Submission to NIST's post-quantum project (2nd round): lattice-based digital signature scheme qTESLA

Name of the cryptosystem: `qTESLA`

Principal and auxiliary submitters:

Nina Bindel, (Principal submitter)	Technische Universität Darmstadt, Hochschulstrasse 10, 64289 Darmstadt, Germany, Email: `nbindel@cdc.informatik.tu-darmstadt.de`, Phone: 004961511620667
	Signature: Nina Bindel
Sedat Akleylek,	Ondokuz Mayis University, Turkey
Erdem Alkim,	Ondokuz Mayis University, Turkey
Paulo S. L. M. Barreto,	University of Washington Tacoma, USA
Johannes Buchmann,	Technische Universität Darmstadt, Germany
Edward Eaton,	ISARA Corporation and Waterloo University, Canada
Gus Gutoski,	ISARA Corporation, Canada
Juliane Krämer,	Technische Universität Darmstadt, Germany
Patrick Longa,	Microsoft Research, USA
Harun Polat,	Technische Universität Darmstadt, Germany
Jefferson E. Ricardini,	University of São Paulo, Brazil
Gustavo Zanon,	University of São Paulo, Brazil

Inventors of the cryptosystem:

All the submitters by name based on a previous scheme by Shi Bai and Steven Galbraith and several other previous works, as explained in the body of this document.

Owners of the cryptosystem:

None (dedicate to the public domain).

Contents

1 Introduction

This document presents a detailed specification of `qTESLA`, a flexible family of post-quantum signature schemes based on the hardness of the decisional Ring Learning With Errors (R-LWE) problem. `qTESLA` is an efficient variant of the Bai-Galbraith signature scheme — which in turn is based on the "Fiat-Shamir with Aborts" framework by Lyubashevsky — adapted to the setting of ideal lattices.

`qTESLA` utilizes *two* different approaches for parameter generation in order to target a wide range of application scenarios. The first approach, referred to as "`heuristic qTESLA`", follows a *heuristic* parameter generation. The second approach, referred to as "`provably-secure qTESLA`", follows a *provably-secure* parameter generation according to existing security reductions.

In addition, `qTESLA` includes the option of using a key compression technique, which we refer to as "public key splitting", that enables a significant reduction in the public key size at the expense of a relatively small increase in the signature size.

Concretely, this document proposes *twelve* parameter sets targeting *four* security levels:

I `Heuristic qTESLA`:

Classical:

(1) `qTESLA-I`: NIST's security category 1.

(2) `qTESLA-II`: NIST's security category 2.

(3) `qTESLA-III`: NIST's security category 3.

(4) `qTESLA-V`: NIST's security category 5.

(5) `qTESLA-V-size`: NIST's security category 5 (option for size).

With public key reduction:

(6) `qTESLA-I-s`: NIST's security category 1.

(7) `qTESLA-II-s`: NIST's security category 2.

(8) `qTESLA-III-s`: NIST's security category 3.

(9) `qTESLA-V-s`: NIST's security category 5.

(10) `qTESLA-V-size-s`: NIST's security category 5 (option for size).

II `Provably-secure qTESLA`:

(1) qTESLA-p-I: NIST's security category 1.

(2) qTESLA-p-III: NIST's security category 3.

The present document is organized as follows. In the remainder of this section, we summarize the main features of qTESLA and describe related previous work. In Section 2, we provide the specification details of the scheme, including a basic and a formal algorithmic description, the functions that are required for the implementation, and the proposed parameter sets. In Section 3, we analyze the performance of our implementations. Section 4 includes the details of the known answer values. Then, we discuss the (provable) security of our proposal in Section 5, including an analysis of the concrete security level and the security against implementation attacks. Section 6 ends this document with a summary of the advantages and limitations of qTESLA.

1.1 qTESLA highlights

qTESLA comes in two flavors: heuristic qTESLA and provably-secure qTESLA. The former is optimized for efficiency and key size while the latter is tailored for high-security applications in which the additional security assurances from the security reduction are valued. In the following paragraphs we highlight relevant properties of each approach.

qTESLA's main features can be summarized as follows:

- **Simplicity.** qTESLA is simple and easy to implement, and its design makes possible the realization of compact and portable implementations that achieve high performance. In addition, the use of a simplified Gaussian sampler is limited to key generation.
- **Parameter flexibility.** qTESLA's flexible design supports parameters defined heuristically or following a provably-secure approach, and supports both power-of-two and non-power-of-two cyclotomic rings.
- **Compactness.** qTESLA signatures are designed to be relatively small, making the combined size of signature and public key competitive with other existing alternatives over ideal lattices. Moreover, qTESLA includes a variant that uses a simple *public key splitting* technique to achieve a dramatic reduction in the public key size at the expense of a relatively small increase in the signature size.
- **Security foundation.** The underlying security of qTESLA is based on the hardness of the decisional R-LWE problem, and comes accompanied by a tight security proof in the (quantum) random oracle model.
- **Practical security.** By design, qTESLA facilitates secure implementations. In particular, it supports *constant-time* implementations (i.e., implementations that are

secure against timing and cache side-channel attacks since their execution time does not depend on secret values), and is inherently protected against certain simple yet powerful fault attacks.

- **Scalability and portability.** qTESLA's simple design makes it straightforward to easily support more than one security level and parameter set with a single, efficient portable implementation.

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

Security. The security of qTESLA is proven using the reductionist approach, i.e., we construct an efficient reduction that turns any successful adversary against qTESLA into one that solves R-LWE. Accordingly, we instantiate heuristic qTESLA such that the corresponding R-LWE parameters (namely the dimension n, the standard deviation of the discrete Gaussian distribution σ, and the modulus q) provide an R-LWE instance of a certain hardness. This approach features high-speed execution and a small memory footprint while requiring relatively compact keys and signatures.
Since our security reductions are also *explicit*, i.e., they explicitly relate an instantiation of qTESLA with an R-LWE instance, we go one step further and choose parameters according to our security reduction. That is, these qTESLA instantiations, which are called provably-secure qTESLA parameter sets, are *provably* secure in the (quantum) random oracle model. For provably-secure qTESLA we present the parameter sets qTESLA-p-I and qTESLA-p-III. Despite these security assurances, provably-secure qTESLA achieves relatively good performance and offers relatively compact signatures. On the downside, this option requires larger public keys.

Flexibility in the underlying ring structure. qTESLA can be defined using the popular ring $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle\Phi_n(x)\rangle = \mathbb{Z}_q[x]/\langle x^n+1\rangle$, with q prime and the dimension n a power of two. This is the definition used to generate most of our parameter sets, namely, qTESLA-I, qTESLA-III, qTESLA-V (and their variants with public key reduction), qTESLA-p-I, and qTESLA-p-III, which use dimensions $n = 512, 1024$ or 2048. This restriction to a power of two, although relatively harmless, does impose some limitations in the parameters that can be used to instantiate qTESLA.
For added flexibility, one can instead use the cyclotomic ring $\mathcal{R}_q = \mathbb{Z}_q[z]/\langle\Phi_k(z)\rangle$, with q prime, $k = 2^\ell \cdot 9$ for a suitably chosen ℓ, and dimension $n = 2^{\ell-1} \cdot 6$. This is the definition used to generate the parameter sets qTESLA-II, qTESLA-II-s, qTESLA-V-size and qTESLA-V-size-s using dimension $n = 768$ and 1536, respectively.

We note that the use of a non-power-of-two dimension requires a slightly different implementation of the number theoretic transform (NTT). Therefore, this option represents a trade-off between an enhanced parameter flexibility and a slight increase in the complexity of the implementation. We describe the non-power-of-two approach in detail in Section 2.6.

In summary, the `qTESLA` family of post-quantum signature schemes offers great flexibility by allowing a selection of parameters and schemes that exhibit different degrees of performance and security. In each case, design decisions have been taken towards enabling efficient yet simple and compact implementations.

1.2 Related work

The signature scheme proposed in this submission is the result of a long line of research. The first work in this line is the signature scheme proposed by Bai and Galbraith [14] which is based on the Fiat-Shamir construction of Lyubashevsky [50]. The scheme by Bai and Galbraith is constructed over standard lattices and comes with a (non-tight) security reduction from the LWE and the short integer solution (SIS) problems in the random oracle model. Dagdelen *et al.* [28] 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 [9], who provided an alternative security reduction from the LWE problem in the quantum random oracle model.

A variant of TESLA over ideal lattices was derived under the name ring-TESLA [1] by Akleylek, Bindel, Buchmann, Krämer, and Marson. Since then, there have appeared subsequent works aimed at improving the efficiency of the scheme [16, 40]. Most notably, a scheme called TESLA# [16] by Barreto, Longa, Naehrig, Ricardini, and Zanon included several implementation improvements. Finally, several works [21, 22, 35] have focused on the analysis of ring-TESLA against side-channel and fault attacks.

Among relevant recent works, we also highlight a recent result by Ducas *et al.* [32], which introduced a compression technique that allows to reduce the size of the public key in some lattice signature schemes. The basic idea consists in splitting the public key in two and moving one the parts to the secret key. During verification the missing information is recovered using a "hint" generated at signing and passed through the signature.

In this document, we consolidate the most relevant features of the prior works with the goal of designing the quantum-secure signature scheme `qTESLA`.

Acknowledgments

We are grateful to Michael Naehrig for his valuable feedback on an earlier version of this document, to Peter Pessl and Matthias Kannwischer for comments and suggestions for a previous version of our implementations, to Kevin Kane and François Gérard for pointing out various typos and suggesting several improvements in a previous version of this document, to Wen Wang for pointing out an optimization in the NTT algorithm, and to Vadim Lyubashevsky for pointing out a typo in an earlier version of the parameter generation script that invalidated the "provably-secure" property of the parameter sets originally submitted to NIST on November 2017 (version 1.0 of this document).

Sedat Akleylek and Erdem Alkim are partially supported by TÜBITAK under grant no. EEEAG-116E279. Nina Bindel and Juliane Krämer are partially supported by the German Research Foundation (DFG) as part of project P1 within the CRC 1119 CROSSING. Edward Eaton is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC). Jefferson Ricardini is supported by the joint São Paulo Research Foundation (FAPESP)/Intel Research grant 2015/50520-6 "Efficient Post-Quantum Cryptography for Building Advanced Security Applications".

2 Specification

In this section, we define basic notation and give an informal description of the basic scheme that is used to specify qTESLA. A formal specification of qTESLA's key generation, signing, and verification algorithms follows in Section 2.3. The correctness of the scheme is discussed in Section 2.4. We describe the implementation of the functions required by qTESLA in Section 2.5, and discuss a variant of qTESLA that allows to use dimensions other than powers of two in Section 2.6. Section 2.7 discusses another variant of qTESLA that yields a smaller public key. Finally, we explain all the system parameters and the proposed parameter sets in Section 2.8.

2.1 Notation

Rings. Let q be an odd prime throughout the document if not stated otherwise. Let $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ denote the quotient ring of integers modulo q, and let $\mathcal{R}$ and $\mathcal{R}_q$ denote the rings $\mathbb{Z}[x]/\langle x^n + 1\rangle$ and $\mathbb{Z}_q[x]/\langle x^n + 1\rangle$, respectively. Given $f = \sum_{i=0}^{n-1} f_i x^i \in \mathcal{R}$, we define the reduction of f modulo q to be $(f \bmod q) = \sum_{i=0}^{n-1}(f_i \bmod q)x^i \in \mathcal{R}_q$. Let $\mathbb{H}_{n,h} = \{f \in \mathcal{R}_q \mid f = \sum_{i=0}^{n-1} f_i x^i,\ f_i \in \{-1,0,1\},\ \sum_{i=0}^{n-1}|f_i| = h\}$, and $\mathcal{R}_{q,[B]} = \{f \in \mathcal{R}_q \mid f = \sum_{i=0}^{n-1} f_i x^i,\ f_i \in [-B,B]\}$.

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

Infinity norm. Given $f \in \mathcal{R}$, the function $\max_k(f)$ returns the k-th largest absolute coefficient of f. That is, if the coefficients of f are reordered as to produce a polynomial g with coefficients ordered (without losing any generality) as $|g_1| \geq |g_2| \geq ... \geq |g_n|$, then $\max_k(f) = g_k$. For an element $c \in \mathbb{Z}_q$, we have that $\|c\|_\infty = |c \bmod^{\pm} q|$, and we 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 $j = 1, ..., k$. In the case of sparse polynomials $c \in \mathbb{H}_{n,h}$, these polynomials are encoded as the two arrays $\mathit{pos\_list} \in \{0, ..., n-1\}^h$ and $\mathit{sign\_list} \in \{-1,1\}^h$ representing the positions and signs of the nonzero coefficients of c, respectively. We denote this by $c \triangleq \{\mathit{pos\_list}, \mathit{sign\_list}\}$.

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 of 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 $\mathcal{D}_\sigma$. For a finite set S, we denote sampling the element s uniformly from S with $s \leftarrow_\$ S$.

2.2 Basic signature scheme

Informal descriptions of the algorithms that give rise to the signature scheme `qTESLA` are shown in Algorithms 1, 2, and 3. These algorithms require two basic terms, namely, *B-short* and *well-rounded*, which are defined below.

Let q, L_E, L_S, E, S, B, and d be system parameters that denote the modulus, the bound constant for error polynomials, the bound constant for the secret polynomial, two rejection bounds used during signing and verification that are related to L_E and L_S, the bound for the random polynomial at signing, and the rounding value, respectively. An integer polynomial y is *B-short* if each coefficient is at most B in absolute value. We call an integer polynomial w *well-rounded* if w is $(\lfloor q/2 \rceil - E)$-short and $[w]_L$ is $(2^{d-1} - E)$-short.

In Algorithms 1–3, we assume for simplicity that the hash oracle $\mathsf{H}(\cdot)$ maps to $\mathbb{H}$, where $\mathbb{H}$ denotes the set of polynomials $c \in \mathcal{R}$ with coefficients in $\{-1,0,1\}$ with exactly h nonzero entries, i.e., we ignore the encoding function Enc introduced in Section 2.3.

Because of the random generation of the polynomial y (see line 1 of Alg. 2), Algorithm 2 is described as a *non-deterministic* algorithm. This property implies that different random-

Algorithm 1 Informal description of the key generation

```
Require: -
Ensure: Secret key $sk = (s, e_1, ..., e_k, a_1, ..., a_k)$, and public key $pk = (a_1, ..., a_k, t_1, ..., t_k)$
1: $a_1, ..., a_k \leftarrow \mathcal{R}_q$ ring elements.
2: Choose $s \in \mathcal{R}$ with entries from $\mathcal{D}_\sigma$. Repeat step if the $h$ largest entries of $s$ sum to at least $L_S$.
3: For $i = 1, ..., k$: Choose $e_i \in \mathcal{R}$ with entries from $\mathcal{D}_\sigma$. Repeat step at iteration $i$ if the $h$ largest
   entries of $e_i$ sum to at least $L_E$.
4: For $i = 1, ..., k$: Compute $t_i \leftarrow a_i s + e_i \in \mathcal{R}_q$.
5: Return $sk = (s, e_1, ..., e_k, a_1, ..., a_k)$ and $pk = (a_1, ..., a_k, t_1, ..., t_k)$.
```

Algorithm 2 Informal description of the signature generation

Require: Message m, secret key $sk = (s, e_1, ..., e_k, a_1, ..., a_k)$
Ensure: Signature (z, c)

```
1: Choose $y$ uniformly at random among $B$-short polynomials in $\mathcal{R}_q$.
2: $c \leftarrow \mathsf{H}([a_1 y]_M, ..., [a_k y]_M, m)$.
3: Compute $z \leftarrow y + sc$.
4: If $z$ is not $(B - S)$-short then retry at step 1.
5: For $i = 1, ..., k$: If $a_i y - e_i c$ is not well-rounded then retry at step 1.
6: Return $(z, c)$.
```

Algorithm 3 Informal description of the signature verification

Require: Message m, public key $pk = (a_1, ..., a_k, t_1, ..., t_k)$, and signature (z, c)
Ensure: "accept" or "reject" signature

```
1: If $z$ is not $(B - S)$-short then return reject.
2: For $i = 1, ..., k$: Compute $w_i \leftarrow a_i z - t_i c \in \mathcal{R}_q$.
3: If $c \neq \mathsf{H}([w_1]_M, ..., [w_k]_M, m)$ then return reject.
4: Return accept.
```

ness is required for each signature. For the formal specification of `qTESLA` we incorporate an additional improvement: `qTESLA` requires a combination of fresh randomness and a fixed value for the generation of y (see Section 2.3). This design feature is added in order to prevent some implementation pitfalls and, at the same time, protect against some simple but devastating fault attacks. We discuss the advantages of our approach in Section 5.3.

2.3 Formal description of `qTESLA`

`qTESLA` is parameterized by λ, κ, n, k, q, σ, L_E, L_S, E, S, B, d, h, and b_{GenA}; see Table 4 in Section 2.8 for a detailed description of all the system parameters. The following functions are required for the implementation of the scheme:

- The pseudorandom function $\mathsf{PRF}_1 : \{0,1\}^\kappa \rightarrow \{0,1\}^{\kappa,k+3}$, 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 $\mathsf{G} : \{0,1\}^* \rightarrow \{0,1\}^{512}$, which maps a message m to a 512-bit string.
- The pseudorandom function $\mathsf{PRF}_2 : \{0,1\}^\kappa \times \{0,1\}^\kappa \times \{0,1\}^{512} \rightarrow \{0,1\}^\kappa$, which takes as inputs seed_y and the random value r, each κ bits long, and the hash G of a message m, which is 512-bit long, and maps them to the κ-bit seed rand.

- The generation function of the public polynomials $a_1, ..., a_k$, $\mathsf{GenA} : \{0,1\}^\kappa \to \mathcal{R}_q^k$, which takes as input the κ-bit seed seed_a and maps it to k polynomials $a_i \in \mathcal{R}_q$.
- The Gaussian sampler function $\mathsf{GaussSampler} : \{0,1\}^\kappa \times \mathbb{Z} \to \mathcal{R}$, which takes as inputs a κ-bit seed $\mathsf{seed} \in \{\mathsf{seed}_s, \mathsf{seed}_{e_1}, \ldots, \mathsf{seed}_{e_k}\}$ and a nonce $\mathsf{counter} \in \mathbb{Z}_{>0}$, and outputs a secret or error polynomial in $\mathcal{R}$ sampled according to the Gaussian distribution $\mathcal{D}_\sigma$. To realize $\mathsf{GaussSampler}$, we propose a simple yet efficient constant-time algorithm. This is described in Section 2.5.4.
- The encoding function $\mathsf{Enc} : \{0,1\}^\kappa \to \{0, ..., n-1\}^h \times \{-1,1\}^h$. This function encodes a κ-bit hash value c' as a polynomial $c \in \mathbb{H}_{n,h}$. The polynomial c is represented as the two arrays *pos_list* $\in \{0, ..., n-1\}^h$ and *sign_list* $\in \{-1,1\}^h$ containing the positions and signs of its nonzero coefficients, respectively.
- The sampling function $\mathsf{ySampler} : \{0,1\}^\kappa \times \mathbb{Z} \to \mathcal{R}_{q,[B]}$ samples a polynomial $y \in \mathcal{R}_{q,[B]}$ taking as inputs a κ-bit seed rand and a nonce $\mathsf{counter} \in \mathbb{Z}_{>0}$.
- The hash-based function $\mathsf{H} : \mathcal{R}_q^k \times \{0,1\}^* \to \{0,1\}^\kappa$. This function takes as inputs k polynomials $v_1, ..., v_k \in \mathcal{R}_q$ and computes $[v_1]_M, ..., [v_k]_M$. The result is then hashed together with the hash G of a given message m to a string κ bits long.
- The correctness check function checkE, which gets an error polynomial e as input and rejects it if $\sum_{k=1}^{h} \max_k(e)$ is greater than some bound L_E; see Algorithm 5. The function checkE guarantees the correctness of the signature scheme by ensuring that $\|e_i c\|_\infty \leq E$ for $i = 1, ..., k$ during key generation, as described in Section 2.4.
- 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 4. checkS ensures that $\|sc\|_\infty \leq S$, 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 6, 7 and 8, 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 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 Gaussian sampling, the corresponding value from $\{\mathsf{seed}_s, \mathsf{seed}_{e_1}, \ldots, \mathsf{seed}_{e_k}\}$ is used as seed, and a counter is used as nonce to provide domain separation between the different calls to the sampler. Accordingly, this counter is initialized at 1 and then increased by 1 after each invocation to the Gaussian sampler. Finally, the secret key sk consists of $s, e_1, \ldots, e_k$ and the seeds seed_a and seed_y, and the public key pk consists of seed_a and the polynomials $t_i = a_i s + e_i \mod q$ for $i = 1, \ldots, k$. All the seeds required during key generation are generated by expanding a pre-seed $\mathsf{pre\text{-}seed}$ using PRF_1.

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

The security check (lines 12–15), also called the *rejection sampling*, is used to ensure that the signature does not leak any information about the secret s. It is realized by checking that $z \notin \mathcal{R}_{q,[B-S]}$. 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) has a *dual* purpose: it ensures that the signature does not leak any secret information and also 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, the algorithm returns the signature (z, c') on m.

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

Algorithm 4 checkS: simplifies the security reduction by ensuring that $\|sc\|_\infty \leq S$.

```
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 5 checkE: ensures correctness of the scheme by checking that $\|ec\|_\infty \leq E$.

```
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 6 qTESLA's key generation

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

Algorithm 7 `qTESLA`'s signature generation

Require: message m, and secret key $sk = (s, e_1, ..., e_k, \mathsf{seed}_a, \mathsf{seed}_y)$
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: $y \leftarrow \mathsf{ySampler}(\mathsf{rand}, \mathsf{counter})$                [Algorithm 12]
 5: $a_1, ..., a_k \leftarrow \mathsf{GenA}(\mathsf{seed}_a)$                    [Algorithm 10]
 6: for $i = 1, ..., k$ do
 7:     $v_i = a_i y \bmod^{\pm} q$
 8: end for
 9: $c' \leftarrow \mathsf{H}(v_1, ..., v_k, \mathsf{G}(m))$                     [Algorithm 13]
10: $c \triangleq \{pos\_list, sign\_list\} \leftarrow \mathsf{Enc}(c')$          [Algorithm 14]
11: $z \leftarrow y + sc$
12: if $z \notin \mathcal{R}_{q,[B-S]}$ then
13:     $\mathsf{counter} \leftarrow \mathsf{counter} + 1$
14:     Restart at step 4
15: end if
16: for $i = 1, ..., k$ do
17:     $w_i \leftarrow v_i - e_i c \bmod^{\pm} q$
18:     if $\|[w_i]_L\|_\infty \geq 2^{d-1} - E \vee \|w_i\|_\infty \geq \lfloor q/2 \rfloor - E$ then
19:         $\mathsf{counter} \leftarrow \mathsf{counter} + 1$
20:         Restart at step 4
21:     end if
22: end for
23: return $(z, c')$
```

Algorithm 8 `qTESLA`'s signature verification

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

```
1: $c \triangleq \{pos\_list, sign\_list\} \leftarrow \mathsf{Enc}(c')$          [Algorithm 14]
2: $a_1, ..., a_k \leftarrow \mathsf{GenA}(\mathsf{seed}_a)$                    [Algorithm 10]
3: for $i = 1, ..., k$ do
4:     $w_i \leftarrow a_i z - t_i c \bmod^{\pm} q$
5: end for
6: if $z \notin \mathcal{R}_{q,[B-S]} \vee c' \neq \mathsf{H}(w_1, ..., w_k, \mathsf{G}(m))$ then
7:     return $-1$
8: end if
9: return 0
```

2.4 Correctness of the scheme

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

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

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

In particular, to guarantee the correctness of `qTESLA` it must hold that for a signature (z, c') of a message m generated by Algorithm 7: (i) $z \in \mathcal{R}_{q,[B-S]}$ and (ii) the output of the hash-based function H at signing (line 9 of Algorithm 7) is the same as the analogous output at verification (line 6 of Algorithm 8). Requirement (i) is ensured by the security check during signing (line 12 of Algorithm 7). To ensure (ii), we need to prove that, for genuine signatures and for all $i = 1, ..., k$ it holds that $[a_i y \bmod^{\pm} q]_M = [a_i z - t_i c \bmod^{\pm} q]_M = [a_i(y+sc) - (a_i s + e_i)c \bmod^{\pm} q]_M = [a_i y + a_i sc - a_i sc - e_i c \bmod^{\pm} q]_M = [a_i y - e_i c \bmod^{\pm} q]_M$. From the definition of $[\cdot]_M$, this means proving that $(a_i y \bmod^{\pm} q - [a_i y \bmod^{\pm} q]_L)/2^d = (a_i y - e_i c \bmod^{\pm} q - [a_i y - e_i c \bmod^{\pm} q]_L)/2^d$, or simply $[a_i y \bmod^{\pm} q]_L = e_i c + [a_i y - e_i c \bmod^{\pm} q]_L$.

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

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

$$[\alpha]_L = \varepsilon + [\alpha - \varepsilon \bmod^{\pm} q]_L. \tag{1}$$

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

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

and therefore

$$[[\varepsilon]_L + [\alpha - \varepsilon \bmod^{\pm} q]_L]_L = [\varepsilon]_L + [\alpha - \varepsilon \bmod^{\pm} q]_L = \varepsilon + [\alpha - \varepsilon \bmod^{\pm} q]_L. \tag{2}$$

Next we prove that

$$[[\varepsilon]_L + [\alpha - \varepsilon \bmod^{\pm} q]_L]_L = [\alpha]_L. \tag{3}$$

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

$$\begin{aligned}
& [[\varepsilon \bmod^{\pm} q]_L + [\alpha - \varepsilon \bmod^{\pm} q]_L]_L & (4)\\
= \; & ((\varepsilon \bmod^{\pm} q) \bmod^{\pm} 2^d + (\alpha - \varepsilon \bmod^{\pm} q) \bmod^{\pm} 2^d) \bmod^{\pm} 2^d & (5)\\
& \text{by the definition of } [\cdot]_L & \\
= \; & (\varepsilon \bmod^{\pm} q + (\alpha - \varepsilon \bmod^{\pm} q)) \bmod^{\pm} 2^d. & (6)
\end{aligned}$$

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 (6) is the same as

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

Combining equations (2) and (3) we deduce that $[\alpha]_L = \varepsilon + [\alpha - \varepsilon \bmod^{\pm} q]_L$, which is the equation we needed to prove. $\square$

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 7, 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 a valid signature, and that Algorithm 6 (line 13) guarantees $\| e_i c \|_\infty \leq E$. Likewise, by definition it holds that $E < \lfloor q/2 \rfloor$; see Section 2.8. Finally, $v_i = a_i y$ is reduced $\bmod^{\pm} q$ in line 7 of Algorithm 7 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 $i = 1, ..., k$ as argued above.

2.5 Realization of the required functions

2.5.1 Hash and pseudorandom functions

In addition to the hash-based functions G and H and the pseudorandom functions PRF_1 and PRF_2, several functions that are used for the implementation of `qTESLA` require pseudorandom bit generation. This functionality is provided by so-called extendable output functions (XOF).

For the remainder, the format that we use to call a XOF is given by $\mathsf{XOF}(\mathsf{X}, \mathsf{L}, \mathsf{D})$, where X is the input string, L specifies the output length in bytes, and D specifies an optional domain separator [1].

Next, we summarize how XOFs are instantiated using SHAKE [33] and cSHAKE [43] in the different functions requiring hashing or pseudorandom bit generation.

- PRF_1: the XOF is instantiated with SHAKE128 (resp. SHAKE256) for Level-`I` and Level-`II` parameter sets (resp. for other parameter sets); see Algorithm 9.
- PRF_2: the same as PRF_1.
- GenA: the XOF is instantiated with cSHAKE128 (see Algorithm 10).

[1]The domain separator D is used with cSHAKE, but ignored when SHAKE is used.

- $\mathsf{GaussSampler}$: the XOF is instantiated with cSHAKE128 (resp. cSHAKE256) for Level-`I` and Level-`II` parameter sets (resp. for other parameter sets); see Algorithm 11.
- Enc: the XOF is instantiated with cSHAKE128 (see Algorithm 14).
- $\mathsf{ySampler}$: the XOF is instantiated with cSHAKE128 (resp. cSHAKE256) for Level-`I` and Level-`II` parameter sets (resp. for other parameter sets); see Algorithm 12.
- Hash G: this function is instantiated with SHAKE128 (resp. SHAKE256) for Level-`I` and Level-`II` parameter sets (resp. for other parameter sets).
- Hash-based function H: the hashing in this function is instantiated with SHAKE128 (resp. SHAKE256) for Level-`I` and Level-`II` parameter sets (resp. for other parameter sets); see Algorithm 13.

In the cases of the functions GenA, Enc, G and H, implementations of `qTESLA` need to follow strictly the XOF specifications based on SHAKE/cSHAKE given above in order to be specification compliant. However, for the rest of the cases (i.e., PRF_1, PRF_2, $\mathsf{ySampler}$ and $\mathsf{GaussSampler}$) users can opt for a different cryptographic PRF.

2.5.2 Pseudorandom bit generation of seeds, PRF_1

`qTESLA` requires the generation of seeds during key generation; see line 3 of Algorithm 6. These seeds are then used to produce the polynomials s, e_i, a_i and y. Specifically, these seeds are:

- seed_s, which is used to generate the polynomial s,
- seed_{e_i}, which are used to generate the polynomials e_i for $i = 1, \ldots, k$,
- seed_a, which is used to generate the polynomials a_i for $i = 1, \ldots, k$, and
- seed_y, which is used to generate the polynomial y.

The size of each of these seeds is κ bits. In the accompanying implementations, the seeds are generated by first calling the system random number generator (RNG) to produce a pre-seed of size κ bits at line 2 of Algorithm 6, and then expanding this pre-seed through Algorithm 9. As explained in Section 2.5.1, in this case the XOF function is instantiated with SHAKE in our implementations.

Algorithm 9 Seed generation, PRF_1

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

2.5.3 Generation of $\mathbf{a_1, ..., a_k}$

In qTESLA, the polynomials $a_1, ..., a_k$ are freshly generated per secret/public keypair using the seed seed_a during key generation; see line 4 of Algorithm 6. This seed is then stored as part of both the private and public keys so that the signing and verification operations can regenerate $a_1, ..., a_k$.

The approach above permits to save bandwidth since we only need κ bits to store seed_a instead of the $k \cdot n \cdot \lceil \log_2 q \rceil$ bits that are required to represent the full polynomials. Moreover, the use of fresh $a_1, ..., a_k$ per keypair makes the introduction of backdoors more difficult and reduces drastically the scope of all-for-the-price-of-one attacks [10, 16].

The procedure depicted in Algorithm 10 to generate $a_1, ..., a_k$ is as follows. The seed seed_a obtained from Algorithm 9 is expanded to $(\text{rate}_{\mathsf{XOF}} \cdot b_{\mathsf{GenA}})$ bytes using cSHAKE128, where $\text{rate}_{\mathsf{XOF}}$ is the SHAKE128 rate constant 168 [33] and b_{GenA} is a qTESLA parameter that represents the number of blocks requested in the first XOF call. Then, the algorithm proceeds to do rejection sampling over each $8\lceil \log_2 q \rceil$-bit string of the cSHAKE output modulo $2^{\lceil \log_2 q \rceil}$, discarding every package that has a value equal or greater than the modulus q. Since there is a possibility that the cSHAKE output is exhausted before all the $k \cdot n$ coefficients are filled out, the algorithm permits successive (and as many as necessary) calls to the function requesting $\text{rate}_{\mathsf{XOF}}$ bytes each time (lines 5–8). 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 to generate $a_1, ..., a_k$ produces polynomials with uniformly random coefficients. Thus, following a standard practice, qTESLA assumes that the resulting polynomials $a_1, ..., a_k$ from Algorithm 10 are already in the NTT domain. This permits an important speedup of the polynomial operations without affecting security. We remark, however, that this assumption does affect the correctness and, hence, implementations should follow this design feature to be specification compliant.
Refer to Section 2.5.8 for details about the NTT computations.

Algorithm 10 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 $\text{rate}_{\mathsf{XOF}} = 168$
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($\mathsf{seed}_a, \text{rate}_{\mathsf{XOF}} \cdot b', D$), where each $\langle c_t\rangle \in \{0,1\}^{8b}$
3:  $i \leftarrow 0$, $pos \leftarrow 0$
4:  while $i < k \cdot n$ do
5:      if $pos > \lfloor(\text{rate}_{\mathsf{XOF}} \cdot b')/b\rfloor - 1$ 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($\mathsf{seed}_a, \text{rate}_{\mathsf{XOF}} \cdot b', D$), where each $\langle c_t\rangle \in \{0,1\}^{8b}$
8:      end if
9:      if $q > c_{pos} \bmod 2^{\lceil \log_2 q\rceil}$ then
10:         $a_{\lfloor i/n\rfloor+1, i-n\cdot\lfloor i/n\rfloor} \leftarrow c_{pos} \bmod 2^{\lceil\log_2 q\rceil}$, where a polynomial $a_x$ is interpreted as a vector of coefficients $(a_{x,0}, a_{x,1}, \ldots, a_{x,n-1})$
11:         $i \leftarrow i + 1$
12:     end if
13:     $pos \leftarrow pos + 1$
14: end while
15: return $(a_1, ..., a_k)$
```

2.5.4 Gaussian sampling

One of the advantages of qTESLA is that Gaussian sampling is only required during key generation to sample $e_1, ..., e_k$, and s (see Alg. 6). 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. In that direction, we propose an efficient and portable "constant-time" 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 Gaussian sample, one picks a uniform sample $u \leftarrow_\$ \mathbb{Z}/2^\beta\mathbb{Z}$, looks it up in the table, and returns the value z such that $\mathsf{CDT}[z] \leqslant u < \mathsf{CDT}[z+1]$.

A CDT-based approach has apparently first been considered for cryptographic purposes by Peikert [56] (in a somewhat more complex form). The approach was assessed and deemed mostly impractical by Ducas *et al.* [30], since it would take $\beta t \sigma$ bits. Yet, they only considered a scenario where the standard deviation σ was at least 107, and as high as 271. As a result, table sizes around 78 Kbytes are reported (presumably for $\sigma = 271$ with roughly 160-bit sampling precision). For the qTESLA parameter sets, however, the values of σ are much smaller, making the CDT approach feasible, as one can see in Table 1.

Table 1: CDT dimensions used in the accompanying `qTESLA` implementations on 32- and 64-bit platforms (targeted precision β : implemented precision in bits : size in bytes). The notation `-s` in parenthesis represents the corresponding parameter sets with public key reduction.

Parameter set	32-bit CPU	64-bit CPU
`qTESLA-I` (`-s`)	64 : 63 : 1656	64 : 63 : 1672
`qTESLA-II` (`-s`)	96 : 94 : 1320	128 : 127 : 2048
`qTESLA-III` (`-s`)	128 : 125 : 2128	128 : 127 : 2160
`qTESLA-V` (`-s`)	224 : 218 : 4956	256 : 253 : 6112
`qTESLA-V-size` (`-s`)	224 : 218 : 4956	256 : 253 : 6112
`qTESLA-p-I`	64 : 63 : 624	64 : 63 : 632
`qTESLA-p-III`	128 : 125 : 1776	128 : 127 : 1792

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

(i) On platforms where a reasonably large number of Gaussian samples can be generated at once, one can sort a list of uniformly random samples together with the CDT itself, then identify the CDT entries between which each sample is located. The cost of sorting, which can be implemented in a constant-time fashion using, e.g., Batcher's odd-even mergesort [18] (also called merge-exchange sorting [45, Section 5.2.2 Algorithm M (Merge exchange)]), is thus amortized among all samples.

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

Amortized sorting approach. Assume that BatcherMergeExchange($\langle sequence\rangle$, key:$\langle key\rangle$, data:$\langle data\rangle$) is a constant-time implementation of the Batcher merge-exchange sorting algorithm for $\langle sequence\rangle$, using the specified $\langle key\rangle$ field of each of its entries for ordering, and carrying the corresponding $\langle data\rangle$ field(s) as associated data. Algorithm 11 generates n Gaussian samples, a chunk of $c \mid n$ samples at a time, in a constant-time fashion.

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

Implementation details. In the accompanying implementations, we implement an optimized version of Algorithm 11 that has been tuned for computer wordsize $w = 32$ and 64. The chunk size is fixed to $c = 512$ when the dimension n is a multiple of 512 (in the case of `qTESLA-II` and `qTESLA-II-s`, for which $n = 768$, we set $c = 256$). In all the cases, the targeted sampling precision β is conservatively set to a value much greater than $\lambda/2$, as can be seen in Table 1. Note that the small gap between the values of β and the actual precision provided in the implementations is due to the use of some of the bits for signs in the CDT table entries. The required precomputed CDT tables are generated using the script provided in the folder `\Supporting_Documentation\Script_for_Gaussian_sampler`.

In our implementations, in order to make the Gaussian sampler constant-time we make sure that basic operations such as comparisons are not implemented with conditional jumps that depend on secret data, and that lookup tables are always fully scanned at each pass extracting entries with logical operations.

As stated in Section 2.5.1, for the pseudorandom bit generation required by Algorithm 11, we use cSHAKE as XOF using a seed seed produced by PRF_1 (see line 3 of Algorithm 6) as input string, and a nonce D (written as $\mathsf{counter}$ in Algorithm 6) as domain separator.

2.5.5 Sampling of y

The sampling of the polynomial y (line 4 of Algorithm 7) can be performed by generating n $(\lceil \log_2 B \rceil + 1)$-bit values uniformly at random, and then correcting each value to the range $[-B, B+1]$ with a subtraction by B. Since values need to be in the range $[-B, B]$, coefficients with value $B+1$ need to be rejected, which in turn might require the generation of additional random bits.

Algorithm 12 depicts the procedure used in our implementations. For the pseudorandom bit generation, the seed rand produced by PRF_2 (see line 3 of Algorithm 7) is used as input string to a XOF, while the nonce D (written as $\mathsf{counter}$ in Algorithm 7) is used 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

Algorithm 11 Constant-time CDT-based Gaussian sampling, GaussSampler

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

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

of the XOF in the signing algorithm.
Our implementations use cSHAKE as the XOF function.

Algorithm 12 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: $y \in \mathcal{R}_{q,[B]}$

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

2.5.6 Hash-based function H

This function takes as inputs k polynomials $v_1, \dots, v_k \in \mathcal{R}_q$ and computes $[v_1]_M, ..., [v_k]_M$. The result is hashed together with the hash G of a message m to a string c' that is κ bits long. The detailed procedure is as follows. Let each polynomial v_i be interpreted as a vector of coefficients $(v_{i,0}, v_{i,1}, \dots, v_{i,n-1})$, where $v_{i,j} \in (-q/2, q/2]$, i.e., $v_{i,j} = v_{i,j} \bmod^{\pm} q$. We first compute $[v_i]_L$ by reducing each coefficient modulo 2^d and subtracting 2^d from the result if it is greater than 2^{d-1}. This guarantees a result in the range $(-2^{d-1}, 2^{d-1}]$, as required by the definition of $[\cdot]_L$. Next, we compute $[v_i]_M$ as $(v_i - [v_i]_L)/2^d$. Since each resulting coefficient is guaranteed to be very small it is stored as a byte, which in total makes up a string of $k \cdot n$ bytes. Finally, SHAKE is used to hash the resulting $k \cdot n$-byte string together with the 64-byte digest $\mathsf{G}(m)$ to the κ-bit string c'. This procedure is depicted in Algorithm 13.

2.5.7 Encoding function

This function maps the bit string c' to a polynomial $c \in \mathbb{H}_{n,h} \subset \mathcal{R}_q$ of degree $n-1$ with coefficients in $\{-1, 0, 1\}$ and weight h, i.e., c has h coefficients that are either 1 or -1. For

Algorithm 13 Hash-based function H

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

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

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 [1,30]. Basically, the idea is to use a 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 14. This works as follows. The algorithm first requests $\text{rate}_{\mathsf{XOF}}$ bytes from a XOF, and the output stream is interpreted as an array of 3-byte packets in little endian format. Each 3-byte packet is then processed as follows, beginning with the least significant packet. The $\lceil \log_2 n \rceil$ least significant bits of the lowest two bytes in every packet are interpreted as an integer value in little endian representing the position *pos* of a nonzero coefficient of c. If such value already exists in the *pos_list* array, the 3-byte packet is rejected and the next packet in line is processed; otherwise, the packet is accepted, the value is added to *pos_list* as the position of a new coefficient, and then the third byte is used to determine the coefficient's sign as follows. If the least significant bit of the third byte is 0, the coefficient is assumed to be positive (+1), otherwise, it is taken as negative (−1). In our implementations, *sign_list* encodes positive and negative coefficients as 0's and 1's, respectively.
The procedure above is executed until *pos_list* and *sign_list* are filled out with h entries each. If the XOF output is exhausted before completing the task then additional calls are invoked, requesting $\text{rate}_{\mathsf{XOF}}$ bytes each time. `qTESLA` uses cSHAKE128 as the XOF function, with the value $D = 0$ as domain separator for the first call. D is incremented by one at each subsequent call.

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

2.5.8 Polynomial multiplication and the number theoretic transform

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

Since qTESLA specifies the generation of the polynomials $a_1, \ldots, a_k$ directly in the NTT domain for efficiency purposes (see Section 2.5.3), we need to define polynomials in such a domain. For the remainder of this section we limit the discussion to the standard case of a power-of-two NTT. The non-power-of-two case is treated in Section 2.6.

Let ω be a primitive n-th root of unity in $\mathbb{Z}_q$, i.e., $\omega^n \equiv 1 \mod q$, and let ϕ be a primitive $2n$-th root of unity in $\mathbb{Z}_q$ such that $\phi^2 = \omega$. Then, given a polynomial $a = \sum_{i=0}^{n-1} a_i x^i$ the forward transform is defined as

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

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

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

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

As mentioned earlier, the outputs $a_1, ..., a_k$ of GenA are assumed to be in the NTT domain. In particular, let $\tilde{a_i}$ be the output a_i in the NTT domain. Polynomial multiplications $a_i \cdot b$ for some polynomial $b \in \mathcal{R}_q$ can be efficiently realized as $\mathsf{NTT}^{-1}(\tilde{a_i} \circ \mathsf{NTT}(b))$.

To compute the NTT in our implementations, we adopt butterfly algorithms to compute the NTT 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 a so-called bit-reversal operation which is required by some implementations [10, 59, 60]. Specifically, we use an algorithm that computes the forward NTT based on the Cooley-Tukey butterfly that absorbs the products of the root powers in bit-reversed ordering. This algorithm receives the inputs of a polynomial a in standard ordering and produces a result in bit-reversed ordering. Similarly, for the inverse NTT we use an algorithm based on the Gentleman-Sande butterfly that absorbs the inverses of the products of the root powers in the bit-reversed ordering. The algorithm receives the inputs of a polynomial $\tilde{a}$ in the bit-reversed ordering and produces an output in standard ordering. Efficient versions of these algorithms, which we follow for our implementations, can be found in [62, Algorithms 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. In our implementations we use Algorithm 15 to realize the multiplications in lines 11 and 17 of Algorithm 7 and in line 4 of Algorithm 8, 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, Algorithm 14).

Algorithm 15 Sparse Polynomial Multiplication

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

Ensure: $f = g \cdot c \in \mathcal{R}_q$

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

2.6 qTESLA variant with non-power-of-two cyclotomic rings

In order to attain greater flexibility in our parameter selection, we propose a qTESLA variant defined over the non-power-of-two cyclotomic ring $\mathcal{R}_q = \mathbb{Z}_q[z]/\langle \Phi_k(z) \rangle$, with $k = 2^\ell \cdot 9$ for a suitably chosen ℓ and for a prime q such that $k \mid q-1$. In this case, the dimension is $n = 2^{\ell-1} \cdot 6$.

Concretely, we propose four instances, namely qTESLA-II, qTESLA-II-s, qTESLA-V-size and qTESLA-V-size-s, that use $\mathcal{R}_q = \mathbb{Z}_q[z]/\langle \Phi_{2^\ell 9}(z) \rangle$. The signature scheme only requires small modifications to work in this setting; see at the end of this subsection for a summary of changes. Thus, the main difference with respect to using the standard power-of-two cyclotomic ring resides on the NTT. This is explained in detail in the following.

The NTT in $\mathbb{Z}_q[z]/\Phi_{2^\ell 9}(z)$. We adopt a tower approach to the ring arithmetic, representing $\mathcal{R}_q$ as the bivariate ring $\mathbb{Z}_q[x,y]/\langle \Phi_{2^\ell}(x), \Phi_9(y) \rangle = (\mathbb{Z}_q[x]/\langle \Phi_{2^\ell}(x) \rangle)[y]/\langle \Phi_9(y) \rangle$. This enables decomposing the NTT on $\mathcal{R}_q$ into six independent, parallel instances of the NTT in the smaller ring $\mathbb{Z}_q[x]/\langle x^{2^{\ell-1}} + 1 \rangle$, linked together by one instance of the NTT in the small ring $\mathbb{Z}_q[y]/\langle y^6 + y^3 + 1 \rangle$.

In other words, to compute the NTT of a given polynomial $u = (u_0, ..., u_{n-1}) \in \mathcal{R}_q$, we first split its $n = 2^{\ell-1} \cdot 6$ polynomial coefficients into six consecutive blocks $(u_{2^{\ell-1}i+j} \mid 0 \leqslant j < 2^{\ell-1})$ of $2^{\ell-1}$ coefficients each for $0 \leqslant i < 6$. Afterwards, we apply the usual binary NTT of order $2^{\ell-1}$ described in Section 2.5.8, or NTT_2 for short, to each of the blocks separately. Then we make $2^{\ell-1}$ sextuples with one coefficient from each block at matching indices, namely $(u_{2^{\ell-1}i+j} \mid 0 \leqslant i < 6)$ for $0 \leqslant j < 2^{\ell-1} - 1$, and apply the NTT of order 6 to each sextuple separately.

Our efficient implementation of NTT of degree 6 is described next. In what follows, assume $\theta \in \mathbb{Z}_q$ to be a primitive 9-th root of unity available as a precomputed table $[\theta^j \mid 0 \leqslant j < 9]$, and $\zeta := \theta^3$.

The NTT of order 6. Let U be the basis matrix of the ideal defined by an element $u \in \mathbb{Z}_q[y]/\langle y^6 + y^3 + 1\rangle$. The matrix U can be diagonalized via multiplication by the Vandermonde matrix $V := \mathrm{vdm}(\theta, \theta^2, \theta^4, \theta^5, \theta^7, \theta^8)$. Namely, the eigenvalues of U are the components of $u' \leftarrow u \cdot V$. This computation can be expedited by noticing that

$$V = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \theta & 0 & 0 & 0 & 0 \\ 0 & 0 & \theta^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & \theta^3 & 0 & 0 \\ 0 & 0 & 0 & 0 & \theta^4 & 0 \\ 0 & 0 & 0 & 0 & 0 & \theta^5 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \theta & \zeta & \zeta\theta & \zeta^2 & \zeta^2\theta \\ 1 & \theta^2 & \zeta^2 & \zeta^2\theta^2 & \zeta & \zeta\theta^2 \\ 1 & \zeta & 1 & \zeta & 1 & \zeta \\ 1 & \zeta\theta & \zeta & \zeta^2\theta & \zeta^2 & \theta \\ 1 & \zeta\theta^2 & \zeta^2 & \theta^2 & \zeta & \zeta^2\theta^2 \end{bmatrix}.$$

Therefore we can write

$$[u'_0, u'_1, u'_2, u'_3, u'_4, u'_5] \leftarrow \mathsf{NTT}\left([u_0, u_1, u_2, u_3, u_4, u_5]\right) = (\mu \circ \psi)\left([u_0, u_1, u_2, u_3, u_4, u_5]\right),$$

where ψ denotes right-multiplication by the first matrix of the product V and μ denotes right-multiplication by the second matrix of the product V. We now provide efficient algorithms for the transforms by ψ and ϕ.

The ϕ transform is defined as

$$[\tilde{u}_0, \tilde{u}_1, \tilde{u}_2, \tilde{u}_3, \tilde{u}_4, \tilde{u}_5] \leftarrow [u_0, u_1, u_2, u_3, u_4, u_5] \cdot \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \theta & 0 & 0 & 0 & 0 \\ 0 & 0 & \theta^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & \theta^3 & 0 & 0 \\ 0 & 0 & 0 & 0 & \theta^4 & 0 \\ 0 & 0 & 0 & 0 & 0 & \theta^5 \end{bmatrix}.$$

The computation of the values $\tilde{u}_0, ..., \tilde{u}_5$ can be implemented (at cost of $5\mathbf{M}$) [2]

$$\begin{array}{llllll} \tilde{u}_0 \leftarrow u_0, & & \tilde{u}_1 \leftarrow u_1 \cdot \theta, & & \tilde{u}_2 \leftarrow u_2 \cdot \theta^2, \\ \tilde{u}_3 \leftarrow u_3 \cdot \theta^3, & & \tilde{u}_4 \leftarrow u_4 \cdot \theta^4, & & \tilde{u}_5 \leftarrow u_5 \cdot \theta^5. \end{array}$$

We note that this algorithm can be merged into the binary ϕ transform typically performed with the NTT_2, in which case its added cost is zero.

Additionally, the μ transform is defined as

$$[u_0', u_1', u_2', u_3', u_4', u_5'] \leftarrow [\tilde{u}_0, \tilde{u}_1, \tilde{u}_2, \tilde{u}_3, \tilde{u}_4, \tilde{u}_5] \cdot \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \theta & \zeta & \zeta\theta & \zeta^2 & \zeta^2\theta \\ 1 & \theta^2 & \zeta^2 & \zeta^2\theta^2 & \zeta & \zeta\theta^2 \\ 1 & \theta^3 & 1 & \theta^3 & 1 & \theta^3 \\ 1 & \theta^4 & \zeta & \zeta\theta^4 & \zeta^2 & \zeta^2\theta^4 \\ 1 & \theta^5 & \zeta^2 & \zeta^2\theta^5 & \zeta & \zeta\theta^5 \end{bmatrix}.$$

The computation of the values $u_0', ..., u_5'$ can be done efficiently (at cost $7\mathbf{M} + 20\mathbf{A}$)[3] by computing intermediate values s_0, s_1, s_2, s_3 as follows:

$$\begin{array}{lll} s_0 \leftarrow \tilde{u}_0 + \tilde{u}_3, & & u_0' \leftarrow s_0 + s_1 + s_2, \\ s_1 \leftarrow \tilde{u}_1 + \tilde{u}_4, & & u_2' \leftarrow s_0 - s_2 + s_3, \\ s_2 \leftarrow \tilde{u}_2 + \tilde{u}_5, & & u_4' \leftarrow s_0 - s_3 - s_1, \\ s_3 \leftarrow (s_1 - s_2) \cdot \zeta, & & \\ & & \\ s_0 \leftarrow \tilde{u}_0 + \tilde{u}_3 \cdot \theta^3, & & u_1' \leftarrow s_0 + s_1 + s_2, \\ s_1 \leftarrow \tilde{u}_1 \cdot \theta + \tilde{u}_4 \cdot \theta^4, & & u_3' \leftarrow s_0 - s_2 + s_3, \\ s_2 \leftarrow \tilde{u}_2 \cdot \theta^2 + \tilde{u}_5 \cdot \theta^5, & & u_5' \leftarrow s_0 - s_3 - s_1, \\ s_3 \leftarrow (s_1 - s_2) \cdot \zeta. & & \end{array}$$

The inverse NTT^{-1} of order 6. To invert the NTT one needs to multiply by the inverse of the Vandermonde matrix, denoted V^{-1}. For convenience, we decompose the inverse as a product of slightly different matrices than the inverses of the ones adopted for the NTT. Thus, we write

$$[u_0, u_1, u_2, u_3, u_4, u_5] \leftarrow \mathsf{NTT}^{-1}\left([u_0', u_1', u_2', u_3', u_4', u_5']\right) = (\psi^\dagger \circ \mu^\dagger)\left([u_0', u_1', u_2', u_3', u_4', u_5']\right),$$

where the $\mu^\dagger$ and $\psi^\dagger$ are related to, but not quite the same as, the inverses of the μ and ψ transforms defined earlier. Next, we define the $\mu^\dagger$ and $\phi^\dagger$ transform. One can verify that

[2] $\mathbf{M}$ stands for Multiplications.

[3] $\mathbf{A}$ stands for Addition.

the decomposition is correct, i.e., indeed $(\psi^\dagger \circ \mu^\dagger) \circ (\mu \circ \psi)$ is the identity transform, even though $\mu^\dagger \neq \mu^{-1}$ and $\psi^\dagger \neq \psi^{-1}$.

We start with the $\mu^\dagger$ that is defined as

$$\left[u_0^\dagger, u_1^\dagger, u_2^\dagger, u_3^\dagger, u_4^\dagger, u_5^\dagger\right] \leftarrow [u_0', u_1', u_2', u_3', u_4', u_5'] \cdot \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ -\zeta^2 & -\theta^5 & -\theta^4 & -1 & -\theta^8 & -\theta^7 \\ 1 & \zeta^2 & \zeta & 1 & \zeta^2 & \zeta \\ -\zeta^2 & -\zeta^2\theta^5 & -\zeta\theta^4 & -1 & -\zeta^2\theta^8 & -\zeta\theta^7 \\ 1 & \zeta & \zeta^2 & 1 & \zeta & \zeta^2 \\ -\zeta^2 & -\zeta\theta^5 & -\zeta^2\theta^4 & -1 & -\zeta\theta^8 & -\zeta^2\theta^7 \end{bmatrix}.$$

The computation of the values $u_0^\dagger, ..., u_5^\dagger$ can be done efficiently (at cost $7\mathbf{M} + 20\mathbf{A}$) by computing intermediate values $t_0, t_1, t_2, t_3, s_0, s_1, s_2, s_3$ as follows:

$$\begin{array}{rcl@{\qquad}rcl} t_0 & \leftarrow & (u_2' - u_4') \cdot \zeta, & u_0^\dagger & \leftarrow & s_0 - s_1 \cdot \zeta^2, \\ t_1 & \leftarrow & (u_3' - u_5') \cdot \zeta, & u_3^\dagger & \leftarrow & s_0 - s_1, \\ s_0 & \leftarrow & u_0' + u_2' + u_4', & & & \\ s_1 & \leftarrow & u_1' + u_3' + u_5', & & & \\ \\ s_0 & \leftarrow & u_0' - u_2' - t_0, & u_1^\dagger & \leftarrow & s_0 - s_1 \cdot \theta^5, \\ s_1 & \leftarrow & u_1' - u_3' - t_1, & u_4^\dagger & \leftarrow & s_0 - s_1 \cdot \theta^8, \\ \\ s_0 & \leftarrow & u_0' - u_4' + t_0, & u_2^\dagger & \leftarrow & s_0 - s_1 \cdot \theta^4, \\ s_1 & \leftarrow & u_1' - u_5' + t_1, & u_5^\dagger & \leftarrow & s_0 - s_1 \cdot \theta^7. \end{array}$$

In addition, we define the $\psi^\dagger$ transform as follows

$$[u_0, u_1, u_2, u_3, u_4, u_5] \leftarrow \left[u_0^\dagger, u_1^\dagger, u_2^\dagger, u_3^\dagger, u_4^\dagger, u_5^\dagger\right] \cdot \frac{1-\zeta}{9} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \theta^{-1} & 0 & 0 & 0 & 0 \\ 0 & 0 & \theta^{-2} & 0 & 0 & 0 \\ 0 & 0 & 0 & -\theta^3 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\theta^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\theta \end{bmatrix}.$$

The computation of the values $u_0, ..., u_5$ can be done efficiently (at cost $6\mathbf{M}$) as follows:

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl} u_0 & \leftarrow & u_0^\dagger \cdot \frac{1-\zeta}{9}, & u_1 & \leftarrow & u_1^\dagger \cdot \frac{1-\zeta}{9} \cdot \theta^{-1}, & u_2 & \leftarrow & u_2^\dagger \cdot \frac{1-\zeta}{9} \cdot \theta^{-2}, \\ u_3 & \leftarrow & u_3^\dagger \cdot \frac{\zeta-1}{9} \cdot \theta^3, & u_4 & \leftarrow & u_4^\dagger \cdot \frac{\zeta-1}{9} \cdot \theta^2, & u_5 & \leftarrow & u_5^\dagger \cdot \frac{\zeta-1}{9} \cdot \theta. \end{array}$$

Note that this algorithm can be merged into the binary ϕ^{-1} transform typically performed with the NTT_2^{-1}, in which case its added cost is zero.

The complete NTT in $\mathbb{Z}_q[z]/\Phi_{2^\ell 9}(z)$ and its estimated cost. Algorithms 16 and 17 summarize the complete NTT transform in the chosen representation of $\mathcal{R}_q$.

Algorithm 16 The NTT on $\mathbb{Z}_q[x,y]/\langle\Phi_{2^\ell}(x),\Phi_9(y)\rangle$

Require: $a \in \mathcal{R}$.
Ensure: $A = \mathsf{NTT}(a) \in \mathcal{R}_q$.

```
1: for $i = 0, ..., 5$ do
2:     $(A[2^{\ell-1}i + j] \mid 0 \leqslant j < 2^{\ell-1}) \leftarrow \mathsf{NTT}_2[(a[2^{\ell-1}i + j] \mid 0 \leqslant j < 2^{\ell-1})]$
3: end for
4: for $j = 0, ..., 2^{\ell-1} - 1$ do
5:     $(A[2^{\ell-1}i + j] \mid 0 \leqslant i < 6) \leftarrow (\mu \circ \psi)[(A[2^{\ell-1}i + j] \mid 0 \leqslant i < 6)]$
6: end for
7: return $A$
```

Algorithm 17 The NTT^{-1} on $\mathbb{Z}_q[x,y]/\langle\Phi_{2^\ell}(x),\Phi_9(y)\rangle$

Require: $A \in \mathcal{R}_q$.
Ensure: $a = \mathsf{NTT}^{-1}(A) \in \mathcal{R}$ in the centered range $(-\lfloor q/2 \rfloor, \lfloor q/2 \rfloor\,]$.

```
1: for $j = 0, ..., < 2^{\ell-1} - 1$ do
2:     $(A[2^{\ell-1}i + j] \mid 0 \leqslant i < 6) \leftarrow (\psi^\dagger \circ \mu^\dagger)[(A[2^{\ell-1}i + j] \mid 0 \leqslant i < 6)]$
3: end for
4: for $i = 0, ..., 5$ do
5:     $(a[2^{\ell-1}i + j] \mid 0 \leqslant j < 2^{\ell-1}) \leftarrow \mathsf{NTT}_2^{-1}[(A[2^{\ell-1}i + j] \mid 0 \leqslant j < 2^{\ell-1})]$
6: end for
7: for $0 \leqslant i < n$ do
8:     $a[i] \leftarrow \mathrm{center}(a[i])$
9: end for
10: return $a$
```

Its total cost, as measured by the number of multiplications in $\mathbb{Z}_q$, is computed as follows:

- The core function NTT_2 of degree $2^{\ell-1}$ (not counting the pre- or post-processing) incurs $2^{\ell-2} \cdot (\ell - 1)$ multiplications in $\mathbb{Z}_q$, and that will be repeated for each of the 6 blocks, totalling $2^{\ell-2} \cdot (\ell - 1) \cdot 6$ multiplications in $\mathbb{Z}_q$.
- The NTT of degree 6 is repeated for each of the $2^{\ell-1}$ sextuples of sub-coefficients, totalling $2^{\ell-1} \cdot 7$ multiplications in $\mathbb{Z}_q$.
- The pre- or post-processing in dimension $n = \varphi(2^\ell)\varphi(9) = 2^{\ell-1}\,6$ incurs $2^{\ell-1}$ multiplications in $\mathbb{Z}_q$ per $2^{\ell-1}$-coefficient block or $2^{\ell-1} \cdot 6$ overall, combining the pre- or post-processing of degree $2^{\ell-1}$ with that of degree 6.

Table 2: NTT cost (in terms of multiplications in $\mathbb{Z}_q$, denoted as $\mathbb{Z}_q\times$) for typical dimensions n.

n	512	768	1024	1536	2048
$\mathbb{Z}_q\times$	2816	4352	6144	9472	13312

Therefore the NTT in $\mathbb{Z}_q[z]/\Phi_{2^\ell 9}(z)$, where the dimension is $n = 2^\ell \cdot 3$, incurs a total cost[4] of $2^{\ell-2} \cdot (\ell - 1) \cdot 6 + 2^{\ell-1} \cdot 7 + 2^{\ell-1} \cdot 6 = 2^{\ell-1} \cdot (3\ell + 10) = (1/2)\, n \lg n + (5/3 - (1/2) \lg 3)\, n$ multiplications in $\mathbb{Z}_q$.

In comparison, the (purely binary) NTT_2 for $n = 2^\ell$ incurs $2^{\ell-1} \cdot (\ell + 2) = (1/2)\, n \lg n + n$ multiplications in $\mathbb{Z}_q$.

The total cost for typical dimensions, as measured by the number of multiplications in $\mathbb{Z}_q$, is listed on Table 2. The settings for `qTESLA-II` and `qTESLA-V-size` are $n = 768$ and $n = 1536$, respectively.

Effects of the ring choice on `qTESLA`. Contrary to the more common ring $\mathbb{Z}_q[x]/\langle\Phi_{2^\ell}(x)\rangle$, which is isometric, the norm of an ideal basis of $\mathbb{Z}_q[z]/\langle\Phi_{2^\ell 9}(z)\rangle$ can vary between basis elements. Specifically, for $u \in \mathbb{Z}_q[z]/\langle\Phi_{2^\ell 9}(z)\rangle$ it holds that $\|z^j \cdot u\|_\infty \leqslant 2\|u\|_\infty$, since individual coefficients of $z^j \cdot u$ can be twice as large in absolute value than the coefficients of u itself.

This requires applying simple modifications to the signing and verification algorithms. Concretely, all conditions based on the values of L_E or L_S must be changed as follows when the underlying ring is $\mathbb{Z}_q[z]/\langle\Phi_{2^\ell 9}(z)\rangle$:

- both the signing and verification algorithms must ensure that $z \in \mathcal{R}_{q,[B-2L_S]}$ (see line 12 in Algorithm 7 and line 6 in Algorithm 8, respectively);
- the signing algorithm must ensure that $\|w_i\|_\infty \leqslant \lfloor q/2 \rfloor - 2L_E$ and $\|[w_i]_L\|_\infty \leqslant 2^{d-1} - 2L_E$ for all $1 \leqslant i \leqslant k$ (see line 18 in Algorithm 7).

These simple changes have already been applied to Algorithms 7 and 8 using a generalization of the bound parameters L_E and L_S via E and S, respectively. These parameters are suitably defined according to the underlying ring, as can be seen in Table 4 .

[4]Excluding the cost of pre- or post-processing, this becomes $2^{\ell-2} \cdot (\ell - 1) \cdot 6 + 2^{\ell-1} \cdot 7 = 2^{\ell-1} \cdot (3\ell + 4)$ multiplications in $\mathbb{Z}_q$.

2.7 qTESLA variant with smaller public key size

Recently, an approach to decrease the size of the public key was proposed by Ducas *et al.* [31]. The basic idea is to push the least significant bits of the public key coefficients to the secret key, and generate a *hint* related to this information. The hint is included in the signature to enable its successful verification. Thus, the public key size decreases at the expense of a larger secret key and a slightly larger signature. In most practical scenarios, the public key size is regarded as more important than the secret key size because the former needs to be transmitted more frequently. Hence, this approach, which we call *public key splitting*, is very attractive. We explain below the details for adapting a very simple version of the technique to qTESLA.

To simplify the description we assume the case with $k = 1$, as in heuristic qTESLA. Given qTESLA's secret polynomials s and e, the corresponding public key, disregarding seed_a, is given by $t = as + e \in \mathcal{R}_q$ (see Algorithm 6). Then, for any t'_0 and t'_1 such that $t = as + e = t'_0 + t'_1$, a qTESLA signature (z, c') can be verified via $[az - tc]_M = [az - t'_0c - t'_1c]_M$ with $c \leftarrow \mathsf{Enc}(c')$ (see Algorithm 7). In [31] the following is suggested: for each polynomial coefficient t_i of t, with $i = 0, \ldots, n-1$, write $t_i = t_{1,i} \cdot 2^b + t_{0,i}$ for some $b \in \mathbb{Z}_{>0}$. Then, we simply set t'_0 as the coefficient vector $(t_{0,0}, \ldots, t_{0,n-1})$, where each coefficient contains the b-least significant bits of the corresponding coefficient from t. Likewise, we set t'_1 as $(t_{1,0}, \ldots, t_{1,n-1})$ where each coefficient contains the remaining top bits of the corresponding coefficient of t. Then, we proceed to include t'_1 in the public key and t'_0 in the secret key. This means that only the most significant bits of the coefficients of t, i.e., t'_1, are directly available during signature verification. Therefore, to be able to verify valid signatures correctly, the verifier needs information about t'_0c. To do this, the value t'_0c is computed during signature generation, and then transmitted as a *hint* through the signature. In order to reduce the potential increase of the signature size, the hint stores the value $[az - t'c]_M - [az - t'_1c]_M$ per coefficient. In heuristic qTESLA this value is bounded by $\pm(q + 2^{d-1})/2^d$; refer to Tables 5 and 6 for specific parameter values. For example, for the proposed heuristic parameters the hint coefficients are in the set $\{-2, -1, 0, 1, 2\}$ which means that each coefficient needs 3 bits of storage.

We note that it is possible to reduce the signature size further by, for instance, avoiding some corner cases in the computation above. This requires blocking those corner cases during the generation of the signature, as done in [31]. Also, Ducas *et al.* [31] limit the hint generation to an evaluation of the equality $[az - t'c]_M = [az - t'_1c]_M$, reducing the hint size to only one bit per coefficient. Furthermore, they include the positions of the 1's instead of the entire n-bit hint in the signature. This, however, requires to bound the number of 1's in the hints and, thus, requires an additional check for rejecting signatures with out-of-bounds hints. qTESLA avoids these optimizations in order to keep a simple design at the expense of a slight increase in the signature size.

In summary, a variant of `qTESLA` using the public key splitting technique has signature sizes increased by $\lceil 3n/8 \rceil$ bytes (in the case of the heuristic parameters). The decrease (resp. increase) of the public (resp. secret) key size is determined by the value of b, i.e., the public key size is decreased by $b \cdot n$ bits while the secret key size is increased by the same amount. In our implementations, we set $b = 16$, which is a convenient value that also helps to minimize the complexity of the software. Our experimental results indicate that the computational overhead of this simplified technique is very small (around 5%); refer to Section 3.4 for more details.

2.8 System parameters and parameter selection

In this section, we describe `qTESLA`'s system parameters and our explicit choice of parameter sets.

Parameter sets. Herein, we propose *twelve* parameter sets which were derived according to *two* different approaches (i) following a "heuristic" parameter generation, and (ii) following a "provably-secure" parameter generation according to a security reduction. The proposed parameter sets are displayed in Table 3 together with their targeted security category, as defined by NIST in [54].

Table 3: Parameter sets and their targeted security.

Heuristic	Provably-secure	Security category
`qTESLA-I`, `qTESLA-I-s`	`qTESLA-p-I`	NIST's category 1
`qTESLA-II`, `qTESLA-II-s`	-	NIST's category 2
`qTESLA-III`, `qTESLA-III-s`	`qTESLA-p-III`	NIST's category 3
`qTESLA-V`, `qTESLA-V-s`	-	NIST's category 5
`qTESLA-V-size`, `qTESLA-V-size-s`	-	NIST's category 5

The proposed provably-secure parameter sets, namely `qTESLA-p-I` and `qTESLA-p-III`, were chosen according to the security reduction provided in Theorem 5, Section 5.1. This implies the following: by virtue of our security reduction, these parameters strictly correspond to an instance of the R-LWE problem. That is, the reduction provably guarantees that our scheme has the selected security level as long as the corresponding R-LWE instance is intractable. In other words, hardness statements for R-LWE instances have a provable consequence for the security levels of our scheme. Moreover, since the presented reduction is tight, the tightness gap of our reduction is equal to 1 for our choice of parameters and, hence, the concrete bit security of our signature scheme is essentially the same as the bit hardness of the underlying R-LWE instance.

Choosing parameters following the security statements, as described above, implies to follow specific security requirements and to take a reduction loss into account. This affects the performance and signature/key sizes of the scheme. In order to offer a more efficient approach, we also propose ten parameter sets, namely `qTESLA-I`, `qTESLA-II`, `qTESLA-III`, `qTESLA-V` and `qTESLA-V-size` (and their corresponding variants with public key reduction), which are chosen *heuristically.* In this case, we assume that the security level of a certain instantiation of the scheme directly corresponds to the hardness level of the corresponding R-LWE instance, without taking into account the security reduction. The assumption is that Theorem 5 still holds for these concrete parameter sets.

The sage script that was used to generate the various parameters is included in the submission package (see the file `parameterchoice.sage` found in the submission folder `\Supporting_Documentation\Script_to_choose_parameters`).

System parameters. `qTESLA`'s system parameters and their corresponding bounds are summarized in Table 4. Concrete parameter values for each of the proposed parameter sets are compiled in Tables 5 and 6.

Let λ be 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 $\ell \in \mathbb{N}$. To extend the flexibility of `qTESLA`'s parameter generation, we also consider the case of $\mathcal{R}_q = \mathbb{Z}_q[z]/\langle \Phi_{2^\ell 9}(z)\rangle$ with non-power-of-two dimension $n = 2^{\ell-1} \cdot 6$; see Section 2.6. Let σ be the standard deviation of the centered discrete Gaussian distribution that is used to sample the coefficients of the secret and error polynomials. Let $k \in \mathbb{Z}_{>0}$ be the number of ring learning with errors samples. For our *heuristic* parameter sets we fix $k = 1$, whereas for our *provably-secure* parameter sets we choose $k \geq 1$. The latter choice allows us to reduce the size of the modulus q, as explained later. 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 h defines the number of nonzero elements in the output of the encoding function described in Section 2.5.7.
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 $a_1, \dots, a_k$ (see Algorithm 10). The values of b_{GenA} were chosen as to allow the generation of (slightly) more bytes than are necessary to fill out all the coefficients of the polynomials $a_1, \dots, a_k$.

Table 4: 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	$= \begin{cases} 2^\ell, & \text{power-of-two case,} \\ 2^{\ell-1} \cdot 6, & \text{non-power-of-two case.} \end{cases}$
σ	standard deviation of centered discrete Gaussian distribution	-
k	#R-LWE samples	-
q	modulus	$q \equiv 1 \bmod 2n$, $q > 4B$ For provably secure parameters: $q^{nk} \geq \lvert\Delta\mathbb{S}\rvert \cdot \lvert\Delta\mathbb{L}\rvert \cdot \lvert\Delta\mathbb{H}\rvert$, $q^{nk} \geq 2^{4\lambda+nkd} 4q_s^3(q_s+q_h)^2$
h	# of nonzero entries of output elements of Enc	$2^h \cdot \binom{n}{h} \geq 2^{2\lambda}$
κ	output length of hash-based function H and input length of GenA, PRF_1, PRF_2, Enc and $\mathsf{ySampler}$	$\kappa \geq \lambda$
L_E, η_E	bound in checkE	$\eta_E \cdot h \cdot \sigma$
L_S, η_S	bound in checkS	$\eta_S \cdot h \cdot \sigma$
S, E	rejection parameters	$= \begin{cases} L_S, L_E & \text{if } n \text{ is power-of-two,} \\ 2L_S, 2L_E & \text{if } n \text{ is non-power-of-two.} \end{cases}$
B	determines interval the randomness is chosen from during signing	near a power-of-two, $B \geq \frac{\sqrt[n]{M}+2S-1}{2(1-\sqrt[n]{M})}$
d	number of rounded bits	$d > \log_2(B)$, $\left(1 - \frac{2\cdot E+1}{2^d}\right)^{k\cdot n} \geq 0.3$
b_{GenA}	number of blocks requested to SHAKE128 for GenA	$b_{\mathsf{GenA}} \in \mathbb{Z}_{>0}$
$\lvert\Delta\mathbb{H}\rvert$	see definition in the text	$\sum_{j=0}^{h} \sum_{i=0}^{h-j} \binom{kn}{2i} 2^{2i} \binom{kn-2i}{j} 2^j$
$\lvert\Delta\mathbb{S}\rvert$		$(4(B-S)+1)^n$
$\lvert\Delta\mathbb{L}\rvert$		$(2^d+1)^{nk}$
δ_z	acceptance probability of z in line 12 during signing	determined experimentally
δ_w	acceptance probability of w in line 18 during signing	determined experimentally
δ_{keygen}	acceptance probability of key pairs during key generation	experimentally
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]	$n(k+1)(\lceil \log_2(t-1) \rceil + 1) + 2\kappa$ with $t = 209, 128, 135, 191, 79$ or 112
sig-split size	theoretical size of signature using splitting [bits]	sig size $+3n$
pk-split size	theoretical size of public key using splitting [bits]	pk size $-16n$
sk-aplit size	theoretical size of secret key using splitting [bits]	sk size $+16n$

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 Section 5.2.1. Both bounds, L_S and L_E (and consequently S and E), impact the rejection probability during the signature generation as follows. If one increases the values of L_S and L_E, the acceptance probability during key generation, referred to as

Table 5: Parameters for each of the proposed *heuristic* parameter sets with $q_h = 2^{128}$ and $q_s = 2^{64}$; we choose $M = 0.3$.

Param.	`qTESLA-I`	`qTESLA-II`	`qTESLA-III`	`qTESLA-V`	`qTESLA-V-size`
λ	95	128	160	225	256
κ	256	256	256	256	256
n	512	768	1 024	2 048	1 536
σ	22.93	9.73	10.2	10.2	10.2
k	1	1	1	1	1
q	4 205 569 $\approx 2^{22}$	8 404 993 $\approx 2^{23}$	8 404 993 $\approx 2^{23}$	16 801 793 $\approx 2^{24}$	33 564 673 $\approx 2^{25}$
h	30	39	48	61	77
L_E, η_E	1 586, 2.306	859, 2.264	1 147, 2.344	1 554, 2.489	1 792, 2.282
L_S, η_S	1 586, 2.306	859, 2.264	1 233, 2.519	1 554, 2.489	1 792, 2.282
E	1 586	1 718	1 147	1 554	3 584
S	1 586	1 718	1 233	1 554	3 584
B	$2^{20}-1$	$2^{21}-1$	$2^{21}-1$	$2^{22}-1$	$2^{23}-1$
d	21	22	22	23	24
b_{GenA}	19	28	38	98	73
δ_w	0.34	0.36	0.43	0.31	0.33
δ_z	0.45	0.55	0.55	0.46	0.55
δ_{sign}	0.16	0.20	0.24	0.14	0.18
δ_{keygen}	0.63	0.23	0.58	0.35	0.19
sig size [bytes]	$1,376$	$2,144$	$2,848$	$5,920$	$4,640$
pk size [bytes]	$1,504$	$2,336$	$3,104$	$6,432$	$5,024$
sk size [bytes]	$1,216$	$1,600$	$2,368$	$4,672$	$3,520$
sig-split size [bytes]	$1,568$	$2,432$	$3,232$	$6,688$	$4,640$
pk-split size [bytes]	480	800	$1,056$	$2,336$	$5,024$
sk-split size [bytes]	$2,240$	$3,136$	$4,416$	$8,768$	$3,520$
classical bit hardness	119	149	204	412	284
quantum bit hardness	111	138	188	377	261

δ_{keygen}, increases (see lines 8 and 13 in Alg. 6), 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 Alg. 7). We determine a good trade-off between the acceptance probabilities during key generation and signing experimentally. To this end, we start by choosing $L_S = \eta_S \cdot h \cdot \sigma$ (resp., $L_E = \eta_E \cdot h \cdot \sigma$) with $\eta_S = \eta_E = 2.8$ and compute the corresponding values for the parameters B, d and q (which are chosen as explained later). We then carefully tune these parameters by trying different values for η_S and η_E in the range $[2.0, \ldots, 3.0]$ until we find a good trade-off between the different probabilities and, hence, runtimes.
The parameter B defines the interval of the random polynomial y (see line 4 of Alg. 7),

Table 6: Parameters for each of the proposed *provably-secure* parameter sets with $q_h = 2^{128}$ and $q_s = 2^{64}$; we choose $M = 0.3$.

Param.	qTESLA-p-I	qTESLA-p-III
λ	95	160
κ	256	256
n	1 024	2 048
σ	8.5	8.5
k	4	5
q	343 576 577 $\approx 2^{28}$	856 145 921 $\approx 2^{30}$
h	25	40
$L_E(=E)$, η_E	554, 2.61	901, 2.65
$L_S(=S)$, η_S	554, 2.61	901, 2.65
B	$2^{19}-1$	$2^{21}-1$
d	22	24
b_{GenA}	108	180
$\|\Delta\mathbb{H}\|$	$\approx 2^{435.8}$	$\approx 2^{750.9}$
$\|\Delta\mathbb{S}\|$	$\approx 2^{21502.4}$	$\approx 2^{47102.7}$
$\|\Delta\mathbb{L}\|$	$\approx 2^{94208.0}$	$\approx 2^{256000.0}$
δ_w	0.37	0.33
δ_z	0.34	0.42
δ_{sign}	0.13	0.14
δ_{keygen}	0.59	0.43
sig size [bytes]	2, 592	5 664
pk size [bytes]	14, 880	38 432
sk size [bytes]	5, 184	12 352
classical bit hardness	151	305
quantum bit hardness	140	279

and it is determined by the parameters M and S as follows:

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

where $M = 0.3$ is a value of our choosing. Once B is chosen, we select the value d that determines the rounding functions $[\cdot]_M$ and $[\cdot]_L$ to be larger than $\log_2(B)$ and such that the acceptance probability of the check $\| [w]_L \|_\infty \geq 2^{d-1} - E$ in line 18 of Algorithm 7 is lower bounded by 0.3. 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} are obtained experimentally, following the procedure above, are summarized in Tables 5 and 6.

The modulus q. This parameter is chosen to fulfill several bounds and assumptions that are motivated by efficiency requirements and `qTESLA`'s security reduction. To enable the use of fast polynomial multiplication using the NTT, q must be a prime integer such that $q \bmod 2n = 1$. Moreover, we choose $q > 4B$. To choose parameters according to the security reduction, i.e., for the case of `provably-secure qTESLA`, it is first convenient to simplify our security statement. To this end we ensure that $q^{nk} \geq |\Delta\mathbb{S}| \cdot |\Delta\mathbb{L}| \cdot |\Delta\mathbb{H}|$ with the following definition of sets: $\mathbb{S}$ is the set of polynomials $z \in \mathcal{R}_{q,[B-S]}$ and $\Delta\mathbb{S} = \{z - z' \ : \ z, z' \in \mathbb{S}\}$, $\mathbb{H}$ is the set of polynomials $c \in \mathcal{R}_{q,[1]}$ with exactly h nonzero coefficients and $\Delta\mathbb{H} = \{c - c' \ : \ c, c' \in \mathbb{H}\}$, and $\Delta\mathbb{L} = \{x - x' : x, x' \in \mathcal{R}$ and $[x]_M = [x']_M\}$. Then, the following equation (see Theorem 5 in Section 5.1) 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 [54, Section 4.A.4], we choose the number of classical queries to the sign oracle to be $q_s = 2^{64}$ for all our parameter sets. Moreover, we choose the number of queries of a hash function to be $q_h = 2^{128}$.

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 CDT tables 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 $n(k+1)(\lceil \log_2(t-1) \rceil + 1) + 2\kappa$ bits. For parameter sets `qTESLA-I`, `qTESLA-II`, `qTESLA-III`, `qTESLA-V`, `qTESLA-V-size`, `qTESLA-p-I`, and `qTESLA-p-III`, t is equal to 209, 128, 135, 191, 191, 79, and 112, respectively [5]. These values of t are then plugged into the equation $n(k+1)(\lceil \log_2(t-1) \rceil + 1) + 2\kappa$ to obtain the values displayed in Tables 5 and 6.

Table 5 also states the sizes of `qTESLA`'s variants with smaller public key sizes which use the public key splitting technique explained in Section 2.7.

3 Performance analysis

3.1 Reference implementations

This document comes accompanied by simple yet efficient reference implementations written exclusively in portable C.

[5]The CDT tables are generated based on the chosen precision β and a given σ; see the script provided in the folder `\Supporting_Documentation\Script_for_Gaussian_sampler`.

An important feature of qTESLA is that it enables very efficient implementations that can work for different security levels with minor changes. For example, our implementations of the heuristic qTESLA parameter sets qTESLA-I, qTESLA-III and qTESLA-V, which use a standard power-of-two NTT, share most of their codebase, and only differ in some packing functions and system constants that can be instantiated at compilation time. This feature is also shared between our implementations for the provably-secure qTESLA parameter sets qTESLA-p-I and qTESLA-p-III. This highlights the simplicity and scalability of software based on qTESLA.

Furthermore, since provably-secure qTESLA uses a generalization of the scheme with $k > 1$, it is possible to merge all the implementations into one. For our current implementations, we separate both approaches, heuristic and provably-secure, in order to maximize the efficiency of heuristic qTESLA. However, we envision applications in which this feature could be exploited with a relatively small performance overhead.

All our implementations avoid the use of secret address accesses and secret branches and, hence, are protected against timing and cache side-channel attacks. Whenever appropriate we write "constant-time" code that is branch-free using masking and logical operations. This is the case of the function H, checkE, checkS, the correctness test for rejection sampling, polynomial multiplication using the NTT, sparse multiplication and all the polynomial operations requiring modular reductions or corrections. Some of the functions that perform some form of rejection sampling, such as the security test at signing, GenA, $\mathsf{ySampler}$ and Enc, potentially leak the timing of the failure to some internal test, but this information is independent of the secret data. Table lookups performed in our implementation of the Gaussian sampler are done with linear passes over the full table and extracting entries via masking with logical operations.

For the polynomial multiplication we use iterative algorithms for the forward and inverse NTTs, as described in Section 2.5.8, using a signed 32-bit datatype for the inputs and outputs. Intermediate results after additions and subtractions are let to grow throughout the execution, and are only reduced or corrected when there is a chance of exceeding 32 bits of length, after a multiplication, or when a result needs to be prepared for final packing (e.g., when outputting secret and public keys). In the NTT and pointwise multiplication the results of multiplications are reduced via Montgomery reductions. To minimize the cost of converting to/from Montgomery representation we use the following approach. First, twiddle factors are scaled *offline* by multiplying with R, where R is the Montgomery constant $2^{32} \bmod q$. Similarly, the coefficients of the outputs a_i from GenA are scaled to remainders $r' = rn^{-1}R \pmod q$ by multiplying with the constant $R^2 \cdot n^{-1}$. This enables an efficient use of Montgomery reductions during the NTT-based polynomial multiplication $\mathsf{NTT}^{-1}(\tilde{a} \circ \mathsf{NTT}(b))$, where $\tilde{a} = \mathsf{NTT}(a)$ is the output in NTT domain of GenA. Multiplications with the twiddle factors during the computation of $\mathsf{NTT}(b)$ naturally cancel out the Montgomery constant. The same happens during the pointwise multiplication with $\tilde{a}$, and finally during the inverse NTT, which naturally outputs values in standard representation without the need for explicit conversions.

In the non-power-of-two case, corresponding to `qTESLA-II`, `qTESLA-II-s`, `qTESLA-V-size` and `qTESLA-V-size-s`, we present reference implementations using a basic algorithm to implement the NTT, and using simple Barrett reductions instead of the more efficient Montgomery reduction. Therefore, there is room for further optimization in these implementations, which is left as future work.

3.2 AVX2-optimized implementations for heuristic, power-of-two parameter sets

This section currently applies to `qTESLA-I`, `qTESLA-III` and `qTESLA-V`, and their variants with smaller public keys. We leave as future work the AVX2-optimized implementation of `qTESLA-II` and `qTESLA-V-size`.

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

Our polynomial multiplication follows the recent approach by Seiler [62], and the realization of the method has some similarities with the implementation from [31]. 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. Let us illustrate the procedure we apply for a polynomial a of dimension $n = 512$ written as the vector of coefficients $(a_0, a_1, \ldots, a_{511})$. We split the coefficients in 8 subsets a'_i equally distributed, namely, $a'_0 = (a_0, \ldots, a_{63}), a'_1 = (a_{64}, \ldots, a_{127})$, and so on. We start by loading the first 4 coefficients of each subset a'_i, filling out 8 AVX2 registers in total, and then performing 3 levels of butterfly computations between the corresponding pairs of subsets according to the Cooley-Tukey algorithm. We repeat this procedure 16 times using the subsequent 4 coefficients from each subset a'_i each time. Note that the 3 levels can be completed at once without the need 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 [31, 62] is that our NTT coefficients are represented as 32-bit *signed* integers, which motivates a speedup in the butterfly computation by avoiding the extra additions that are required to make the result of subtractions positive when using an unsigned representation. Moreover, we implement a *full* polynomial multiplication $\mathsf{NTT}^{-1}(\tilde{a} \circ \mathsf{NTT}(b))$ that integrates the pointwise multiplication and the forward and inverse NTTs. This allows us to further optimize the implementation by eliminating multiple load/store operations and some data processing to pack coefficients in the AVX2 registers.

With our approach we reduce the cost of the reference polynomial multiplication from $25,300$ to only $5,800$ cycles for dimension $n = 512$ on an Intel Skylake processor using gcc for compilation. For $n = 1024$, we reduce the cost from $58,200$ to $12,700$ cycles.

Sampling of y is sped up by using the AVX2 implementation of SHAKE by Bertoni *et al.* [19], which allows us to sample up to 4 coefficients in parallel.

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

3.3 Performance of `qTESLA` on x64 Intel

We evaluated the performance of our implementations on two machines powered by: (i) a 3.4GHz Intel Core i7-6700 (Skylake) processor and (ii) a 3.4GHz Intel Core i7-4770 (Haswell) processor, both running Ubuntu 16.04.3 LTS. As is standard practice, TurboBoost was disabled during the tests. For compilation we used gcc version 7.2.0 with the command `gcc -O3 -march=native -fomit-frame-pointer`.

The results for the reference and AVX2-optimized implementations are summarized in Tables 7 and 8, respectively.

Our results showcase the high performance of `heuristic qTESLA` with a simple and compact implementation written entirely in portable C: the combined (median) time of signing and verification on the Skylake platform is of approximately 133.8, 205.0 and 600.3 microseconds for `qTESLA-I`, `qTESLA-III` and `qTESLA-V`, respectively. Likewise, `provably-secure qTESLA` computes the same operations in approximately 0.87 and 2.43 milliseconds with `qTESLA-p-I` and `qTESLA-p-III`, respectively. This demonstrates that the speed of `provably-secure qTESLA`, although slower, can still be considered practical for most applications.

The AVX2 optimizations improve the performance by a factor between 1.5–1.7x, approximately. The speedup is mainly due to the AVX2 implementation of the polynomial mul-

Scheme	`keygen`	`sign`	`verify`	**total** (`sign + verify`)
`qTESLA-I`	1,123.9 (1,146.2)	372.8 (492.2)	82.2 (82.8)	455.0 (575.0)
`qTESLA-II`	5,415.2 (5,984.3)	1,769.9 (2,427.3)	448.2 (448.5)	2,218.1 (2,875.8)
`qTESLA-III`	2,919.7 (3,255.3)	527.0 (695.0)	170.1 (170.4)	697.1 (865.4)
`qTESLA-V`	14,836.4 (16,926.7)	1,644.1 (2,151.8)	396.9 (397.6)	2,041.0 (2,549.4)
`qTESLA-V-size`	20,872.9 (24,197.0)	4,088.8 (5,688.3)	990.3 (990.6)	5,079.1 (6,678.9)
`qTESLA-p-I`	5,106.3 (5,317.5)	2,294.9 (3,132.3)	669.1 (669.3)	2,964.0 (3,801.6)
`qTESLA-p-III`	25,452.7 (25,879.7)	6,421.5 (8,561.2)	1,842.4 (1,842.4)	8,263.9 (10,403.6)

Table 7: Performance (in thousands of cycles) of the reference implementations of `qTESLA` on a 3.4GHz Intel Core i7-6700 (Skylake) processor. Results for the median and average (in parenthesis) are rounded to the nearest 10^2 cycles. Signing is performed on a message of 59 bytes.

tiplication, which is responsible for $\sim 70\%$ of the total speedup. The combined (median) time of signing and verification on the Skylake platform is of about 90.9, 132.9 and 348.9 microseconds for `qTESLA-I`, `qTESLA-III` and `qTESLA-V`, respectively.

Similar results were observed on an Intel Haswell processor. These results are summarized in Tables 9 and 10 for the the reference and AVX2-optimized implementations, respectively.

As we remarked in previous sections, the current reference implementations of the non-power-of-two options, namely `qTESLA-II` and `qTESLA-V-size`, have not been fully optimized. This, as well as the development of their AVX2-optimized implementations, is left as future work. As reference, we mention that the "optimized" implementations of `qTESLA-II` and `qTESLA-V-size` provided in this submission include a Barrett reduction written in assembly. With this simple optimization, the combined (median) time of signing and verification of `qTESLA-II` on the Skylake platform is of 1,331.7 thousand cycles (compare to 2,218.1 thousand cycles from Table 7 corresponding to the reference implementation).

Scheme	`keygen`	`sign`	`verify`	**total** (`sign + verify`)
`qTESLA-I`	$1,105.3$ $(1,123.6)$	245.5 (317.8)	63.4 (64.0)	308.9 (381.8)
`qTESLA-III`	$2,869.3$ $(3,153.5)$	324.4 (407.5)	127.3 (127.9)	451.7 (535.4)
`qTESLA-V`	$14,765.2$ $(17,041.4)$	887.9 $(1,137.1)$	298.5 (299.2)	$1,186.4$ $(1,436.3)$

Table 8: Performance (in thousands of cycles) of the AVX2 implementations of `qTESLA` on a 3.4GHz Intel Core i7-6700 (Skylake) processor. Results for the median and average (in parenthesis) are rounded to the nearest 10^2 cycles. Signing is performed on a message of 59 bytes.

3.4 Performance of `qTESLA` with smaller public keys on x64 Intel

We also evaluated the performance of our implementations of the `qTESLA` variant with smaller public keys (as described in Section 2.7) on the same machine powered by a 3.4GHz Intel Core i7-6700 (Skylake) processor running Ubuntu 16.04.3 LTS. Again, TurboBoost was disabled during the tests. For compilation we used gcc version 7.2.0 with the command `gcc -O3 -march=native -fomit-frame-pointer`.

The results for the reference and AVX2-optimized implementations are summarized in Tables 11 and 12, respectively. Our results showcase the very small overhead of the `qTESLA` variant with smaller public key compared with the original `qTESLA`. In most cases, this overhead is below 5%. This high performance, on top of the significant reduction in public key size, makes these variants very attractive for many applications.

Scheme	`keygen`	`sign`	`verify`	**total** (`sign + verify`)
`qTESLA-I`	1,139.9 (1,162.7)	392.3 (529.6)	85.8 (86.2)	478.1 (615.8)
`qTESLA-II`	5,393.1 (5,894.3)	1,880.0 (2,583.1)	478.2 (478.9)	2,358.2 (3,062.0)
`qTESLA-III`	2,910.3 (3,190.2)	561.5 (725.1)	177.7 (178.1)	739.2 (903.2)
`qTESLA-V`	14,952.1 (17,367.3)	1,707.1 (2,319.6)	409.3 (411.5)	2,116.4 (2,731.1)
`qTESLA-V-size`	21,089.6 (25,008.5)	4,417.1 (6,029.4)	1,078.5 (1,079.0)	5,495.6 (7,108.4)
`qTESLA-p-I`	5,125.0 (5,362.0)	2,391.5 (3,194.3)	686.0 (690.3)	3,077.5 (3,884.6)
`qTESLA-p-III`	25,553.0 (26,029.8)	6,818.6 (9,194.4)	1,939.3 (1,940.8)	8,757.9 (11,135.2)

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

Scheme	`keygen`	`sign`	`verify`	**total** (`sign + verify`)
`qTESLA-I`	1,120.1 (1,146.5)	257.3 (337.9)	66.0 (66.5)	323.3 (404.4)
`qTESLA-III`	2,860.1 (3,158.6)	338.2 (424.5)	133.6 (134.3)	471.8 (558.8)
`qTESLA-V`	14,888.7 (17,474.9)	955.2 (1,236.8)	317.2 (318.4)	1,274.4 (1,555.2)

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

Scheme	`keygen`	`sign`	`verify`	**total** (`sign + verify`)
`qTESLA-I-s`	1,124.5 (1,138.0)	385.1 (502.5)	83.3 (83.7)	468.4 (586.2)
`qTESLA-II-s`	5,396.8 (5,886.2)	1,912.4 (2,539.2)	446.9 (447.3)	2,359.3 (2,986.4)
`qTESLA-III-s`	2,921.7 (3,222.5)	551.9 (719.9)	170.7 (171.1)	722.6 (891.0)
`qTESLA-V-s`	14,940.3 (17,244.1)	1,684.0 (2,189.8)	394.2 (395.1)	2,078.2 (2,584.9)
`qTESLA-V-size-s`	20,931.9 (24,662.0)	4,409.7 (6,007.4)	991.6 (992.0)	5,401.3 (6,999.4)

Table 11: Performance (in thousands of cycles) of the reference implementations of `qTESLA` variant with smaller public key size on a 3.4GHz Intel Core i7-6700 (Skylake) processor. Results for the median and average (in parenthesis) are rounded to the nearest 10^2 cycles. Signing is performed on a message of 59 bytes.

Scheme	`keygen`	`sign`	`verify`	**total** (`sign + verify`)
`qTESLA-I-s`	1,109.3 (1,133.2)	257.8 (334.2)	64.1 (64.6)	321.9 (398.8)
`qTESLA-III-s`	2,901.7 (3,208.5)	348.5 (420.6)	127.9 (128.4)	476.4 (549.0)
`qTESLA-V-s`	14,717.8 (16,937.7)	943.8 (1,182.2)	298.9 (299.6)	1,242.6 (1,481.8)

Table 12: Performance (in thousands of cycles) of the AVX2 implementations of `qTESLA` variant with smaller public key size 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.

4 Known answer values

The submission includes KAT values with tuples that contain message size (`mlen`), message (`msg`), public key (`pk`), secret key (`sk`), signature size (`smlen`) and signature (`sm`) values for all the proposed parameter sets.

The KAT files for the reference implementations can be found in the media folder:

- `qTESLA-I`: `\KAT\ref\<KATxx>\PQCsignKAT_qTesla-I.rsp`,
- `qTESLA-I-s`: `\KAT\ref\<KATxx>\PQCsignKAT_qTesla-I-s.rsp`,
- `qTESLA-II`: `\KAT\ref\<KATxx>\PQCsignKAT_qTesla-II.rsp`,
- `qTESLA-II-s`: `\KAT\ref\<KATxx>\PQCsignKAT_qTesla-II-s.rsp`,
- `qTESLA-III`: `\KAT\ref\<KATxx>\PQCsignKAT_qTesla-III.rsp`,
- `qTESLA-III-s`: `\KAT\ref\<KATxx>\PQCsignKAT_qTesla-III-s.rsp`,
- `qTESLA-V`: `\KAT\ref\<KATxx>\PQCsignKAT_qTesla-V.rsp`,
- `qTESLA-V-s`: `\KAT\ref\<KATxx>\PQCsignKAT_qTesla-V-s.rsp`,
- `qTESLA-V-size`: `\KAT\ref\<KATxx>\PQCsignKAT_qTesla-V-size.rsp`,
- `qTESLA-V-size-s`: `\KAT\ref\<KATxx>\PQCsignKAT_qTesla-V-size-s.rsp`,
- `qTESLA-p-I`: `\KAT\ref\<KATxx>\PQCsignKAT_qTesla-p-I.rsp`, and
- `qTESLA-p-III`: `\KAT\ref\<KATxx>\PQCsignKAT_qTesla-p-III.rsp`.

The KAT files for the AVX2-optimized implementations can be found in the media folder:

- `qTESLA-I`: `\KAT\avx2\KAT64\PQCsignKAT_qTesla-I.rsp`,
- `qTESLA-I-s`: `\KAT\avx2\KAT64\PQCsignKAT_qTesla-I-s.rsp`,
- `qTESLA-III`: `\KAT\avx2\KAT64\PQCsignKAT_qTesla-III.rsp`,
- `qTESLA-III-s`: `\KAT\avx2\KAT64\PQCsignKAT_qTesla-III-s.rsp`,
- `qTESLA-V`: `\KAT\avx2\KAT64\PQCsignKAT_qTesla-V.rsp`,
- `qTESLA-V-s`: `\KAT\avx2\KAT64\PQCsignKAT_qTesla-V-s.rsp`.

KATxx is either KAT32 (KATs for 32-bit platforms) or KAT64 (KATs for 64-bit platforms).

Our different implementations produce different KATs because the CDT tables in the Gaussian sampler are optimized for a given computer wordsize ($w = 32$ or $w = 64$ in our implementations). Moreover, our AVX2 implementation differ from the reference implementation in that it employs a vector-friendly implementation of the function that samples y. Since qTESLA signatures are probabilistic, this difference in the implementation produces different signatures and, hence, different KATs.

5 Expected security strength

It this section we discuss the expected security strength of and possible attacks against `qTESLA`. This includes two statements about the theoretical security and the parameter choices depending on them. To this end we first define the hardness assumptions `qTESLA` is based on. This includes the ring short integer solution (R-SIS) problem and the *decisional* ring learning with errors (decisional R-LWE) problem.

Definition 1 (Ring short integer solution problem R-SIS$_{n,k,q,\beta}$). *Let $a_1, ..., a_k \leftarrow_\$ \mathcal{R}_q$. The ring short integer solution problem $R{-}SIS_{n,k,q,\beta}$ is to find solutions $u_1, ..., u_{k+1} \in \mathcal{R}_q$, where $u_i \neq 0$ for at least one i, such that $(a_1, ..., a_k, 1) \cdot (u_1, ..., u_{k+1})^T = a_1u_1 + ... + a_ku_k + u_{k+1} = 0 \mod q$ and $\|u_1\|, ..., \|u_{k+1}\| \leq \beta$.*

Definition 2 (Learning with Errors Distribution). *Let $n, q > 0$ be integers, $s \in \mathcal{R}$, and χ be a distribution over $\mathcal{R}$. We define by $\mathcal{D}_{s,\chi}$ the LWE distribution which outputs $(a, \langle a, s\rangle + e) \in \mathcal{R}_q \times \mathcal{R}_q$, where $a \leftarrow_\$ \mathcal{R}_q$ and $e \leftarrow \chi$.*

Definition 3 (Decisional Ring Learning with Errors Problem R-LWE$_{n,k,q,\chi}$). *Let $n, q > 0$ be integers and χ be a distribution over $\mathcal{R}$. Moreover, let $s \leftarrow \chi$ and $\mathcal{D}_{s,\chi}$ be the learning with errors distribution. Given k tuples $(a_1, t_1), ..., (a_k, t_k)$, the decisional ring learning with errors problem 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.*

5.1 Provable security in the (quantum) random oracle model

The asymptotic security for `qTESLA` is expected to follow from [29,49], where it is shown how the security of the Fiat-Shamir transformation transfers over to the quantum setting.

In addition, the concrete security of `qTESLA` is supported by *two* statements reducing the hardness of lattice-based assumptions to the security of our proposed signature scheme in the (quantum) random oracle model. In this subsection, we describe these two statements. Formal security proofs are not included in this document because these are very close to the original results. The interested reader is referred to [9, 14] for more details.

The first reduction (see Theorem 4), which follows closely the approach proposed by Bai and Galbraith [14], gives a non-tight reduction from R-LWE and R-SIS to the existential unforgeability under chosen-message attack (EUF-CMA) of `qTESLA` in the random oracle model.

Theorem 4. *Let $2^n \cdot \binom{n}{h} \geq 2^\lambda$, $(2R+1)^{k+1} \geq kq^n 2^\kappa$, and $q > 4B$. If there exists an adversary A that forges a signature of the signature scheme* `qTESLA` *described in Section 2.3 in time t_Σ and with success probability ϵ_Σ, then there exists a reduction R that solves either*

- *the $R{-}LWE_{n,k,q,\sigma}$ problem in time $t_{LWE} \approx t_\Sigma$ with $\epsilon_{LWE} \geq \epsilon_\Sigma/2$, or*
- *the $R{-}SIS_{n,k,q,\beta}$ problem with $\beta = \max\{k2^{d-1}, 2(B-E)\} + 2hR$ in time $t_{SIS} \approx 2t_\Sigma$ with $\epsilon_{SIS} \geq \frac{1}{2}(\epsilon_\Sigma - \frac{1}{2^\kappa})\left(\frac{(\epsilon_\Sigma - \frac{1}{2^\kappa})}{q_h} - \frac{1}{2^\kappa}\right) + \epsilon_\Sigma/2$ with our choice of parameters.*

The second security reduction (see Theorem 5) gives a tight reduction from the R-LWE problem to the EUF-CMA security of `qTESLA` in the *quantum* random oracle model. In our opinion, this second theorem is much stronger since it guarantees security against adversaries that have quantum access to a quantum random oracle. Accordingly, we always refer to Theorem 5 when discussing the security of the scheme. We emphasize that Theorem 5 gives a reduction from the decisional R-LWE problem only, while in Theorem 4 the decisional R-SIS problem is additionally used. Currently, Theorem 5 holds assuming a conjecture, as explained below.

Theorem 5. *Let the parameters be as in Table 4. Furthermore, assume that Conjecture 6 holds. If there exists an adversary A that forges a signature of the signature scheme* `qTESLA` *described in Section 2.3 in time t_Σ and with success probability ϵ_Σ, then there exists a reduction R that solves the $R{-}LWE_{n,k,q,\sigma}$ problem in time $t_{LWE} \approx t_\Sigma$ with $\epsilon_\Sigma \leq \frac{2^{3\lambda+nkd} \cdot 4 \cdot q_s^3(q_s+q_h)^2}{q^{nk}} + \frac{2q_h+5}{2^\lambda} + \epsilon_{LWE}$.*

The proof follows the approach proposed in [9], 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 $\mathcal{R}_{q,[B]}$ as in Algorithm 2. Recall that we define $\Delta\mathbb{L}$ to be the set $\{x - x' : x, x' \in \mathcal{R} \text{ and } [x]_M = [x']_M\}$, and that we call a polynomial f *well-rounded* if f is in $\mathcal{R}_{q,[\lfloor q/2 \rfloor - E]}$ and $[f]_L \in \mathcal{R}_{q,[(2^{d-1}-E)]}$. For our discussion we also define the following sets of polynomials:

$\mathbb{Y}$:	The set of polynomials $y \in \mathcal{R}_{q,[B]}$.
$\Delta\mathbb{Y}$:	The set $\left\{y - y' \ : \ y, y' \in \mathcal{R}_{q,[B]}\right\} = \mathcal{R}_{q,[2B]}$.
$\mathbb{H}$:	The set of polynomials $c \in \mathcal{R}_{q,[1]}$ with exactly h nonzero coefficients.
$\Delta\mathbb{H}$:	The set $\{c - c' \ : \ c, c' \in \mathbb{H}\}$.
$\mathbb{W}$:	The set $\{[w]_M : w \in \mathcal{R}_q\}$ of integer polynomials obtained from the high bits of a polynomial in $\mathcal{R}_q$.

We define the following quantities for keys $(a_1, ..., a_k, t_1, ..., t_k)$, $(s, e_1, ..., e_k)$, where we denote $\overrightarrow{a} = (a_1, ..., a_k)$ and $\overrightarrow{e} = (e_1, ..., e_k)$:

$$\mathrm{nwr}(\overrightarrow{a}, \overrightarrow{e}) = \Pr_{(y,c)\in\mathcal{R}_{q,[B]}\times\mathbb{H}} [a_i y - e_i c \text{ not well-rounded for at least one } i \in \{1, ..., k\}] \quad (7)$$

$$\mathrm{coll}(\overrightarrow{a}, \overrightarrow{e}) = \max_{(w_1,...,w_k)\in\mathbb{W}^k} \left\{ \Pr_{(y,c)\in\mathcal{R}_{q,[B]}\times\mathbb{H}} [[a_i y - e_i c]_M = w_i \text{ for } i = 1, ..., k] \right\}. \quad (8)$$

Informally speaking $\mathrm{nwr}(\overrightarrow{a}, \overrightarrow{e})$ refers to the probability over random (y, c) that $a_i y - e_i c$ is not well-rounded for some i. This quantity varies as a function of $a_1, ..., a_k, e_1, ..., e_k$. In contrast to [9], we cannot upper bound this in general in the ring setting. Hence, we first assume that $\mathrm{nwr}(\overrightarrow{a}, \overrightarrow{e}) < 3/4$ and afterwards check experimentally that this holds true. We check it by verifying that our acceptance probability for w_i in line 18 of Algorithm 7 (signature generation) is at least $1/4$ for our provably-secure parameter sets (see Table 6), which implies that the bound $\mathrm{nwr}(\overrightarrow{a}, \overrightarrow{e}) < 3/4$ holds.

Secondly, we need to bound the probability $\mathrm{coll}(\overrightarrow{a}, \overrightarrow{e})$. In [9, Lemma 4] the corresponding probability $\mathrm{coll}(A, E)$ for standard lattices is upper bounded, with $A \in \mathbb{Z}_q^{m\times n}$, $E \in \mathbb{Z}_q^{m\times n'}$, 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 [9, Lemma 4], it is used that if the randomness $y \in [-B, B]^n$ is not equal to 0, the vector Ay is uniformly random distributed over $\mathbb{Z}_q$ and hence also $Ay - Ec$ is uniformly random distributed over $\mathbb{Z}_q$. This does not necessarily hold if the *polynomial* y is chosen uniformly in $\mathcal{R}_{q,[B]}$. Moreover, in Equation (99) in [9], ψ denotes the probability that a random vector $x \in \mathbb{Z}_q^m$ is in $\Delta\mathbb{L}$, i.e.,

$$\psi = \Pr_{x\in\mathbb{Z}_q^m} [x \in \Delta\mathbb{L}] \leq \left(\frac{2^d + 1}{q}\right)^m. \quad (9)$$

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 6. *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 $a_i \cdot y$ is in $\{-2^d - 2E + 1, \ldots, 2^d + 2E - 1\}$.*

More formally,

$$\Pr_{(\overrightarrow{a},y)\leftarrow_{\$}\mathcal{R}_q^k\times\mathcal{R}_{q,[2B]}}[\forall i\in\{1,\ldots,k\}:\ a_i\cdot y\in R_{q,[2^d+2E-1]}]\leq\frac{1}{2^{-(n+2\lambda)}}\cdot\frac{|\mathbb{H}|}{|\Delta\mathbb{H}|}. \tag{10}$$

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 $\mathrm{coll}(\overrightarrow{a},\overrightarrow{e})$. This value corresponds to the maximum likelihood that the values a_iy-e_ic 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_iy-e_ic is 0, and this is a poor choice of a public key.

In the proof of TESLA [9], 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

$$\mathbb{G}(\overrightarrow{a},\overrightarrow{e})=\{(y,c)\in\Delta\mathbb{Y}\times\Delta\mathbb{H}:\forall i,a_iy-e_ic\in\Delta\mathbb{L}\}. \tag{11}$$

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 [9, Lemma 5], we can see that

$$\mathrm{coll}(\overrightarrow{a},\overrightarrow{e})\leq\frac{\mathbb{G}(\overrightarrow{a},\overrightarrow{e})}{|\mathbb{Y}|\cdot|\mathbb{H}|}. \tag{12}$$

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_ic is always quite small, we can see that the rounding $[a_iy]_M$ of a_iy is almost always the same as $[a_iy-e_ic]_M$, and ignore the effects of e_ic. By considering the maximum difference between two elements that round to the same value, we can replace $\Delta\mathbb{L}$ with $R_{q,[2^d-1]}$. Then since each coefficient of e_ic for a $\in\Delta\mathbb{H}$ is at most $2E$, we can see that $a_iy-e_ic\in\Delta\mathbb{L}$ implies that $a_iy\in\mathcal{R}_{q,[2^d+2E-1]}$. This allows us to define the set $\mathbb{G}(\overrightarrow{a})=\{y\in\Delta\mathbb{Y}:\forall i,a_iy\in\mathcal{R}_{q,[2^d+2E-1]}$ and establish that $|\mathbb{G}(\overrightarrow{a},\overrightarrow{e})|\leq|\Delta\mathbb{H}|\cdot|\mathbb{G}(\overrightarrow{a})|$.

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 in Appendix B.9 of [9], we get that the expected size of $\mathbb{G}(\overrightarrow{a})$ is equal to $|\Delta\mathbb{Y}|$ times the probability that appears in Equation 10.

If this probability is lower than the bound in Equation 10, then we can employ Markov's inequality to establish that

$$\Pr_{\overrightarrow{a},\overrightarrow{e}}[\mathrm{coll}(\overrightarrow{a},\overrightarrow{e})\geq 2^{-\lambda}]\leq 2^{-\lambda}. \tag{13}$$

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 to experiment with our parameters, we include a script in this submission[6] that samples such a y and a and checks if their product is in $\mathcal{R}_{q,[2^d+2E-1]}$. We do this to establish basic checks on the accuracy of the conjecture and enable the community to examine the problem for themselves. We have run our experiments script over the parameter sets `qTESLA-p-I` and `qTESLA-p-III` 10,000 times each, and have not observed an instance where a uniform element of $\mathcal{R}_q$ and $\mathcal{R}_{q,[2B]}$ was in $\mathcal{R}_{q,[2^d+2E-1]}$. This supports the claim that our conjecture holds true for the provably-secure instantiations of `qTESLA`.

Remark 7. *Our explanation above assumes an "expanded" public key* $(a_1,...,a_k,t_1,...,t_k)$*. In the description of* `qTESLA`*, however, the public polynomials* $a_1,...,a_k$ *are generated from* seed_a *which is part of the secret and public key. This assumption can be justified by another reduction in the QROM: assume there exists an algorithm* $\mathcal{A}$ *that breaks the original* `qTESLA` *scheme with public key* $(\mathsf{seed}_a,t_1,...,t_k)$*. Then we can construct an algorithm* $\mathcal{B}$ *that breaks a variant of* `qTESLA` *with "expanded" public key* $(a_1,...,a_k,t_1,...,t_k)$*. To this end, we model* $\mathsf{GenA}(\cdot)$ *as a (programmable) random oracle. The algorithm* $\mathcal{B}$ *chooses first* $\mathsf{seed}'_a \leftarrow_\$ \{0,1\}^\kappa$ *and reprograms* $\mathsf{GenA}(\mathsf{seed}'_a)=(a_1,...,a_k)$*. Afterwards, it forwards* $(\mathsf{seed}'_a,t_1,...,t_k)$ *as the input tuple to* $\mathcal{A}$*. Quantum queries to* $\mathsf{GenA}(\cdot)$ *by* $\mathcal{A}$ *can be simulated by* $\mathcal{B}$ *according to the construction of Zhandry based on* $2q_h$*-wise independent functions [64]. Hence, the assumption above also holds in the QROM.*

5.2 Bit security of our proposed parameter sets

In the following, we describe how we estimate the concrete security of the proposed parameters described in Section 2.8. To this end, we first describe how the security of our scheme depends on the hardness of R-LWE and afterwards we describe how we derive the bit hardness of the underlying R-LWE instance.

[6]This script can be found at `/Supporting_Documentation/Script_for_conjecture/Script_for_experiments_supporting_the_security_conjecture.py`.

5.2.1 Correspondence between security and hardness

The security reduction given by Theorem 5, in Section 5.1, provides a reduction from the hardness of the decisional ring learning with errors problem and bounds *explicitly* the forging probability with the success probability of the reduction. More formally, let ϵ_Σ and t_Σ denote the success probability and the runtime (resp.) of a forger against our signature scheme, and let ϵ_{LWE} and t_{LWE} denote analogous quantities for the reduction presented in the proof of Theorem 5. We say that R-LWE is *η-bit hard* if $t_{LWE}/\epsilon_{LWE} \geq 2^\eta$; and we say that the signature scheme is *λ-bit secure* if $t_\Sigma/\epsilon_\Sigma \geq 2^\lambda$.

For our *provably-secure* parameter sets, we choose parameters such that $\epsilon_{LWE} \approx \epsilon_\Sigma$ and $t_\Sigma \approx t_{LWE}$, that is, 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. Hence, the security reduction provably guarantees that our scheme instantiated with the provably-secure parameter sets has the selected security level as long as the corresponding R-LWE instance gives the assumed hardness level. This approach provides a *stronger* security argument.

For our *heuristic* parameter sets, we *assume* that the bit security of our scheme instantiated with these parameter sets is *theoretically* the same as the bit hardness of the corresponding R-LWE instance, without taking into account the security reduction. So far no attack that exploits this heuristic is known.

Remark 8. *In practical instantiations of* qTESLA*, the bit security does not exactly match the bit hardness of R-LWE (see Tables 5 and 6). This is because the bit security does not only depend on the bit hardness of R-LWE (as explained above), 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, heuristic and provably-secure, the key space is reduced by the rejection of polynomials $s, e_1, ..., e_k$ 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.*

Accordingly, we claim a bit security that is strictly smaller than the hardness of the corresponding R-LWE instance. For example, the hardness of the R-LWE instance corresponding to qTESLA-p-I is 140 bits but we claim a bit security of 95.

5.2.2 Estimation of the R-LWE hardness

Since the introduction of the learning with errors problem over rings [52], it has remained an open question to determine whether the R-LWE problem is as hard as the LWE problem. Several results exist that exploit the structure of some ideal lattices [24,27,34,36]. However, up to now, these results do not seem to apply to R-LWE instances that are typically used in signature schemes and, therefore, do not apply to the proposed `qTESLA` instances. 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 [8] presented the *LWE-Estimator*, a software to estimate the hardness of LWE given the matrix dimension n, the modulus q, the relative error rate $\alpha = \frac{\sqrt{2\pi}\sigma}{q}$, and the number of given LWE samples. The LWE-Estimator estimates the hardness against the fastest 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 [41], the dual lattice attacks recently published in [2], the enumeration approach by Linder and Peikert [48], the primal attack described in [6, 15], the Arora-Ge algorithm [12] using Gröbner bases [3], and the latest analysis to compute the block sizes used in the lattice basis reduction BKZ recently published by Albrecht *et al.* [7]. Moreover, quantum speedups for the sieving algorithm used in BKZ [46, 47] are also considered.

Arguably the most important building block in most efficient attacks against the underlying R-LWE instance in `qTESLA` is BKZ. Hence, the model used to estimate the cost of BKZ determines the overall hardness estimation of our instances. While many different cost models for BKZ exist [5], we decided to adopt the BKZ cost model of $0.265\beta + 16.4 + \log_2(8d)$, where β is the BKZ blocksize and d is the lattice dimension, for the hardness estimation of our parameters. In the LWE-Estimator this corresponds to using the option `cost_model = BKZ.qsieve`. This cost model is very conservative in the following sense: it only takes into account the number of operations needed to solve a certain instance but it assumes that the attacker can handle huge amounts of quantum memory. At the same time it matches practical state-of-the-art attacks, where (classical) experiments [26,57] show that during BKZ an SVP oracle is required to be called several times instead of only once as it is assumed in even more conservative models such as the cost model proposed in [10]. Still, to deal with potential future advances in cryptanalysis, we choose instances that currently provide a much higher hardness level than the targeted security level. For example, the hardness of `qTESLA-III` is estimated to be 188 bits (see Table 13) while we target a security of 160 bits for security category 3. We compare our chosen hardness estimation with the BKZ model from [10] in Table 13 (in the LWE-Estimator this cost model corresponds to using the option `cost_model = partial(BKZ.ADPS16,mode="quantum")`).

Table 13: Security estimation (bit hardness) under different BKZ cost models of the proposed parameter sets

BKZ cost model	qTESLA-I	qTESLA-II	qTESLA-III	qTESLA-V	qTESLA-V-size	qTESLA-p-I	qTESLA-p-III
BKZ.qsieve	111	138	188	377	261	140	279
BKZ.ADPS16,mode="quantum"	81	108	157	345	230	109	248
Targeted bit security	95	128	160	225	256	95	160
Security category	I	II	III	V	V	I	III

We note that another recent quantum attack, called quantum hybrid attack, by Göpfert, van Vredendaal, and Wunderer [39] is not considered in our analysis and the LWE-Estimator. This hybrid attack is most efficient on the learning with errors problem with very small secret and error, e.g., binary or ternary. Since the coefficients of the secret and error polynomials of `qTESLA` are chosen Gaussian distributed, the attack is not efficient for our instances.

The LWE-Estimator is the result of many different contributions and contributors. It is open source and hence easily checked and maintained by the community. Hence, we find the LWE-Estimator to be a suitable tool to estimate the hardness of our chosen LWE instances. We integrated the LWE-Estimator with commit-id `3019847` on 2019-02-14 in our sage script that is also included in this submission.

In the following we describe very briefly the most efficient LWE solvers for our instances, i.e., the decoding attack and the embedding approach, following closely the description of [20]. The Blum-Kalai-Wasserman algorithm [4, 44] is omitted since it requires exponentially many samples.

The embedding attack. The standard embedding attack solves LWE via reduction to the unique shortest vector problem (uSVP). During the reduction an $(m+1)$-dimensional lattice that contains the error vector e is created. Since e is very short for typical LWE instances, this results in a uSVP instance that is usually solved by applying basis reduction.

Let $(A, c = As + e \mod q)$ and t be the distance $\operatorname{dist}(c, L(A)) = \|c - x\|$ where $x \in L(A)$, such that $\|c - x\|$ is minimized. Then the lattice $L(A)$ can be embedded in the lattice $L(A')$, with $A' = \begin{pmatrix} A & c \\ 0 & t \end{pmatrix}$. If $t < \frac{\lambda_1(L(A))}{2\gamma}$, the higher-dimensional lattice $L(A')$ has a unique shortest vector $c' = (-e, t) \in Z_q^{m+1}$ with length $\|c'\| = \sqrt{m\alpha^2 q^2/(2\pi) + |t|^2}$ [28,51]. In the LWE-Estimator $t = 1$ is used. Therefore, e can be extracted from c', the value As is known, and s can be solved for. Based on Albrecht *et al.* [6], Göpfert shows [38, Section 3.1.3] that the standard embedding attack succeeds with non-negligible probability if $\delta_0 \leq$

$\left(\frac{q^{1-\frac{n}{m}}\sqrt{\frac{1}{e}}}{\tau\alpha q}\right)^{\frac{1}{m}}$, where m is the number of LWE samples. The value τ is experimentally determined to be $\tau \leq 0.4$ for a success probability of $\epsilon = 0.1$ [6].

The efficiency of the embedding attack highly depends on the number of samples. In the case of LWE instances with limited number of samples, the lattice $\Lambda_q^{\perp}(A_o) = \{v \in \mathbb{Z}^{m+n+1} | A_o \cdot v = 0 \bmod q\}$ with $A_o =[A|I|b]$ can be used as the embedding lattice.

The decoding attack. The decoding attack treats an LWE instance as an instance of the bounded distance decoding problem (BDD). The attack can be divided into two phases: basis reduction and finding the closest vector to the target vector. In the first phase, basis reduction algorithms like BKZ [61] are applied. Afterwards, in the second phase, the nearest plane algorithm [13] (or its variants) is applied to find the closest vector to As and thereby eliminate the error vector e of the LWE instance. The secret can then be accessed, as the closest vector equals the value As of an LWE instance.

5.3 Resistance to implementation attacks

Besides the theoretical security against computational attacks, such as lattice reduction, it is important for a cryptographic scheme to be secure against implementation attacks. These attacks come in two flavors: side-channel and fault analysis attacks.

5.3.1 Side-channel analysis

These attacks exploit physical information such as timing or power consumption, electromagnetic emanation, etc., that is correlated to some secret information during the execution of a cryptographic scheme. Simple and differential side-channel attacks that rely on power and electromagnetic emanations are very powerful but typically require physical access (or close proximity) to the targeted device. Protecting lattice-based schemes against this class of attacks is a very active area of research.

In contrast, attacks that exploit timing leakage, such as timing and cache attacks, are easier to carry out remotely. Hence, these attacks represent a more immediate danger for most applications and, consequently, it has become a minimum security requirement for a cryptographic implementation to be secure against this class of attacks. One effective approach to provide such a protection is by guaranteeing so-called *constant-time* execution. In practice, this means that an implementation should avoid the use of secret address accesses and conditional branches based on secret information and that the execution time should be independent of secret data.

One of the main advantages of qTESLA is that the Gaussian sampler, arguably the most complex part of the scheme, is restricted to key generation. This reduces drastically the attack surface to carry out a timing and cache attack against qTESLA. Moreover, we emphasize that qTESLA's Gaussian sampler is relatively simple and can be implemented securely in a constant-time manner, as can be observed in the accompanying implementations. Other functions of qTESLA, such as polynomial arithmetic operations, are easy to implement in constant-time as well.

Recently, the scheme ring-TESLA [1] was analyzed with respect to cache side channels with the software tool CacheAudit [22]. It was the first time that a post-quantum scheme was analyzed with program analysis. The authors found potential cache side channels, proposed countermeasures, and showed the effectiveness of their mitigations with CacheAudit. In our implementations, we apply similar techniques to those proposed in [22] with some additional optimizations.

It is relevant to note that qTESLA includes *built-in* defenses against several attack scenarios, thanks to its probabilistic nature. Specifically, the seed used to generate the randomness y is produced by hashing the value seed_y that is part of the secret key, some fresh randomness r, and the digest $\mathsf{G}(m)$ of the message m. The random value r guarantees the use of a fresh y at each signing operation, which increases the difficulty to carry out side-channel attacks against the scheme. Moreover, this fresh y prevents some easy-to-implement but powerful fault attacks against deterministic signature schemes, as explained next.

5.3.2 Fault analysis

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 [25].

Recently, some studies have exposed the vulnerability of lattice-based schemes to fault attacks. We describe a simple yet powerful attack that falls in this category of attacks [23].

Assume that line 3 of Algorithm 7 is computed without the random value r, i.e., as $\mathsf{rand} \leftarrow \mathsf{PRF}_2(\mathsf{seed}_y, m)$. Assume that a signature (z, c) is generated for a given message m. Afterwards, a signature is requested again for the same message m, but this time, a fault is injected on the computation of the hash value c yielding the value c_{faulted}. This second signature is $(z_{\text{faulted}}, c_{\text{faulted}})$. Computing $z - z_{\text{faulted}} = sc - sc_{\text{faulted}} = s(c - c_{\text{faulted}})$, reveals the secret s since $c - c_{\text{faulted}}$ is known to the attacker. As stated in [58], this attack has broad implications since it is generically applicable to deterministic *Schnorr-like* signatures.

It is easy to see that, to prevent this (and other similar) fault attacks, every signing operation should be injected with fresh randomness, as precisely specified in line 3 of Algorithm 7. This makes qTESLA implicitly resilient to this line of attacks.

6 Advantages and limitations

In this section, we summarize some advantages and limitations of qTESLA. schemes.

Security foundation. qTESLA comes accompanied by an *explicit* and *tight* security reduction in the quantum random oracle model, i.e., a quantum adversary is allowed to ask the random oracle in superposition. This reduction is based on a variant of our scheme over standard lattices [9]. To port the reduction given in [9], we use a heuristic argument as explained in Section 5.1. Since our security reduction is *explicit*, we can explicitly give the relation between the success probabilities of solving the R-LWE problem and forging qTESLA signatures. Moreover, the explicitness of the reduction enables choosing so-called *provably-secure* parameters according to the reduction. The tightness of the reduction enables smaller parameters and, thus, better performance of the provably-secure instantiations.

Flexible choice of parameters. We offer *two* approaches to instantiate qTESLA: using "heuristic" parameters and using "provably-secure" parameters that follow existing security reductions.

The heuristic approach identifies the security level of an instantiation of the scheme by a certain parameter set with the hardness level of the corresponding R-LWE instance, regardless of the tightness gap of the provided security reduction. This approach supports implementations that achieve very high-speed execution while requiring relatively compact keys and signatures.

The parameter choice according to the reduction, i.e., the provably-secure parameters, can be considered as providing a *stronger* security argument since it provably guarantees that the scheme has the selected security level as long as the corresponding R-LWE instance gives a certain hardness level. This approach supports implementations that are relatively slower and require larger public keys, but that can still be considered practical for many high-security applications.

In summary, when choosing between both options, provably-secure and heuristic, there is a trade-off between provably security assurance and high efficiency.

In addition, `qTESLA` does not only support the use of standard power-of-two cyclotomic rings, but also, for added flexibility in its parameterization, has been designed to support the use of a non-power-of-two cyclotomic ring. This flexibility allows us to expand our choices to produce very efficient instantiations.

The bit security of the proposed parameters was estimated against known state-of-the-art classical and quantum algorithms that solve the LWE problem. The proposed parameters, especially in the case of `provably-secure qTESLA`, incorporate a comfortable margin between the targeted and the estimated bit security in order to deal with future or unknown LWE solvers as well.

Finally, the parameter generation of `qTESLA` is easy to audit: the choice of parameters and their relation are clearly explained; and the generation procedure is simple and easy to follow. In order to facilitate the generation of new parameters (if needed), and also for transparency, we make our sage script for parameter generation available[7].

Ease of implementation and scalability. `qTESLA` has a very compact and simple structure consisting of a few, easy-to-implement functions. The Gaussian sampler, arguably the most complex function in `qTESLA`, is only required during key generation. Therefore, even if the efficient Gaussian sampler included in this document is not used, most applications will not be impacted by the use of a slower Gaussian sampler.

`qTESLA` exhibits great scalability, i.e., it is easy to support different security levels with a common codebase. For instance, our reference implementations of `heuristic qTESLA`, written in portable C, consist of about 350 lines of code shared among all the heuristic parameter sets [8].

High-speed and key and signature size compactness. Despite the simplicity of the scheme, `qTESLA` (especially with "heuristic" parameters) supports high-speed implementations with relatively compact keys and signatures. The superb performance of `heuristic qTESLA` is specifically achieved in the computation of signing and verification, and is obtained at the expense of a relatively less efficient key generation. This approach suits perfectly most scenarios in which key generation is executed less frequently or is done offline.

`qTESLA` signatures are also relatively compact, making them ideal for applications in which signature size is prioritized over public key size. The combined size of signature and

[7]The parameter generation script can be found at `\Supporting_Documentation\Script_to_choose_parameters\parameterchoice.sage`.

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

public key for `heuristic` `qTESLA` is competitive with other efficient schemes based on ideal lattices.

Furthermore, the `qTESLA` variants with smaller public keys using the splitting technique feature significantly reduced public keys at the expense of a slightly larger signature size (see Section 2.7). These variants offer some of the most compact public keys among post-quantum signature schemes, while exhibiting competitive signature sizes.

As previously stated, `provably-secure` `qTESLA` implementations are slower and require larger public keys, but they are still practical for many applications and come with provably secure guarantees that other signature schemes do not provide.

Security against implementation attacks. `qTESLA`'s simplicity facilitates its implementation in constant-time and, arguably, its protection against more powerful physical attacks such as differential power analysis. As stated before, Gaussian sampling is only required during key generation, which reduces significantly the attack surface over this function. Moreover, `qTESLA` requires a *simpler* Gaussian sampler which further eases implementation and protection against side-channel attacks.

`qTESLA` comes equipped with built-in measures that provide a first layer of defense against some side-channel and fault attacks. Since `qTESLA` generates a fresh y per signing operation, some simple side-channel attacks are more difficult to mount against the scheme. More importantly, this feature immediately renders some powerful and easy-to-carry out fault attacks unfeasible. At the same time, `qTESLA` is resilient to a catastrophic failure of the RNG during generation of the fresh randomness that is required to generate y.

Cryptographic libraries and hybrid mode. The reference implementation of `qTESLA` is integrated in several cryptographic libraries. In particular, the C reference implementation is integrated to the library *pqm4* [42] that targets microcontrollers, and the post-quantum cryptography library *liboqs* [11] as part of the *Open Quantum Safe* project [53]. Also, an implementation of `qTESLA` written in Java (not included in this submission) is integrated in the cryptographic library *BouncyCastle*.

Another contribution of the *Open Quantum Safe* project is an OpenSSL fork that integrates *liboqs* to OpenSSL and includes post-quantum and hybrid authentication in TLS [55]. Hybrid authentication enables to authenticate using classical and post-quantum signature schemes in order to preserve classical security, while adding post-quantum security. The post-quantum/hybrid authentication can already be instantiated with `qTESLA-I` and `qTESLA-III`. Additional implementations of hybrid certificates and hybrid authentication in TLS 1.3 can be found in [37] and [63] using the `qTESLA` integrations in *liboqs* and *BouncyCastle*, respectively. This shows the suitability of `qTESLA` used in hybrid schemes, an approach that seems to be favored by industry to enable a smooth and secure transition to post-quantum cryptography.

Diverse applications. qTESLA is present in several applications and projects from which we shortly mention two of them. qTESLA is used in a field study to protect medical health data through an experimental implementation of TLS. This collaboration between TU Darmstadt, the National Institute of Information and Communications Technology (NICT) and ISARA has been presented at the 6th ETSI/IQC Workshop on Quantum-Safe Cryptography.[9]

A variant of qTESLA was used as basis in the design of a new post-quantum certificate provisioning process for vehicle-to-everything (V2X) communications. This application allows provisioning of pseudonym certificates such that vehicles are able to engage in V2X communications in an environment with privacy by design, meaning that vehicles cannot be tracked by other entities in the system by the use of these pseudonym certificates [17].

References

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

[2] Martin R. Albrecht. On dual lattice attacks against small-secret LWE and parameter choices in HElib and SEAL. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part II*, volume 10211 of *Lecture Notes in Computer Science*, pages 103–129, Paris, France, April 30 – May 4, 2017. Springer, Heidelberg, Germany.

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

[4] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. On the complexity of the BKW algorithm on LWE. *Designs, Codes and Cryptography*, 74(2):325–354, 2015.

[5] Martin R. Albrecht, Benjamin R. Curtis, Amit Deo, Alex Davidson, Rachel Player, Eamonn W. Postlethwaite, Fernando Virdia, and Thomas Wunderer. Estimate all the

[9]Contact persons: Atsushi Yamada (ISARA corporation), Atsushi.Yamada@isara.com and Masahide Sasaki (NICT), psasaki@nict.go.jp

LWE, NTRU schemes! In Dario Catalano and Roberto De Prisco, editors, *SCN 18: 11th International Conference on Security in Communication Networks*, volume 11035 of *Lecture Notes in Computer Science*, pages 351–367, Amalfi, Italy, September 5–7, 2018. Springer, Heidelberg, Germany.

[6] Martin R. Albrecht, Robert Fitzpatrick, and Florian Göpfert. On the efficacy of solving LWE by reduction to unique-SVP. In Hyang-Sook Lee and Dong-Guk Han, editors, *ICISC 13: 16th International Conference on Information Security and Cryptology*, volume 8565 of *Lecture Notes in Computer Science*, pages 293–310, Seoul, Korea, November 27–29, 2014. Springer, Heidelberg, Germany.

[7] Martin R. Albrecht, Florian Göpfert, Fernando Virdia, and Thomas Wunderer. Revisiting the expected cost of solving uSVP and applications to LWE. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 297–322, Hong Kong, China, December 3–7, 2017. Springer, Heidelberg, Germany.

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

[9] Erdem Alkim, Nina Bindel, Johannes A. Buchmann, Özgür Dagdelen, Edward Eaton, Gus Gutoski, Juliane Krämer, and Filip Pawlega. Revisiting TESLA in the quantum random oracle model. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017*, pages 143–162, Utrecht, The Netherlands, June 26–28 2017. Springer, Heidelberg, Germany.

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

[11] Nicholas Allen, Maxime Anvari, Eric Crockett, Nir Drucker, Vlad Gheorghiu, Shay Gueron, Christian Paquin, Tancréde Lepoint, Shravan Mishra, and Douglas Stebila. liboqs – nist-branch: C library for quantum-resistant cryptographic algorithms, 2018. GitHub at `https://github.com/open-quantum-safe/liboqs`, commit-id: 86c6ab1.

[12] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP 2011: 38th International Colloquium on Automata, Languages and Programming, Part I*, volume 6755 of *Lecture Notes in Computer Science*, pages 403–415, Zurich, Switzerland, July 4–8, 2011. Springer, Heidelberg, Germany.

[13] László Babai. On Lovász' lattice reduction and the nearest lattice point problem. In K. Mehlhorn, editor, *STACS 1985*. Springer, 1985.

[14] Shi Bai and Steven D. Galbraith. An improved compression technique for signatures based on learning with errors. In Josh Benaloh, editor, *Topics in Cryptology – CT-RSA 2014*, volume 8366 of *Lecture Notes in Computer Science*, pages 28–47, San Francisco, CA, USA, February 25–28, 2014. Springer, Heidelberg, Germany.

[15] Shi Bai and Steven D. Galbraith. Lattice decoding attacks on binary LWE. In Willy Susilo and Yi Mu, editors, *ACISP 14: 19th Australasian Conference on Information Security and Privacy*, volume 8544 of *Lecture Notes in Computer Science*, pages 322–337, Wollongong, NSW, Australia, July 7–9, 2014. Springer, Heidelberg, Germany.

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

[17] Paulo S. L. M. Barreto, Jefferson E. Ricardini, Marcos A. Simplicio Jr., and Harsh Kupwade Patil. qscms: Post-quantum certificate provisioning process for v2x. Cryptology ePrint Archive, Report 2018/1247, 2018.

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

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

[20] Nina Bindel, Johannes Buchmann, Florian Göpfert, and Markus Schmidt. Estimation of the hardness of the learning with errors problem with a restricted number of samples. Cryptology ePrint Archive, Report 2017/140, 2017. https://eprint.iacr.org/2017/140.

[21] Nina Bindel, Johannes Buchmann, and Juliane Krämer. Lattice-based signature schemes and their sensitivity to fault attacks. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016*, pages 63–77. IEEE Computer Society, 2016.

[22] Nina Bindel, Johannes Buchmann, Juliane Krämer, Heiko Mantel, Johannes Schickel, and Alexandra Weber. Bounding the cache-side-channel leakage of lattice-based signature schemes using program semantics. In *Proceedings of the 10th International Symposium on Foundations & Practice of Security (FPS)*, 2017. To appear.

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

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

[25] H.M. Cantero, S. Peter, Bushing, and 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.

[26] Yuanmi Chen. *Réduction de réseau et sécurité concrète du chiffrement com-plètement homomorphe*. PhD thesis, Paris, France, 2013.

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

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

[29] Jelle Don, Serge Fehr, Christian Majenz, and Christian Schaffner. Security of the Fiat-Shamir Transformation in the Quantum Random-Oracle Model. *CoRR*, abs/1902.07556, 2019.

[30] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal Gaussians. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 40–56, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Heidelberg, Germany.

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

[32] Léo Ducas, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS – Dilithium: Digital signatures from module lattices, 2017. http://cryptojedi.org/papers/#dilithium.

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

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

[35] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Loop-abort faults on lattice-based fiat-shamir and hash-and-sign signatures. In Roberto Avanzi and Howard M. Heys, editors, *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference*, volume 10532 of *Lecture Notes in Computer Science*, pages 140–158. Springer, 2017.

[36] Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 1–17, Athens, Greece, May 26–30, 2013. Springer, Heidelberg, Germany.

[37] Luca Gladiator and Tobias Stöckert. Fork of OQS-OpenSSL_1_1_1-stable, 2019. GitHub at https://github.com/CROSSINGTUD/openssl-hybrid-certificates, commit-id: 6ea0607.

[38] Florian Göpfert. *Securely Instantiating Cryptographic Schemes Based on the Learning with Errors Assumption*. PhD thesis, Darmstadt University of Technology, Germany, 2016.

[39] Florian Göpfert, Christine van Vredendaal, and Thomas Wunderer. A hybrid lattice basis reduction and quantum search attack on LWE. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017*, volume 10346 of *LNCS*, pages 184–202. Springer, 2017.

[40] Shay Gueron and Fabian Schlieker. Optimized implementation of ring-TESLA. GitHub at https://github.com/fschlieker/ring-TESLA, 2016.

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

[42] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4–Post-quantum crypto library for the ARM Cortex-M4, 2018. GitHub at https://github.com/mupq/pqm4, commit-id: 133c0e8.

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

[44] Paul Kirchner and Pierre-Alain Fouque. An improved BKW algorithm for lwe with applications to cryptography and lattices. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, volume 9215 of *LNCS*, pages 43–62. Springer, 2015.

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

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

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

[48] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-based encryption. In Aggelos Kiayias, editor, *Topics in Cryptology – CT-RSA 2011*, volume 6558 of *Lecture Notes in Computer Science*, pages 319–339, San Francisco, CA, USA, February 14–18, 2011. Springer, Heidelberg, Germany.

[49] Qipeng Liu and Mark Zhandry. Revisiting Post-Quantum Fiat-Shamir. Cryptology ePrint Archive, Report 2019/262, 2019. https://eprint.iacr.org/2019/262.

[50] Vadim Lyubashevsky. Lattice signatures without trapdoors. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 738–755, Cambridge, UK, April 15–19, 2012. Springer, Heidelberg, Germany.

[51] Vadim Lyubashevsky and Daniele Micciancio. On bounded distance decoding, unique shortest vectors, and the minimum distance problem. In *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference*, pages 577–594. Springer, 2009.

[52] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23, French Riviera, May 30 – June 3, 2010. Springer, Heidelberg, Germany.

[53] Michele Mosca and Douglas Stebila. Open quantum safe – software for prototyping quantum-resistant cryptography, 2018. https://openquantumsafe.org/. Accessed: 2018-07-26.

[54] 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.

[55] Christian Paquin and Douglas Stebila. OQS-OpenSSL_1_1_1-stable, 2018. GitHub at https://github.com/open-quantum-safe/openssl, commit-id: 57a9724.

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

[57] Rachel Player. *Parameter selectionin lattice-based cryptography.* PhD thesis, Royal Holloway, University of London, London, United Kingdom, 2018.

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

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

[60] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact Ring-LWE cryptoprocessor. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014*, volume 8731 of *Lecture Notes in Computer Science*, pages 371–391. Springer, 2014.

[61] Claus P. Schnorr and Taras Shevchenko. Solving subset sum problems of densioty close to 1 by randomized BKZ-reduction. Cryptology ePrint Archive, Report 2012/620, 2012. http://eprint.iacr.org/2012/620.

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

[63] Johannes Wirth. HybridCertificates, 2018. GitHub at https://github.com/jojowi/HybridCertificate, commit-id: 8518175.

[64] Mark Zhandry. Secure identity-based encryption in the quantum random oracle model. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 758–775, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.