Instantiating Treeless Signature Schemes

Preliminary Version

Patrick Weiden, Andreas Hülsing,
Daniel Cabarcas, and Johannes Buchmann

Technische Universität Darmstadt
Department of Computer Science
Darmstadt, Germany
{pweiden,huelsing,cabarcas,buchmann}@cdc.informatik.tu-darmstadt.de

Abstract. We study the efficiency of the treeless signature schemes [Lyu08], [Lyu09], [Lyu12] and evaluate their practical performance. We explain how to implement them, e.g., how to realize discrete Gaussian sampling and how to instantiate the random oracles. Our software implementation as well as extensive experimental results are presented. In particular, we compare the treeless signature schemes with currently used schemes and other post-quantum signature schemes. As the experimental data shows non-competitiveness, a discussion of possible improvements concludes the paper.

Keywords: Treeless Signature Schemes, Lattice-Based Cryptography, Efficiency, Implementation, Discrete Gaussians, Random Oracles.

1 Introduction

Digital signatures are one of the most important primitives used in cryptography today. There are many applications of digital signatures in practice, e.g., signatures on updates of operating systems or other software distributed over the Internet, authentication of online banking systems or online marketplaces to the customers, and so on. Lattice-based signature schemes constitute an interesting alternative to RSA- and discrete logarithm-based systems used in practice today, since the lattice-based schemes are assumed to resist quantum-attacks, are equipped with strong security proofs and are asymptotically efficient in theory. Unfortunately, only little is known about the practical efficiency of those schemes and to what extent asymptotic and practical efficiency coincide.

In this work we study the practical efficiency of lattice-based signature schemes and evaluate their performance. We implement the treeless signature schemes [Lyu08,Lyu09,Lyu12] since especially the last one is believed to be a very efficient lattice-based signature scheme in practice and at the same time provably secure (in the random oracle model). We detail the technical solutions to the main

challenges, since the realization of these challenges in practice is not as straightforward as assumed in theory. We show how to correctly realize sampling integers following a discrete Gaussian distribution and instantiating the different random oracles. We also show how to implement the random oracles used in the schemes, as the common way of replacing them by a cryptographic hash function [BR93] does not work. The reason is that the ranges of the random oracles do not consist of bit strings, but of polynomials with certain constraints. We present detailed experimental results for running times, sizes and space consumption of the three implementations.

There already exist a few implementations of lattice-based schemes. In a recent work Güneysu et al. [GLP12] presented a tailored and highly optimized hardware (FPGA) implementation of a lattice-based signature scheme on constraint devices. An implementation of a lattice-based encryption scheme was presented by Göttert et al. in [GFS+12]. Implementations of other post-quantum secure schemes can be found in [BDH11,HBB13,BERW08,CCC+09].

Organization. In Section 2 we define our notation. Section 3 recalls the descriptions of the treeless signature schemes. Section 4 presents our implementations, including details on the main challenges. In Section 5 experimental results are shown, and Section 6 concludes our work.

2 Preliminaries

We first introduce the notation used througout the paper. For an introduction and definition of lattices and ideal lattices as well as the underlying hardness problems we refer to [MG02,MR08]. Regarding attacks on lattices we advise the reader to [GN08,GNR10,CN11].

2.1 Notation

We use the rings $R := \mathbb{Z}[x]/\langle f(x)\rangle$ and $R_q := \mathbb{Z}_q[x]/\langle f(x)\rangle$ for an integer q and a polynomial $f(x)$ that is monic and irreducible over $\mathbb{Z}[x]$, where $\langle f(x)\rangle := f(x)\mathbb{Z}[x]$ denotes the ideal generated by $f(x)$. Two commonly used polynomials are $f(x) = x^n + 1$ for n being a power of 2 and $f(x) = \sum_{i=0}^{n-1} x^i$ for n being prime. By $R_{q,d}$ we denote the subset of polynomials in R_q with coefficients in $[-d, \ldots, d] \cap \mathbb{Z}$. Reductions modulo an integer q end up in $(-q/2, \ldots, q/2] \cap \mathbb{Z}$.

We denote ring elements by boldface lower case letters, e.g., $\mathbf{y}$, and identify them both as polynomials $\mathbf{y} = y_0 + y_1 x + \cdots + y_{n-1}x^{n-1}$ of maximal degree smaller than n as well as (column) vectors $\mathbf{y} =$

$(y_0, \ldots, y_{n-1})^T$ with n entries. For better distinction, we denote vectors of ring elements by a hat, e.g., $\widehat{\mathbf{p}}$. For $\widehat{\mathbf{p}} = (\mathbf{p}_1, \ldots, \mathbf{p}_m)$, we denote the associated coefficient-vector of dim. mn by $\bar{\mathbf{p}} = (\mathbf{p}_1^T, \ldots, \mathbf{p}_m^T)^T$. Matrices are denoted by boldface capital letters $\mathbf{A}$. The usual scalar product of $\mathbf{x}$ and $\mathbf{y}$ is written as $\langle \mathbf{x}, \mathbf{y} \rangle$, the ℓ_∞-norm is $\|\mathbf{y}\|_\infty := \max_i(|y_i|)$ and the ℓ_1-norm $\|\mathbf{y}\|_1 := |y_0| + \cdots + |y_{n-1}|$. By simply writing $\|\mathbf{x}\|$ we intentionally mean the Euclidean length $\|\mathbf{x}\|_2 = \sqrt{|x_0|^2 + \ldots + |x_{n-1}|^2}$.

Let $x \leftarrow D$ denote sampling element x according to distribution D and let $x \xleftarrow{\$} S$ denote uniformly sampling element x from set S. The latter is also written $x \leftarrow \mathcal{U}(S)$, or $x \leftarrow \mathcal{U}(a, b)$ if $S = [a, b]$, or $\mathcal{U}_b$ short for $\mathcal{U}(0, b)$. The statistical distance of two distributions F and G is the absolute difference of both functions evaluated at the (common) domain D, i.e., $\Delta(F, G) := \sum_{x \in D} \frac{1}{2}|F(x) - G(x)|$.

Let $D_{v,\sigma}$ be the discrete Gaussian distribution over $\mathbb{Z}$, centered at $v \in \mathbb{R}$, with standard deviation $\sigma \in \mathbb{R}_{>0}$. For $x \in \mathbb{Z}$, $D_{v,\sigma}$ assigns the probability $D_{v,\sigma}(x) := \rho_{v,\sigma}(x) / \sum_{z \in \mathbb{Z}} \rho_{v,\sigma}(z)$, where $\rho_{v,\sigma}(x) = \exp\left(-\frac{1}{2}|x - v|^2/\sigma^2\right)$ denotes the Gaussian function. We write D_σ for $D_{0,\sigma}$ and ρ_σ for $\rho_{0,\sigma}$.[1]

Let $[m]$ be the set $\{1, \ldots, m\}$ and $a||b$ the usual concatenation of a and b. Let furthermore $O(g)$ be the standard big O-notation and let $\tilde{O}(g)$ denote $O(g \log^c(g))$ for a constant $c > 0$. Finally, we mean $\log_2(x)$ when simply writing $\log(x)$.

3 Treeless Signature Schemes

In this section, we briefly describe the three treeless signature schemes to be instantiated. We state their security assumptions, list the parameter choices and highlight the challenges we faced for their implementations.

We first establish some notation common to all the schemes. The schemes are parametrized by lattice dimension n being a power of 2 and modulus q, which define R_q via $f(x) = x^n + 1$. The parameters $u_\mathcal{K}$, $u_\mathcal{M}$, $u_\mathcal{S}$ and m are integers and $\mathcal{A}_i \subset R_q$ are special scheme-dependent sets defined later. All schemes share the following features.

[1] Some authors use a slightly different definition for the Gaussian function with $\rho_{v,s}(x) = \exp\left(-\pi|x - v|^2/s^2\right)$. The two definitions are equivalent with $s = \sigma\sqrt{2\pi}$.

signing keys $\mathcal{K} := \{\mathbf{f} \in R_q : \|\mathbf{f}\|_\infty \leq u_\mathcal{K}\}$
masking elements $\mathcal{M} := \{\mathbf{f} \in R_q : \|\mathbf{f}\|_\infty \leq u_\mathcal{M}\}$
signature outputs $\mathcal{S} := \{\mathbf{f} \in R_q : \|\mathbf{f}\|_\infty \leq u_\mathcal{S}\}$
hash function family $\mathcal{H}(R_q, m) := \{h_{\widehat{\mathbf{a}}} : \widehat{\mathbf{a}} \in R_q^m\}$ with $h_{\widehat{\mathbf{a}}} : R_q^m \to R_q, \widehat{\mathbf{z}} \mapsto h_{\widehat{\mathbf{a}}}(\widehat{\mathbf{z}}) := \mathbf{a}_1\mathbf{z}_1 + \cdots + \mathbf{a}_m\mathbf{z}_m$
random oracle outputs $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$
random oracles $H : \{0,1\}^* \to \mathcal{A}_i$

We note that $\mathcal{H}(R_q, m)$ is a collision-resistant hash function family for certain choices of m and q (see e.g., [Lyu08] or [Mic07]), and each $h \in \mathcal{H}(R_q, m)$ can be described via $\widehat{\mathbf{a}} = (\mathbf{a}_1, \ldots, \mathbf{a}_m) \in R_q^m$.

3.1 Treeless Signatures 2008 and 2009

The following description covers both the treeless signature scheme [Lyu08] (TSS08) and [Lyu09] (TSS09), which are operationally the same.

KeyGen(1^n): Sample $\widehat{\mathbf{s}} \xleftarrow{\$} \mathcal{K}^m$ and choose $h \xleftarrow{\$} \mathcal{H}(R_q, m)$. Output the signing key $\widehat{\mathbf{s}}$ and the verification key $(h, \mathbf{t})$ with $\mathbf{t} := h(\widehat{\mathbf{s}})$.

Sign($\mu, h, \widehat{\mathbf{s}}$): Given message μ, hash function h and signing key $\widehat{\mathbf{s}}$, sample $\widehat{\mathbf{y}} \xleftarrow{\$} \mathcal{M}^m$ and compute the values $\mathbf{e} = H(h(\widehat{\mathbf{y}})\|\mu)$ and $\widehat{\mathbf{z}} = \widehat{\mathbf{s}}\mathbf{e} + \widehat{\mathbf{y}}$. Repeat these steps until $\widehat{\mathbf{z}} \in \mathcal{S}^m$. Then, output the signature $(\widehat{\mathbf{z}}, \mathbf{e})$.

Verify($\mu, (\widehat{\mathbf{z}}, \mathbf{e}), (h, \mathbf{t})$): Accept the signature $(\widehat{\mathbf{z}}, \mathbf{e})$ iff both $\widehat{\mathbf{z}} \in \mathcal{S}^m$ and $\mathbf{e} = H(h(\widehat{\mathbf{z}}) - \mathbf{te}\|\mu)$ hold.

One of the challenges we faced for the implementation of the key generation algorithm was how to uniformly sample elements from $\mathcal{K}^m$. Similarly challenging is sampling a hash function from the hash function family $\mathcal{H}(R_q, m)$. But perhaps the biggest challenge is the instantiation of the random oracle in the signing algorithm. As we will see, this is challenging due to the particularities of the different output sets $\mathcal{A}_i$, which cannot simply be sampled by using a hash function.

Security and Parameter Choice for TSS08. The security of TSS08 is based on the hardness of SVP_γ, as stated in the following theorem.

Theorem 1 (Cor. 6.7 in [Lyu08])**.** *If TSS08 is not strongly unforgeable for the parameters of Table 1, then there is a polynomial-time algorithm that solves $SVP_\gamma(\mathcal{L})$ for $\gamma = \tilde{O}(n^{2.5})$ for every ideal lattice $\mathcal{L} \subset R$.*

For TSS08, parameters and variable definitions are given in Table 1. As for our implementation, we let q be prime numbers of bitlength $\log(n^4)$.[2] The other parameters are set as in Table 1. Since

[2] We rely on the function `GenPrime_ZZ` of NTL [Sho] for this.

	TSS08	TSS09
n	power of 2	power of 2
m	$3\log_2(n)$	any integer
σ	-	any integer
κ	-	smallest integer s.t. $2^\kappa \binom{n}{\kappa} \geq 2^{160}$
q	prime of order $\Theta(n^4)$	integer $\approx (2\sigma+1)^m \cdot 2^{-128/n}$
$u_\mathcal{K}$	1	σ
$u_\mathcal{M}$	$mn^{1.5} \cdot \log_2(n)$	$mn\sigma\kappa$
$u_\mathcal{S}$	$mn^{1.5} \cdot \log_2(n) - \sqrt{n} \cdot \log_2(n)$	$mn\sigma\kappa - \sigma\kappa$
$\mathcal{A}_i$	$\mathcal{A}_1 = \{\mathbf{f} \in R_q : \|\mathbf{f}\|_\infty \leq 1\}$	$\mathcal{A}_2 = \{\mathbf{f} \in R_q : \|\mathbf{f}\|_1 \leq \kappa\}$

Table 1. Parameter Definitions for TSS08 and TSS09 (see [Lyu08,Lyu09])

we want to retain the underlying security proof, we did not further investigate different parameter choices.

Security and Parameter Choice for TSS09. The security for TSS09 is the same as given for TSS08 in Theorem 1 with $\gamma = \tilde{O}(n^2)$ (see Theorem 3 in [Lyu09]), but the parameter set for this scheme is different as listed in Table 1. Two new parameters σ, κ are introduced which have an impact on the sets $\mathcal{K}$, $\mathcal{M}$ and $\mathcal{S}$. Furthermore, we note that the set of random oracle outputs $\mathcal{A}_2$ is different from $\mathcal{A}_1$. This apparently minor change in theory leads to a major change in practice (see Section 4.3 below).

For our implementation, we chose the parameters $m = \log_2(n)$, $\sigma = 127$, and $q = \lceil (2\sigma+1)^m \cdot 2^{-128/n} \rceil$, and κ, $u_\mathcal{K}$, $u_\mathcal{M}$, and $u_\mathcal{S}$ as in Table 1, which is a reasonable adaption of the definitions and instantiations in Figure 2 of [Lyu09].

3.2 Treeless Signatures 2012

The treeless signature scheme in [Lyu12] (TSS12) has the same KeyGen- and nearly the same Verify-algorithm as TSS08 and TSS09. The difference in the latter is that the condition $\widehat{\mathbf{z}} \in \mathcal{S}^m$ is replaced by $\|\bar{\mathbf{z}}\|_2 \leq 2\sigma\sqrt{m}$, while the condition on the random oracle remains. The Sign-algorithm of TSS12 is as follows.

Sign$(\mu, h, \widehat{\mathbf{s}})$**:** Given message μ, hash function h and signing key $\widehat{\mathbf{s}}$, sample $\widehat{\mathbf{y}} = (\mathbf{y}_1, \ldots, \mathbf{y}_m) \leftarrow (D_\sigma^n)^m$ and compute $\mathbf{e} = H(h(\widehat{\mathbf{y}})||\mu)$ and $\widehat{\mathbf{z}}$ with $\mathbf{z}_i = \mathbf{s}_i\mathbf{e} + \mathbf{y}_i$ for $i \in [\gamma]$. Output the signature $(\widehat{\mathbf{z}}, \mathbf{e})$ with probability

$$\min\left(1, \frac{D_\sigma^p(\bar{\mathbf{z}})}{M D_{\bar{\mathbf{e}},\sigma}^p(\bar{\mathbf{z}})}\right), \tag{1}$$

where $p = mn$, and $\bar{\mathbf{z}} = (\mathbf{z}_1^T, \ldots, \mathbf{z}_m^T)^T$ and $\bar{\mathbf{e}} = ((\mathbf{s}_1\mathbf{e})^T, \ldots, (\mathbf{s}_m\mathbf{e})^T)^T$.

Note that the masking values $\mathbf{y}_i$ are chosen from a discrete Gaussian distribution instead of uniformly from the set $\mathcal{M}$. Sampling elements from a discrete Gaussian distribution is another challenge of the implementation, which is addressed in detail in Section 4.2. Also notice that the random oracle output is $\mathcal{A}_3$, defined below. A last challenge for TSS12 is the conditional output of the signature which we address in Section 4.4.

Security and Parameter Choice for TSS12. The security of this variant of the treeless signature scheme is based on the hardness of Search-SIS$^{\ell_2}_{q,f,m,\beta}$ with $\beta = (4\sigma + 2((2\alpha + 1)u_{\mathcal{K}} + \alpha)\kappa)\sqrt{p}$ for $p = mn$ and some positive integer α, and on the hardness of Decision-SIS$_{q,f,m,u_{\mathcal{K}}}$ (see [Lyu12]).

n	power of 2	512	512
q	*see below*	2^{24}	2^{33}
$u_{\mathcal{K}}$		1	31
p	$2n$	1024	1024
κ	smallest integer s.t. $2^{\kappa}\binom{n}{\kappa} \geq 2^{100}$	14	14
σ	$12u_{\mathcal{K}} \cdot \kappa\sqrt{p}$	5376	166656
M	$\exp(12u_{\mathcal{K}} \cdot \kappa\sqrt{p}/\sigma + (u_{\mathcal{K}} \cdot \kappa\sqrt{p}/(2\sigma))^2)$	2.72	2.72
$\mathcal{A}_3$	$\{\mathbf{f} \in R_q : \|\mathbf{f}\|_\infty \leq 1 \text{ and } \|\mathbf{f}\|_1 \leq \kappa\}$		

Table 2. TSS12: Parameters according to Figure 2 of [Lyu12], columns IV and V.

Parameter definitions and concrete choices for this scheme can be found in Table 2. The author chose the Hermite factor $\delta = 1.007$ as origin to deduce the given values according to the extensive experimental results of [GN08] and to be equivalent to 80-bit security (see [GLP12]).

In order for the SIS$_{q,f,m,u_{\mathcal{K}}}$ problem to be hard, one can deduce that $q < \delta^{m^2n}(\frac{2}{3}u_{\mathcal{K}}(u_{\mathcal{K}} + 1)\pi e)^{m/2}$. Therefore, we chose q to be the largest power of 2 that fulfills the previous condition. With $m = 2$, $u_{\mathcal{K}} = 1$[3] and p, κ and σ as in Table 2 we obtained the following values for q in our implementation:

n	8	16	32	64	128	256	512	1024	2048	4096
$\log_2(q)$	3	4	4	6	8	13	24	44	85	168

[3] For our implementation, we solely chose the case $u_{\mathcal{K}} = 1$, but our method can be adapted to the case $u_{\mathcal{K}} = 31$ as used by the author (see Table 2).

4 Implementing the Schemes

In this section we present relevant details on our implementations of the treeless signature schemes. After some general considerations, we detail the technical solution to the main challenges faced for their implementation, namely, sampling random numbers, sampling discrete Gaussian distributed integers, and instantiating the different random oracles. When available, we explain further tweaks used for optimizing the implementation.

General Considerations. All implementations were developed in C++ using the Number Theory Library (NTL) from [Sho]. Tasks like polynomial multiplication and reduction (as in R_q) are out of this work's scope. We note that NTL performs multiplication and reduction in R_q using the Fast Fourier Transform (FFT) which suffices for this purpose.

Thus, we rely on NTL's internal data structures for the representation of reals, (modular) integers, and polynomials as well as internal functions that can be applied to them. Since some functions, e.g., computing $\binom{n}{k}$, a special infinity norm on polynomials, or conversion of polynomial representations, did not already exist in NTL, we implemented them.

Everything that is not explicitly mentioned in this Section can be implemented in a straightforward way. We only detail those issues that are of particular interest.

4.1 Sampling Uniformly Random Numbers

Integers chosen uniformly random from $[0, b) \cap \mathbb{Z}$ (or $[-b/2, b/2) \cap \mathbb{Z}$, equivalently) are needed in all schemes, e.g., the coefficients for polynomials in R_q as for the secret and public keys created in the KeyGen-algorithm. In order to guarantee a uniform distribution we use rejection sampling.

`simple_rejection_sampling`(b):

1. Read y bits $b_{y-1} \dots b_0$ from a "random source" with $y = \lceil \log_2(b) \rceil$.
2. Compute the decimal representation $x = 2^{y-1}b_{y-1} + \cdots + 2^0 b_0$.
3. If $x < b$, output x; otherwise goto step 1.

Notice that the naive approach of sampling a large integer (via native C++ for example) and reducing it modulo b is not a good idea. The modular reduction increases the probability of small numbers to occur, and thus spoils the uniform distribution.

We have to rely on some "random source" from which we assume that it returns random bits. In our case, the operating system's source of randomness achieves this goal.

Tweak. To improve the efficiency of uniformly sampling we use an internal randomness buffer which for $\pi > 1$ requests πy bits from a single syscall and which is used by all internal routines until it is empty.

4.2 Sampling Discrete Gaussians

In order to sample integers according to a discrete Gaussian distribution D_σ as needed in the Sign-algorithm in TSS12, we adapt rejection sampling described in [GPV08] in the following way.

`normal_sampling`(τ, ω, σ):

1. Sample an integer x uniformly at random from $[-\tau\sigma, \tau\sigma)$.
2. Sample an integer y uniformly at random in $[0, 2^\omega)$.
3. If $y < 2^\omega \cdot \rho_\sigma(x)$, output x; otherwise goto step 1.

The parameter ω determines the precision in the rejection step, which aims at accepting x with probability $\rho_\sigma(x)$. The value of ω dictates how many decimal places of $\rho_\sigma(x)$ will be taken into consideration. We synchronize ω with the precision of the floating point arithmetic used.

The parameter τ (tailcut) affects the distribution and the running time. A larger τ yields a better fit to the Gaussian distribution, but increases the rejection rate and therefore the running time. In [GPV08] it is shown that the number of repetitions for a single sample is proportional to τ and independent of σ. Moreover, if $\tau = \omega(\sqrt{\log(n)})$ the output of `normal_sampling` is statistically close to D_σ, such that we used $\tau = \sqrt{n}$.

Tweak. For improvement we implemented a space-time-tradeoff: We pre-compute a lookup table consisting of all pairs $(x, \rho_\sigma(x))$ with x in $[0, \tau\sigma]$. In step 3. of `normal_sampling` we replace the calculation of $\rho_\sigma(x)$ by a lookup in the table. Note that the table has to store only half of the values of $\rho_\sigma(x)$ for $x \in [-\tau\sigma, \tau\sigma]$ due to its symmetry.

4.3 Random Oracle Instantiations

Since the treeless signatures schemes are proven secure in the random oracle model, we have to instantiate random oracles. This is challenging because the output of the ROs has to be uniformly distributed over their ranges, which for the treeless signature schemes consist of elements in R_q with certain constraints, and we have to preserve the security properties of a cryptographic hash function, i.e., collision-resistance, second-preimage resistance and one-wayness.

In the following we assume that we already have (an instantiation of) a "basic" random oracle $\mathrm{RO}_{2,k} : \mathbb{Z} \to \mathbb{Z}_p$ with $p = 2^k$ for some $k \in \mathbb{Z}$ following the common approach using a hash function.

TSS08: Polynomials with Bounded Coefficients. In TSS08, the output of the RO is the set $\mathcal{A}_1 = \{\mathbf{f} \in R_q : \|\mathbf{f}\|_\infty \leq 1\}$. We present two generic approaches that can be adapted to the set $\mathcal{A}_1$ (set $d = 3$). We assume that $d > 2^\star$ and $p \gg d^n$, i.e., that $\mathrm{RO}_{2,k}$ gives us as many output bits as needed. Otherwise, the output can be extended as in (3) below.

Construction 1. Let t' be the decimal representation of the first $\ell = \lceil 2\log_2(d^n) \rceil$ bits of the output of $\mathrm{RO}_{2,k}(\mu)$ for message μ. Round t' to an element $t \in \mathbb{Z}_{d^n}$ by computing $t \equiv t' \bmod d^n$. Take the base-d representation $t = (t_1 \dots t_n)$, and return $\mathrm{RO}_{d,n}(\mu) = \sum_{i=1}^{n} t_i x^{i-1}$, the polynomial whose coefficients are the representatives t_i from t.

Analysis of Construction 1. The goal is to obtain an output uniformly distributed over $\mathbb{Z}_{d^n}$, assuming a uniformly distributed bit string of length ℓ (via $\mathrm{RO}_{2,k}$) as input. After the modular reduction, we have the following output probability for any $e' \in \mathbb{Z}_{d^n}$:

$$\frac{\left\lfloor \frac{2^\ell}{d^n} \right\rfloor}{2^\ell} \leq \Pr_{e \leftarrow \mathrm{RO}_{d,n}(\mathcal{U}_{2^\ell})}\left[e = e'\right] \leq \frac{\left\lfloor \frac{2^\ell}{d^n} \right\rfloor + 1}{2^\ell}.$$

Hence, for two different values, the difference in the output probability is at most $1/2^\ell$. In the worst case, the first $d^n/2$ values occur with probability $(\lfloor \frac{2^\ell}{d^n} \rfloor + 1)/2^\ell$ and the remaining $d^n/2$ values with $\lfloor \frac{2^\ell}{d^n} \rfloor / 2^\ell$. Overall, we get for the statistical distance between the uniform distribution on $[0, d^n)$ and the output of $\mathrm{RO}_{d,n}$:

$$\begin{aligned} \Delta(\mathcal{U}_{d^n}, \mathrm{RO}_{d,n}(\mathcal{U}_{2^\ell})) &= \frac{1}{2}\left(\frac{d^n}{2}\left|\frac{1}{d^n} - \frac{\left\lfloor \frac{2^\ell}{d^n} \right\rfloor + 1}{2^\ell}\right| + \frac{d^n}{2}\left|\frac{1}{d^n} - \frac{\left\lfloor \frac{2^\ell}{d^n} \right\rfloor}{2^\ell}\right|\right) \\ &= \frac{d^n}{2^{\ell+2}} \leq \frac{d^n}{d^{2n}} = \frac{1}{d^n}. \end{aligned} \tag{2}$$

Thus, the statistical distance above is negligible in n.

While the first solution leads to a direct transform from $\mathrm{RO}_{2,k}$ to $\mathrm{RO}_{d,n}$, the second solution instantiates $\mathrm{RO}_{d,n}$ indirectly by a modification of its querying signing algorithm $\mathcal{B}_1$, called $\mathcal{B}_2$, which performs as follows.

* The cases $d = 1$ and $d = 2$ are trivial.

Construction 2. Sample $y' \leftarrow D$ as $\mathcal{B}_1$. Next, use the binary RO to compute $\mathrm{RO}_{2,k}(f(y'), \mu)$. Take the first $\ell' = \lceil \log_2(d^n) \rceil$ output bits and treat them as an integer t. If $t < d^n$, take the base-d representation $t = (t_1 \dots t_n)$, set $e' = \sum_{i=1}^{n} t_i x^{i-1}$ and output (y', e'). If $t \geq d^n$, restart by sampling a new y'.

Analysis of Construction 2. Let D be a distribution and f some function with noticeable range. Algorithm $\mathcal{B}_1$ first samples $y \leftarrow D$, then evaluates $e = \mathrm{RO}_{d,n}(f(y), \mu)$ for message μ, and finally outputs (y, e).

The goal is that the outputs (y, e) of $\mathcal{B}_1$ and (y', e') of $\mathcal{B}_2$ are computationally indistinguishable. The distribution of e equals that of e' as both are uniformly random over $\mathbb{Z}_{d^n}$. Furthermore, y and y' are equally distributed, although y' has to be re-chosen sometimes. This is because for a fixed μ, the rejection probability is the same for every y', since $\mathrm{RO}_{2,k}$ outputs a uniformly random value. Hence, the rejection step does not change the distribution of y'. Besides, $\mathcal{B}_2$ rejects with probability $< 1/2$ because of the choice of ℓ' and the output distribution of the RO.

Comparison between the Constructions. We note that our two constructions for instantiating the random oracle are nearly equally fast, e.g., for $n = 512$ the average running time of Construction 2 is only 3% higher than of Construction 1. Thus, we used the values of Construction 2 in the experimental results.

Addressing the Length Problem. We now address the case $\ell > k$ (or $\ell' > k$, resp.), i.e., the output length of $\mathrm{RO}_{2,k}$ is not large enough. Our construction $\mathrm{RO}_{2,k'}$ extends the output using a PRNG. More specifically, if $\mathfrak{h}$ is a collision-resistant hash function instantiating $\mathrm{RO}_{2,k}$, e.g., $\mathfrak{h} = \text{SHA-256}$, we obtain $\mathrm{RO}_{2,k'}$ by an iterative construction from [BDK+07]:

$$\begin{aligned} \mathit{Rand} &\leftarrow \mathfrak{h}(\mathit{Seed}) \\ \mathit{Seed} &\leftarrow (1 + \mathit{Seed} + \mathit{Rand}) \bmod 2^k. \end{aligned} \tag{3}$$

The first *Seed* is the message μ, and the function is repeated until enough random output *Rand* is generated, i.e., $k' > \ell$ (or $k' > \ell'$, resp.). Our construction is collision-resistant if $\mathfrak{h}$ is.

TSS09/TSS12: Bounded Polynomials (and Coefficients). The random oracle in TSS09 outputs elements uniformly at random from the set $\mathcal{A}_2 = \{\mathbf{f} \in R_q : \|\mathbf{f}\|_1 \leq \kappa\}$, and in TSS12 it outputs uniformly random elements of $\mathcal{A}_3 = \{\mathbf{f} \in R_q : \|\mathbf{f}\|_\infty \leq 1 \text{ and } \|\mathbf{f}\|_1 \leq \kappa\}$. For

notational purpose, let $n_s := 2^s\binom{n}{s}$, let $S_i := \sum_{j=0}^{i} n_j$ be the i-th subsum of the n_j's for $i = 0, \ldots, \kappa$, and let $S := S_\kappa$.

Construction for TSS12. Use $\mathrm{RO}_{2,k'}$ (e.g., as built in (3)) to output at least $k' \geq \lceil \log_2(S) \rceil + \kappa \cdot (\lceil \log_2(n) \rceil + 1)$ bits. First sample the outcome $s = \|\mathbf{f}\|_1$ for $0 \leq s \leq \kappa$ using $\lceil \log_2(S) \rceil$ bits (see below). Then sample the indices i of the s non-zero coefficients f_i (skipping already chosen ones) as well as their signs, since $f_i \in \{\pm 1\}$, using the $s \cdot (\lceil \log_2(n) \rceil + 1)$ bits.

Construction for TSS09. Use $\mathrm{RO}_{2,k'}$ (e.g., as built in (3)) to output at least $k' \geq \lceil \log_2(S) \rceil + \kappa \cdot \log_2(\kappa) + \kappa \cdot (\lceil \log_2(n) \rceil + 1)$ bits. First sample the outcome $s = \|\mathbf{f}\|_1$ for $0 \leq s \leq \kappa$ using $\lceil \log_2(S) \rceil$ bits (see below). Let $I = \emptyset$, then iteratively sample the non-zero coefficients f_i as well as their indices i and signs using the $\kappa(\log_2(\kappa) + \lceil \log_2(n) \rceil + 1)$ bits as follows:

Repeat while $s > 0$ and $I \neq \{0, \ldots, n-1\}$:
- Uniformly sample $i \in \{0, \ldots, n-1\} \setminus I$ and add it to I.
- Uniformly sample $|f_i|$ from $[1, s]$ and uniformly sample its sign.
- Set $s = s - |f_i|$.

In both constructions (for TSS09 and TSS12) one needs to sample $s = \|\mathbf{f}\|_1$ which is done using one of the following two methods.

I. Uniformly sample a number r_s in $[0, S]$ and find the value s with $S_{s-1} < r_s \leq S_s$. (The second step can be done efficiently using binary search with depth $\log_2(\kappa)$.)

II. Uniformly sample s in $[0, \kappa]$ and r_s in $[0, S]$. Output s if $r_s \leq n_s$, otherwise restart.

Analysis. Sampling s is a challenge because the values for s are not uniformly distributed, but according to some other distribution. This is due to the fact that for larger values of s there are more possible polynomials $\mathbf{f}$ with $\|\mathbf{f}\|_1 = s$. We have to assure to output s with the correct probability. Therefore we need to compute these probabilities first, via the number of (overall) possibilities n_s for each s. Then, one of the above methods is used to obtain s.

For the constructions for TSS09 and TSS12, respectively, we have the following. Since we use rejection sampling for obtaining s, the indices i, the non-zero coefficients $|f_i|$ as well as their signs, we do not spoil the according distribution. Unfortunately, the methodology can entail multiple calls to $\mathrm{RO}_{2,k'}$ due to the rejection of s or some indices i. Using a randomness buffer with a larger chunk of bits improves the situation.

We have to compute the values n_j and S_i for both methods for sampling $s = \|\mathbf{f}\|_1$, but these can be precomputed for fixed values of n and κ. For our implementation, we prefer method I since the second leads to more rejections due to the choice of r_s.

4.4 Other Issues

Handling Keys. In our implementation, we changed the handling of the keys, i.e., for TSS08 the signing key is $(n, q, \widehat{\mathbf{a}}, \widehat{\mathbf{s}})$ instead of $\widehat{\mathbf{s}}$ and the verification key is $(n, q, \widehat{\mathbf{a}}, \mathbf{t})$ instead of $(\widehat{\mathbf{a}}, \mathbf{t})$ (TSS09 and TSS12 were modified similarly). This is due to the fact that the value $\widehat{\mathbf{a}}$ is needed (at least temporarily in the cache) of the signing device, e.g., when a smartcard downloads the "publicly known" $\widehat{\mathbf{a}}$ from a webpage in order to use it, and the system parameters are needed to operate at all.

Conditional Output of Signatures in TSS12. In the Sign-algorithm, a signature $(\widehat{\mathbf{z}}, \mathbf{e})$ is output only with probability

$$\min\left(1, \frac{D^p_\sigma(\overline{\mathbf{z}})}{M D^p_{\mathbf{e},\sigma}(\overline{\mathbf{z}})}\right) \overset{(*)}{=} \min\left(1, \frac{E}{M}\right) = \begin{cases} 1 & \text{if } E \geq M \\ E/M & \text{if } E < M \end{cases}$$

where $E = \exp\left((\|\overline{\mathbf{z}} - \overline{\mathbf{e}}\|_2^2 - \|\overline{\mathbf{z}}\|_2^2)/(2\sigma^2)\right)$ and $(*)$ holds because of the invariance property of Gaussian distributions in higher dimensions.

In practice, we first compute E. If $E \geq M$, we immediately accept the signature $(\widehat{\mathbf{z}}, \mathbf{e})$; otherwise we sample $x \leftarrow \mathcal{U}_{M+1}$ and accept the signature if $x < E$; else the signature is rejected.

5 Experimental Results

In this section we analyze the efficiency of the three schemes. We describe the experiments that lead to this analysis, present the results and analyze the data. Finally, we compare our results to those of other practically used signature schemes, e.g., RSA and DSA, and other post-quantum signature schemes, e.g., XMSS and Rainbow.

The experiments were performed on a Sun XFire 4400 server with 16 Quad-Core AMD Opteron 8356 CPUs running at 2.3GHz, having 64GB of RAM and running a 64bit version of Debian 6.0.6. For our experiments we only used one of the 16 available cores.

Each of the three schemes performed 10^6 full cycles (KeyGen, Sign, Verify) on randomly chosen messages of length 512 bits. For

any cryptographic operation we measured the running time, averaging the final results of all 10^6 cycles. For the key and signature sizes we did similar measurements. We measured the running time using the Linux function `clock_gettime` called with the clock `CLOCK_PROCESS_CPUTIME_ID`. We used `Valgrind` with tool `callgrind` and `Kcachegrind` to analyze the contribution of parts of the code to the total, and measured peak memory consumption using `Valgrind` with tool `massif`.

5.1 Key and Signature Sizes

Table 3 shows the sizes of keys and signatures for our implementation of the three treeless signature schemes in kilobytes.

	TSS08			TSS09			TSS12		
n	sk	pk	sig.	sk	pk	sig.	sk	pk	sig.
8	0.55	0.39	0.32	0.41	0.39	0.27	0.08	0.06	0.05
16	1.72	1.21	0.98	1.17	1.07	0.69	0.17	0.12	0.12
32	5.00	3.48	2.72	3.62	3.45	1.70	0.31	0.24	0.23
64	14.00	9.62	6.97	10.17	9.72	4.18	0.68	0.56	0.52
128	38.14	26.88	18.13	26.96	26.08	8.66	1.61	1.38	1.19
256	95.85	67.91	44.76	69.64	67.51	20.12	4.03	3.75	3.01
512	235.68	168.13	107.46	172.09	168.22	46.01	12.93	12.78	8.10
1024	560.72	400.14	257.69	418.49	407.34	103.31	42.07	43.89	22.45

Table 3. Key and Signature Sizes for the Treeless Signature Schemes (in kB)

As one can see, the public and secret key sizes as well as the signature sizes of TSS08 are huge, i.e., hundreds of kilobytes, for secure parameters like $n = 512$. Compared to these results, the sizes of keys and signatures in TSS09 have been lowered by a factor of 1.34, but still remain bad. In TSS12, the sizes of signatures and keys have been reduced to at most 40 or 20 kilobytes, resp. For signatures the reduction factor is 4.5 and higher; for keys it is 9.25 and higher which is quite a significant improvement. When compared to RSA and other schemes which have sizes around hundreds of bytes or small kilobytes, TSS12 has much larger sizes.

We note that we did not efficiently store keys and signatures due to their straightforward use, i.e., we store a number in its $\mathbb{Z}_q$ representation rather than its bit-representation. Using the latter, one could improve the sizes at least by a factor of 2.

5.2 Running Times

Table 4 shows the running times for our implementations of the treeless signature schemes.

n	TSS08			TSS09			TSS12		
	KeyGen	Sign	Verify	KeyGen	Sign	Verify	KeyGen	Sign	Verify
8	1.99	5.71	0.13	4.06	4.16	0.09	1.39	1.55	0.09
16	2.16	7.22	0.34	4.15	4.33	0.18	1.73	2.00	0.19
32	2.91	6.72	1.11	4.39	4.99	0.59	1.87	2.40	0.45
64	5.19	16.55	3.41	4.99	6.79	1.78	2.09	3.87	1.16
128	12.80	50.98	10.13	7.10	13.19	4.91	2.92	5.82	1.89
256	16.34	96.65	11.04	12.04	26.99	9.13	3.09	11.67	2.81
512	43.47	177.67	31.18	25.58	58.02	20.48	4.77	40.71	5.68
1024	102.06	507.23	71.65	63.34	145.98	53.71	9.89	56.53	14.54

Table 4. Running Times of the Treeless Signature Schemes (in ms)

Notice the huge running times for TSS08. For example, for $n = 512$, key generation takes about 44ms, signing about 178ms and verification about 32ms. For higher dimensions, the situation gets even worse.

In TSS09, the running times are large, but compared to TSS08 there was a significant reduction of the running times for KeyGen and Sign.

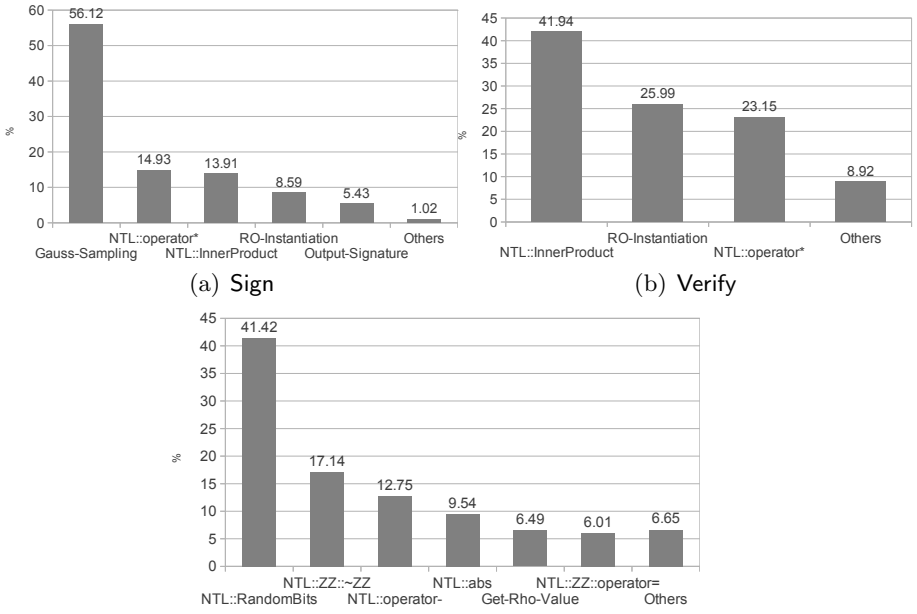
Further improvement in TSS12 compared to TSS09 is that the running times of all methods have been decreased: for KeyGen by a factor of 6, for Sign by a factor of about 1.5, and for Verify by a factor of 3.5.

In order to identify the bottleneck in the running time, we investigated the contribution of different parts of the code to the total. Figure 1 summarizes some of the most interesting results for TSS12. As can be seen in Figure 1(a), the bottleneck of the Sign-algorithm is sampling discrete Gaussians. For our implementation of the Gaussian sampling algorithm as in Section 4.2, using a lookup table, one can see in Figure 1(c) that the limiting factors are NTL-internal functions. In the Verify-algorithm, the most time-consuming function is `NTL::InnerProduct`, which computes the product of two vectors of polynomials. This also matches our expectations, regarding multiplication of polynomials and vector of polynomials, respectively, which are complex functions due to conversions to/from the FFT-representation.

5.3 Memory Consumption

We also measured the memory consumption of our implementations.

In TSS08, for $n = 512$, key generation needed 2.5MB at maximum and KeyGen used in average 86.37% of this memory; the remaining part was used by memory allocations of the program execution. For signature creation the program needed 3.6MB, but Sign used



(a) Sign (b) Verify

(c) Gauss-Sampling

Fig. 1. Running time-Partitioning of the Sign- and Verify-Algorithms and Gauss-Sampling

only 53.66% of this memory. This is due to inputting/outputting the keys from/to files. Verify only used at max. 5% of the peak memory of 2.9MB (mostly consuming: in-/output). For TSS09, the situation is almost the same with minor difference in the allocated memory.

For TSS12, the situation is different, since the most memory-consuming part is storing the lookup table in RAM. Thus, we put particular interest to the consumption of the table-creation method. For $n = 512$, of an overall memory of 4.5MB for the whole program, the lookup table consumed on average 77,78% of it. For all cycles, the maximum percentage for the lookup table was 78.15% compared to the complete memory allocation.

5.4 Comparison to State-of-the-Art Signatures

In order to contextualize our results, we compare them to other signature schemes. We collected efficiency measures for different signature schemes which are considered state-of-the-art, are used in practice, or are post-quantum secure. A summary is presented in Table 5.

As one can derive, the key generation in the treeless signature schemes is quite fast compared to other schemes. Unfortunately, this procedure can be performed offline and is thus of less interest. Furthermore, we can deduce that the fastest Sign-algorithm (of TSS12)

	Running times [ms]			Sizes [KB]		
	KeyGen	Sign	Verify	sk	pk	sig.
TSS08 ($n = 512$)	43.47	177.67	31.18	235.68	168.13	107.46
TSS09 ($n = 512$)	25.58	58.02	20.48	172.09	168.22	46.01
TSS12 ($n = 512$)	4.77	40.71	5.68	12.93	12.78	8.10
RSA-2048	279.30	5.31	0.06	2.05	0.26	0.26
DSA-2048	109.20	18.03	10.63	2.10	3.87	1.16
XMSS ($H = 20, w = 4$,AES-128)	158,208.49	2.87	0.22	0.02	0.91	2.45
Rainbow(?)	69.28	0.36	0.67	20.11	31.68	0.04
Rainbow($2^4, 24, 20, 20$)	1600.00	0.93	0.73	91.50	83.00	0.26
Rainbow($31, 24, 20, 20$)	120.00	0.071	0.02	57.00	150.00	0.04?

Table 5. Comparison of the Treeless to State-of-the-Art Signature Schemes. The values for RSA and DSA are from [EBA], the values for XMSS are from [BDH11], and the values for Rainbow are from [EBA,BERW08,CCC+09].

is outperformed by nearly all other schemes by at least a factor 2. Even for verification it is beaten outstandingly by current schemes.

The same can be derived for sizes: Regarding key sizes, the most efficient treeless signature scheme is definitely worse than RSA or DSA, but better or comparable to the other schemes. However, the signature sizes are much too large.

6 Conclusion and Future Work

In this work we presented a collection of implementation data for three signature schemes in the area of lattice-based cryptography. Our work points out that indeed there are useful lattice-based primitives that are ready for implementation.

Nonetheless, there are many open problems left. Regarding the running times, nearly all other signature schemes outperform the treeless signature schemes. In addition, the size of the keys of these schemes is still large compared to classical schemes like RSA and DSA. For the signature size, the situation is even worse. Memory consumption is good for PCs, but still far from optimal for small equipment. Lowering the sizes of these objects is a fundamental goal to further expand lattice-based schemes to constraint devices, e.g., smartcards. Approaches like the tailored and highly optimized one in [GLP12] must be considered for software implementations, but concurrently the impact of parameter selection of these schemes in regard to attacks has to be investigated. Further progress on implementing discrete Gaussian sampling, maybe parallelisation techniques, can entail improvement to signature creation time due to increased number of samples per clock cycle. The efficient creation

of random bits in a secure fashion is very interesting, since at least the treeless schemes heavily rely on them.

A further interesting topic is the multiplication of polynomials. We consider the routines of NTL sufficient for first implementations, but believe it is possible to obtain faster implementations using more dedicated routines, such as the Karatsuba method. Lattice-based signature schemes in different rings, like cyclotomic rings, could also be interesting.

Acknowledgements

We would like to thank Michael Schneider and Rachid El Bansarkhani for fruitful discussions on several topics of this work.

References

[BDH11] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - a practical forward secure signature scheme based on minimal security assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography*, volume 7071 of *Lecture Notes in Computer Science*, pages 117–129. Springer Berlin / Heidelberg, 2011.

[BDK+07] Johannes Buchmann, Erik Dahmen, Elena Klintsevich, Katsuyuki Okeya, and Camille Vuillaume. Merkle signatures with virtually unlimited signature capacity. In Jonathan Katz and Moti Yung, editors, *ACNS 07: 5th International Conference on Applied Cryptography and Network Security*, volume 4521 of *Lecture Notes in Computer Science*, pages 31–45. Springer, June 2007.

[BERW08] Andrey Bogdanov, Thomas Eisenbarth, Andy Rupp, and Christopher Wolf. Time-area optimized public-key engines: Cryptosystems as replacement for elliptic curves? In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2008*, volume 5154 of *Lecture Notes in Computer Science*, pages 45–61. Springer, August 2008.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 93: 1st Conference on Computer and Communications Security*, pages 62–73. ACM Press, November 1993.

[CCC+09] Anna Inn-Tung Chen, Ming-Shing Chen, Tien-Ren Chen, Chen-Mou Cheng, Jintai Ding, Eric Li-Hsiang Kuo, Frost Yu-Shuang Lee, and Bo-Yin Yang. SSE implementation of multivariate PKCs on modern x86 CPUs. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems – CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 33–48. Springer, September 2009.

[CN11] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 1–20. Springer, December 2011.

[EBA] eBACS: ECRYPT benchmarking of cryptographic systems, homepage: http://bench.cr.yp.to/ebats.html.

[GFS+12] Norman Göttert, Thomas Feller, Michael Schneider, Johannes Buchmann, and Sorin A. Huss. On the design of hardware building blocks for modern lattice-based encryption schemes. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 512–529. Springer, September 2012.

[GLP12] Tim Güneysu, Vadim Lyubashevsky, and Thomas Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428 of *Lecture Notes in Computer Science*, pages 530–547. Springer, September 2012.

[GN08] Nicolas Gama and Phong Q. Nguyen. Predicting lattice reduction. In Nigel P. Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 31–51. Springer, April 2008.

[GNR10] Nicolas Gama, Phong Q. Nguyen, and Oded Regev. Lattice enumeration using extreme pruning. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 257–278. Springer, May 2010.

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In Richard E. Ladner and Cynthia Dwork, editors, *40th Annual ACM Symposium on Theory of Computing*, pages 197–206. ACM Press, May 2008.

[HBB13] Andreas Hülsing, Christoph Busold, and Johannes Buchmann. Forward secure signatures on smart cards. In LarsR. Knudsen and Huapeng Wu, editors, *Selected Areas in Cryptography*, volume 7707 of *Lecture Notes in Computer Science*, pages 66–80. Springer Berlin Heidelberg, 2013.

[Lyu08] Vadim Lyubashevsky. Towards practical lattice-based cryptography. PhD thesis, University of California, San Diego, 2008.

[Lyu09] Vadim Lyubashevsky. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 598–616. Springer, December 2009.

[Lyu12] Vadim Lyubashevsky. Lattice signatures without trapdoors. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 738–755. Springer, April 2012.

[MG02] Daniele Micciancio and Shafi Goldwasser. *Complexity of Lattice Problems: a Cryptographic Perspective*, volume 671 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, Massachusetts, March 2002.

[Mic07] Daniele Micciancio. Generalized compact knapsacks, cyclic lattices, and efficient one-way functions. *Computational Complexity*, 16(4):365–411, December 2007. Prelim. in FOCS 2002.

[MR08] Daniele Micciancio and Oded Regev. Lattice-based cryptography. In Daniel J. Bernstein, Johannes A. Buchmann, and Erik Dahmen, editors, *Post-Quantum Cryptography*, pages 147–191. Springer, 2008.

[Sho] Victor Shoup. Number theory library (NTL) for C++. `http://www.shoup.net/ntl/`.