Submission to NIST’s post-quantum project:
lattice-based digital signature scheme qTESLA

Name of the cryptosystem: qTESLA
Principal and auxiliary submitters:

Nina Bindel, Technische Universität Darmstadt,
(Principal submitter) 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, Ege University, Turkey
Paulo S. L. M. Barreto, University of Washington Tacoma, USA
Johannes Buchmann, Technische Universität Darmstadt, Germany
Edward Eaton, ISARA Corporation, 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).
## Changelog

This is the changelog of this document and the corresponding implementation of qTESLA.

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Description of changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>11/30/2017</td>
<td>• Original submission to NIST.</td>
</tr>
<tr>
<td>2.0</td>
<td>06/14/2018</td>
<td>• qTESLA described generically using ( k &gt; 1 ) R-LWE samples.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Signing algorithm changed to probabilistic (instead of deterministic).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• New parameter sets proposed: three heuristic and two provably-secure parameter sets.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Improved explanation of the realization of the different functions (Section 2.4).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Minor changes and refinements throughout the document.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• C-only reference implementation corrected; e.g., to have proper protection against timing and cache attacks.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• C-only reference implementation improved; e.g., to have more resilience against certain fault attacks.</td>
</tr>
<tr>
<td>2.1</td>
<td>06/30/2018</td>
<td>• Corrected typo that assumed a exponent ( d ) or ( d + 1 ) instead of ( d − 1 ) or ( d ) (resp.) in some places.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Small fix in the bounds of the signature rejection evaluation, line 18 of Algorithm 7. Updated KATs accordingly.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Applied notation ( \text{mod}^{±} ) to denote the use of a centered representative in Algorithms 7 and 8.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated correctness proof in Section 2.3.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Replaced bit-hardness by corresponding bit-security figures for the qTESLA parameter sets in Table 6.</td>
</tr>
<tr>
<td>2.2</td>
<td>08/27/2018</td>
<td>• Corrected typo in the definition of ( \text{mod}^{±} ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Corrected typo in the signature verification algorithm, line 6 of Algorithm 8.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Some corrections in Algorithm 10. Rearranged if-blocks to maximize use of cSHAKE128’s output. Updated KATs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Corrected typos in Algorithm 13. Rearranged if-blocks to maximize use of cSHAKE128’s output.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added rejection of value ( B + 1 ) during sampling of ( y ), Algorithm 14.</td>
</tr>
</tbody>
</table>
## Contents

1 Introduction 4
   1.1 qTESLA highlights ................................................. 4
   1.2 Related work ..................................................... 6

2 Specification 8
   2.1 Basic signature scheme ........................................... 8
   2.2 Formal description of qTESLA .................................... 9
   2.3 Correctness of the scheme ....................................... 13
   2.4 Realization of the required functions .......................... 14
      2.4.1 Hash and pseudorandom functions .......................... 14
      2.4.2 Pseudorandom bit generation of seeds .................... 15
      2.4.3 Generation of $a_1, ..., a_k$ .............................. 16
      2.4.4 Gaussian sampling ......................................... 17
      2.4.5 Encoding function ......................................... 20
      2.4.6 Sampling of $y$ ............................................ 22
      2.4.7 Hash $H$ ................................................... 23
      2.4.8 Polynomial multiplication and the number theoretic transform .... 23
   2.5 System parameters and parameter selection ..................... 24

3 Performance analysis 29
   3.1 Associated implementations .................................... 29
   3.2 Performance on x64 Intel ....................................... 30

4 Known answer values 31

5 Expected security strength 32
   5.1 Provable security in the (quantum) random oracle model ........ 32
   5.2 Bit security of our proposed parameter sets .................... 34
      5.2.1 Correspondence between security and hardness ............ 35
      5.2.2 Estimation of the R-LWE hardness ......................... 35
   5.3 Resistance to implementation attacks .......................... 37
      5.3.1 Side-channel analysis .................................... 37
      5.3.2 Fault analysis ........................................... 38

6 Advantages and limitations 39
1 Introduction

This document presents a detailed specification of qTESLA, a 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.

Concretely, qTESLA includes five parameter sets targeting two security levels:

I Heuristic qTESLA:
(1) qTESLA-I: NIST’s security category 1.
(2) qTESLA-III-speed: NIST’s security category 3 (option for speed).
(3) qTESLA-III-size: NIST’s security category 3 (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 applica-
tions in which the additional security assurances from the provably-secure proof 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.

- **Compactness of signatures.** qTESLA signatures are designed to be relatively small, at the expense of (slightly) larger public keys.

- **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), and is inherently protected against certain simple yet powerful fault attacks.

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

In addition, Heuristic qTESLA, comprising the parameters sets qTESLA-I, qTESLA-III-speed and qTESLA-III-size, features:

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

For added flexibility, we present two options for Heuristic qTESLA for the NIST’s security category 3, namely, qTESLA-III-speed, which prioritizes raw speed over signature and key sizes, and qTESLA-III-size, which prioritizes signature and key sizes over speed.

**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 \(\sigma\), 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.

In summary, the qTESLA family of post-quantum signature schemes offers great flexibility by allowing a selection of 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 [47]. 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. [26] 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, 39]. Most notably, a scheme called TESLA# [16] by Barreto, Longa, Naehrig, Ricardini, and Zanon included several implementation improvements. Finally, several works [19, 20, 34] have focused on the analysis of ring-TESLA against side-channel and fault attacks.

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 some typos in a previous version of this document, 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).

SA and EA are partially supported by TÜBITAK under grant no. EEEAG-116E279. NB is partially supported by the German Research Foundation (DFG) as part of project P1 within the CRC 1119 CROSSING. JR 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

Next, we give an informal description of the basic scheme that is used to specify \texttt{qTESLA}. A formal specification of \texttt{qTESLA}'s key generation, signing and verification algorithms then follows in Section 2.2. The correctness of the scheme is discussed in Section 2.3. We describe the implementation of the functions required by \texttt{qTESLA} in Section 2.4, and explain all the system parameters and the proposed parameter sets in Section 2.5.

2.1 Basic signature scheme

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

Let \( q, L_E, L_S, \) and \( d \) be system parameters that denote the modulus, the bound constant for error polynomials, the bound constant for the secret polynomial, and the rounding value, respectively. An integer polynomial \( y \) is \textit{B-short} if each coefficient is at most \( B \) in absolute value. We call an integer polynomial \( w \) \textit{well-rounded} if \( w \) is \( (\lfloor q/2 \rfloor - L_E) \)-short and \( \lfloor w \rfloor \) is \( (2^{d-1} - L_E) \)-short, where \( \lfloor w \rfloor \) denotes the unique integer in \((-2^{d-1}, 2^{d-1}) \subset \mathbb{Z}\) such that \( w = \lfloor w \rfloor \mod 2^d \). Likewise, \( \lfloor w \rfloor_M \) is the value represented by all but the \( d \) least significant bits of \((w - \lfloor w \rfloor)\). Let \( R = \mathbb{Z}[x]/(x^n + 1) \) and \( R_q = \mathbb{Z}_q[x]/(x^n + 1) \). For simplicity we assume that the hash oracle \( H(\cdot) \) maps from \( \{0, 1\}^* \) to \( \mathbb{H} \), where \( \mathbb{H} \) denotes the set of polynomials \( c \in R \) with coefficients in \( (-1, 0, 1) \) with exactly \( h \) nonzero entries, i.e., we ignore the encoding function \( F \) introduced in Section 2.2.

Because of the random generation of the polynomial \( y \) (see line 1 of Alg. 2), Algorithm 2 is described as a \textit{non-deterministic} algorithm. This property implies that different randomness is required for each signature. For the formal specification of \texttt{qTESLA} we incorporate an additional improvement: \texttt{qTESLA} requires a combination of fresh randomness and a fixed value for the generation of \( y \) (see Section 2.2). This design feature is added in order to pre-

\begin{algorithm}
Informal description of the key generation

\textbf{Require:} -

\textbf{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: \( a_1, ..., a_k \leftarrow R_q \) invertible ring elements.
2: Choose \( s \in R \) with entries from \( D_\sigma \). Repeat step at the \( h \) largest entries of \( s \) sum to \( L_S \).
3: For \( i = 1, ..., k \): Choose \( e_i \in R \) with entries from \( D_\sigma \). Repeat step at iteration \( i \) if the \( h \) largest entries of \( e_i \) sum to \( L_E \).
4: For \( i = 1, ..., k \): Compute \( t_i \leftarrow a_i s + e_i \in R_q \).
5: Return \( sk = (s, e_1, ..., e_k, a_1, ..., a_k) \) and \( pk = (a_1, ..., a_k, t_1, ..., t_k) \).
\end{algorithm}
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 \( R_q \).
2: \( c \leftarrow H([a_1y]_M, ..., [a_ky]_M, m) \).
3: Compute \( z \leftarrow y + sc \).
4: If \( z \) is not \((B - L_S)\)-short then retry at step 1.
5: For \( i = 1, ..., k \): If \( a_iy - e_ic \) 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 \((B - L_S)\)-short then return reject.
2: For \( i = 1, ..., k \): Compute \( w_i \leftarrow a_i z - t_i c \in R_q \).
3: If \( c \neq H([w_1]_M, ..., [w_k]_M, m) \) then return reject.
4: Return accept.

vent 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.

2.2 Formal description of qTESLA

Below, we define all the necessary functions, sets, and system parameters in qTESLA.

The description of the scheme depends on the following system parameters: \( \lambda, \kappa, n, k, q, \sigma, L_E, L_S, B, d, h \), and \( b_{GenA} \). Let \( \mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z} \), \( R = \mathbb{Z}[x]/(x^n + 1) \), \( R_q = \mathbb{Z}_q[x]/(x^n + 1) \), \( R_{q,l} = \{ f \in \mathbb{R}_q \mid f = \sum_{i=0}^{n-1} f_i x^i, f_i \in [-I, I] \} \), and \( \mathbb{H}_{n,h} = \{ f \in \mathbb{R}_q \mid f = \sum_{i=0}^{n-1} f_i x^i, f_i \in \{-1, 0, 1\}, \sum_{i=0}^{n-1} f_i = h \} \). Let \( R \) be a ring, and let inverse elements in this ring be represented by \( R^\times \). Let \( f = \sum_{i=0}^{n-1} f_i x^i \in R \). Then we define the reduction of \( f \) modulo \( q \) to be \( (f \mod q) = \sum_{i=0}^{n-1} (f_i \mod q) x^i \in R_q \). Let \( d \in \mathbb{N} \) and \( c \in \mathbb{Z} \). For an even (odd) modulus \( m \in \mathbb{Z}_{\geq 0} \), define \( c' = c \mod \frac{m}{2} \) as the unique element \(-m/2 < c' \leq m/2 \) (resp. \(-|m/2| \leq c' \leq |m/2| \)) such that \( c' = c \mod m \). As defined before, we denote by \([c]_L \) the unique integer in \((-2^{d-1}, 2^{d-1}] \subset \mathbb{Z} \) such that \( c = [c]_L \mod 2^d \); that is, \([c]_L = c \mod 2^d \). Let \([\cdot]_M : \mathbb{Z} \rightarrow \mathbb{Z}, c \mapsto (c \mod 2^d - [c]_L)/2^d \). Furthermore, let \( f = \sum_{i=0}^{n-1} f_i x^i \in R_q \), for which \( [f]_L = \sum_{i=0}^{n-1} [f_i]_L x^i \) and \([f]_M = \sum_{i=0}^{n-1} [f_i]_M x^i \). Let \( f \in R_q \) be a polynomial with coefficients being ordered (without losing any generality) as \( |f_1| \geq |f_2| \geq ... \geq |f_n| \). Then we define \( \max_i(f) = f_i \). Parameter \( b_{GenA} \in \mathbb{Z}_{\geq 0} \) represents
the number of blocks requested to cSHAKE128 in a first call during generation of the public polynomials $a_1, \ldots, a_k$.

In the remainder, we write $f = \sum_{i=0}^{n-1} f_i x^i$ to denote a polynomial in $\mathbb{R}_q$ (i.e., a polynomial of degree at most $n - 1$ with coefficients from the ring $\mathbb{Z}_q$). We use the same symbol $f$ to also denote the coefficient vector $f = (f_0, f_1, \ldots, f_{n-1}) \in \mathbb{Z}_q^n$. In some instances we represent polynomials as $f_j$ (e.g., to represent polynomials $a_1, \ldots, a_k$). In these cases, we write $f_j = \sum_{i=0}^{n-1} f_{j,i} x^i$, and the corresponding vector representation is given by $f_j = (f_{j,0}, f_{j,1}, \ldots, f_{j,n-1}) \in \mathbb{Z}_q^n$. Likewise, $s$-bit strings $r \in \{0, 1\}^s$ and $r' \in \{-1, 0, 1\}^s$ are written as vectors over the sets $\{0, 1\}$ and $\{-1, 0, 1\}$ (resp.), in which an element in the $i$-th position is represented by $r_i$ ($r'_i$, resp.). 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 $\{0, 1\}$.

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

qTESLA’s algorithms for key generation, signing and signature verification are given in Algorithms 4, 7, and 8, respectively. The two subroutines checkE and checkS that are called during key generation are depicted in Algorithms 6 and 5, respectively.

**Remark 1.** In an earlier description of qTESLA, we described the algorithms in a less general way using only one sample of the ring learning with errors problem. In particular, the public key consisted of seed$_a$ (corresponding to $a$) and $t$, and the secret key consisted of the polynomials $s$ and $e$, together with seed$_y$ and seed$_a$.

We define the following functions that are required in the implementation of qTESLA (refer to the specified sections for explicit details about their realization):

- The pseudorandom function $\text{PRF}_1 : \{0, 1\}^\kappa \rightarrow \{0, 1\}^{\kappa,k+3}$. This function takes as input a seed pre-seed that is $\kappa$ bits long and maps it to $(k + 3)$ seeds of $\kappa$ bits each (cf. Section 2.4.2).

- The pseudorandom function $\text{PRF}_2 : \{0, 1\}^\kappa \times \{0, 1\}^\kappa \times \{0, 1\}^* \rightarrow \{0, 1\}^\kappa$. This function takes as inputs the seed seed$_y$ and the random value $r$, each $\kappa$ bits long, and a message $m$ and maps them to the seed rand of $\kappa$ bits (cf. line 3 of Algorithm 7).

- The generation function of the public polynomials $a_1, ..., a_k$ GenA : $\{0, 1\}^\kappa \rightarrow \mathcal{R}_q^k$. This function takes as input the seed seed$_a$ that is $\kappa$ bits long and maps it to $k$ polynomials $a_i \in \mathcal{R}_q$ (cf. Section 2.4.3).

- The Gaussian sampler function GaussSampler : $\{0, 1\}^\kappa \times \mathbb{Z} \rightarrow \mathbb{Z}$. This function
samples the polynomials $s, e_1, \ldots, e_k$ according to the Gaussian distribution $D_\sigma$ taking as inputs a $\kappa$-bit seed $\text{rand}$ and a nonce $S \in \mathbb{Z}_{>0}$ (cf. Section 2.4.4).

- The encoding function $\text{Enc} : \{0, 1\}^\kappa \to \{-1, 0, 1\}^{k, 2}$. This function encodes a $\kappa$-bit hash value $c'$ as a polynomial $c \in \mathbb{H}_{n, h}$. The polynomial $c$ is in turn encoded as the two lists $\text{post\_list}$ and $\text{sign\_list} \in \{-1, 0, 1\}^h$ containing the positions and signs of its nonzero coefficients, respectively (cf. Section 2.4.5).

- The sampling function of the polynomial $y$, $\text{ySampler} : \{0, 1\}^\kappa \times \mathbb{Z} \to \mathbb{R}_{q, [B]}$. This function samples a polynomial $y \in \mathbb{R}_{q, [B]}$ taking as inputs a $\kappa$-bit seed $\text{rand}$ and a nonce $S \in \mathbb{Z}_{>0}$ (cf. Section 2.4.6).

- The hash function $H : \mathbb{R}_{q, [B]}^k \times \{0, 1\}^* \to \{0, 1\}^\kappa$. This function takes as inputs $k$ polynomials $v_1, \ldots, v_k \in \mathbb{R}_{q, [B]}$ and computes $[v_1]_M, \ldots, [v_k]_M$. The result is then hashed together with a message $m$ to a string $\kappa$ bits long (cf. Section 2.4.7).

\begin{algorithm}
\caption{qTESLA’s key generation}
\begin{algorithmic}
\Require {-} \\
\Ensure secret key $sk = (s, e_1, \ldots, e_k, \text{seed}_a, \text{seed}_y)$, and public key $pk = (\text{seed}_a, t_1, \ldots, t_k)$
\State $\text{pre\_seed} \leftarrow \{0, 1\}^\kappa$
\State $\text{seed}_s, \text{seed}_{e_1}, \ldots, \text{seed}_{e_k}, \text{seed}_a, \text{seed}_y \leftarrow \text{PRF}_1(\text{pre\_seed})$ \hspace{1cm} [Algorithm 9]
\State $a_1, \ldots, a_k \leftarrow \text{GenA}($ $\text{seed}_a)$ \hspace{1cm} [Algorithm 10]
\State $\text{do}$
\State $s \leftarrow \sigma \mathcal{R}$ \hspace{1cm} [Algorithm 11]
\State $\text{while checkS}(s) \neq 0$ \hspace{1cm} [Algorithm 5]
\State $\text{for } i = 1, \ldots, k \text{ do}$
\State $e_i \leftarrow \sigma \mathcal{R}$ \hspace{1cm} [Algorithm 11]
\State $\text{while checkE}(e_i) \neq 0$ \hspace{1cm} [Algorithm 6]
\State $t_i \leftarrow a_i s + e_i \mod q$
\State $\text{end for}$
\State $sk \leftarrow (s, e_1, \ldots, e_k, \text{seed}_a, \text{seed}_y)$
\State $pk \leftarrow (\text{seed}_a, t_1, \ldots, t_k)$
\State $\text{return } sk, pk$
\end{algorithmic}
\end{algorithm}
**Algorithm 5** `checkS`: simplifies the security reduction by ensuring that $\|sc\|_\infty \leq L_S$.

**Require:** $s \in \mathcal{R}$

**Ensure:** $\{0, 1\} \triangleright \text{true, false}$

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

---

**Algorithm 6** `checkE`: ensures correctness of the scheme by checking that $\|ec\|_\infty \leq L_E$.

**Require:** $e \in \mathcal{R}$

**Ensure:** $\{0, 1\} \triangleright \text{true, false}$

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

---

**Algorithm 7** qTESLA’s signature generation

**Require:** message $m$, and secret key $sk = (s, e_1, ..., e_k, seed_a, seed_y)$

**Ensure:** signature $(z, c')$

1: counter $\leftarrow 0$
2: $r \leftarrow \{0, 1\}^c$
3: rand $\leftarrow$ PRF$_2(seed_y, r, m)$
4: $y \leftarrow y_{\text{Sampler}}(\text{rand}, \text{counter})$ \hspace{1em} \[\text{Algorithm 14}\]
5: $a_1, ..., a_k \leftarrow \text{GenA}(seed_a)$ \hspace{1em} \[\text{Algorithm 10}\]
6: for $i = 1, ..., k$ do
7: \hspace{1em} $v_i = a_i y \mod q$
8: end for
9: $c' \leftarrow H([v_1]_M, ..., [v_k]_M, m)$ \hspace{1em} \[\text{Algorithm 15}\]
10: $c \leftarrow \text{Enc}(c')$ \hspace{1em} \[\text{Algorithm 13}\]
11: $z \leftarrow y + sc$
12: if $z \notin \mathcal{R}_{q,[B-L_S]}$ then
13: \hspace{1em} counter++
14: \hspace{1em} Restart at step 4
15: end if
16: for $i = 1, ..., k$ do
17: \hspace{1em} $w_i \leftarrow v_i - e_i c \mod q$
18: \hspace{1em} if $\|w_i\|_L \geq 2^{d-1} - L_E$ or $\|w_i\|_\infty \geq \lfloor q/2 \rfloor - L_E$ then
19: \hspace{2em} counter++
20: \hspace{2em} Restart at step 4
21: \hspace{1em} end if
22: end for
23: return $(z, c')$
Thus, from $\lfloor \alpha \rfloor$, we assume that for integers $\alpha$, proving that $(1)$. 

**Proof.** To establish the correctness of qTESLA we need to prove that the nonce input to the hash function $H$ at signing (line 9 of Algorithm 7) is the same as the nonce input to the hash function $H$ at verification (line 6 of Algorithm 8). That is, we need to prove that, for genuine signatures, $[ay mod^\pm q]_M = [az - tc mod^\pm q]_M = [a(y + sc) - (as + c) mod^\pm q]_M = [ay + asc - asc - ec mod^\pm q]_M = [ay - ec mod^\pm q]_M$. From the definition of $[.]_M$, this means proving that $((ay mod^\pm q - [ay mod^\pm q]_L)/2^d = (ay - ec mod^\pm q - [ay - ec mod^\pm q]_L)/2^d$, or simply $[ay mod^\pm q]_L = ec + [ay - ec mod^\pm q]_L$.

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

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

$$[\alpha]_L = \varepsilon + [\alpha - \varepsilon mod^\pm q]_L. \quad (1)$$

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

$$|[\varepsilon]_L + [\alpha - \varepsilon mod^\pm q]_L| = |\varepsilon|_L + [\alpha - \varepsilon mod^\pm q]_L = \varepsilon + [\alpha - \varepsilon mod^\pm q]_L. \quad (2)$$

Next we prove that

$$|[\varepsilon]_L + [\alpha - \varepsilon mod^\pm q]_L| = [\alpha]_L. \quad (3)$$

---

**Algorithm 8** qTESLA’s signature verification

**Require:** message $m$, signature $(z, c')$, and public key $pk = (seed_a, t_1, ..., t_k)$

**Ensure:** $\{0, -1\} \triangleright$ accept, reject signature

1. $c \leftarrow Enc(c')$
2. $a_1, ..., a_k \leftarrow GenA(seed_a)$
3. for $i = 1, ..., k$ do
4.   $w_i \leftarrow a_i z - t_i c \mod q$
5. end for
6. if $z \not\in R_q\{B - L_E\} \lor c' \neq H([w_1]_M, ..., [w_k]_M, m)$ then
7.   return $-1$
8. end if
9. return $0$

**2.3 Correctness of the scheme**

To establish the correctness of qTESLA we need to prove that the nonce input to the hash function $H$ at signing (line 9 of Algorithm 7) is the same as the nonce input to the hash function $H$ at verification (line 6 of Algorithm 8). That is, we need to prove that, for genuine signatures, $[ay mod^\pm q]_M = [az - tc mod^\pm q]_M = [a(y + sc) - (as + c) mod^\pm q]_M = [ay + asc - asc - ec mod^\pm q]_M = [ay - ec mod^\pm q]_M$. From the definition of $[.]_M$, this means proving that $((ay mod^\pm q - [ay mod^\pm q]_L)/2^d = (ay - ec mod^\pm q - [ay - ec mod^\pm q]_L)/2^d$, or simply $[ay mod^\pm q]_L = ec + [ay - ec mod^\pm q]_L$.

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

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

$$[\alpha]_L = \varepsilon + [\alpha - \varepsilon mod^\pm q]_L. \quad (1)$$

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

$$|[\varepsilon]_L + [\alpha - \varepsilon mod^\pm q]_L| = |\varepsilon|_L + [\alpha - \varepsilon mod^\pm q]_L = \varepsilon + [\alpha - \varepsilon mod^\pm q]_L. \quad (2)$$

Next we prove that

$$|[\varepsilon]_L + [\alpha - \varepsilon mod^\pm q]_L| = [\alpha]_L. \quad (3)$$

---

13
We note that since $|\varepsilon| \leq L_E < \lfloor q/2 \rfloor$ it holds that $[\varepsilon]_L = [\varepsilon \mod \pm q]_L$. It holds further that

$$\begin{align*}
([\varepsilon \mod \pm q]_L + [\alpha - \varepsilon \mod \pm q]_L)_L \\
= ([\varepsilon \mod \pm q] \mod \pm 2^d + (\alpha - \varepsilon \mod \pm q) \mod \pm 2^d) \mod \pm 2^d \\
\text{by the definition of } [.]_L \\
= ([\varepsilon \mod \pm q + (\alpha - \varepsilon \mod \pm q)) \mod \pm 2^d].
\end{align*}$$

Since $|\varepsilon| \leq L_E$ and $|\alpha - \varepsilon \mod \pm q| < \lfloor q/2 \rfloor - L_E$, it holds that $|\alpha - \varepsilon| + |\varepsilon| < (\lfloor q/2 \rfloor - L_E) + L_E = \lfloor q/2 \rfloor$. Hence, equation (6) is the same as

$$\begin{align*}
= ([\varepsilon + \alpha - \varepsilon \mod \pm q] \mod \pm 2^d = (\alpha \mod \pm q) \mod \pm 2^d = \alpha \mod \pm 2^d \\
= [\alpha]_L.
\end{align*}$$

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

Now define $\alpha := (ay)_i$ and $\varepsilon := (ec)_i$. From line 18 of Algorithm 7, we know that $\|\[ay - ec][\infty < 2^d - 1 - L_E$ and $\|ay - ec\|_\infty < [q/2] - L_E$ for a valid signature, and that Algorithm 4 (line 10) guarantees $\|ec\|_\infty \leq L_E$. Likewise, by definition it holds that $L_E < \lfloor q/2 \rfloor$; see Section 2.5. Finally, $v = ay$ is reduced mod $\pm q$ in line 7 of Algorithm 7 and, hence, $v$ is in the centered range $-\lfloor q/2 \rfloor < ay \leq \lfloor q/2 \rfloor$.

In conclusion, we get the desired condition for ring elements, $[ay]_L = ec + [ay - ec]_L$, which in turn means $[az - tc]_M = [ay]_M$ as argued above.

### 2.4 Realization of the required functions

#### 2.4.1 Hash and pseudorandom functions

In addition to the hash function $H$ and the pseudorandom functions $\text{PRF}_1$ and $\text{PRF}_2$, several functions that are used for the implementation of $\text{qTESLA}$ 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 $\text{XOF}(X, L, S)$, where $X$ is the input string, $L$ specifies the output length in bytes, and $S$ specifies an optional domain separator $^1$.

Next, we summarize how XOFs are instantiated using SHAKE [31] and cSHAKE [42] in the different functions requiring hashing or pseudorandom bit generation.

$^1$Specifically, the domain separator $S$ is used with cSHAKE, but ignored when SHAKE is used.
• **PRF$_1$**: the XOF is instantiated with SHAKE128 (resp. SHAKE256) for parameter sets qTESLA-I and qTESLA-p-I (resp. for other parameter sets); cf. Algorithm 9.

• **PRF$_2$**: the same as PRF$_1$.

• **GenA**: the XOF is instantiated with cSHAKE128 (cf. Algorithm 10).

• **GaussSampler**: the XOF is instantiated with cSHAKE128 (resp. cSHAKE256) for parameter sets qTESLA-I and qTESLA-p-I (resp. for other parameter sets); cf. Algorithm 11.

• **Enc**: the XOF is instantiated with cSHAKE128 (cf. Algorithm 13).

• **ySampler**: the XOF is instantiated with cSHAKE128 (resp. cSHAKE256) for parameter sets qTESLA-I and qTESLA-p-I (resp. for other parameter sets); cf. Algorithm 14.

• **Hash H**: the hash is instantiated with cSHAKE128 (resp. cSHAKE256) for parameter sets qTESLA-I and qTESLA-p-I (resp. for other parameter sets); cf. Algorithm 15.

In the cases of the functions GenA, Enc, and H, implementations of qTESLA need to follow strictly the XOF specifications based on SHAKE/cSHAKE given above in order to be fully compatible. However, for the rest of the cases (i.e., PRF$_1$, PRF$_2$, and ySampler) users can opt for a different cryptographic PRF.

### 2.4.2 Pseudorandom bit generation of seeds

qTESLA requires the generation of seeds during key generation; see line 2 of Algorithm 4. 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$_{a_i}$**, 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 $\kappa$ 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 $\kappa$ bits at line 1 of Algorithm 4, and then expanding this pre-seed through Algorithm 9. As explained in Section 2.4.1, in this case the XOF function is instantiated with SHAKE in our implementations.
Algorithm 10 Seed generation

Require: pre-seed \( \in \{0,1\}^\kappa \)
Ensure: \((\text{seed}_s, \text{seed}_{e_1}, \ldots, \text{seed}_{e_k}, \text{seed}_a)\), where each seed is \(\kappa\) bits long

1: \(\langle \text{seed}_s \rangle \parallel \langle \text{seed}_{e_1} \rangle \parallel \ldots \parallel \langle \text{seed}_{e_k} \rangle \parallel \langle \text{seed}_a \rangle \parallel \langle \text{seed}_y \rangle \leftarrow \text{XOF}(\text{pre-seed}, \kappa \cdot (k + 3)/8)\), where each \(\langle \text{seed} \rangle \in \{0,1\}^\kappa\)
2: return \((\text{seed}_s, \text{seed}_{e_1}, \ldots, \text{seed}_{e_k}, \text{seed}_a)\)

2.4.3 Generation of \(a_1, \ldots, a_k\)

In qTESLA, the polynomials \(a_1, \ldots, a_k\) are freshly generated per secret/public keypair using the seed \(\text{seed}_a\) during key generation; see line 3 of Algorithm 4. This seed is then stored as part of both the private and public keys so that the signing and verification operations can regenerate \(a_1, \ldots, a_k\).

The approach above permits to save bandwidth since we only need \(\kappa\) bits to store \(\text{seed}_a\) 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 \(a_1, \ldots, a_k\) per keypair makes the introduction of backdoors more difficult and reduces drastically the scope of all-for-the-price-of-one attacks [10,16].

The procedure depicted in Algorithm 10 to generate \(a_1, \ldots, a_k\) is as follows. The seed \(\text{seed}_a\) obtained from Algorithm 9 is expanded to \((\text{rate}_{\text{XOF}} \cdot b_{\text{GenA}})\) bytes using cSHAKE128, where \(\text{rate}_{\text{XOF}}\) is the SHAKE128 rate constant (i.e., the value 168 [31]) and \(b_{\text{GenA}}\) is a qTESLA parameter that represents the number of blocks requested in the first XOF call. The values of \(b_{\text{GenA}}\) for the different parameter sets were chosen as to allow the generation of slightly more bytes than are necessary to fill out all the coefficients of the polynomials (see Tables 2 and 3). Then, the algorithm proceeds to do rejection sampling over each 8\(b\)-bit string of the cSHAKE output, discarding every package that has a value greater than the modulus \(q\), where \(b\) is the number of bytes needed to represent \(q\). 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 \(\text{rate}_{\text{XOF}}\) bytes each time (lines 5–8). The particular form of the expression to evaluate if more cSHAKE128 calls are necessary at line 5 facilitates high-performance implementations that evaluate up to four coefficients at a time in the if-loop of lines 9–13. It should be noted that the first call to cSHAKE128 uses the value \(S = 0\) as domain separator. This value is incremented by one at each subsequent call.
Algorithm 10 Generation of public polynomials $a_i$, GenA

Require: $\text{seed}_{a} \in \{0,1\}^\kappa$. Set $b = \lceil(\log_2 q)/8 \rceil$ and the SHAKE128 rate constant $\text{rate}_{XOF} = 168$

Ensure: $a_i \in \mathbb{R}_q$ for $i = 1, \ldots, k$

1: $S \leftarrow 0$, $b' \leftarrow b_{\text{GenA}}$
2: $\langle c_0 \rangle \| \langle c_1 \rangle \| \ldots \| \langle c_T \rangle \leftarrow \text{cSHAKE128}(\text{seed}_{a}, \text{rate}_{XOF} \cdot b', S)$, where each $\langle c_i \rangle \in \{0,1\}^{8b}$
3: $i \leftarrow 0$, pos $\leftarrow 0$
4: while $i < k \cdot n$ do
5: \hspace{1em} if pos > $\lceil \text{rate}_{XOF} \cdot b' \rceil - 4$ then
6: \hspace{2em} $S \leftarrow S + 1$, pos $\leftarrow 0$, b$'$ $\leftarrow 1$
7: \hspace{2em} $\langle c_0 \rangle \| \langle c_1 \rangle \| \ldots \| \langle c_T \rangle \leftarrow \text{cSHAKE128}(\text{seed}_{a}, \text{rate}_{XOF} \cdot b', S)$, where each $\langle c_i \rangle \in \{0,1\}^{8b}$
8: \hspace{2em} end if
9: \hspace{1em} if $q > c_{\text{pos}} \mod 2^{\lceil \log_2 q \rceil}$ then
10: \hspace{2em} $a_{i \cdot [i/n] + 1:i-n:[i/n]} \leftarrow c_{\text{pos}} \mod 2^{\lceil \log_2 q \rceil}$, where a polynomial $a_x$ is interpreted as a vector of coefficients $(a_{x,0},a_{x,1},\ldots,a_{x,n-1})$
11: \hspace{2em} $i \leftarrow i + 1$
12: \hspace{1em} end if
13: \hspace{1em} pos $\leftarrow$ pos + 1
14: end while
15: return $(a_1, \ldots, a_k)$

The procedure to generate $a_1, \ldots, a_k$ produces polynomials with uniformly random coefficients. Thus, following a standard practice, qTESLA assumes that the resulting polynomials $a_1, \ldots, a_k$ from Algorithm 10 are in the NTT domain. This permits an important speedup of some of the polynomial arithmetic operations. We remark that, although this assumption does not affect the security of the scheme, it does affect the correctness. In other words, implementations of the scheme need to follow this specification in order to be compatible with the qTESLA design. In particular, polynomial multiplications with these polynomials need to consider that they are expressed in NTT domain. For example, in the accompanying implementations multiplications in line 11 of Algorithm 4, line 7 of Algorithm 7 and line 4 of Algorithm 8 are done by computing $a_i \cdot b = \text{NTT}^{-1}(a_i \circ \text{NTT}(b))$ for some polynomial $b \in \mathbb{R}_q$ (see Section 2.4.8 for details about the NTT computations).

2.4.4 Gaussian sampling

One of the advantages of qTESLA is that Gaussian sampling is only required during key generation to sample $e_1, \ldots, e_k$, and $s$ (see Alg. 4). Nevertheless, certain applications might still require an efficient and secure implementation of key generation and one that is, in particular, protected against timing and cache side-channel attacks. In that direction, we adopt the “constant-time” Gaussian sampler proposed in [16], which is an improvement upon the sampler proposed by Ducas et al. [27, Section 6]. We give the pseudocode of the proposed Gaussian sampler in Algorithms 11 and 12.
Algorithm 11 Sampling $D_{\sigma}$, GaussSampler

Require: seed $\text{rand} \in \{0, 1\}^n$, and nonce $S \in \mathbb{Z}_{>0}$. Set $\xi \in \mathbb{Z}$, the computer wordsize $w$, and the precomputed CDT table.

Ensure: $z \in \mathbb{Z}$ according to $D_{\sigma}$

1: $S \leftarrow S \cdot 2^8$
2: do
3: do
4: $y \leftarrow \text{XOF}(\text{rand}, \text{sizeof}(\xi), S)$
5: $S \leftarrow S + 1$
6: while $y < \xi - 1$
7: $r \leftarrow \text{XOF}(\text{rand}, w/8, S)$
8: $S \leftarrow S + 1$
9: $x \leftarrow 0$
10: while $r < \text{CDT}[x]$ do
11: $x \leftarrow x + 1$
12: end while
13: $z \leftarrow \xi \cdot x + y$
14: $r \leftarrow \text{XOF}(\text{rand}, w/8, S)$
15: $S \leftarrow S + 1$
16: while $(\text{BerSampler}(r, y(y + 2\xi x)) = 0)$
17: Generate a random bit $b$
18: if $z = 0 \land b = 0$ then
19: Restart at step 3
20: end if
21: Generate a random bit $b$
22: return $(-1)^b \cdot z$

In our implementations we target $w = 64$. The precomputed CDT table for this case is generated by the Magma script gaussSigma2Sample_table.magma found in the folder \Supporting_Documentation\Scripts_for_Gaussian_sampler.
Algorithm 12 Sampling $B_{\exp(-t/(2\sigma^2))}$ for $t \in [0, 2^w)$, BerSampler

Require: $r \in \{0, 1\}^w$ and $t \in [0, 2^w)$. Set the computer wordsize $w$, and the precomputed Bernoulli table $B$, which consists of $B_{\text{tables}}$ sub-tables with $B_{\text{rows}}$ rows each.

1: $c \leftarrow 2^{w-2}$
2: $s \leftarrow t$
3: for $i = 0, 1, \ldots, B_{\text{tables}} - 1$ do
4: \hspace{1em} $c \leftarrow c \cdot B_{i,s \mod B_{\text{rows}}}$, where an element at the $j$-th row of the $i$-th sub-table is represented by $B_{i,j}$
5: \hspace{1em} $s \leftarrow s / B_{\text{rows}}$
6: end for
7: if $r \mod (w - 1) \geq \lfloor c \rfloor$ then
8: \hspace{1em} return 0
9: else
10: \hspace{1em} return 1
11: end if

In our implementations we target $w = 64$. For our parameters, we use $B_{\text{tables}} = 3$ and $B_{\text{rows}} = 32$. The precomputed Bernoulli table $B$ for this case is generated by the Magma script `gaussBernoulliSample_table.magma` found in the folder `Supporting_Documentation\Scripts_for_Gaussian_sampler`.

The basic idea behind the Gaussian sampler by Ducas et al. [27, Algorithms 10–12] is to start from a distribution that approximates the desired Gaussian distribution. Namely, Ducas et al. use a stepwise uniform distribution where the steps have width $\xi$ and height distributed according to a $\xi$-scaled binary Gaussian that can be implemented efficiently.

The binary Gaussian sampler, which was introduced by Ducas et al., is a discrete Gaussian with specific variance $\sigma^2 = \frac{1}{\sqrt{2\ln 2}} \approx 0.849$.

We follow the original Gaussian sampler, which focuses only on the positive half of $D_{\sigma^2}$ denoted by $D_{\sigma^2}^+ = \{x \leftarrow D_{\sigma^2} : x \geq 0\}$. From there, a Gaussian distribution $D_{\sigma}$ with the desired quality is obtained by rejection sampling guided by Bernoulli distributions $B_{\rho}$ with parameter $\rho$ related to the standard deviation $\sigma$ of the desired Gaussian distribution. Ducas et al. implement those Bernoulli distributions by decomposing them into $\ell$ certain base distributions ($B_{\rho_0}, B_{\rho_1}, \ldots, B_{\rho_{\ell-1}}$) where the $\rho$ constants are precomputed to the desired accuracy, and then sampling from those base distributions to that accuracy. Even though this Bernoulli decomposition is reportedly quite efficient, its running time highly depends on the private bits. Besides that, each $B_{\rho_p}$ must be sampled to the same precision as the target distribution, which is why the total amount of entropy needed to obtain one Gaussian sample is much higher than theoretically necessary, roughly $O(\ell \lambda)$ bits rather than $O(\lambda)$ for security level $\lambda$.

Since qTESLA only needs a basic Gaussian sampler for key generation, it is possible to use
the much simpler construction explained in [16]. In particular, only one kind of Bernoulli distribution \( B_\rho \) is needed in our case, namely, a distribution \( B_{\exp(-t/2\sigma^2)} \) where \( t \) is an \( \ell \)-bit integer.\(^2\) Hence, we can simplify the sampler by Ducas et al. by just computing the exponential bias \( \rho = \exp(-t/2\sigma^2) \) using well-known exponentiation techniques, where \( \rho \) is an approximation of a real number in the interval \([0, 1]\) to the desired precision according to the security level \( \lambda \). Specifically, we need to perform one uniform sample \( y \leftarrow \mathbb{Z} \) in \( \{0, \ldots, \xi - 1\} \) (lines 3–6 of Algorithm 11), one binary Gaussian sample \( x \in \mathbb{Z} \) according to \( D_{\sigma^2}^+ \) (lines 7–12 of Algorithm 11) and, finally, one uniform sample to the desired precision based on the bias \( \rho \) with \( t = y(\xi x) \) (Algorithm 12 invoked from line 16 of Algorithm 11). At the end of this process we have the desired Gaussian sample with \( \sigma = \xi \sigma_2 = \frac{\xi}{\sqrt{2\ln 2}} \), and the total entropy consumption is \( O(\lambda) \) bits.

In the accompanying implementations, we implement an optimized version of Algorithm 11 that has been finely tuned for a computer wordsize \( w = 64 \). The precomputed tables that are used can be generated with the scripts provided in the folder \Supporting\Documentation\Scripts_for_Gaussian_sampler. In order to make the Gaussian sampler constant-time in our implementations, 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. These modifications are straightforward in the case of qTESLA, given the significantly simpler Gaussian sampler algorithm. As explained in Section 2.4.1, our implementations use cSHAKE as the XOF function.

### 2.4.5 Encoding function

In the signature generation we need to map the hash input \( ([v_1]_M, \ldots, [v_k]_M, m) \) to a polynomial \( c \in \mathbb{H}_{n,h} \subset \mathbb{R}_q \) (cf. line 9 and 10 of Algorithm 7). In order to obtain smaller signatures \((z, c') \in \{0, 1\}^n \times \mathbb{R}_q\), we break up this operation into \( \text{Enc}(H([v_1]_M, \ldots, [v_k]_M, m)) = \text{Enc}(c') = c \), where the encoding function \( \text{Enc} \) takes the output of the hash function \( H \) and maps it to a vector with entries in \( \{-1, 0, 1\} \) of length \( n \) and weight \( h \) (i.e., it has \( h \) entries that are either 1 or −1). The vector represents a polynomial of degree \( n - 1 \).

We implement \( \text{Enc} \) as in [1] which in turn follows an algorithm originally proposed in [30, Section 4.4]; see Algorithm 13. Basically, the algorithm maps the output of the hash function \( H \) to a vector \( c \) by using values generated uniformly at random to fix the positions and signs of the \( h \) nonzero entries. The outputs are the two list arrays \( \text{pos\_list} \) and \( \text{sign\_list} \), which contain the positions and signs of the nonzero entries of \( c \), respectively. Accordingly, the algorithm first requests \text{rateXOF} \) bytes from a XOF, and the output stream is interpreted as an array of 3-byte packets in little endian format. The lowest two bytes in every packet

\(^2\)The Gaussian sampler originally designed for the signature scheme BLISS has the additional complication that Bernoulli distributions with inverse hyperbolic cosine biases \( B_{1/cosh(t/f)} \) are required as well.
is used to calculate a position \( pos \) of a nonzero element in the \( n \)-dimension vector \( c \). The third byte is used to determine its sign as follows. If the \( (pos) \)-th element of \( c \) is “empty” (indicated by the value 0), then the sign entry is filled out with 1 (+) if the least significant bit of the third byte is 0 which is interpreted as a positive element 1 of \( c \). Otherwise, if the least significant bit of the third byte is 1 then the sign entry is set to \(-1 \) (–), which means that a bit 1 is interpreted as a negative coefficient \(-1 \) of \( c \). These procedure is executed until \( h \) nonzero entries are filled out. If the XOF output is exhausted before completing the task then additional calls are invoked, requesting \( rate_{XOF} \) bytes each time. It should be noted that the first call to the XOF uses the value \( S = 0 \) as domain separator. This value is incremented by one at each subsequent call. As explained in Section 2.4.1, \( qTESLA \) uses \( cSHAKE128 \) as the XOF function.

**Algorithm 13** Encoding function, \( Enc \)

**Require:** \( c' \in \{0, 1\}^\kappa \)

**Ensure:** lists \( pos \) \(_{\!list} \in \{0, ..., n - 1\}^h \) and \( sign \) \(_{\!list} \in \{-1, 1\}^h \) containing the positions and signs, resp., of the nonzero elements of \( c \in \mathbb{F}_{n,h}^1 \)

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

The sampling of the polynomial $y$ at line 4 of Algorithm 7 can be simply 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 simple 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 14 depicts the procedure used in the accompanying implementations based on a XOF. The value $S + 2^8$, for $S \in \mathbb{Z} > 0$, is used as domain separator or nonce. In order to provide proper domain separation, the first invocation to sample $y$ in Algorithm 7 is done with $S$ initialized at 1, and then each subsequent invocation increases $S$ by 1. This means that the successive calls to the sampler use nonces initialized to $2^8, 2 \cdot 2^8, 3 \cdot 2^8$, and so on, providing domain separation between the sampling of $y$ and other uses of the XOF in the signing algorithm. If additional calls to the XOF function are required as explained above (see lines 4–7), the nonce is increased by 1. As explained in Section 2.4.1, our implementations use cSHAKE as the XOF function.

Algorithm 14 receives as input a seed $\text{rand}$ which is the result of hashing seed $y$, a random value $r$ and the message $m$, as can be seen at line 3 of Algorithm 7.

Algorithm 14: Sampling $y$, $y_{\text{Sampler}}$

**Require:** seed $\text{rand} \in \{0, 1\}^\kappa$ and nonce $S \in \mathbb{Z}_{>0}$. Set $b = [(\log_2 B + 1)/8]$

**Ensure:** $y \in \mathcal{R}_{q,[B]}$

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

The hash \( H \) required by the signing and verification algorithms takes as inputs \( k \) polynomials \( v_1, \ldots, v_k \in \mathbb{R}_q \) and computes \([v_1]_M, \ldots, [v_k]_M\). The result is hashed together with a message \( m \) to a string \( c' \) that is \( \kappa \) bits long. The procedure is depicted in Algorithm 15. Let each polynomial \( v_i \) be interpreted as a vector of coefficients \((v_{i,0}, v_{i,1}, \ldots, v_{i,n-1})\). We first compute \([v_{i,j}]_L\) for each coefficient \( v_{i,j} \in (-q/2, q/2] \) following the details from Section 2.2: each coefficient is reduced modulo \( 2^d \) if it is greater than \( 2^d - 1 \). This guarantees a result in the range \((-2^d - 1, 2^d - 1]\), as required by the definition of \([\cdot]_L\). Next, we compute \([v_{i,j}]_M\) as \((v_{i,j} - [v_{i,j}]_L)/2^d\) and store it as a byte, which in the end makes up a full string of \( k \cdot n \) bytes. Finally, SHAKE is used to hash the resulting string concatenated with the message \( m \) to the \( \kappa \)-bit string \( c' \).

**Algorithm 15 Hash \( H \)**

**Require:** polynomials \( v_1, \ldots, v_k \in \mathbb{R}_q \), where \( v_{i,j} \in (-q/2, q/2] \), for \( i = 1, \ldots, k \) and \( j = 0, \ldots, n - 1 \), and a message \( m \) of length \( mlen \) 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: \( \text{val} \leftarrow v_{i,j} \mod 2^d \)
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: for \( i = 0,1,\ldots,mlen - 1 \) do
11: \( w_{k \cdot n + i} \leftarrow m_i \), where \( m \) is interpreted as the byte array \( \langle m_0 \rangle \parallel \langle m_1 \rangle \parallel \ldots \parallel \langle m_{mlen - 1} \rangle \)
12: end for
13: \( c' \leftarrow \text{SHAKE}(w, \kappa/8) \), where \( w \) is the byte array \( \langle w_0 \rangle \parallel \langle w_1 \rangle \parallel \ldots \parallel \langle w_{k \cdot n + mlen - 1} \rangle \)
14: return \( c' \in \{0,1\}^\kappa \)

2.4.8 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 2.4.3), we need to define polynomials in such a domain. Let \( \omega \) be a primitive \( n \)-th root of unity in \( \mathbb{Z}_q \), i.e., \( \omega^n \equiv 1 \pmod{q} \), 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

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

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

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

It then holds that $\text{NTT}^{-1}(\text{NTT}(a)) = a$ 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 $a \cdot b = \text{NTT}^{-1}(\text{NTT}(a) \circ \text{NTT}(b))$, where $\cdot$ is the polynomial multiplication in $\mathcal{R}_q$ and $\circ$ is the coefficient wise multiplication in $\mathbb{Z}_n^q$.

In the accompanying implementations, we adopt butterfly algorithms to compute the NTT that efficiently merge the powers of $\phi$ and $\phi^{-1}$ with the powers of $\omega$, and that at the same time avoid the need of a so-called bit-reversal operation which is required by some implementations [10, 51, 52]. 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 [54, Algorithms 1 and 2].

Sparse multiplication. In our implementations, standard polynomial multiplications are carried out using the NTT as explained above. However, qTESLA also requires specialized multiplications with the polynomial $c$ which by definition only contains $h$ nonzero coefficients (cf. lines 11 and 17 in Algorithm 7, and line 4 in Algorithm 8). These sparse multiplications can be realized efficiently with the specialized polynomial algorithm depicted in Algorithm 16.

2.5 System parameters and parameter selection

In this section, we describe qTESLA’s system parameters and our explicit choice of parameter sets.
**Algorithm 16** Sparse Polynomial Multiplication

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

**Ensure:** \( f = a \cdot c \in \mathcal{R}_q \)

1: Set all coefficients of \( f \) to 0
2: for \( i = 0, \ldots, h - 1 \) do
3: \( pos \leftarrow pos\_list_i \)
4: for \( j = 0, \ldots, pos - 1 \) do
5: \( f_j \leftarrow f_j - sign\_list_i \cdot a_j + pos \)
6: end for
7: for \( j = pos, \ldots, n - 1 \) do
8: \( f_j \leftarrow f_j + sign\_list_i \cdot a_j - pos \)
9: end for
10: end for
11: return \( f \)

**Parameter sets.** Herein, we propose five 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 1 together with their targeted security category, as defined by NIST in [56].

<table>
<thead>
<tr>
<th>Parameter sets</th>
<th>Heuristic</th>
<th>Provably-secure</th>
<th>Security category</th>
</tr>
</thead>
<tbody>
<tr>
<td>qTESLA-I</td>
<td>qTESLA-p-I</td>
<td>NIST’s category 1</td>
<td></td>
</tr>
<tr>
<td>qTESLA-III-speed, qTESLA-III-size</td>
<td>qTESLA-p-III</td>
<td>NIST’s category 3</td>
<td></td>
</tr>
</tbody>
</table>

The proposed provably-secure parameter sets, namely qTESLA-p-I and qTESLA-p-III, were chosen according to the security reduction provided in Theorem 6, 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 three parameter sets, namely qTESLA-I, qTESLA-III-speed, and qTESLA-III-size, which are chosen heuristically. In this case, we assume that the security level of an instantiation of the scheme by a certain parameter set directly corresponds to the hardness level of the instance of the underlying lattice problem that corresponds to those parameters, without taking into account the security reduction. The assumption is that Theorem 6 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).

Table 2: Description and bounds of all the system parameters.

<table>
<thead>
<tr>
<th>Param.</th>
<th>Description</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda )</td>
<td>security parameter</td>
<td>-</td>
</tr>
<tr>
<td>( q_h, q_s )</td>
<td>number of hash and sign queries</td>
<td>power-of-two</td>
</tr>
<tr>
<td>( n )</td>
<td>dimension (( n-1 ) is the poly. degree)</td>
<td>( \sigma = \frac{\xi}{\sqrt{2\ln 2}} )</td>
</tr>
<tr>
<td>( \sigma, \xi )</td>
<td>standard deviation of centered discrete Gaussian distribution</td>
<td>-</td>
</tr>
<tr>
<td>( k )</td>
<td>#R-LWE samples</td>
<td>( q \equiv 1 \mod 2n, q &gt; 4B )</td>
</tr>
<tr>
<td>( q )</td>
<td>modulus</td>
<td>For provably secure parameters: ( q^{nk} \geq</td>
</tr>
<tr>
<td>( h )</td>
<td># of nonzero entries of output elements of ( \text{Enc} )</td>
<td>( 2^k \binom{n}{h} \geq 2^{2\lambda} )</td>
</tr>
<tr>
<td>( \kappa )</td>
<td>output length of hash function ( H ) and input length of ( \text{GenA} ), PRF₁, PRF₂, ( \text{Enc} ) and ( \text{ySampler} )</td>
<td>( \kappa \geq \lambda )</td>
</tr>
<tr>
<td>( L_E, \eta_E )</td>
<td>bound in ( \text{checkE} )</td>
<td>( \eta_E \cdot h \cdot \sigma )</td>
</tr>
<tr>
<td>( L_S, \eta_S )</td>
<td>bound in ( \text{checkS} )</td>
<td>( \eta_S \cdot h \cdot \sigma )</td>
</tr>
<tr>
<td>( B )</td>
<td>determines interval the randomness is chosen from during signing</td>
<td>( B \geq \frac{k \sqrt{2L_S} - 2L_S - 1}{2(1 - k \sqrt{2L_S})} ), near a power-of-two</td>
</tr>
<tr>
<td>( d )</td>
<td>number of rounded bits</td>
<td>( \left( 1 - \frac{2L_S + 1}{2d \sqrt{\sigma}} \right)^{1/n} \geq 0.3, \ d &gt; \log_2(B) )</td>
</tr>
<tr>
<td>( b_{\text{GenA}} )</td>
<td>number of blocks requested to SHAKE128 for ( \text{GenA} )</td>
<td>( b_{\text{GenA}} \in \mathbb{Z}_{&gt;0} )</td>
</tr>
<tr>
<td>(</td>
<td>\Delta_S</td>
<td>)</td>
</tr>
<tr>
<td>(</td>
<td>\Delta_L</td>
<td>)</td>
</tr>
<tr>
<td>(</td>
<td>\Delta_H</td>
<td>)</td>
</tr>
<tr>
<td>( \delta_z )</td>
<td>acceptance probability of ( z ) in line 12 during signing</td>
<td>experimentally</td>
</tr>
<tr>
<td>( \delta_w )</td>
<td>acceptance probability of ( w ) in line 18 during signing</td>
<td>experimentally</td>
</tr>
<tr>
<td>( \delta_k_{\text{gen}} )</td>
<td>acceptance probability of key pairs during key generation</td>
<td>experimentally</td>
</tr>
<tr>
<td>( \text{sig size} )</td>
<td>theoretical size of signature [byte]</td>
<td>( \kappa + n(\log_2(B - L_S)) + 1) )</td>
</tr>
<tr>
<td>( \text{pk size} )</td>
<td>theoretical size of public key [byte]</td>
<td>( kn(\log_2(q)) + \kappa )</td>
</tr>
<tr>
<td>( \text{sk size} )</td>
<td>theoretical size of secret key [byte]</td>
<td>( n(k + 1)(\log_2(t \cdot \sigma + 1)) + 2\kappa )</td>
</tr>
</tbody>
</table>

with \( t = 11.6 \) or 15.
Table 3: Parameters for each of the proposed heuristic and provably-secure parameter sets with \(q_h = 2^{128}\) and \(q_s = 2^{64}\); we choose \(M = 0.3\).

<table>
<thead>
<tr>
<th>Param.</th>
<th>qTESLA-I</th>
<th>qTESLA-III-speed</th>
<th>qTESLA-III-size</th>
<th>qTESLA-p-I</th>
<th>qTESLA-p-III</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\lambda)</td>
<td>95</td>
<td>160</td>
<td>160</td>
<td>95</td>
<td>160</td>
</tr>
<tr>
<td>(\kappa)</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>(n)</td>
<td>512</td>
<td>1 024</td>
<td>1 024</td>
<td>1 024</td>
<td>2 048</td>
</tr>
<tr>
<td>(\sigma, \xi)</td>
<td>23.78, 27.9988</td>
<td>10.2, 12</td>
<td>8.49, 9.9962</td>
<td>8.5, 10</td>
<td>8.5, 10</td>
</tr>
<tr>
<td>(k)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>(q)</td>
<td>(\approx 2^{22})</td>
<td>(\approx 2^{21})</td>
<td>(\approx 2^{22})</td>
<td>(\approx 2^{29})</td>
<td>(\approx 2^{30})</td>
</tr>
<tr>
<td>(h)</td>
<td>30</td>
<td>48</td>
<td>48</td>
<td>25</td>
<td>40</td>
</tr>
<tr>
<td>(L_E, \eta_E)</td>
<td>1 586, 2.223</td>
<td>1 147, 2.34</td>
<td>910, 2.23</td>
<td>554, 2.61</td>
<td>901, 2.65</td>
</tr>
<tr>
<td>(L_S, \eta_S)</td>
<td>1 586, 2.223</td>
<td>1 233, 2.52</td>
<td>910, 2.23</td>
<td>554, 2.61</td>
<td>901, 2.65</td>
</tr>
<tr>
<td>(B)</td>
<td>(2^{20} - 1)</td>
<td>(2^{21} - 1)</td>
<td>(2^{20} - 1)</td>
<td>(2^{21} - 1)</td>
<td>(2^{23} - 1)</td>
</tr>
<tr>
<td>(d)</td>
<td>21</td>
<td>22</td>
<td>21</td>
<td>22</td>
<td>24</td>
</tr>
<tr>
<td>(b_{GenA})</td>
<td>19</td>
<td>38</td>
<td>38</td>
<td>108</td>
<td>180</td>
</tr>
<tr>
<td>(</td>
<td>\Delta H</td>
<td>)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(</td>
<td>\Delta S</td>
<td>)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(</td>
<td>\Delta L</td>
<td>)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(\delta_w)</td>
<td>0.31</td>
<td>0.38</td>
<td>0.25</td>
<td>0.33</td>
<td>0.34</td>
</tr>
<tr>
<td>(\delta_z)</td>
<td>0.44</td>
<td>0.56</td>
<td>0.37</td>
<td>0.78</td>
<td>0.81</td>
</tr>
<tr>
<td>(\delta_{sign})</td>
<td>0.14</td>
<td>0.21</td>
<td>0.09</td>
<td>0.26</td>
<td>0.28</td>
</tr>
<tr>
<td>(\delta_{keygen})</td>
<td>0.45</td>
<td>0.60</td>
<td>0.39</td>
<td>0.59</td>
<td>0.44</td>
</tr>
<tr>
<td>sig size</td>
<td>1 376</td>
<td>2 848</td>
<td>2 720</td>
<td>2 848</td>
<td>6 176</td>
</tr>
<tr>
<td>pk size</td>
<td>1 504</td>
<td>3 104</td>
<td>2 976</td>
<td>14 880</td>
<td>39 712</td>
</tr>
<tr>
<td>sk size</td>
<td>1 216</td>
<td>2 112</td>
<td>2 112</td>
<td>4 576</td>
<td>12 320</td>
</tr>
<tr>
<td>classical bit hardness</td>
<td>104</td>
<td>178</td>
<td>188</td>
<td>132</td>
<td>247</td>
</tr>
<tr>
<td>quantum bit hardness</td>
<td>97</td>
<td>164</td>
<td>169</td>
<td>123</td>
<td>270</td>
</tr>
</tbody>
</table>

**System parameters.** Table 2 summarizes all the system parameters, including all relevant bounds. Concrete parameter values for each of the proposed parameter sets are compiled in Table 3. Let \(\lambda\) be the security parameter, i.e., the targeted bit security of a given instantiation. Let \(n \in \mathbb{Z}_{>0}\) be the dimension, such that \(n - 1\) is the polynomial degree. In the targeted R-LWE setting with \(R_q = \mathbb{Z}_q[x]/(x^n + 1)\), the polynomial degree is set to a power of two, i.e., \(n = 2^l\) for \(l \in \mathbb{N}\). Let \(\sigma\) be the standard deviation of the centered discrete Gaussian distribution that is used to sample the coefficients of the secret and error polynomials. To use the fast Gaussian sampler described in Section 2.4.4, we choose \(\sigma = \frac{\xi}{\sqrt{2\ln 2}}\) for some \(\xi \in \mathbb{Z}_{>0}\). Let \(k \in \mathbb{Z}_{>0}\) be the number of ring learning with errors samples. For our heuristic parameter sets qTESLA-I, qTESLA-III-speed, and qTESLA-III-size, we fix...
\[ k = 1, \text{ whereas for our provably-secure parameter sets } q_{\text{TESLA-p-I}} \text{ and } q_{\text{TESLA-p-III}}, \text{ we choose } k > 1. \] The latter choice allows us to reduce the size of the modulus \( q \), as explained later. Depending on the specific function, the parameter \( \kappa \) defines the outputs or inputs of the hash function and pseudorandom functions described in Section 2.4.1. The parameter \( h \) defines the number of nonzero elements in the output of the encoding function described in Section 2.4.5.

The values \( L_E \) and \( L_S \) are used to bound the coefficients in the error and secret polynomials in the evaluation functions checkE and checkS, 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. Both bounds, \( L_E \) and \( L_S \), impact the rejection probability during the signature generation as follows. If one increases the values of \( L_E \) and \( L_S \), the acceptance probability during key generation increases (lines 6 and 10 in Algorithm 4), while the acceptance probability during signature generation decreases (lines 12 and 18 in Algorithm 7). We determine the best trade-off between these two acceptance probabilities experimentally. We start choosing \( L_E = \eta_E \cdot h \cdot \sigma \) (resp., \( L_S = \eta_S \cdot h \cdot \sigma \)) with \( \eta_E = \eta_S = 2.8 \) and try different values for \( \eta_E, \eta_S \in [2.0, 3.0] \). Let \( M = 0.3 \) be a value of our choosing that determines (together with \( L_S \) and \( B \)) the acceptance probability of the rejection sampling in line 12 of Algorithm 7. The parameter \( B \) defines the interval of the random polynomial \( y \) (cf. line 4 of Algorithm 7) and it is determined by \( M \) and the parameter \( L_S \) as follows:

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

We select the rounding value \( d \) to be larger than \( \log_2(B) \) and such that the acceptance probability of the check \( \| [w]_L \|_\infty \geq 2^{d-1} - L_E \) in line 18 of Algorithm 7 is upper bounded by 0.7 when using the sage script to choose parameters. Changing the value \( L_E \) as described above, impacts the rejection probability of \( w \) as well. Experimentally, we determine the acceptance probability \( \delta_z \) of \( z \) and \( \delta_w \) of \( w \) during signing and the acceptance probability of key pairs \( \delta_{\text{keygen}} \). The results are summarized in Table 3.

Finally, \( b_{\text{GenA}} \in \mathbb{Z}_{>0} \) represents the number of blocks requested to cSHAKE128 in a first call during the generation of the public polynomials \( a_1, \ldots, a_k \) (cf. Algorithm 10). The values of \( b_{\text{GenA}} \) for the different parameter sets were chosen as to allow the generation of (slightly) more bytes than are necessary to fill out all the coefficients of the polynomials \( a_1, \ldots, a_k \).

The modulus \( q \). The parameter \( q \) is the modulus in the R-LWE instance. Depending on whether parameters are chosen heuristically or according to our security reduction in Theorem 6, \( q \) is chosen to fulfill several bounds and assumptions that are motivated by the security reduction or efficient implementation requirements. To be able to use fast
polynomial multiplication we choose \( q \) to be a prime integer such that \( q \mod 2n = 1 \).

To choose parameter sets according to the security reduction, it is first convenient to simplify our security statement. To this end we ensure that \( q^{nk} \geq |\Delta S| \cdot |\Delta L| \cdot |\Delta H| \) with the following definition of sets: \( S \) is the set of polynomials \( z \in \mathcal{R}_{q[B-\mathcal{U}]} \) and \( \Delta S = \{ z - z' : z, z' \in S \} \), \( H \) is the set of polynomials \( c \in \mathcal{R}_{q[1]} \) with exactly \( h \) nonzero coefficients and \( \Delta H = \{ c - c' : c, c' \in H \} \), and \( \Delta L = \{ x - x' : x, x' \in \mathcal{R} \text{ and } [x]_M = [x']_M \} \). Then, the following equation (cf. Theorem 6) has to hold:

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

Following the NIST’s call for proposals [56, 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** We are now in position to determine the key and signature sizes. The theoretical bit length of the signatures and public keys are given by \( \kappa + n \cdot ([\log_2(B - L_S)] + 1) \) bits and \( k \cdot n \cdot ([\log_2(q)]) + \kappa \) bits, respectively. To determine the size of the secret key we note that for \( t > 0 \) it holds that \( Pr_{x \leftarrow \mathcal{Z}}[|x| > t\sigma] \leq 2e^{-t^2/2} \).

For example, for \( t = 11.6 \) and \( t = 15 \), the probability \( Pr_{x \leftarrow \mathcal{Z}}[|x| > t\sigma] \) is less or equal to \( 2^{-95} \) and \( 2^{-160} \), respectively. Therefore, the theoretical size of the secret key is given by \( n(k + 1)([\log_2(t \cdot \sigma + 1)]) + 2\kappa \) bits with \( t = 11.6 \) for qTESLA-I and qTESLA-p-I, and with \( t = 15 \) for qTESLA-III-speed, qTESLA-III-size and qTESLA-p-III. Table 3 details the key and signature sizes according to these theoretical estimates.

### 3 Performance analysis

#### 3.1 Associated implementations

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

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-speed and qTESLA-III-size 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 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.

### 3.2 Performance on x64 Intel

To evaluate the performance of the provided implementations, we ran our benchmarking suite on two machines powered by: (i) a 3.40 GHz Intel Core i7-6700 (Skylake) processor and (ii) a 3.40 GHz 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`.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>keygen</th>
<th>sign</th>
<th>verify</th>
<th>total (sign + verify)</th>
</tr>
</thead>
<tbody>
<tr>
<td>qTESLA-I</td>
<td>1,585.0</td>
<td>470.1</td>
<td>97.7</td>
<td>567.8</td>
</tr>
<tr>
<td></td>
<td>(1,744.2)</td>
<td>(630.4)</td>
<td>(98.1)</td>
<td>(728.5)</td>
</tr>
<tr>
<td>qTESLA-III-speed</td>
<td>3,578.8</td>
<td>665.4</td>
<td>200.7</td>
<td>866.1</td>
</tr>
<tr>
<td></td>
<td>(3,884.8)</td>
<td>(860.1)</td>
<td>(201.0)</td>
<td>(1,061.1)</td>
</tr>
<tr>
<td>qTESLA-III-size</td>
<td>6,043.8</td>
<td>1,246.9</td>
<td>201.8</td>
<td>1,448.7</td>
</tr>
<tr>
<td></td>
<td>(6,282.1)</td>
<td>(1,752.3)</td>
<td>(202.3)</td>
<td>(1,954.6)</td>
</tr>
<tr>
<td>qTESLA-p-I</td>
<td>6,416.3</td>
<td>1,247.2</td>
<td>507.2</td>
<td>1,754.4</td>
</tr>
<tr>
<td></td>
<td>(6,624.7)</td>
<td>(1,589.6)</td>
<td>(507.8)</td>
<td>(2,097.4)</td>
</tr>
<tr>
<td>qTESLA-p-III</td>
<td>30,682.2</td>
<td>5,045.1</td>
<td>2,553.5</td>
<td>7,598.6</td>
</tr>
<tr>
<td></td>
<td>(32,466.4)</td>
<td>(6,146.7)</td>
<td>(2,554.0)</td>
<td>(8,700.7)</td>
</tr>
</tbody>
</table>

Table 4: Performance (in thousands of cycles) of the reference implementation of qTESLA on a 3.40 GHz 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.

The results in Table 4 showcase the high performance of heuristic qTESLA: the (median) time of signing and verification on the Intel Skylake platform is of approximately 167.0, 254.7 and 426.1 microseconds for qTESLA-I, qTESLA-III-speed and qTESLA-III-size, respectively. Likewise, provably-secure qTESLA computes the same operations in approximately 0.52 and 2.24 milliseconds with qTESLA-p-I and qTESLA-p-III, respectively.
This demonstrates that the speed of provably-secure qTESLA, although slower, is still practical for many applications. Similar results were observed in our tests on the Intel Haswell platform (see Table 5).

We highlight that these results are only obtained with generic C implementations. Future work includes the development of optimized implementations exploiting assembly and vector instructions to get further performance improvements.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>keygen</th>
<th>sign</th>
<th>verify</th>
<th>total (sign + verify)</th>
</tr>
</thead>
<tbody>
<tr>
<td>qTESLA-I</td>
<td>1,664.8</td>
<td>518.9</td>
<td>106.4</td>
<td>625.3</td>
</tr>
<tr>
<td></td>
<td>(1,869.1)</td>
<td>(715.4)</td>
<td>(110.0)</td>
<td>(825.4)</td>
</tr>
<tr>
<td>qTESLA-III-speed</td>
<td>3,750.7</td>
<td>810.4</td>
<td>240.4</td>
<td>1,050.8</td>
</tr>
<tr>
<td></td>
<td>(4,072.1)</td>
<td>(1,074.5)</td>
<td>(241.3)</td>
<td>(1,315.8)</td>
</tr>
<tr>
<td>qTESLA-III-size</td>
<td>6,155.2</td>
<td>1,411.7</td>
<td>217.8</td>
<td>1,629.5</td>
</tr>
<tr>
<td></td>
<td>(6,459.8)</td>
<td>(1,957.6)</td>
<td>(219.5)</td>
<td>(2,177.1)</td>
</tr>
<tr>
<td>qTESLA-p-I</td>
<td>6,702.3</td>
<td>1,356.9</td>
<td>548.4</td>
<td>1,905.3</td>
</tr>
<tr>
<td></td>
<td>(6,880.2)</td>
<td>(1,736.5)</td>
<td>(549.4)</td>
<td>(2,285.9)</td>
</tr>
<tr>
<td>qTESLA-p-III</td>
<td>31,587.5</td>
<td>5,461.5</td>
<td>2,710.3</td>
<td>8,171.8</td>
</tr>
<tr>
<td></td>
<td>(33,778.8)</td>
<td>(6,637.7)</td>
<td>(2,722.9)</td>
<td>(9,360.6)</td>
</tr>
</tbody>
</table>

Table 5: Performance (in thousands of cycles) of the reference implementation of qTESLA on a 3.40 GHz 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.

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 (slen) and signature (sm) values for all the proposed parameter sets.

The KAT files can be found in the media folder:

- qTESLA-I: \KAT\PQCsignKAT_qTesla-I.rsp,
- qTESLA-III-speed: \KAT\PQCsignKAT_qTesla-III-speed.rsp,
- qTESLA-III-size: \KAT\PQCsignKAT_qTesla-III-size.rsp,
- qTESLA-p-I: \KAT\PQCsignKAT_qTesla-p-I.rsp, and
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 2 (Ring short integer solution problem R-SIS). Let \( a_1, ..., a_k \leftarrow \mathcal{R}_q \). The ring short integer solution problem \( R-\text{SIS}_{n,k,q,\beta} \) 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 \((a_1, ..., a_k, 1) \cdot (u_1, ..., u_{k+1})^T = a_1 u_1 + ... + a_k u_k + u_{k+1} = 0 \mod q \) and \( \|u_1\|, ..., \|u_{k+1}\| \leq \beta \).

Definition 3 (Learning with Errors Distribution). Let \( n, q > 0 \) be integers, \( s \in \mathcal{R} \), and \( \chi \) be a distribution over \( \mathcal{R} \). We define by \( \mathcal{D}_{s,\chi} \) the LWE distribution which outputs \((a, (a, s) + e) \in \mathcal{R}_q \times \mathcal{R}_q \), where \( a \leftarrow \mathcal{R}_q \) and \( e \leftarrow \chi \).

Definition 4 (Decisional Ring Learning with Errors Problem R-LWE). Let \( n, q > 0 \) be integers and \( \chi \) 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 \( R-\text{LWE}_{n,k,q,\chi} \) is to distinguish whether \((a_i, t_i) \leftarrow U(\mathcal{R}_q \times \mathcal{R}_q) \) or \((a_i, t_i) \leftarrow \mathcal{D}_{s,\chi} \) for all \( i \).

5.1 Provable security in the (quantum) random oracle model

The security of qTESLA instantiated with the provably-secure parameters sets (see Section 2.5) 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. Note that 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 (cf. Theorem 5), which follows closely the approach proposed by Bai and Galbraith [14], gives a non-tight reduction from R-LWE and R-SIS to the existentially unforgeability under chosen-message attack (EUF-CMA) of qTESLA in the random oracle model.
Theorem 5. Let \( 2^n : \binom{n}{k} \geq 2^{\lambda} \), \( (2R + 1)^{k+1} \geq q^n 2^k \), and \( q > 4B \). If there exists an adversary \( A \) that forges a signature of the signature scheme \( \text{qTESLA} \) described in Section 2.2 in time \( t_\Sigma \) and with success probability \( \epsilon_\Sigma \), then there exists a reduction \( R \) that solves either

- the \( \text{R-LWE}_{n,k,q,\sigma} \) problem in time \( t_{\text{LWE}} \approx t_\Sigma \) with \( \epsilon_{\text{LWE}} \geq \epsilon_\Sigma/2 \), or
- the \( \text{R-SIS}_{n,k,q,\beta} \) problem with \( \beta = \max\{k 2^{d-1}, 2(B - L_E)\} + 2h R \) in time \( t_{\text{SIS}} \approx 2t_\Sigma \) with \( \epsilon_{\text{SIS}} \geq \frac{1}{2} \left( \epsilon_\Sigma - \frac{1}{2^\lambda} \right) + \epsilon_\Sigma/2 \) with our choice of parameters.

The second security reduction (cf. Theorem 6) gives a tight reduction in the quantum random oracle model from \( \text{R-LWE} \) to \( \text{EUF-CMA of qTESLA} \). 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 6 when discussing the security of the scheme. We emphasize that Theorem 6 gives a reduction from the decisional \( \text{R-LWE} \) problem only, while in Theorem 5 the decisional \( \text{R-SIS} \) problem is additionally used. Currently, Theorem 6 holds assuming a conjecture, as explained below.

Theorem 6. Let the parameters be as in Table 2. Furthermore, assume that Conjecture 7 holds. If there exists an adversary \( A \) that forges a signature of the signature scheme \( \text{qTESLA} \) described in Section 2.2 in time \( t_\Sigma \) and with success probability \( \epsilon_\Sigma \), then there exists a reduction \( R \) that solves the \( \text{R-LWE}_{n,k,q,\sigma} \) problem in time \( t_{\text{LWE}} \approx t_\Sigma \) with \( \epsilon_{\text{LWE}} \leq \frac{2^{3\lambda+nkd}}{q^{n\kappa}} + \frac{2q_0}{2^\lambda} + \epsilon_\Sigma/2 \) with parameters as in Table 2.

The proof follows the approach proposed in [9] except for the computation of the two probabilities \( \text{coll}(a,e) \) and \( \text{nwr}(a,e) \) that we explain in the following. For simplicity we assume that the randomness is sampled uniformly random in \( \mathcal{R}_{q,[B]} \) as in Algorithm 2. We define \( \Delta L \) to be the set \( \{ x - x' : x, x' \in \mathcal{R} \text{ and } [x]_M = [x']_M \} \). Furthermore, we call a polynomial \( w \) well-rounded if \( w \) is in \( \mathcal{R}_{q,[[q/2] - L_E]} \) and \( [w]_L \in \mathcal{R}_{q,[[2^{d-1} - L_E]]} \). 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 \( \overrightarrow{e} = (e_1, ..., e_k) \):

\[
\text{nwr}(\overrightarrow{a}, \overrightarrow{e}) \overset{\text{def}}{=} \Pr_{(y,c) \in \mathcal{Y} \times \mathcal{E}}[a_i y - e_i c \text{ not well-rounded for at least one } i \in \{1, ..., k\}] \tag{7}
\]

\[
\text{coll}(\overrightarrow{a}, \overrightarrow{e}) \overset{\text{def}}{=} \max_{(w_1, ..., w_k) \in \mathcal{W}^k} \left\{ \Pr_{(y,c) \in \mathcal{Y} \times \mathcal{E}}[[a_1 y - e_1 c]_M = w_1, ..., [a_k y - e_k c]_M = w_k] \right\} \tag{8}
\]

Informally speaking \( \text{nwr}(\overrightarrow{a}, \overrightarrow{e}) \) refers to the probability over random \((y, c)\) that \( a_i y - e_i c \) is not well-rounded for some \( i \). This quantity varies as a function of \( a_1, ..., a_k, e_1, ..., e_k \). In contrast to [9], we cannot upper bound this in general in the ring setting. Hence, we first assume that \( \text{nwr}(\overrightarrow{a}, \overrightarrow{e}) < 3/4 \) and afterwards check experimentally that this holds true. As our acceptance probability of \( w_i \) in line 18 of Algorithm 7 (signature generation) is at least 1/4 for all parameter sets (cf. \( \delta_w \) in Table 2), the bound \( \text{nwr}(\overrightarrow{a}, \overrightarrow{e}) < 3/4 \) holds.
Secondly, we need to bound the probability \( \text{coll}(\vec{a}, \vec{e}) \). In \cite[Lemma 4]{9} the corresponding probability \( \text{coll}(A, E) \) for standard lattices is upper bounded. Unfortunately, we were not able to transfer the proof to the ring setting for the following reason. In the proof of \cite[Lemma 4]{9}, it is used that if the randomness \( y \) is not equal to 0 the vector \( Ay \) is uniformly random distributed over \( \mathbb{Z}_q \) and hence also \( Ay - Ec \) is uniformly random distributed over \( \mathbb{Z}_q \). This does not necessarily hold if the polynomial \( y \) is chosen uniformly in \( \mathcal{R}_{q,(B)} \). Moreover, in Equation (99) in \cite{9}, \( \psi \) denotes the probability that a random vector \( x \in \mathbb{Z}_q^m \) is in \( \Delta L \):

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

The quantity \( \psi \) 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 \( \text{qTESLA} \) over ideals. Instead, we need to conjecture the following.

**Conjecture 7.** Let \( I \) be a nonzero ideal in \( \mathcal{R}_q \) and let \( r \in \mathcal{R}_q \) be a fixed choice of ring elements. Then, it holds that the probability that \( x + r \in \Delta L \) for a uniformly distributed element \( x \leftarrow I \) is negligibly small.

The intuition behind our conjecture is as follows. Let \( \psi_I \) denote the probability that a random element from the ideal \( I \) lands in \( \Delta L \). We know that \( \psi_I \) is small when the ideal \( I = \mathcal{R}_q \), i.e., a negligibly small fraction of elements from \( \mathcal{R}_q \) are in \( \Delta L \). Furthermore, the set \( \Delta L \) appears to have no relationship with the ideal structure of the ring, so it seems reasonable to view each ideal as a “random” subset of \( \mathcal{R}_q \) in the following sense: no larger or smaller portion of elements in the ideal \( I \) is in \( \Delta L \) than that portion of elements of \( \mathcal{R}_q \) that is in \( \Delta L \).

Hence, the corresponding statement described above and needed in \cite[Lemma 4]{9} translates for \( \text{qTESLA} \) to the following. If \( y \neq 0 \) then \( a_i y \) is a uniformly random element of some nonzero ideal \( I \) for all \( i \). The polynomial \( c \) is fixed and the polynomials \( e_1, \ldots, e_k \) are independent of the polynomials \( a_1, \ldots, a_k \), and \( y \). Hence, by our conjecture (with \( x = a_i y \) and \( r = e_i c \)) it holds that the probability of Equation (107) in \cite{9} is negligibly small. Thus, assuming that our conjecture holds true, \cite[Lemma 4]{9} and hence the security reduction in \cite{9} holds for \( \text{qTESLA} \) as well.

### 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 2.5. 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.
5.2.1 Correspondence between security and hardness

The security reduction given by Theorem 6, 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 \( \epsilon_{\Sigma} \) and \( t_{\Sigma} \) denote the success probability and the runtime (resp.) of a forger against our signature scheme, and let \( \epsilon_{LWE} \) and \( t_{LWE} \) denote analogous quantities for the reduction presented in the proof of Theorem 6. We say that R-LWE is \( \eta \)-bit hard if \( t_{LWE}/\epsilon_{LWE} \geq 2^\eta \); and we say that the signature scheme is \( \lambda \)-bit secure if \( t_{\Sigma}/\epsilon_{\Sigma} \geq 2^\lambda \).

For our provably-secure parameter sets \( q_{\text{TESLA-}p-I} \) and \( q_{\text{TESLA-}p-III} \), 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. Hence, the security reduction provably guarantees that our scheme instantiated with the two provably-secure parameter sets has the selected security level as long as the corresponding R-LWE instance is intractable.

For our heuristic parameter sets \( q_{\text{TESLA-I}}, q_{\text{TESLA-}III-\text{speed}}, \) and \( q_{\text{TESLA-}III-\text{size}} \), we assume that the bit security of our scheme instantiated with these three parameter sets is theoretically the same as the bit hardness of the corresponding R-LWE instance. So far no attack that exploits this heuristic is known. However, this is a somewhat less trustworthy approach that trades provable security assurance for performance.

Remark 8. In practical instantiations of \( q_{\text{TESLA}} \), the bit security does not exactly match the bit hardness of R-LWE (see Table 3). 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 \( \text{Enc} \). First, in all our parameter sets, heuristic and provably-secure, the key space is reduced by the rejection of polynomials \( s, e_1, \ldots, e_k \) with large coefficients via \( \text{checkE} \) and \( \text{checkS} \). We compensate this security loss by choosing an R-LWE instance of larger bit hardness. Moreover, we instantiate the encoding function \( \text{Enc} \) such that it is \( \lambda \)-bit secure; cf. Section 2.5.

5.2.2 Estimation of the R-LWE hardness

Since the introduction of the learning with errors problem over rings [49], 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 [22,25,33,35]. 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 \( q_{\text{TESLA}} \) 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 \( \alpha = \sqrt{2\pi \sigma q} \), 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 [40], the dual lattice attacks recently published in [3], the enumeration approach by Linder and Peikert [46], the primal attack described in [6, 15], the Arora-Ge algorithm [11] using Gröbner bases [4], and the latest analysis to compute the block sizes used in the lattice basis reduction BKZ recently published by Albrecht et al. [2]. Moreover, quantum speed-ups for the sieving algorithm used in BKZ [44, 45] are also considered.

We note that another recent quantum attack, called quantum hybrid attack, by Göpfert, van Vredendaal, and Wunderer [38] 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 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 9302d42 on 2017-09-27 in our sage script.

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 [18]. The Blum-Kalai-Wasserman algorithm [5, 43] 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 \( \text{dist}(c, L(A)) = \|c - x\| \) 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 \( c' = (-e, t) \in \mathbb{Z}_q^{m+1} \) with length \( \|c'\| = \sqrt{m\alpha^2 q^2/(2\pi)} + |t|^2 \) [26,48].

In the LWE-Estimator \( t = 1 \) is used. Therefore, \( e \) can be extracted from \( c' \), the value \( A_s \) is known, and \( s \) can be solved for. Based on Albrecht et al. [7], Göpfert shows [37, Section 3.1.3] that the standard embedding attack succeeds with non-negligible probability if \( \delta_0 \leq \frac{q^1 - \frac{1}{\sqrt{q^2}} \sqrt{\frac{1}{\tau_o q}}}{\tau_o q} \), where \( m \) is the number of LWE samples. The value \( \tau \) is experimentally determined to be \( \tau \leq 0.4 \) for a success probability of \( \epsilon = 0.1 \) [7].

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 \( \Lambda_q^\perp(A_o) = \{v \in \mathbb{Z}_q^{m+n+1} | A_o \cdot v = 0 \mod q \} \) 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 [53] 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 guarantying 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.

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. Still, we remark 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.

We note that, recently, the scheme ring-TESLA [1] was analyzed with respect to cache side channels with the software tool CacheAudit [20]. 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 [20] with some additional optimizations. For example, our implementation of the rejection sampling in the signing algorithm (line 12 of Algorithm 7) protects the sign of each coefficient of $z$ by doing a masked conversion to obtain its absolute value before checking against a unique positive bound. The function leaks, however, the position of the coefficient that fails the test, but this information is independent of the secret data.

Notably, qTESLA has implicitly an additional line of defense, thanks to its non-deterministic nature. To generate the polynomial $y$, qTESLA hashes together fresh randomness, a value $\text{seed}_y$ passed through the secret key and the message $m$. The use of $\text{seed}_y$ makes qTESLA resilient to a catastrophic failure of the RNG during generation of the fresh randomness (e.g., a similar failure of ECDSA signatures was demonstrated in [23]). The randomness guarantees the use of a fresh $y$ at each signing operation, which makes a timing attack (or even more generally, any side-channel attack) more difficult to carry out. More importantly, this implicitly protects against some powerful and easy-to-carry out fault attacks, as explained next.

5.3.2 Fault analysis

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 [21].

Assume that line 3 of Algorithm 7 is computed without the random value $r$, i.e., as $\text{rand} \leftarrow \text{PRF}_2(\text{seed}_y, m)$. 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 $c_{\text{faulted}}$. 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 [50], 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.

**Remark 9.** In an earlier description of qTESLA, the scheme was specified as a deterministic signature scheme and, hence, was susceptible to the fault attacks described in [21].

### 6 Advantages and limitations

In this section, we summarize some advantages and limitations of the proposed signature scheme qTESLA. In addition, we compare our scheme with other post-quantum and classical signatures.

**Security of our signature scheme.** qTESLA comes accompanied by a tight security proof (cf. Theorem 6) 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. The tightness in the proof enables the use of smaller parameters and, thus, better performance when choosing parameters according to security reductions.

Our signature scheme instantiated with provably-secure parameters is provably EUF-CMA secure by virtue of the provided security reduction from the hardness of the decisional R-LWE problem to EUF-CMA security. Moreover, 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. This provides a high-level of security assurance to the proposed signature scheme.

**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 a scheme by a certain parameter set with the hardness level of the instance of the underlying lattice problem.
that corresponds to these parameters, 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 a reduction 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 is intractable. This approach supports implementations that are slower and require larger public keys, but that can still be considered practical for many high-security applications.

In summary, the choice between both options represent a trade-off between provably security assurance (provided by the provably-secure parameter sets) and greater efficiency (provided by the heuristic parameter sets).

The bit-security of the proposed parameters was estimated against known state-of-the-art classical and quantum algorithms that solve the learning with errors problem. The proposed parameters, especially in the case of provably-secure qTESLA, provide 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\(^3\).

**Ease of implementation and scalability.** qTESLA has a very compact 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 fast 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 to support different security levels with a unique, efficient implementation. The simplicity of qTESLA makes it easy to support any of the proposed security levels with a common codebase. For example, our implementations for qTESLA-I, qTESLA-III-speed and qTESLA-III-size share about 325 lines of C code, differing in only about 100 lines of C code corresponding to easy-to-implement packing functions, and a few NTT constants and system parameters which are instantiated at compilation time. Furthermore, the packing functions can be optionally implemented in a more generic way, which would virtually eliminate any differences in the C code (beside the NTT constants and system parameters). This also highlights the compactness of realizations of qTESLA;

\(^3\)The parameter generation script can be found at `\Supporting_Documentation\Script_to_choose_parameters\parameterchoice.sage`. 

40
for example, our reference implementations of heuristic qTESLA require about 430 lines of C code \(^4\), which is significantly shorter than other schemes based on ideal lattices.

**High-speed and key compactness.** Despite the compactness 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 more expensive 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. We note that the combined size of signature and public key for heuristic qTESLA is still competitive with other efficient schemes based on ideal lattices.

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 and compactness facilitates its implementation in constant-time and, arguably, its protection against more powerful 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 protect 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.

**Comparison with selected state-of-the-art signature schemes.** In this subsection we compare qTESLA with other post-quantum and classical signature schemes.

Table 6 summarizes the most relevant features of selected signature schemes, including the underlying computational assumption, the bit security against classical and quantum attacks, and key and signature sizes.

\(\text{41}\)This count excludes header files, the NTT constants and the code for the additional symmetric primitives.
As can be seen, qTESLA is among the most compact post-quantum schemes with regard to signature size. In particular, the signature size of qTESLA is several magnitudes smaller than hash-based and multivariate alternatives. Among post-quantum lattice based schemes, heuristic qTESLA offers the smallest signatures, and represents a very competitive option when the combined size of public keys and signatures is taken into account. The lattice-based signature scheme BLISS has noticeably smaller signatures, but the bit security is not estimated against quantum adversaries. Likewise, if one takes into account the provided security reduction, provably-secure qTESLA offers very competitive key and signature sizes, and is quantum-resistant.

We note, however, that classical schemes such as RSA and ECDSA feature significantly smaller signature and public key sizes than all the available post-quantum alternatives.
Table 6: Overview of selected post-quantum and classical signature schemes; signature and key sizes are given in bytes; we write ‘−’ if no corresponding data is available

<table>
<thead>
<tr>
<th>Software/Scheme</th>
<th>Comp. Assum.</th>
<th>Bit Security</th>
<th>Key Size (bytes)</th>
<th>Sig. Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>qTESLA-I</td>
<td>R-LWE</td>
<td>95(^b)</td>
<td>pk: 1 504 sk: 1 216</td>
<td>1 376</td>
</tr>
<tr>
<td>qTESLA-III-speed</td>
<td>R-LWE</td>
<td>160(^b)</td>
<td>pk: 3 104 sk: 2 112</td>
<td>2 848</td>
</tr>
<tr>
<td>qTESLA-III-size</td>
<td>R-LWE</td>
<td>160(^b)</td>
<td>pk: 2 976 sk: 2 112</td>
<td>2 720</td>
</tr>
<tr>
<td>qTESLA-p-I</td>
<td>R-LWE(^a)</td>
<td>95(^b)</td>
<td>pk: 14 880 sk: 4 576</td>
<td>2 848</td>
</tr>
<tr>
<td>qTESLA-p-III</td>
<td>R-LWE(^a)</td>
<td>160(^b)</td>
<td>pk: 39 712 sk: 12 320</td>
<td>6 176</td>
</tr>
<tr>
<td>Dilithium-II (medium) [28]</td>
<td>module SIS, module LWE</td>
<td>91(^b)</td>
<td>pk: 1 184 sk: –</td>
<td>2 044</td>
</tr>
<tr>
<td>Dilithium-II (recommended) [28]</td>
<td>module SIS, module LWE</td>
<td>125(^b)</td>
<td>pk: 1 472 sk: –</td>
<td>2 700</td>
</tr>
<tr>
<td>Dilithium-III (very high)[28]</td>
<td>module SIS, module LWE</td>
<td>158(^b)</td>
<td>pk: 1 760 sk: –</td>
<td>3 366</td>
</tr>
<tr>
<td>GPV-poly [32,36]</td>
<td>R-SIS(^a)</td>
<td>96(^c)</td>
<td>pk: 55 705 sk: 26 316</td>
<td>32 972</td>
</tr>
<tr>
<td>BLISS-B-IV [29,57]</td>
<td>R-SIS, NTRU</td>
<td>182(^c)</td>
<td>pk: 896 sk: 384</td>
<td>812</td>
</tr>
<tr>
<td>gravity-SPHINCS [12]</td>
<td>Hash collisions, 2nd preimage</td>
<td>128(^b)</td>
<td>pk: 32 sk: 64</td>
<td>22 304</td>
</tr>
<tr>
<td>SPHINCS-256 [17]</td>
<td>Hash collisions, 2nd preimage</td>
<td>128(^b)</td>
<td>pk: 1 056 sk: 1 088</td>
<td>41 000</td>
</tr>
<tr>
<td>MQDSS-31-64 [24]</td>
<td>Multivariate Quadratic system</td>
<td>128(^b)</td>
<td>pk: 72 sk: 64</td>
<td>40 952</td>
</tr>
<tr>
<td>RSA-3072 [55]</td>
<td>Integer Factorization</td>
<td>128(^d)</td>
<td>pk: 384 sk: 1 728</td>
<td>384</td>
</tr>
<tr>
<td>ECDSA (P-256) [41]</td>
<td>Elliptic Curve Discrete Logarithm</td>
<td>128(^d)</td>
<td>pk: 64 sk: 96</td>
<td>64</td>
</tr>
</tbody>
</table>

\(^a\)Parameters are chosen according to given security reduction in the quantum random oracle model.
\(^b\)Bit security analyzed against classical and quantum adversaries.
\(^c\)Bit security analyzed against classical adversaries.
\(^d\)Broken by quantum computers (bit security analyzed against classical adversaries).
References


[38] Florian Göpfert, Christine van Vredendaal, and Thomas Wunderer. A hybrid lattice basis reduction and quantum search attack on LWE. In Tanja Lange and Tsuyoshi


