Enhanced Lattice-Based Signatures on Reconfigurable Hardware
Extended Version

Thomas Pöppelmann[1], Léo Ducas[2], and Tim Güneysu[1]

[1] Horst Görtz Institute for IT-Security, Ruhr-University Bochum, Germany
thomas.poeppelmann@rub.de,tim.gueneysu@rub.de
[2] University of California, San-Diego
lducas@eng.ucsd.edu

Abstract. The recent Bimodal Lattice Signature Scheme (BLISS) showed that lattice-based constructions have evolved to practical alternatives to RSA or ECC. It offers small signatures of 5600 bits for a 128-bit level of security, and proved to be very fast in software. However, due to the complex sampling of Gaussian noise with high precision, it is not clear whether this scheme can be mapped efficiently to embedded devices. Even though the authors of BLISS also proposed a new sampling algorithm using Bernoulli variables this approach is more complex than previous methods using large precomputed tables. The clear disadvantage of using large tables for high performance is that they cannot be used on constrained computing environments, such as FPGAs, with limited memory. In this work we thus present techniques for an efficient Cumulative Distribution Table (CDT) based Gaussian sampler on reconfigurable hardware involving Peikert's convolution lemma and the Kullback-Leibler divergence. Based on our enhanced sampler design, we provide a scalable implementation of BLISS signing and verification on a Xilinx Spartan-6 FPGA supporting either 128-bit, 160-bit, or 192-bit security. For high speed we integrate fast FFT/NTT-based polynomial multiplication, parallel sparse multiplication, Huffman compression of signatures, and Keccak as hash function. Additionally, we compare the CDT with the Bernoulli approach and show that for the particular BLISS-I parameter set the improved CDT approach is faster with lower area consumption. Our BLISS-I core uses 2,291 slices, 5.5 BRAMs, and 5 DSPs and performs a signing operation in 114.1 µs on average. Verification is even faster with a latency of 61.2 µs and 17,101 supported verification operations per second.

Keywords: Ideal Lattices, Gaussian Sampling, Digital Signatures, FPGA

1 Introduction and Motivation

Virtually all currently used digital signature schemes rely either on the factoring (RSA) or the discrete logarithm problem (DSA/ECDSA). However, with Shor's algorithm [49] sufficiently large quantum computers can solve these problems in polynomial time which potentially puts billions of devices and users at risk. Although powerful quantum computers will certainly not become available soon, significant resources are definitely spent by various organizations to boost their further development [44]. Also motivated by further advances in classical cryptanalysis (e.g., [6, 27]), it is important to investigate potential alternatives now, to have secure constructions and implementations at hand when they are finally needed.

In this work we deal with such a promising alternative, namely the Bimodal Lattice Signature Scheme (BLISS) [17], and specifically address implementation challenges for constrained devices and reconfigurable hardware. First efforts in this direction were made in 2012 by Güneysu et al. [21] (GLP). Their scheme was based on work by Lyubashevsky [34] and tuned for practicability and efficiency in embedded systems. This was achieved by a new signature compression mechanism, a more "aggressive", non-standard hardness assumption, and the decision to use uniform (as in [33])

This article is based on an earlier article: Enhanced Lattice-Based Signatures on Reconfigurable Hardware, Thomas Pöppelmann, Léo Ducas, Tim Güneysu, CHES 2014 © IACR 2014 [40] instead of Gaussian noise to hide the secret key contained in each signature via rejection sampling. While GLP allows high performance on low-cost FPGAs [21, 22] and CPUs [23] it later turned out that the scheme is suboptimal in terms of signature size and its claimed security level compared to BLISS. The main reason for this is that Gaussian noise, which is prevalent in almost all lattice-based constructions, allows more efficient, more secure, and also smaller signatures. However, while other techniques relevant for lattice-based cryptography, like fast polynomial arithmetic on ideal lattices received some attention [3, 41, 45], it is currently not clear how efficient Gaussian sampling can be done on reconfigurable and embedded hardware for large standard deviations. Results from electrical engineering (e.g., [25,51]) are not directly applicable, as they target continuous Gaussians. Applying these algorithms for the discrete case is not trivial (see, e.g., [12] for a discrete version of the Ziggurat algorithm). First progress was recently made by Roy et al. [46] based on work by Galbraith and Dwarakanath [18] but the implemented sampler is only evaluated for very low standard deviations commonly used for lattice-based encryption. We would also like to note that for lattice-based digital signature schemes large tables in performance optimized implementations might imply the impression that Gaussian-noise based schemes are a suboptimal choice on constrained embedded systems. A recent example is a microcontroller implementation [9] of BLISS that requires tables for the Gaussian sampler of roughly 40 to 50 KB on an ATxmega64A3. Thus, despite the necessity of improving Gaussian sampling techniques (which is one contribution of this work) BLISS seems to be currently the most promising scheme with a signatures length of 5,600 bit, equally large public keys, and 128-bit of equivalent symmetric security based on a reasonable security assumption. Signature schemes with explicit reductions to weaker assumptions/standard lattice problems [19, 32, 37] seem to be currently too inefficient in terms of practical signature and public key sizes (see [5] for an implementation of [37]). There surely is some room for theoretical improvement, as suggested by the new compression ideas developed by Bai and Galbraith [4]; one can hope that all those techniques can be combined to further improve lattice-based signatures.

Contribution. A first contribution of this work are improved techniques for efficient sampling of Gaussian noise that support parameters required for digital signature schemes such as BLISS and similar constructions. First, we detail how to accelerate the binary search on a cumulative distribution table (CDT) using a shortcut table of intervals (also known as guide table [14, 16]) and develop an optimal data structure that saves roughly half of the table space by exploiting the properties of the Kullback-Leibler divergence. Furthermore, we apply a convolution lemma [38] for discrete Gaussians that allows even smaller tables of less than 2.1 KB for BLISS-I parameters. Based on these techniques we provide an implementations of the BLISS-I,BLISS-III, and BLISS-IV parameter set on reconfigurable hardware that are tweaked for performance and offer 128 bits to 192 bits of security. For practical evaluation we compare our improvements for the CDT-based Gaussian sampler to the Bernoulli approach presented in [17]. Our implementation includes an FFT/NTT-based polynomial multiplier (contrary to the schoolbook approach from [21]), more efficient sparse multiplication, and the KECCAK-f[1600] hash function to provide the full picture of the performance that can be achieved by employing latest lattice-based signature schemes on reconfigurable hardware. Our BLISS-I implementation on a Xilinx Spartan-6 FPGA supports up to 8,761 signatures per second using 7,193 LUTs, 6,420 flip-flops, 5 DSPs, and 5.5 block RAMs, includes Huffman encoding and decoding of the signature and outperforms previous work in time [22] and area [21].

Table 1: BLISS parameter proposals from [17].

Name of the scheme	**BLISS-I**	**BLISS-II**	**BLISS-III**	**BLISS-IV**
Security	128 bits	128 bits	160 bits	192 bits
Optimized for	Speed	Size	Security	Security
(n, q)	(512,12289)	(512,12289)	(512,12289)	(512,12289)
Secret key densities δ_1, δ_2	0.3 , 0	0.3 , 0	0.42 , 0.03	0.45, 0.06
Gaussian std. dev. σ	215.73	107.86	250.54	271.93
Max Shift/std. dev. ratio α	1	.5	.7	.55
Weight of the challenge κ	23	23	30	39
Secret key N_κ-Threshold C	1.62	1.62	1.75	1.88
Dropped bits d in $\mathbf{z}_2$	10	10	9	8
Verif. thresholds B_2, B_∞	12872, 2100	11074, 1563	10206,1760	9901, 1613
Repetition rate	1.6	7.4	2.8	5.2
Entropy of challenge $\mathbf{c} \in \mathbb{B}^n_\kappa$	132 bits	132 bits	161 bits	195 bits
Signature size	5.6kb	5kb	6kb	6.5kb
Secret key size	2kb	2kb	3kb	3kb
Public key size	7kb	7kb	7kb	7kb

Remark. In order to allow third-party evaluation of our results, source code, test-benches, and documentation is available on our website[3]. This paper is an extended version of [40] which appeared at CHES 2014 and focused on the BLISS-I (128-bit security) parameter set. The most important changes compared to the conference version are area improvements of the underlying NTT multiplier, a full description of the Bernoulli sampler, Huffman encoding and decoding of BLISS-I signatures, and additional support for BLISS-III (160-bit security), BLISS-IV (192-bit security), as well as an extended comparison section.

2 The Bimodal Lattice Signature Scheme

The most efficient instantiation of the BLISS signature scheme [17] is based on ideal-lattices [35] and operates on polynomials over the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$. For quick reference, the BLISS key generation, signing as well as verification algorithms are given in Figure 1 and implementation relevant parameters as well as achievable signature and key sizes are listed in Table 1. Note that for the remainder of this work, we will focus solely on BLISS-I. The BLISS key generation basically involves uniform sampling of two small and sparse polynomials $\mathbf{f}, \mathbf{g}$, computation of a certain rejection condition ($N_\kappa(\mathbf{S})$), and computation of an inverse. For signature generation two polynomials $\mathbf{y}_1, \mathbf{y}_2$ of length n are sampled from a discrete Gaussian distribution with standard deviation σ. Note that the computation of $\mathbf{ay}_1$ can still be performed in the FFT-enabled ring $\mathcal{R}_q$ instead of $\mathcal{R}_{2q}$. The result $\mathbf{u}$ is then hashed with the message μ. The output of the hash function is interpreted as sparse polynomial $\mathbf{c}$. The polynomials $\mathbf{y}_1, \mathbf{y}_2$ are then used to mask the secret keys $\mathbf{s}_1, \mathbf{s}_2$ which are multiplied with the polynomial $\mathbf{c}$ and thus "sign" the hash of the message. In order to prevent any leakage of information on the secret key, rejection sampling is performed and signing might restart. Finally, the signature is compressed and $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$ returned. For verification the norms of the signature are first validated, then the input to the hash function is reconstructed and it is checked whether the corresponding hash output matches $\mathbf{c}$ from the signature.

[3] See http://www.sha.rub.de/research/projects/lattice/

```
Algorithm KeyGen()
1: Choose $\mathbf{f}, \mathbf{g}$ as uniform polynomials with exactly $d_1 = \lceil \delta_1 n \rceil$ entries in $\{\pm 1\}$ and $d_2 = \lceil \delta_2 n \rceil$
   entries in $\{\pm 2\}$
2: $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2)^t \leftarrow (\mathbf{f}, 2\mathbf{g} + 1)^t$
3: if $N_\kappa(\mathbf{S}) \geq C^2 \cdot 5 \cdot (\lceil \delta_1 n \rceil + 4 \lceil \delta_2 n \rceil) \cdot \kappa$ then restart
4: $\mathbf{a}_q = (2\mathbf{g} + 1)/\mathbf{f} \bmod q$ (restart if $\mathbf{f}$ is not invertible)
5: Return($pk = \mathbf{A}, sk = \mathbf{S}$) where $\mathbf{A} = (\mathbf{a}_1 = 2\mathbf{a}_q, q - 2) \bmod 2q$
```

```
Alg. Sign($\mu, pk = \mathbf{A}, sk = \mathbf{S}$)
1: $\mathbf{y}_1, \mathbf{y}_2 \leftarrow D_{\mathbb{Z}^n, \sigma}$
2: $\mathbf{u} = \zeta \cdot \mathbf{a}_1 \cdot \mathbf{y}_1 + \mathbf{y}_2 \bmod 2q$
3: $\mathbf{c} \leftarrow H(\lfloor \mathbf{u} \rceil_d \bmod p, \mu)$
4: Choose a random bit $b$
5: $\mathbf{z}_1 \leftarrow \mathbf{y}_1 + (-1)^b \mathbf{s}_1 \mathbf{c}$
6: $\mathbf{z}_2 \leftarrow \mathbf{y}_2 + (-1)^b \mathbf{s}_2 \mathbf{c}$
7: Continue with probability
   $1 \Big/ \left( M \exp\left( -\frac{\|\mathbf{Sc}\|^2}{2\sigma^2} \right) \cosh\left( \frac{\langle \mathbf{z}, \mathbf{Sc} \rangle}{\sigma^2} \right) \right)$
   otherwise restart
8: $\mathbf{z}_2^\dagger \leftarrow (\lfloor \mathbf{u} \rceil_d - \lfloor \mathbf{u} - \mathbf{z}_2 \rceil_d) \bmod p$
9: Return $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$
```

```
Alg. Verify($\mu, pk = \mathbf{A}, (\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$)
1: if $\|(\mathbf{z}_1 | 2^d \cdot \mathbf{z}_2^\dagger)\|_2 > B_2$ then Reject
2: if $\|(\mathbf{z}_1 | 2^d \cdot \mathbf{z}_2^\dagger)\|_\infty > B_\infty$ then Reject
3: Accept iff $\mathbf{c} = H\left( \lfloor \zeta \cdot \mathbf{a}_1 \cdot \mathbf{z}_1 + \zeta \cdot q \cdot \mathbf{c} \rceil_d + \mathbf{z}_2^\dagger \bmod p, \mu \right)$
```

Fig. 1: The Bimodal Lattice Signature Scheme [17].

3 Improving Gaussian Sampling for Lattice-Based Digital Signatures

Target distribution. We recall that the centered discrete Gaussian distribution $D_{\mathbb{Z},\sigma}$ is defined by a weight proportional to $\rho_\sigma(x) = \exp(\frac{-x^2}{2\sigma^2})$ for all integers x. Our goal is to efficiently sample from that distribution for a constant value $\sigma \approx 215.73$ as specified in BLISS-I (precisely $\sigma = 254 \cdot \sigma_{\text{bin}}$ where $\sigma_{\text{bin}} = \sqrt{1/(2\ln 2)}$ is the parameter of the so-called binary-Gaussian; see Appendix C). This can easily be reduced to sampling from a distribution over $\mathbb{Z}^+$ proportional to $\rho(x)$ for all $x > 0$ and to $\rho(0)/2$ for $x = 0$.

Overview. Gaussian sampling using a large cumulative distribution table (CDT) has been shown to be an efficient strategy for the software implementation of BLISS given in [17]. In this section, we further enhance CDT-based Gaussian sampling for use on constrained devices. For simplicity, we explicitly refer to the parameter set BLISS-I although we remark that our enhancements can be transferred to any other parameter set as well. To increase performance, we first analyze and improve the binary search step to reduce the number of comparisons (cf. Section 3.1). Secondly, we decrease the size of the precomputed tables. In Section 3.3 we therefore apply a convolution lemma for discrete Gaussians adapted from [39] that enables the use of a sampler with much smaller standard deviation $\sigma' \approx \sigma/11$, reducing the table size by a factor 11. In Section 3.4 we finally reduce the size of the precomputed table further by roughly a factor of two using floating-point representation by introducing an *adaptive mantissa size*.

For those last two steps we require the "measure of distance"[4] for a distribution, called Kullback-Leibler divergence [15, 30], that offers tighter proof than the usual statistical distance (cf. Section 3.2). Kullback-Leibler is a standard notion in information theory and already played a role in cryptography, mostly in the context of symmetric cryptanalysis [7, 52].

[4] Technically, Kullback-Leibler divergence is not a distance; it is not even symmetric.

3.1 Binary Search with Shortcut Intervals

The CDT sampling algorithm uses a table $0 = T[0] \leq T[i] \leq \cdots \leq T[S+1] = 1$ to sample from a uniform real $r \in [0,1)$. The unique result x is obtained from a binary search satisfying that $T[x] \leq r < T[x+1]$ so that each output x has a probability $T[x+1] - T[x]$. For BLISS-I we need a table with $S = 2891 \approx 13.4\sigma$ entries to dismiss only a portion of the tail less than 2^{-128}. As a result, the naive binary search would require $C \in [\lfloor \log_2 S \rfloor, \lceil \log_2 S \rceil] = [11, 12]$ comparisons on average.

As an improvement we propose to combine the binary search with a hash map based on the first bits of r to narrow down the search interval in a first step (an idea that is not exactly new [14,16], also known as guide tables). For the given parameters and memory alignment reasons, we choose the first byte of r for this hash map: the unique $v \in \{0 \ldots 255\}$ such that $v/256 \leq r < (v+1)/256$. This table I of intervals has length 256 and each entry $I[v]$ encodes the smallest interval (a_v, b_v) such that $T[a_v] \leq v/256$ and $T[b_v] \geq (v+1)/256$. With this approach, the search can be directly reduced to the interval (a_v, b_v). By letting C denote the number of comparison on average, we have that $\sum_v \frac{\lfloor \log_2(b_v - a_v) \rfloor}{256} \leq C \leq \sum_v \frac{\lceil \log_2(b_v - a_v) \rceil}{256}$. For this distribution this would give $C \in [1.3, 1.7]$ comparisons on average.

3.2 Preliminaries on the Kullback-Leibler Divergence

We now present the notion of Kullback-Leibler (KL) divergence that is later used to further reduce the table size. Detailed proofs of following lemmata are given in Appendix C.1.

Definition 1 (Kullback-Leibler Divergence). *Let $\mathcal{P}$ and $\mathcal{Q}$ be two distributions over a common countable set Ω, and let $S \subset \Omega$ be the strict support of $\mathcal{P}$ ($\mathcal{P}(i) > 0$ iff $i \in S$). The Kullback-Leibler divergence, noted D_{KL} of $\mathcal{Q}$ from $\mathcal{P}$ is defined as:*

$$D_{KL}(\mathcal{P} \| \mathcal{Q}) = \sum_{i \in S} \ln \left(\frac{\mathcal{P}(i)}{\mathcal{Q}(i)} \right) \mathcal{P}(i)$$

with the convention that $\ln(x/0) = +\infty$ for any $x > 0$.

The Kullback-Leibler divergence shares many useful properties with the more usual notion of statistical distance. First, it is additive so that $D_{\mathrm{KL}}(\mathcal{P}_0 \times \mathcal{P}_1 \| \mathcal{Q}_0 \times \mathcal{Q}_1) = D_{\mathrm{KL}}(\mathcal{P}_0 \| \mathcal{Q}_0) + D_{\mathrm{KL}}(\mathcal{P}_1 \| \mathcal{Q}_1)$ and, second, non-increasing under any function $D_{\mathrm{KL}}(f(\mathcal{P}) \| f(\mathcal{Q})) \leq D_{\mathrm{KL}}(\mathcal{P} \| \mathcal{Q})$ (see Lemmata 4 and 5 of Appendix C.1). An important difference though is that it is not symmetric. Choosing parameters so that the theoretical distribution $\mathcal{Q}$ is at KL-divergence about 2^{-128} from the actually sampled distribution $\mathcal{P}$, the next lemma will let us conclude the following[5]: if the ideal scheme $\mathcal{S}^{\mathcal{Q}}$ (*i.e.* BLISS with a perfect sampler) has about 128 bits of security, so has the implemented scheme $\mathcal{S}^{\mathcal{P}}$ (*i.e.* BLISS with our imperfect sampler).

Lemma 1 (Bounding Success Probability Variations). *Let $\mathcal{E}^{\mathcal{P}}$ be an algorithm making at most q queries to an oracle sampling from a distribution $\mathcal{P}$ and returning a bit. Let $\epsilon \geq 0$, and $\mathcal{Q}$ be a distribution such that $D_{KL}(\mathcal{P} \| \mathcal{Q}) \leq \epsilon$. Let x (resp. y) denote the probability that $\mathcal{E}^{\mathcal{P}}$ (resp. $\mathcal{E}^{\mathcal{Q}}$) outputs 1. Then, $|x - y| \leq \sqrt{q\epsilon/2}$.*

[5] Apply the lemma to an attacker with success probability 3/4 against $\mathcal{S}^{\mathcal{P}}$ and number of queries $< 2^{127}$ (amplifying success probability by repeating the attack if necessary), and deduce that it also succeeds against $\mathcal{S}^{\mathcal{Q}}$ with probability at least 1/4.

In certain cases, the KL-divergence can be as small as the square of the statistical distance. For example, noting $\mathcal{B}_c$ the Bernoulli variable that returns 1 with probability c, we have $D_{\mathrm{KL}}(\mathcal{B}_{\frac{1-\epsilon}{2}} \| \mathcal{B}_{\frac{1}{2}}) \approx \epsilon^2/2$. In such a case, one requires $q = O(1/\epsilon^2)$ samples to distinguish those two distribution with constant advantage. Hence, we yield higher security using KL-divergence than statistical distance for which the typical argument would only prove security up to $q = O(1/\epsilon)$ queries. Intuitively, statistical distance is the sum of absolute errors, while KL-divergence is about the sum of squared relative errors.

Lemma 2 (Kullback-Leibler divergence for bounded relative error). *Let $\mathcal{P}$ and $\mathcal{Q}$ be two distributions of same countable support. Assume that for any $i \in S$, there exists some $\delta(i) \in (0, 1/4)$ such that we have the relative error bound $|\mathcal{P}(i) - \mathcal{Q}(i)| \leq \delta(i)\mathcal{P}(i)$. Then*

$$D_{KL}(\mathcal{P} \| \mathcal{Q}) \leq 2 \sum_{i \in S} \delta(i)^2 \mathcal{P}(i).$$

Using floating-point representation, it seems now possible to halve the storage ensuring a relative precision of 64 bits instead of an absolute precision of 128 bits. Indeed, storing data with slightly more than of relative 64 bits of precision (that is, mantissa of 64 bits in floating-point format) one can reasonably hope to obtain relative errors $\delta(i) \leq 2^{-64}$ resulting in a KL-divergence less than 2^{-128}. We further exploit this idea in Section 3.4. But first, we will also use KL-divergence to improve the convolution Lemma of Peikert [39] and construct a sampler using convolutions.

3.3 Reducing Precomputed Data by Gaussian Convolution

Given that x_1, x_2 are variables from continuous Gaussian distributions with variances σ_1^2, σ_2^2, then their combination $x_1 + cx_2$ is Gaussian with variance $\sigma_1^2 + c^2\sigma_2^2$ for any c. While this is not generally the case for discrete Gaussians, there exists similar convolution properties under some smoothing condition as proved in [38,39]. Yet those lemmata were designed with asymptotic security in mind; for practical purpose it is in fact possible to improve the $O(\epsilon)$ statistical distance bound to a $O(\epsilon^2)$ KL-divergence bound. We refer to [39] for the formal definition of the smoothing parameter η; for our purpose it only matters that $\eta_\epsilon(\mathbb{Z}) \leq \sqrt{\ln(2 + 2/\epsilon)/\pi}$ and thus our adapted lemma allows to decrease the smoothing condition by a factor of about $\sqrt{2}$.

Lemma 3 (Adapted from Thm. 3.1 from [39]). *Let $x_1 \leftarrow D_{\mathbb{Z},\sigma_1}$, $x_2 \leftarrow D_{k\mathbb{Z},\sigma_2}$ for some positive reals σ_1, σ_2 and let $\sigma_3^{-2} = \sigma_1^{-2} + \sigma_2^{-2}$, and $\sigma^2 = \sigma_1^2 + \sigma_2^2$. For any $\epsilon \in (0, 1/2)$ if $\sigma_1 \geq \eta_\epsilon(\mathbb{Z})/\sqrt{2\pi}$ and $\sigma_3 \geq \eta_\epsilon(k\mathbb{Z})/\sqrt{2\pi}$, then distribution $\mathcal{P}$ of $x_1 + x_2$ verifies*

$$D_{KL}(\mathcal{P} \| D_{\mathbb{Z},\sigma}) \leq 2\Big(1 - \Big(\frac{1+\epsilon}{1-\epsilon}\Big)^2\Big)^2 \approx 32\epsilon^2.$$

Remark. The factor $1/\sqrt{2\pi}$ in our version of this lemma is due to the fact that we use the standard deviation σ as the parameter of Gaussians and not the renormalized parameter $s = \sqrt{2\pi}\sigma$ often found in the literature.

Proof. The proof is similar to the one of [39], with $\Lambda_1 = \mathbb{Z}$, $\Lambda_2 = k\mathbb{Z}$, $\mathbf{c}_1 = \mathbf{c}_2 = \mathbf{0}$; but for the last argument of the proof where we replace statistical distance by KL-divergence. As in [39], we first establish that for any $\bar{x} \in \mathbb{Z}$ one has the following relative error bound

$$\mathbb{P}_{x \leftarrow \mathcal{P}}[x = \bar{x}] \in \left[\Big(\frac{1-\epsilon}{1+\epsilon}\Big)^2, \Big(\frac{1+\epsilon}{1-\epsilon}\Big)^2\right] \cdot \mathbb{P}_{x \leftarrow D_{\mathbb{Z},\sigma}}[x = \bar{x}].$$

It remains to conclude using Lemma 2. $\square$

To exploit this lemma, for BLISS-I we set $k = 11$, $\sigma' = \sigma/\sqrt{1+k^2} \approx 19.53$, and sample $x = x_1' + kx_2'$ for $x_1', x_2' \leftarrow D_{\mathbb{Z},\sigma'}$ (equivalently $k \cdot x_2' = x_2 \leftarrow D_{k\mathbb{Z},k\sigma'}$). The smoothness conditions are verified for $\epsilon = \sqrt{2^{-128}/32}$ and $\eta_\epsilon(\mathbb{Z}) \leq 3.860$[6]. Due to usage of the much smaller σ' instead of σ the size of the precomputation table reduces by a factor of about $k = 11$ at the price of sampling twice. However, the running time does not double in practice since the enhancement based on the shortcut intervals reduces the number of necessary comparisons to $C \in [0.22, 0.25]$ on average. For a majority of first bytes v the interval length $b_v - a_v$ is reduced to 1 and x is determined without any comparison.

Asymptotics cost. If one considers the asymptotic costs in σ our methods allow one to sample using a table size of $\Theta(\sqrt{\sigma})$ rather than $\Theta(\sigma)$ by doubling the computation time. Actually, for much larger σ one could use $O(\log \sigma)$ samples of constant standard deviation and thus achieve a table size of $O(1)$ for computational cost in $O(\log \sigma)$.

3.4 CDT Sampling with Reduced Table Size

We recall that when doing floating-point error analysis, the relative error of a computed value v is defined as $|v - v_e|/v_e$ where v_e is the exact value that was meant to be computed. Using the table $0 = T[0] \leq T[i] \leq \cdots \leq T[S+1] = 1$, the output of a CDT sampler follows the distribution $\mathcal{P}$ with $\mathcal{P}(i) = T[i+1] - T[i]$. When applying the results from KL-divergence obtained above, the relative error of $T[i+1] - T[i]$ might be significantly larger than the one of $T[i]$. This is particularly true for the tail, where $T[i] \approx 1$ but $\mathcal{P}(i)$ is very small. Intuitively, we would like the smallest probability to come first in the CDT. A simple workaround is to reverse the order of the table so that $1 = T[0] \geq T[i] \geq \cdots \geq T[S+1] = 0$ with a slight modification of the algorithm so that $\mathcal{P}(i) = T[i] - T[i+1]$. With this trick, the subtraction only increases the relative error by a factor of roughly σ. Indeed, leaving aside the details relative to discrete Gaussian, for $x \geq 0$ we have

$$\int_{y=x}^{\infty} \rho_s(y)dy \Big/ \rho_s(x) \leq \sigma \quad \text{whereas} \quad \int_{y=0}^{x} \rho_s(y)dy \Big/ \rho_s(x) \underset{x\to\infty}{\longrightarrow} +\infty.$$

The left term is an estimation of the relative-error blow-up induced by the subtraction with the CDT in the reverse order and the right term the same estimation for the CDT in the natural order. We aim to have a variable precision in the table $T[i]$ so that $\delta(i)^2\mathcal{P}(i)$ is about constant around $2^{-128}/|S|$ as suggested by Lemma 2 while $\delta(i)$ denotes the relative error $\delta(i) = |\mathcal{P}(i) - \mathcal{Q}(i)|/\mathcal{P}(i)$. As a trade-off between optimal variable precision and hardware efficiency, we propose the following data-structure. We define 9 tables $M_0 \ldots M_8$ of bytes for the mantissa with respective lengths $\ell_0 \geq \ell_1 \geq \cdots \geq \ell_8$ and another byte table E for exponents, of length ℓ_0. The value $T[i]$ is defined as

$$T[i] = 256^{-E[i]} \cdot \sum_{k=0}^{8} 256^{-(k+1)} \cdot M_k[i]$$

where $M_k[i]$ is defined as 0 when the index is out of bound $i \geq \ell_k$. In other term, the value of $T[i]$ is stored with $p(i) = 9 - \min\{k | \ell_k > i\}$ bytes of precisions. More precisely, lengths are defined as $[\ell_0, \ldots, \ell_8] = [262, 262, 235, 223, 202, 180, 157, 125, 86]$ so that we store at least two bytes for each

[6] In a previous version we stated $\eta_\epsilon(\mathbb{Z}) \leq 3.92$ which is not accurate and has been fixed in this version.

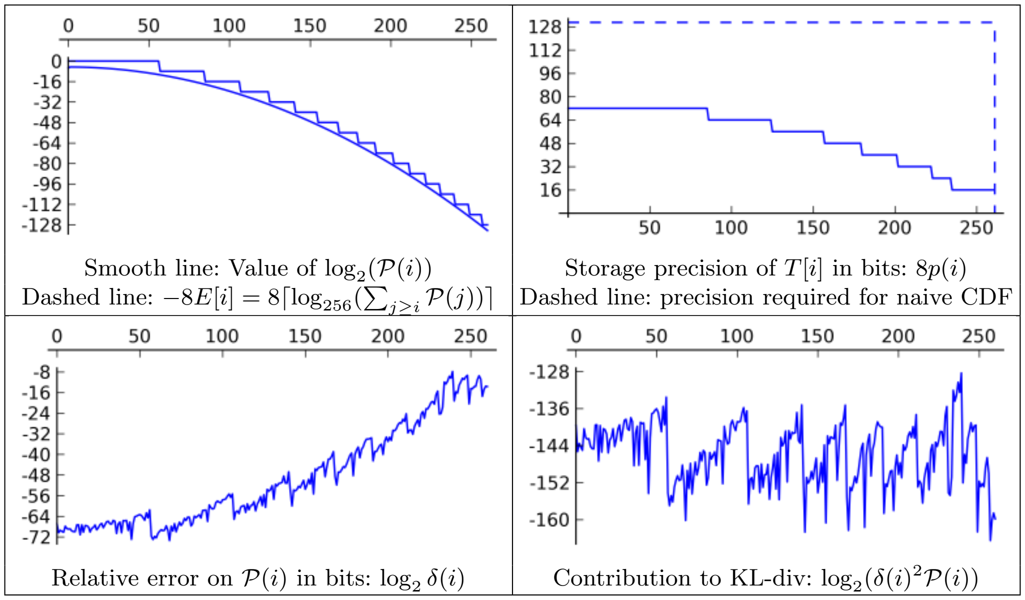

Fig. 2: Data of our optimized CDT sampler for discrete Gaussian of parameter $\sigma' \approx 19.53$.

entry up to $i < 262$, three bytes up to $i < 213$ and so forth. Note that no actual computation is involved in constructing $T[i]$ following the plain CDT algorithm.

For evaluation, we used the closed formula for KL-divergence and measured $D_{\mathrm{KL}}(\mathcal{P}\|\mathcal{Q}) \leq 2^{-128}$. The storage required by the table is $2\ell_0 + \ell_1 + \cdots + \ell_8 \approx 2.0$ KB. The straightforward CDF approach requires each entry up to $i < 262$ to be stored with $128 + \log_2 \sigma$ bits of precisions and thus requires a total of at least 4.4 KB. The storage requirements are graphically depicted by the area under the curves in the top-right quadrant of Figure 2.

4 Implementation on Reconfigurable Hardware

In this section we provide details on our implementation of the BLISS signature scheme on a Xilinx Spartan-6 FPGA. We include the enhancements from the previous section to achieve a design that is tweaked for high-performance at moderate resource costs.

4.1 Gaussian Sampling

In this section we present implementation details on the CDT sampler and the Bernoulli sampler proposed in previous work [17] to evaluate and validate our results in practice.

Enhanced CDT Sampling. Along the lines of the previous section our hardware implementation operates on bytes in order to use the 1024x8-bit mode of operation of the Spartan-6 block RAMs. The design of our CDT sampler is depicted in Figure 3 and uses the aforementioned convolution lemma. Thus two samples with $\sigma' \approx 19.53$ are combined into a sample with standard deviation $\sigma \approx 215.73$. The `BinSearch` component performs a binary search on the table T as described in Section 3.4 for a random byte vector r to find a c such that $T[c] \geq r > T[c+1]$. It accesses T byte-wise and thus $T_j[i] = M_{j-E[i]}[i]$ denotes the entry at index $i \in (0, 261)$ and byte j where $T_j[i] = 0$

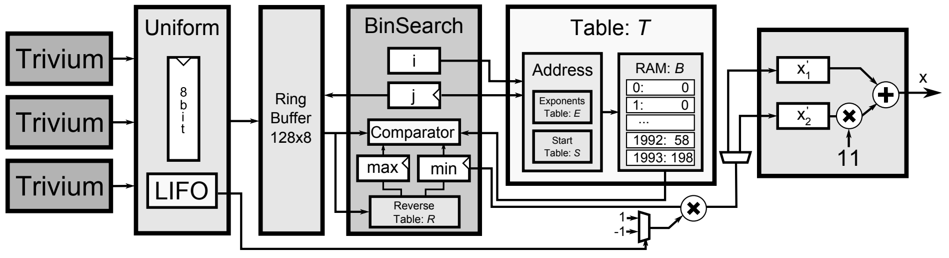

Fig. 3: Block diagram of the CDT sampler which generates two samples x'_1, x'_2 of standard deviation $\sigma' \approx 19.53$ which are combined to a sample $x = x'_1 + 11x'_2$ with standard deviation $\sigma = 215.73$. The sampling is performed using binary search on the size optimized Table T.

when $j - E[i] < 0$ or $i \geq \ell_{j-E[i]}$. When a sampling operation is started in the `BinSearch` component we set $\mathtt{j} = 0$ and initialize the pointer registers `min` and `max` with the values stored in the reverse interval table $R[r_0]$ where r_0 is the first random byte. The reverse interval table is realized as 256x15-bit single port distributed ROM (6 bits for the minimum and 9 bits for the maximum). The index of the middle element of the search radius is $\mathtt{i} = (\mathtt{min}{+}\mathtt{max})/2$. In case $T_j[\mathtt{i}] > r_\mathtt{j}$ we set $(\mathtt{min} = \mathtt{i}, \mathtt{i} = (\mathtt{i} + \mathtt{max})/2, \mathtt{max} = \mathtt{max}, \mathtt{j} = 0)$. Otherwise, for $T_j[\mathtt{i}] < r_\mathtt{j}$ we set $(\mathtt{i} = (\mathtt{min} + \mathtt{i})/2, \mathtt{min} = \mathtt{min}, \mathtt{max} = \mathtt{i}, \mathtt{j} = 0)$ until $\mathtt{max} - \mathtt{min} < 2$. In case of $T_\mathtt{j}[\mathtt{i}] = r_\mathtt{j}$ we increase $\mathtt{j} = \mathtt{j} + 1$ and thus compare the next byte. The actual entries of $M_0 \ldots M_8$ are consecutively stored in block memory B and the address is computed as $a = S[j - E[i] + i]$ where we store the start addresses of each byte group in a small additional LUT-based table $S = [0, 262, 524, 759, 982, 1184, 1364, 1521, 1646]$. Some control logic takes care that all invalid/out of bound requests to S and B return a zero.

For random byte generation we use three instantiations of the Trivium stream cipher [13] (each `Trivium` instantiation outputs one bit per clock cycle) to generate a uniformly random byte every third clock cycle and store spare bits in a `LIFO` for later use as sign bits. The random values r_j are stored in a 128x8 bit ring buffer realized as simple dual-port distributed RAM. The idea is that the sampler may request a large number of random bytes in the worst-case but usually finishes after one or two comparisons due to the lazy search. As the `BinSearch` component keeps track of the maximum number of accessed random bytes, it allows the `Uniform` sampler to refresh only the used $\max(j) + 1$ bytes in the buffer. In case the buffer is empty, we stop the Gaussian sampler until a sufficient amount of randomness becomes available. In order to compute the final sample x we assign individual random signs to two samples x'_1, x'_2 and finally output $x = x'_1 + 11x'_2$.

To achieve a high clock frequency, a comparison in the binary search step could not be performed in one cycle due to the excessive number of tables and range checks involved. We therefore allow two cycles per search step which are carefully balanced. For example, we precompute the indices $\mathtt{i}' = (\mathtt{min}{+}\mathtt{i})/2$ and $\mathtt{i}'' = (\mathtt{i}{+}\mathtt{max})/2$ in the cycle prior to a comparison to relax the critical paths. Note also that we are still accessing the two ports of the block RAM holding B only every two clock cycles which would enable another sampler to operate on the same table using time-multiplexing. Note that the implementations of the CDT sampler for $\sigma \approx 250.54$ and $\sigma \approx 271.93$ just differ by different tables and constants which are selected during synthesis using a `generic` statement.

Bernoulli Approach. In [17] Ducas et al. proposed an efficient Gaussian sampling algorithm which can be used to lower the size of precomputed tables to $\lambda \log_2(2.4\tau\sigma^2)$ bits without the need for long-integer arithmetic and with low entropy consumption ($\approx 6+3\log_2\sigma$). A detailed description and additional background on the sampler is contained in Appendix C. The general advantage of this sampler is a new technique to reduce the probability of rejections by first sampling from an (easy to sample) intermediate distribution and then from the target distribution.

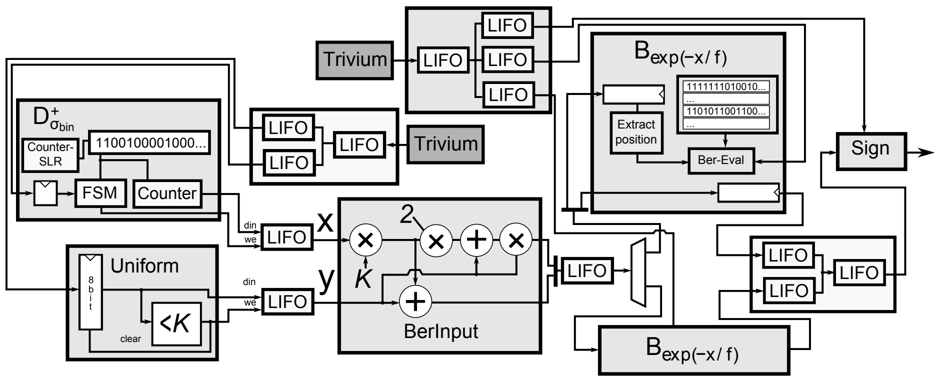

Fig. 4: Block diagram of the Bernoulli sampler using two instantiations of Trivium as PRNG and two $\mathtt{B_{exp(-x/f)}}$ components (only one is shown in more detail).

The block diagram of the the implemented sampler is given in Figure 4. In the $\mathtt{D}^+_{\sigma_{\mathrm{bin}}}$ component a $x \in D^+_{\sigma_{\mathrm{bin}}}$ is sampled according to Algorithm 2. However, on-the-fly construction of the binary distribution of $\rho_{\sigma_{\mathrm{bin}}}(\{0,\dots,j\}) = 1.1001000010000001...$ (see Algorithm 2 of Appendix C) is not necessary as we use two 64-bit shift registers (LUTM) to store the expansion precomputed up to a precision of 128 bits. Uniformly random values $y \in \{0,\dots,k-1\}$ are sampled in the `Uniform` component using rejection sampling (for $k = 254$ with $\frac{2}{256}$ the probability of a rejection is low [7]). The pipelined `BerInput` component takes a (y,x) tuple as input and computes $t = kx$ and outputs $z = t + y$ as well as $j = y(y + 2t)$. While z is retained in a register, the $\mathtt{B_{exp(-x/f)}}$ module evaluates the Bernoulli distribution of $b \leftarrow B_{\exp(-j/2\sigma^2)}$. Only if $b = 1$ the value z is passed to the output and discarded otherwise. The evaluation of $B_{\exp(-x/f)}$ requires independent evaluations of Bernoulli variables. Sampling from $\mathcal{B}_c$ is easy and can be done by just evaluating $s < c$ for a uniformly random $s \in [0, 1)$ and a precomputed c. The precomputed tables $c_i = \exp(-2^i/f)$ for $0 \le i \le l, f = 2\sigma^2$ where l is $\lceil \log_2(\max(j)) \rceil$ are stored in a distributed RAM. The $\mathtt{B_{exp(-x/f)}}$ module (Algorithm 1) then searches for one-bit positions u in j and evaluates the Bernoulli variable B_{c_u}. This is done in a lazy manner so that the evaluation aborts when the first bit has been found that differs between a random s and c. This techniques saves randomness and also runtime. As the chance of rejection is larger for the most significant bits we scan them first in order to abort as quickly as possible. As the last step the `Sign` component samples a sign bit and rejects half of the samples where $z = 0$.

The Bernoulli sampler is suitable for hardware implementation as most operations work on single bits (mostly comparisons) only. However, due to the non-constant time behavior of rejection sampling we had to introduce buffers between each element (see Figure 4) to allow parallel execution and maximum utilization of every component. This includes the distribution and buffering of

[7] Rejection sampling could be avoided completely by setting $k = 256$ and thus by sampling using $\sigma = k\sigma_{\mathrm{bin}} \approx 217.43$. However, we decided to stick to the original parameter as the costs of rejection sampling are low.

random bits. In order to reduce the impact of buffering on resource consumption we included Last-In-First-Out (LIFO) buffers that solely require a single port RAM and a counter as the ordering of independent random elements does not need to be preserved by the buffer (what would be the case with a FIFO). For maximum utilization we have evaluated optimal combinations of sub-modules and finally implemented two `B_exp(-x/f)` modules fed by two instantiations of the Trivium stream cipher to generate pseudo random bits. A detailed analysis is given in Section 5.

4.2 Signing and Verification Architecture

The architecture of our implementation of a high-speed BLISS signing engine is given in Figure 5 and the same for all supported parameter sets (I,III,IV). Similar to the GLP design [21] we implemented a two stage pipeline for signing where the polynomial multiplication $\mathbf{a}_1\mathbf{y}_1$ runs in parallel to the hashing $H(\lfloor \mathbf{u} \rceil_d, \mu)$ and sparse multiplication $\mathbf{z}_1 = \mathbf{s}_1\mathbf{c} \pm \mathbf{y}_1$ and $\mathbf{z}_2 = \mathbf{s}_2\mathbf{c} \pm \mathbf{y}_2$[8].

Polynomial multiplication. For polynomial multiplication [3, 41, 45] of $\mathbf{a}_1\mathbf{y}_1$ we rely on a publicly available microcode engine that uses an FFT/NTT-based approach (`PolyMul`)[9]. However, the implementation presented in [42] was designed for slightly more complex operations than required for BLISS and as been used to realize Ring-LWE based public key encryption and a more efficient implementation of the GLP signature scheme [22]. As the public key $\mathbf{a}_1$ is already stored in NTT format, for BLISS we just have to perform a sampling operation, a forward transformation, point-wise multiplication, and one backward transformation. In contrast to the conference version [40] we thus removed unnecessary flexibility of the core (e.g., polynomial addition or subtraction), fixed some generic options to ($n = 512, q = 12289$), and also support only one NTT enabled register (instead of two). Special NTT registers are necessary in the multiplier implementation to provide two write and two read ports required by the NTT butterfly (see [45] for an improvement of this aspect). As the registers are comprised of two block RAMs which are only filled with $n/2$ coefficients (thus $n/2 \cdot \log_2(q) = 3584$ bits) this saves one block memory. Moreover, we have optimized the implemented polynomial reduction over the previous approach by using Barrett reduction (a discussion of options can be found in [10]) where VHDL code was created using Vivado High-Level Synthesis (HLS) for a special description of the operations written in C code. The microcode engine also instantiates either the Bernoulli or the CDT Gaussian sampler (configurable by a VHDL `generic`) and an intermediate FIFO for buffering.

Hash block. When a new triple $(\mathbf{a}_1\mathbf{y}_1, \mathbf{y}_1, \mathbf{y}_2)$ is available the data is transferred into the block memories `BRAM-U`, `BRAM-Y1` and `BRAM-Y2` and the small polynomial $\mathbf{u} = \zeta\mathbf{a}_1\mathbf{y}_1 + \mathbf{y}_2$ is computed on-the-fly and stored in `BRAM-U` for later use. The lower order bits $\lfloor \mathbf{u} \rceil_d \bmod p$ of $\mathbf{u}$ are saved in the `RAM-U`. As random oracle instantiation we have chosen the KECCAK-f[1600] hash function for its security and speed in hardware [29, 48]. A configurable hardware implementation[10] is provided by the KECCAK project and the `mid-range` core is parametrized so that the KECCAK state it split into 16 pieces ($Nb = 16$). To simplify control logic and padding we just hash multiples of 1024-bit

[8] Another option would be a three stage pipeline with an additional buffer between the hashing and sparse multiplication. As a tradeoff this would allows to use a slower and thus more area efficient hash function but also imply a longer delay and require pipeline flushes in case of an accepted signature. See [22] for an implementation of GLP where a three stage pipeline is used.

[9] See `http://www.sha.rub.de/research/projects/lattice/`

[10] See `http://keccak.noekeon.org/mid_range_hw.html` for more information on the core.

blocks and rehash in case of a rejection. Storing the state of the hash function after hashing the message (and before hashing $\lfloor \mathbf{u} \rceil_d \bmod p$) would be possible but is not done due to the state size of KECCAK. After hashing the `ExtractPos` component extracts the κ positions of $\mathbf{c}$ which are one from the binary hash output and stores them in the 23x9-bit memory `RAM-Pos`.

Sparse multiplication. For the computation of $\mathbf{z}'_1 = \mathbf{s}_1\mathbf{c}$ and $\mathbf{z}'_2 = \mathbf{s}_2\mathbf{c}$ we then exploited that $\mathbf{c}$ has mainly zero coefficients and only $\kappa = 23$ coefficients set to one. Moreover, only $d_1 = \lceil \delta_1 n \rceil = 154$ coefficients in $\mathbf{s}_1$ are ± 1 and $\mathbf{s}_2$ has d_1 entries in ± 2 where the first coefficient is from $\{-1, 1, 3\}$. The simplest and, in this case, also best suited algorithm for sparse polynomial multiplication is the row- or column-wise schoolbook algorithm. While row-wise multiplication would benefit from the sparsity of $\mathbf{s}_{1,2}$ *and* $\mathbf{c}$, more memory accesses are necessary to add and store inner products. Since memory that has more than two ports is extremely expensive, this also prevents efficient and configurable parallelization. As a consequence, our implementation consists of a configurable number of cores (C) which perform column-wise multiplication to compute $\mathbf{z}_1$ and $\mathbf{z}_2$, respectively. Each core stores the secret key (either $\mathbf{s}_1$ or $\mathbf{s}_2$) efficiently in a distributed RAM and accumulates inner products in a small multiply-accumulate unit (`MAC`). Positions of $\mathbf{c}$ are fed simultaneously into the cores. Another advantage of our approach is that we can compute the norms and scalar products for rejection sampling parallel to the sparse multiplication. In Figure 5 a configuration with $C = 2$ is shown for simplicity but our experiments show that $C = 8$ leads to an optimal trade-off between speed and resource consumption.

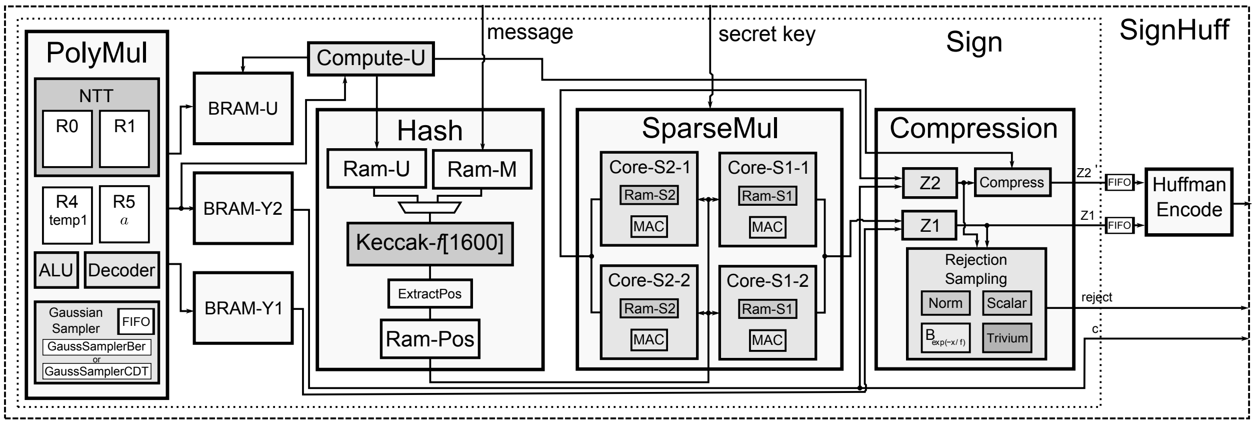

Fig. 5: Block diagram of the implemented BLISS-I signing engine.

Signature verification. Our verification engine uses only the `PolyMul` (without a Gaussian sampler) and the `Hash` component and is thus much more lightweight compared to signing. The polynomial $\mathbf{c}$ stored as (unordered) positions is expanded into a 512x1-bit distributed RAM and the input to the hash function is computed in a pipelined manner when `PolyMul` outputs $\mathbf{a}_1\mathbf{y}_1$.

4.3 Huffman Encoding

The `Sign` and `Verify` components described above operate on signatures $(c, \mathbf{z}_1, \mathbf{z}_2^\dagger)$ that consist of κ positions of bits that are one in the polynomial $\mathbf{c}$, the Gaussian distributed polynomial $\mathbf{z}_1$ (std. deviation σ), and the small polynomial $\mathbf{z}_2^\dagger$ where most lower order bits have already been dropped. Storing the signature in this format would require $\approx \kappa \cdot \log_2(n) + n \cdot \lceil (1 + \log_2(\tau\sigma)) \rceil + n \cdot \lceil (\log_2(3) \rceil$ bits which is ≈ 8399 bits for BLISS-I. However, in order to achieve the signature size stated in Table 1 of 5600 bits for BLISS-I additional Huffman encoding of $\mathbf{z}_1, \mathbf{z}_2^\dagger$ is necessary (c does not contain any easily removable redundancy). Savings are possible as small coefficients are much more likely than large ones in a Gaussian distribution.

In order to perform the Huffman encoding efficiently, we aim at getting a small Huffman table by not encoding too many values. Therefore, we focus on the higher-order bits of coefficients of $\mathbf{z}_1, \mathbf{z}_2$ (for $\mathbf{z}_2$ already given by $\mathbf{z}_2^\dagger$), since the lower order bits are almost uniform and thus not efficiently compressible. As the distributions are symmetric, we encode only absolute values, and deal with the signs separately which further reduces the table size. To decrease the overhead between theoretical entropy and actual efficiency of Huffman encoding we group two coefficients from $\mathbf{z}_1$ and two coefficients from $\mathbf{z}_2^\dagger$ together.

To encode a signature we thus split $\mathbf{z}_1, \mathbf{z}_2^\dagger$ into $n/2$ blocks of the form $b[i] = (\mathbf{z}_1[2i], \mathbf{z}_1[2i+1], \mathbf{z}_2^\dagger[2i], \mathbf{z}_2^\dagger[2i+1])$ and focus from now on a single block $b = (z_1, z_1', z_2, z_2')$. The components z_1, and z_1', respectively, are then decomposed as a triple of higher-order bits h_{z_1}, lower-order bits l_{z_1}, and sign $s_{z_1} \in \{-1, 0, 1\}$ where $z_1 = s_{z_1}(\cdot h_{z_1} \cdot B + l_{z_1})$ with $B = 2^\beta$ and $l_{z_1} \in [0, \dots, B-1]$. Note that the value of s_{z_1} is irrelevant in case the decomposed value is zero and that the coefficients from $\mathbf{z}_2$ already have their lower-order bits dropped (thus $h_{z_2} = z_2$ and $h_{z_2'} = z_2'$). For possible values of $(h_{z_1}, h_{z_1'}, h_{z_2}, h_{z_2'})$ we have calculated the frequencies and accordingly a variable length Huffman encoding where $\tilde{e} = \text{ENCODE}(h_{z_1}, h_{z_1'}, h_{z_2}, h_{z_2'})$. The sign bits are also stored as a variable length $\tilde{s}$ string where sign bits are only stored if the associated coefficient is non zero (maximum four bits). As a consequence, the encoding of a whole block $b[i]$ is the concatenation $v = \tilde{s}|\tilde{e}|l_{z_1'}|l_{z_1}$. During decoding the values of $(h_{z_1}, h_{z_1'}, h_{z_2}, h_{z_2'})$ have to be recovered from $\tilde{e}$ and (z_1, z_1', z_2, z_2') can be computed using $l_{z_1'}, l_{z_1}$ and the sign information.

For our FPGA implementation we have realized a separate encoder `HuffmanEncode` and decoder `HuffmanDecode` component exclusively for the BLISS-I parameter set and developed wrappers `SignHuff` and `VerifyHuff` using them on top of `Sign` and `Verify`, respectively (see Figure 5). For faster development time we relied on high-level synthesis to implement both cores[11]. The encoder requests pairs of $\mathbf{z}_1, \mathbf{z}_2^\dagger$ from a small fifo necessary for short term buffering and because $\mathbf{z}_2^\dagger$ coefficients are slightly more delayed dues to compression. The $\tilde{e} = \text{ENCODE}(h_{z_1}, h_{z_1'}, h_{z_2}, h_{z_2'})$ function is realized as a look-up table where the concatenated value $h_{z_1}|h_{z_1'}|h_{z_2}|h_{z_2'}$ is used as an address for a table with 64 entries (see Appendix B). The final encoded signature is then written to the top-level component in 32-bit chunks. The maximum size of one v is 39 bits with a maximum length of $\tilde{e}$ of 19 bits, 2 times 8 bits for the $l_{z_1'}|l_{z_1}$ and 4 bits for signs. It can happen that from a previous run at

[11] While a hand-optimized plain VHDL implementation would probably be more efficient, we opted for the HLS design flow mainly due to much higher development speed and faster verification using a C testbench. As Huffman encoding is not a core component of the signature scheme and not a particularly new technique it did not seem worthwhile to spend a large amount of time with low-level design of such a component. However, in order to provide a complete implementation that achieves the theoretical signature size, Huffman encoding is required and by using a HLS tool we can give good estimates for resource consumption and running time (or at least an upper bound). However, in future work it would certainly be interesting to compare our implementation to a hand-optimized low-level VHDL implementation.

maximum 31 bits stay in the internal buffer of the encoder (with 32 or more bits the buffer would have been written to the output). For decoding we first request chunks of the encoded signature into a shift register. The values of $(h_{z_1}, h_{z_1'}, h_{z_2}, h_{z_2'})$ for a given e are recovered by linear searching in a the Huffman table ordered by the size/probability of resulting bit strings. Using this information the signature can be completely recovered and is stored in an internal dual-block memory instantiated by the `HuffmanDecode` component. In the `VerifyHuff` top-level component this buffer is connected to the `Verify` component and the buffer allows parallel decoding of one signature and verification of an already decoded signature after it has been read from the `HuffmanDecode` buffer.

5 Results and Comparison

In this section we discuss our results which were obtained post place-and-route (PAR) on a Spartan-6 LX25 (speed grade -3) with Xilinx ISE 14.7.

Gaussian Sampling. Detailed results on area consumption and timing of our two Gaussian sampler designs (`SamplerBER`/`SamplerCDT`) are given in Table 2. The results show that the enhanced CDT sampler consumes less logic resources than the Bernoulli sampler at the cost of one 18k block memory to store the table B. This is a significant improvement in terms of storage size compared to a naive implementation without the application of the Kullback-Leibler divergence and Gaussian convolution. A standard CDT implementation would require at least $\sigma\tau\lambda = 370$ kbits (that is about 23 18K block RAMs) for the defined parameters matching a standard deviation $\sigma = 215.73$, tailcut $\tau = 13.4$ and precision $\lambda = 128$.

Regarding randomness consumption the CDT sampler needs on average 21 bits for one sample (using two smaller samples and the convolution theorem) which are generated by three instantiations of Trivium. The Bernoulli sampler on the other hand consumes 33 bits on average where 12% of the random bits are consumed by the $\mathtt{D}_{\sigma_{\mathtt{bin}}}$ module, 42% by the uniform sampler, 43% by both $\mathtt{B}_{\mathtt{exp(-x/f)}}$ units and 2.8% for the final sign determination and zero rejection. As illustrated in Figure 4, we feed the Bernoulli sampler with the pseudo-random output of two Trivium instances and a significant amount of the logic consumption can be attributed to additional buffers to compensate for possible rejections and distribution of random bits to various modules. With respect to the averaged performance, 7.5 and 17.95 cycles are required by the CDT and the Bernoulli sampler to provide one sample, respectively. We also provide results for instantiations of the CDT sampler for larger standard deviations required by BLISS-III and BLISS-IV that show that the performance impact caused by the increased standard deviation σ is small. In general, the `SamplerBER` component could also be instantiated for larger standard deviations but in this case random number generation and distribution and thus the desing would have to be changed for a fair comparison (e.g., sampling of k in Algorithm 3 requires more random bits for $\sigma > 217.34$). As a consequence, we just give results for the optimized and balanced instantiation with $\sigma \approx 215$.

With regard to the `SamplerCDT` component, by combining the convolution lemma and KL-divergence we were able to maintain the advantage of the CDT, namely high speed and relative simple implementation, but significantly reduced the memory requirements (from ≈ 23 18K block RAMs to one 18K block RAM). The convolution lemma works especially well in combination with the reverse tables as the overall table sizes shrink and thus the number of comparisons is reduced. Thus, we do not expect a CTD sampler that samples directly from standard deviation σ to be significantly faster. Additionally, larger tables would require more complex address generation which might lower the achievable clock frequency. The Bernoulli approach on the other hand does not seem as suitable for an application of the convolution lemma as the CDT. The reason is that the tables are already very small and thus a reduction would not significantly reduce the area usage. Moreover, sampling from the binary Gaussian distribution σ_{bin} ($\mathtt{D}^{+}_{\sigma_{\text{bin}}}$ component) is independent of the target distribution and does not profit from a smaller σ.

Previous implementations of Gaussian sampling for lattice-based public key encryption can be found in [43, 46]. However, both works target a smaller standard deviation of $\sigma = 3.3$. The work of Roy et al. [46] uses the Knuth-Yao algorithm (see [18] for more details), is very area-efficient (47 slices on a Virtex-5), and consumes few randomness but requires 17 clock cycles for one sample. In [43] Bernoulli sampling is used to optimize simple rejection sampling by using Bernoulli evaluation instead of computation of exp(). However, without usage of the binary Gaussian distribution (see [17]) the rejection rate is high and one sample requires 96 random bits and 144 cycles. This is acceptable for a relatively slow encryption scheme and possible due to the high output rate (one bit per cycle) of the used stream cipher but not a suitable architecture for BLISS. The discrete Ziggurat [12] performs well in software and might also profit from the techniques introduced in this work but does not seem to be a good target for a hardware implementation due to its infrequent rejection sampling operations and its costly requirement on high precision floating point arithmetic.

BLISS Operations. Results for our implementation of the BLISS signing and verification engine and sub-modules can be found in Table 2 including averaged cycle counts and possible operation per second (sometimes considering pipelining). [12] The `SignHuff` and `VerifyHuff` top-level modules include the Huffman encoding for very short signatures which is just implemented for the BLISS-I parameter set. The impact of Huffman encoding on the signing performance is negligible as the encoding is performed in parallel to the signing process. For verification we save the decoded signature obtained from the `DecodeHuff` component in a buffer which is then accessed by the verification core. As a consequence, the latency of the verification operation is $\mathit{Cycles}(\mathtt{Verify}) + \mathit{Cycles}(\mathtt{DecodeHuff})$ but for high throughput a signature can be decoded while another signature is verified. Thus the number of verification operations per second is not affected and similar to the amount of operations possible without Huffman decoding. For BLISS-I one signing attempt takes roughly 10,000 cycles and on average 1.6 trials are necessary using the BLISS-I parameter set. To evaluate the impact of the sampler used in the design, we instantiated two signing engines (`Sign`) of which one employs a CDT sampler and the other one two Bernoulli samplers to match the speed of the multiplier. For a similar performance of roughly 9,000 signing operations per second, the signing instance based on the Bernoulli sampler has a higher resource consumption (about 600 extra slices). Due to the two pipeline stages involved, the runtime of both instances is determined by $\max(\mathit{Cycles}(\mathtt{PolyMul}), \mathit{Cycles}(\mathtt{Hash})) + \mathit{Cycles}(\mathtt{SparseMul})$ where the rejection sampling in `Compression` is performed in parallel. Further design space exploration (e.g., evaluating the impact of a different number of parallel sparse multiplication operations or a faster configuration

[12] We also give size, runtime, and achievable clock frequency estimates for sub modules. However, due to cross-module optimization, different design strategies, and constraint options the resource consumption can not just be added and varies slightly (e.g., achievable timing of a stand-alone instantiation might be lower than when instantiated by a top-level module). Moreover, the target clock frequency in the constraints file can heavily influence the achievable clock frequency (if too low, PAR optimization stops early, if too high, PAR optimization gives up too quickly), and while we tried to find a good configuration for the top level modules we just synthesized the sub modules as they are with a generic constraints file.

of KECCAK) always identified the `PolyMul` component as performance bottleneck or did not provide significant savings in resources for reduced versions. In order to further increase the clock rate it would of course also be possible to instantiate the Gaussian sampler in a separate clock domain. The verification runtime is determined by $Cycles(\texttt{PolyMul}) + Cycles(\texttt{Hash})$ as no pipelining is used inside of `Verify`. The `PolyMul` is slightly faster than for signing as no Gaussian sampling is needed. It is also worth noting that for higher security parameter sets like BLISS-III (160-bit security) and BLISS-IV (192-bit security) the resource consumption does not increase significantly. Only the signing performance suffers due to a higher repetition rate (see Table 1). Verification speed and area consumption are almost constant for all security levels.

Table 2: Performance and resource consumption of our BLISS implementation.

Configuration and Operation	**Slices**	**LUT /FF**	**BRAM18/ DSP**	**MHz**	**Cycles**	**Operations per second (output)**
`SignHuff` (BLISS-I, CDT, $C = 8$)	2,291	7,193/6,420	5.5/5	139	≈15,864	8,761(signature)
`VerifyHuff` (BLISS-I)	1,687	5,065/4,312	4/3	166	≈16,346	17,101* (valid/invalid)
`Sign` (BLISS-I, 2×BER, $C = 8$)	2,646	8,313/7,932	5/7	142	≈15,840	≈8,964 (signature)
`Sign` (BLISS-I, CDT, $C = 8$)	2,047	6,309/6,127	6/5	140	≈15,864	≈8,825 (signature)
`Sign` (BLISS-III, CDT, $C = 8$)	2,079	6,397/6,179	6.5/5	133	≈27,547	≈4,828 (signature)
`Sign` (BLISS-IV, CDT, $C = 8$)	2,141	6,438/6,198	7/5	135	≈47,528	≈2,840 (signature)
`Verify` (BLISS-I)	1,482	4,366/3,887	3/3	172	9,607	17,903 (valid/invalid)
`Verify` (BLISS-III)	1,435	4,298/3,867	3/3	172	9,628	17,760 (valid/invalid)
`Verify` (BLISS-IV)	1,399	4,356/3,886	3/3	171	9,658	17,809 (valid/invalid)
`SamplerBER` (BLISS-I)	452	1,269/1,231	0/1	137	≈17.95	≈7,632,311 (sample)
`SamplerCDT` (BLISS-I)	299	928/1,121	1/0	129	≈7.5	≈17,100,00 (sample)
`SamplerCDT` (BLISS-III)	265	880/1,122	1.5/0	133	≈7.56	≈17,592,593 (sample)
`SamplerCDT` (BLISS-IV)	281	922/1,123	2/0	133	≈7.78	≈17,095,116 (sample)
`PolyMul` (CDT, BLISS-I)	835	2,557/2,707	4.5/1	145	9,307	15,579 ($\mathbf{a} \cdot \mathbf{y}_1$)
`Butterfly`	127	410/213	0/1	195	6	$195 \cdot 10^6$ [pipelined]
`Hash` ($Nb = 16$)	752	2,461/2,134	0/0	149	1,931	77,162 ($\mathbf{c}$)
`SparseMul` ($C = 1$)	64	162/125	0/0	274	15,876	17,258 ($\mathbf{c} \cdot \mathbf{s}_{1,2}$)
`SparseMul` ($C = 8$)	308	918/459	0/0	267	2,436	109,605 ($\mathbf{c} \cdot \mathbf{s}_{1,2}$)
`SparseMul` ($C = 16$)	628	1,847/810	0/0	254	1,476	172,086 ($\mathbf{c} \cdot \mathbf{s}_{1,2}$)
`Compression`**	232	700/626	0/4	150	-	parallel to `SparseMul`
`EncodeHuff` (BLISS-I)	244	752/244	0/0	150	-	parallel to `Sign`
`DecodeHuff` (BLISS-I)	259	795/398	0/0	159	≈5,639	28,196 ($\mathbf{z}_1, \mathbf{z}_2^\dagger$)

By ≈ we denote averaged cycle counts. The post PAR results where synthesized on a Spartan-6 LX25-3 for a 1024 bit message. (*) Regarding throughput the cycle count of the Huffman enabled verification core is equal to the standard core as the a decoded signature is saved in a buffer RAM and thus the decoding and verification can work in parallel. However, the latency of `VerifyHuff` is $Cycles(\texttt{Verify}) + Cycles(\texttt{DecodeHuff})$. (**) In the conference version `Compression` also contained the area count of the `Hash` and `SparseMul` components. However, this does not match the block diagram in Figure 5.

Comparison In comparison with the GLP implementations from [21, 22], the design of this work achieves higher throughput with a lower or equal number of block RAMs and DSPs. The structural advantage of BLISS is a smaller polynomial modulus (GLP: $q = 8383489$/BLISS-I: $q = 12289$), less iterations necessary for a valid signature (GLP: 7/BLISS-I: 1.6), and a higher security level (GLP: 80 bit/BLISS-I: 128 bit). Furthermore and contrary to [21], we remark that our implementation takes the area costs and timings of a hash function (KECCAK) into account. The only advantage of the implementation of [22] which improved [21] by using the NTT and the QUARK lightweight hash function, is that one core is a signing/verification hybrid which might be useful for some application scenarios. However, proving a signing/verification hybrid core (even supporting multiple security levels) would also be possible for BLISS but would require a significant additional engineering and testing effort. In summary, our implementation of BLISS is superior to [21] and also [22] in almost all aspects.

Table 3: Signing or verification speed of comparable signature scheme implementations.

Operation	**Security**	**Algorithm**	**Device**	**Resources**	**Ops/s**
Our work [`SignHuff`]	128	BLISS-I + Huffman	XC6SLX25	7,193 LUT/ 6,420 FF 5 DSP/ 5.5 BRAM18	8,761
Our work [`VerHuff`]	128	BLISS-I + Huffman	XC6SLX25	5,065 LUT/4,312 FF 3 DSP/ 4 BRAM18	17,101
Conf. version [`Sign`] [40]	128	BLISS-I	XC6SLX25	7,491 LUT/7,033 FF 6 DSP/ 7.5 BRAM18	7,958
Conf. version [`Verify`] [40]	128	BLISS-I	XC6SLX25	5,275 LUT/1,727 FF 3 DSP/ 4.5 BRAM18	14,438
GLP [sign/ver] [22]	80	GLP-I	XC6SLX25	6,088 LUT/ 6,804 FF/ 4 DSP/ 19.5 BRAM18	1,627/7,438
GLP [sign] [21]	80	GLP-I	XC6SLX16	7,465 LUT/ 8,993 FF/ 28 DSP/ 29.5 BRAM18	931
GLP [ver] [21]	80	GLP-I	XC6SLX16	6,225 LUT/ 6,663 FF/ 8 DSP/ 15 BRAM18	998
ECDSA [sign] [20]	128	Full ECDSA; secp256r1	XC5VLX110T	32,299 LUT/FF pairs	139
ECDSA [ver] [20]	128	Full ECDSA; secp256r1	XC5VLX110T	32,299 LUT/FF pairs	110
ECDSA [sign/ver] [28]	80	Full ECDSA; B-163	Cyclone II EP2C20	15,879 LE / 8,472 FF/ 36 M4K	1,063/621
RSA [sign] [50]	80/103	RSA-1024/2048 private key operation	XC4VFX12-10	4190 slices 17 DSP/7 BRAM18	548/79
ECDSA point mult.* [2]	80	NIST-B163;	XC4VLX200	7,719 LUT/ 1,502 FF	47,619
ECDSA point mult.* [24]	112	NIST-P224;	XC4VFX12-12	1,580 LS/ 26 DSP	2,739
ECDSA point mult.* [36]	128	generic 256-bit prime curve	XC5LX100	4,177 LUT/ 4,792 FF 37 DSP/ 18 BRAM36	2,631
ECDSA point mult.* [47]	128	Curve25519	XC7Z020	2,783 LUT/ 3,592 FF 20 DSP/ 2 BRAM36	2,518

(*) The overall cost of an ECDSA signing operation is dominated by one point multiplication but a full core would also require a hash function, logic for arithmetic operations, and inversion. ECDSA verification requires one or two point multiplications depending on the curve representation but also a hash function and logic for arithmetic operations.

In addition to that Glas et al. [20] report a vehicle-to-X communication accelerator based on an ECDSA signature over 256-bit prime fields. With respect to this, our BLISS implementation shows higher performance at less resource cost. An ECDSA implementation on a binary curve for an 80-bit security level on an Altera FPGA is given in [28] and achieves similar speeds and area consumption compared to our work. Other ECC implementations over 256-bit prime or binary fields (e.g., such as [24] on a Xilinx Virtex-4) only implement the point multiplication operation and not the full ECDSA protocol. Finally, a fast RSA-1024/2048 hybrid core was presented for Virtex-4 devices in [50] which requires more logic/DSPs and provides significantly lower performance (12.6 ms per 2048-bit private key operation) than our lattice-based signature instance. Implementations of several post quantum multivariate quadratic ($\mathcal{MQ}$) signature schemes (UOV, Rainbow, TTS) were given in [8]

5.1 Conclusion and Future Work

With this work we have shown that lattice-based digital signature schemes supporting high security level (128-bit to 192-bit security) can be implemented efficiently on low-cost FPGAs. Moreover, we have given an approach for efficient and theoretically sound discrete Gaussian sampling using the a small Cumulative Distribution Table (CDT) that might also be applicable in software or for other schemes (e.g., [11]).

For future work it seems worthwhile to investigate the properties of other samplers (e.g., Knuth-Yao [18]) and to implement different signature schemes like PASS [26] or NTRUsign with secure rejection sampling [1]. Moreover, for practical adoption of BLISS protection against side-channels is required. Using new compression techniques or other tricks it might also be possible to further reduce the size of the signature BLISS signature. For practical applications an (open-source) BLISS core supporting multiple security levels and signing as well as verification might be useful. In general, also other configuration options could be explored (e.g., more piplining stages for the high-repetition rate BLISS-IV parameter set), usage of different hash functions, or a faster NTT multiplier. As this paper focuses on speed it is also not clear how small and fast a lightweight implementation of BLISS would be.

Acknowledgment

We express our gratitude to David Xiao and Vadim Lyubashevsky for helpful conversations. We also thank the anonymous CHES'14 reviewers for detailed comments. This research was supported in part by the DARPA PROCEED program, NSF grant CNS-1117936, German Research Foundation (DFG), DFG Research Training Group GRK 1817/1 and the German Federal Ministry of Economics and Technology (Grant 01ME12025 SecMobil). Opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA, NSF, DFG, or German Federal Ministry of Economics and Technology.

References

1. C. Aguilar-Melchor, X. Boyen, J. Deneuville, and P. Gaborit. Sealing the leak on classical NTRU signatures. Cryptology ePrint Archive, Report 2014/484, 2014. http://eprint.iacr.org/, to appear in PQCrypto'14. 18
2. B. Ansari and M. A. Hasan. High-performance architecture of elliptic curve scalar multiplication. *IEEE Trans. Computers*, 57(11):1443–1453, 2008. 17
3. A. Aysu, C. Patterson, and P. Schaumont. Low-cost and area-efficient FPGA implementations of lattice-based cryptography. In *HOST*, pages 81–86. IEEE, 2013. 2, 11
4. S. Bai and S. D. Galbraith. An improved compression technique for signatures based on learning with errors. In J. Benaloh, editor, *CT-RSA*, volume 8366 of *Lecture Notes in Computer Science*, pages 28–47. Springer, 2014. 2

5. R. E. Bansarkhani and J. Buchmann. Improvement and efficient implementation of a lattice-based signature scheme. In Lange et al. [31], pages 48–67. 2
6. R. Barbulescu, P. Gaudry, A. Joux, and E. Thomé. A heuristic quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. In P. Q. Nguyen and E. Oswald, editors, *EUROCRYPT*, volume 8441 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2014. 1
7. C. Blondeau and B. Gérard. On the data complexity of statistical attacks against block ciphers (full version). Cryptology ePrint Archive, Report 2009/064, 2009. http://eprint.iacr.org/2009/064. 4
8. A. Bogdanov. Multiple-differential side-channel collision attacks on AES. In E. Oswald and P. Rohatgi, editors, *CHES 2008*, volume 5154 of *LNCS*, pages 30–44, Washington, D.C., USA, Aug. 10–13, 2008. Springer, Berlin, Germany. 18
9. A. Boorghany and R. Jalili. Implementation and comparison of lattice-based identification protocols on smart cards and microcontrollers. *IACR Cryptology ePrint Archive*, 2014:78, 2014. 2
10. A. Boorghany, S. B. Sarmadi, and R. Jalili. On constrained implementation of lattice-based cryptographic primitives and schemes on smart cards. *IACR Cryptology ePrint Archive*, 2014:514, 2014. 11
11. J. W. Bos, C. Costello, M. Naehrig, and D. Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. *IACR Cryptology ePrint Archive*, 2014:599, 2014. 18
12. J. Buchmann, D. Cabarcas, F. Göpfert, A. Hülsing, and P. Weiden. Discrete ziggurat: A time-memory trade-off for sampling from a Gaussian distribution over the integers. In Lange et al. [31], pages 402–417. 2, 15
13. C. D. Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. In S. K. Katsikas, J. Lopez, M. Backes, S. Gritzalis, and B. Preneel, editors, *ISC*, volume 4176 of *Lecture Notes in Computer Science*, pages 171–186. Springer, 2006. 9
14. H.-C. Chen and Y. Asau. On generating random variates from an empirical distribution. *AIIE Transactions*, 6(2):163–166, 1974. 2, 5
15. T. M. Cover and J. Thomas. *Elements of Information Theory*. Wiley, 1991. 4
16. L. Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, 1986. http://luc.devroye.org/rnbookindex.html. 2, 5
17. L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky. Lattice signatures and bimodal gaussians. In R. Canetti and J. A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 40–56, Santa Barbara, CA, USA, Aug. 18–22, 2013. Springer, Berlin, Germany. 1, 2, 3, 4, 8, 9, 15, 21, 22
18. N. C. Dwarakanath and S. D. Galbraith. Sampling from discrete Gaussians for lattice-based cryptography on a constrained device. *Applicable Algebra in Engineering, Communication and Computing*, pages 1–22, 2014. 2, 15, 18
19. C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In R. E. Ladner and C. Dwork, editors, *40th ACM STOC*, pages 197–206, Victoria, British Columbia, Canada, May 17–20, 2008. ACM Press. 2
20. B. Glas, O. Sander, V. Stuckert, K. D. Müller-Glaser, and J. Becker. Prime field ECDSA signature processing for reconfigurable embedded systems. *Int. J. Reconfig. Comp.*, 2011, 2011. 17
21. T. Güneysu, V. Lyubashevsky, and T. Pöppelmann. Practical lattice-based cryptography: A signature scheme for embedded systems. In E. Prouff and P. Schaumont, editors, *CHES 2012*, volume 7428 of *LNCS*, pages 530–547, Leuven, Belgium, Sept. 9–12, 2012. Springer, Berlin, Germany. 1, 2, 11, 16, 17, 19
22. T. Güneysu, V. Lyubashevsky, and T. Pöppelmann. Lattice-based signatures: Optimization and implementation on reconfigurable hardware. *IEEE Transactions on Computers*, 0(0):1–1, 0 2014. Journal version of [21]. 2, 11, 16, 17
23. T. Güneysu, T. Oder, T. Pöppelmann, and P. Schwabe. Software speed records for lattice-based signatures. In P. Gaborit, editor, *PQCrypto*, volume 7932 of *LNCS*, pages 67–82. Springer, 2013. 2
24. T. Güneysu and C. Paar. Ultra high performance ECC over NIST primes on commercial FPGAs. In E. Oswald and P. Rohatgi, editors, *CHES 2008*, volume 5154 of *LNCS*, pages 62–78, Washington, D.C., USA, Aug. 10–13, 2008. Springer, Berlin, Germany. 17, 18
25. R. Gutierrez, V. Torres-Carot, and J. Valls. Hardware architecture of a Gaussian noise generator based on the inversion method. *IEEE Trans. on Circuits and Systems*, 59-II(8):501–505, 2012. 2
26. J. Hoffstein, J. Pipher, J. M. Schanck, J. H. Silverman, and W. Whyte. Practical signatures from the partial Fourier recovery problem. In I. Boureanu, P. Owesarski, and S. Vaudenay, editors, *ACNS*, volume 8479 of *Lecture Notes in Computer Science*, pages 476–493. Springer, 2014. 18
27. A. Joux. A new index calculus algorithm with complexity $l(1/4 + o(1))$ in very small characteristic. Cryptology ePrint Archive, Report 2013/095, 2013. http://eprint.iacr.org/2013/095. 1
28. T. M. K. Jrvinen and J. Skytt. Final project report: Cryptoprocessor for elliptic curve digital signature algorithm (ECDSA), 2007. http://www.altera.com/literature/dc/2007/in_2007_dig_signature.pdf. 17, 18

29. B. Jungk and J. Apfelbeck. Area-efficient FPGA implementations of the SHA-3 finalists. In P. M. Athanas, J. Becker, and R. Cumplido, editors, *ReConFig*, pages 235–241. IEEE Computer Society, 2011. 11
30. S. Kullback and R. A. Leibler. On information and sufficiency. *Ann. Math. Statist.*, 22(1):79–86, 1951. 4
31. T. Lange, K. Lauter, and P. Lisonek, editors. *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, volume 8282 of *LNCS*, Burnaby, BC, Canada, August 14-16, 2013, 2014. Springer, Berlin, Germany. 19, 20
32. V. Lyubashevsky. Lattice-based identification schemes secure under active attacks. In R. Cramer, editor, *PKC 2008*, volume 4939 of *LNCS*, pages 162–179, Barcelona, Spain, Mar. 9–12, 2008. Springer, Berlin, Germany. 2
33. V. Lyubashevsky. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In M. Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 598–616, Tokyo, Japan, Dec. 6–10, 2009. Springer, Berlin, Germany. 1
34. V. Lyubashevsky. Lattice signatures without trapdoors. In D. Pointcheval and T. Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 738–755, Cambridge, UK, Apr. 15–19, 2012. Springer, Berlin, Germany. 1
35. V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In H. Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23, French Riviera, May 30 – June 3, 2010. Springer, Berlin, Germany. 3
36. Y. Ma, Z. Liu, W. Pan, and J. Jing. A high-speed elliptic curve cryptographic processor for generic curves over GF(p). In Lange et al. [31], pages 421–437. 17
37. D. Micciancio and C. Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In D. Pointcheval and T. Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 700–718, Cambridge, UK, Apr. 15–19, 2012. Springer, Berlin, Germany. 2
38. D. Micciancio and C. Peikert. Hardness of SIS and LWE with small parameters. In R. Canetti and J. A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 21–39, Santa Barbara, CA, USA, Aug. 18–22, 2013. Springer, Berlin, Germany. 2, 6
39. C. Peikert. An efficient and parallel Gaussian sampler for lattices. In T. Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 80–97, Santa Barbara, CA, USA, Aug. 15–19, 2010. Springer, Berlin, Germany. 4, 6
40. T. Pöppelmann, L. Ducas, and T. Güneysu. Enhanced lattice-based signatures on reconfigurable hardware. In E. Prouff and P. Schaumont, editors, *CHES*, volume 8731 of *Lecture Notes in Computer Science*. Springer, 2014. 1, 3, 11, 17
41. T. Pöppelmann and T. Güneysu. Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In A. Hevia and G. Neven, editors, *LATINCRYPT 2012*, volume 7533 of *LNCS*, pages 139–158, Santiago, Chile, Oct. 7–10, 2012. Springer, Berlin, Germany. 2, 11
42. T. Pöppelmann and T. Güneysu. Towards practical lattice-based public-key encryption on reconfigurable hardware. In Lange et al. [31], pages 68–85. 11
43. T. Pöppelmann and T. Güneysu. Area optimization of lightweight lattice-based encryption on reconfigurable hardware. In *ISCAS*, pages 2796–2799. IEEE, 2014. 15
44. S. Rich and B. Gellman. NSA seeks quantum computer that could crack most codes. *The Washington Post*, 2013. `http://wapo.st/19DycJT`. 1
45. S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede. Compact hardware implementation of Ring-LWE cryptosystems. *IACR Cryptology ePrint Archive*, 2013:866, 2013. 2, 11
46. S. S. Roy, F. Vercauteren, and I. Verbauwhede. High precision discrete Gaussian sampling on FPGAs. In Lange et al. [31], pages 383–401. 2, 15
47. P. Sasdrich and T. Güneysu. Efficient elliptic-curve cryptography using Curve25519 on reconfigurable devices. In D. Goehringer, M. D. Santambrogio, J. M. P. Cardoso, and K. Bertels, editors, *ARC*, volume 8405 of *Lecture Notes in Computer Science*, pages 25–36. Springer, 2014. 17
48. R. Shahid, M. U. Sharif, M. Rogawski, and K. Gaj. Use of embedded FPGA resources in implementations of 14 round 2 SHA-3 candidates. In R. Tessier, editor, *FPT*, pages 1–9. IEEE, 2011. 11
49. P. W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *35th FOCS*, pages 124–134, Santa Fe, New Mexico, Nov. 20–22, 1994. IEEE Computer Society Press. 1
50. D. Suzuki and T. Matsumoto. How to maximize the potential of FPGA-based DSPs for modular exponentiation. *IEICE Transactions*, 94-A(1):211–222, 2011. 17, 18
51. D. B. Thomas, W. Luk, P. H. W. Leong, and J. D. Villasenor. Gaussian random number generators. *ACM Comput. Surv.*, 39(4), 2007. 2
52. S. Vaudenay. Decorrelation: A theory for block cipher security. *Journal of Cryptology*, 16(4):249–286, Sept. 2003. 4

A Appendix

B Huffman Encoding

The Huffman table for parameter set BLISS-I is given in Table 4.

Table 4: Huffman table for the BLISS-I parameter set.

$(h_{z_1}, h_{z'_1}, h_{z_2}, h_{z'_2})$	binary string	$(h_{z_1}, h_{z'_1}, h_{z_2}, h_{z'_2})$	binary string
(0, 0, 0, 0)	100	(2, 0, 0, 0)	00011
(0, 0, 0, 1)	01000	(2, 0, 0, 1)	0000111
(0, 0, 1, 0)	01001	(2, 0, 1, 0)	0000101
(0, 0, 1, 1)	0011100	(2, 0, 1, 1)	000001001
(0, 1, 0, 0)	110	(2, 1, 0, 0)	00110
(0, 1, 0, 1)	01101	(2, 1, 0, 1)	0001000
(0, 1, 1, 0)	01011	(2, 1, 1, 0)	0001011
(0, 1, 1, 1)	0011110	(2, 1, 1, 1)	000001101
(0, 2, 0, 0)	00100	(2, 2, 0, 0)	0000001
(0, 2, 0, 1)	0000100	(2, 2, 0, 1)	0000011111
(0, 2, 1, 0)	0000110	(2, 2, 1, 0)	000000000
(0, 2, 1, 1)	000001010	(2, 2, 1, 1)	000001111001
(0, 3, 0, 0)	000000011	(2, 3, 0, 0)	000001111000
(0, 3, 0, 1)	00000000101	(2, 3, 0, 1)	00000001011010
(0, 3, 1, 0)	000001111011	(2, 3, 1, 0)	00000001011000
(0, 3, 1, 1)	00000111101000	(2, 3, 1, 1)	00000111101011010
(1, 0, 0, 0)	101	(3, 0, 0, 0)	000001000
(1, 0, 0, 1)	01100	(3, 0, 0, 1)	00000000100
(1, 0, 1, 0)	01010	(3, 0, 1, 0)	00000000110
(1, 0, 1, 1)	0011101	(3, 0, 1, 1)	00000001011011
(1, 1, 0, 0)	111	(3, 1, 0, 0)	000001011
(1, 1, 0, 1)	01110	(3, 1, 0, 1)	00000001010
(1, 1, 1, 0)	01111	(3, 1, 1, 0)	00000000111
(1, 1, 1, 1)	0011111	(3, 1, 1, 1)	00000111101001
(1, 2, 0, 0)	00101	(3, 2, 0, 0)	000000010111
(1, 2, 0, 1)	0001001	(3, 2, 0, 1)	00000001011001
(1, 2, 1, 0)	0001010	(3, 2, 1, 0)	000001111010111
(1, 2, 1, 1)	000001110	(3, 2, 1, 1)	00000111101011011
(1, 3, 0, 0)	000001100	(3, 3, 0, 0)	00000111101011001
(1, 3, 0, 1)	00000001000	(3, 3, 0, 1)	000001111010110000
(1, 3, 1, 0)	00000001001	(3, 3, 1, 0)	0000011110101100011
(1, 3, 1, 1)	00000111101010	(3, 3, 1, 1)	0000011110101100010

Computed for standard deviation $\sigma \approx 215$, with $B = 2^{\beta}$ for $\beta = 8$.

C Bernoulli Sampling

In this section we briefly recall the Bernoulli Sampling algorithms and refer to [17] for a detailed analysis. The first tool for sampling in [17] is an algorithm to sample according to $\mathcal{B}_{\exp(-x/f)}$ for any positive integer x using $\log_2 x$ precomputed values as described in Algorithm 1.

Algorithm 1 Sampling $\mathcal{B}_{\exp(-x/f)}$ for $x \in [0, 2^\ell)$

```
Input: $x \in [0, 2^\ell)$ an integer in binary form $x = x_{\ell-1} \cdots x_0$
Precomputation: $c_i = \exp(-2^i/f)$ for $0 \le i \le \ell - 1$
  for $i = \ell - 1$ to $0$
    if $x_i = 1$ then
      sample $A_i \leftarrow \mathcal{B}_{c_i}$
      if $A_i = 0$ then return 0
  return 1
```

Next, this algorithm is used to transform a simple Gaussian distribution to discrete Gaussian of arbitrary parameter σ. This simple Gaussian (called the binary Gaussian because the probability of each x is proportional to 2^{-x^2}) has parameter $\sigma_{\text{bin}} = \sqrt{1/(2\ln 2)} \approx 0.849$. It is straightforward to apply (Alg. 2) because of the form of its (unnormalized) cumulative distribution

$$\rho_{\sigma_{\text{bin}}}(\{0, \ldots, j\}) = \sum_{i=0}^{j} 2^{-i^2} = 1\,.\,1\,0\,0\,1\,\underbrace{0\ldots0}_{4}1\underbrace{0\ldots0}_{6}1 \quad \ldots \quad \underbrace{0\ldots0}_{2(j-2)}1\underbrace{0\ldots0}_{2(j-1)}1\,.$$

From there, one easily builds the distribution $k \cdot D_{\mathbb{Z}^+,\sigma_{\text{bin}}} + \mathcal{U}(\{0 \ldots k-1\})$ as an approximation of $D_{\mathbb{Z}^+,k\sigma_{\text{bin}}}$ which is corrected using rejection sampling technique (Alg. 3). This rejection only requires variables of the form $\mathcal{B}_{\exp(-x/f)}$ for integers x. The last step is to extend the distribution from $\mathbb{Z}^+$ to the whole set of integers $\mathbb{Z}$ as done by Algorithm 4.

Algorithm 2 Sampling $D_{\mathbb{Z}^+,\sigma_{\text{bin}}}$

```
Output: An integer $x \in \mathbb{Z}^+$ according to $D^+_{\sigma_{\text{bin}}}$
  Generate a bit $b \leftarrow \mathcal{B}_{1/2}$
  if $b = 0$ then return 0
  for $i = 1$ to $\infty$ do
    draw random bits $b_1 \ldots b_k$ for $k = 2i - 1$
    if $b_1 \ldots b_{k-1} \neq 0 \ldots 0$ then restart
    if $b_k = 0$ then return $i$
  end for
```

Algorithm 3 Sampling $D_{\mathbb{Z}^+,k\sigma_{\text{bin}}}$ for $k \in \mathbb{Z}$

```
Input: An integer $k \in \mathbb{Z}$ ($\sigma = k\sigma_{\text{bin}}$)
Output: An integer $z \in \mathbb{Z}^+$ according to $D^+_\sigma$
  sample $x \in \mathbb{Z}$ according to $D^+_{\sigma_{\text{bin}}}$
  sample $y \in \mathbb{Z}$ uniformly in $\{0, \ldots, k-1\}$
  $z \leftarrow kx + y$
  sample $b \leftarrow \mathcal{B}_{\exp(-y(y+2kx)/(2\sigma^2))}$
  if $\neg b$ then restart
  return $z$
```

Algorithm 4 Sampling $D_{\mathbb{Z},k\sigma_{\text{bin}}}$ for $k \in \mathbb{Z}$

```
  Generate an integer $z \leftarrow D^+_{k\sigma_{\text{bin}}}$
  if $z = 0$ restart with probability 1/2
  Generate a bit $b \leftarrow \mathcal{B}_{1/2}$ and return $(-1)^b z$
```

Algorithm 5 Sampling $\mathcal{B}_a \oslash \mathcal{B}_b$

```
  sample $A \leftarrow \mathcal{B}_a$; if $A$ then return 1
  sample $B \leftarrow \mathcal{B}_b$; if $\neg B$ then return 0
  restart
```

Final Rejection Step with $\mathcal{B}_{1/\cosh(X)}$*:* To avoid explicit computation of $1/\cosh$ for the final rejection step, the authors of [17] suggested the following algorithm: $\mathcal{B}_{1/\cosh(X)} = \mathcal{B}_{\exp(-|X|)} \oslash \left(\mathcal{B}_{1/2} \vee \mathcal{B}_{\exp(-|X|)}\right)$. It is shown that it requires at most 3 calls to $\mathcal{B}_{\exp(-|X|)}$ on average.

C.1 Complements on the KL-Divergence

We now provides the essential properties and proofs for our KL-based statistical arguments.

Lemma 4 (Additivity of Kullback-Leibler Divergence). *Let $\mathcal{P}_0, \mathcal{P}_1, \mathcal{Q}_0, \mathcal{P}_1$ be independent distributions over some countable set Ω. Then*

$$D_{KL}(\mathcal{P}_0 \times \mathcal{P}_1 \| \mathcal{Q}_0 \times \mathcal{Q}_1) = D_{KL}(\mathcal{P}_0 \| \mathcal{Q}_0) + D_{KL}(\mathcal{P}_1 \| \mathcal{Q}_1).$$

Lemma 5 (Data-Processing Inequality). *Let $\mathcal{P}, \mathcal{Q}$ be independent distributions over some countable set Ω. Then for any function f*

$$D_{KL}(f(\mathcal{P}) \| f(\mathcal{Q})) \leq D_{KL}(\mathcal{P} \| \mathcal{Q})$$

equality holds when f is injective over the support of $\mathcal{P}$.

Lemma (restatement of Lemma 2). *Let $\mathcal{P}$ and $\mathcal{Q}$ be two distribution of same countable support S. Assume that for any $i \in S$, there exists some $\delta(i) \in (0, 1/4)$ such that we have the relative error bound $|\mathcal{P}(i) - \mathcal{Q}(i)| \leq \delta(i)\mathcal{P}(i)$. Then*

$$D_{KL}(\mathcal{P} \| \mathcal{Q}) \leq 2 \sum_{i \in S} \delta(i)^2 \mathcal{P}(i).$$

Proof. We rely on second order Taylor bounds. For any $y > 0$, we have

$$\begin{aligned} \frac{d}{dx} y \ln \frac{y}{y+x} &= \frac{-y}{x+y} && = -1 \text{ at } x = 0 \\ \frac{d^2}{dx^2} y \ln \frac{y}{y+x} &= \frac{y}{(x+y)^2} && \leq \frac{2}{y} \text{ if } 4|x| \leq y \text{ since } (4/3)^2 \leq 2. \end{aligned}$$

Therefore, we conclude that for any x such that $|x| \leq \epsilon |y|$ for $\epsilon \in (0, 1/4)$,

$$|y \ln \frac{y}{y+x} + x| \leq \frac{2}{y} \epsilon^2 y^2 = 2y\epsilon^2.$$

One now sets $y = \mathcal{P}(i)$ and $x = \mathcal{Q}(i) - \mathcal{P}(i)$ and sums over $i \in S$

$$\left| \sum_{i \in S} P(i) \ln \frac{\mathcal{P}(i)}{\mathcal{Q}(i)} + (\mathcal{Q}(i) - \mathcal{P}(i)) \right| \leq 2 \sum_{i \in S} \delta(i)^2 \mathcal{P}(i).$$

Since S is the support of both $\mathcal{P}$ and $\mathcal{Q}$, we have $\sum_{i \in S} \mathcal{P}(i) = \sum_{i \in S} \mathcal{Q}(i) = 1$, therefore we conclude that

$$\sum_{i \in S} P(i) \ln \frac{\mathcal{P}(i)}{\mathcal{Q}(i)} \leq 2 \sum_{i \in S} \delta(i)^2 \mathcal{P}(i).$$

□

Lemma (restatement of Lemma 1). *Let $\mathcal{E}^{\mathcal{P}}$ being an algorithm making at most q queries an oracle sampling from a distribution $\mathcal{P}$ and outputting a bit. Let $\epsilon \geq 0$, and $\mathcal{Q}$ be a distribution $D_{KL}(\mathcal{P} \| \mathcal{Q}) \leq \epsilon$. Let x (resp. y) denote the probability that $\mathcal{E}^{\mathcal{P}}$ (resp. $\mathcal{E}^{\mathcal{Q}}$) outputs 1. Then,*

$$|x - y| \leq \frac{1}{\sqrt{2}} \sqrt{q\epsilon}.$$

Proof. By the additive and non-increasing properties of the Kullback-Leibler divergence, we have

$$D_{\mathrm{KL}}(\mathcal{B}_x, \mathcal{B}_y) = D_{\mathrm{KL}}\left(\mathcal{E}^{\mathcal{P}} \| \mathcal{E}^{\mathcal{Q}}\right) \leq D_{\mathrm{KL}}\left(\mathcal{P}^q \| \mathcal{Q}^q\right) \leq q\epsilon.$$

We conclude using Taylor bounds; from the identities

$$\begin{aligned} D_{\mathrm{KL}}(\mathcal{B}_z, \mathcal{B}_y) &= 0 \quad \text{ at } z = y \\ \frac{d}{dz} D_{\mathrm{KL}}(\mathcal{B}_z, \mathcal{B}_y) &= \ln\left(\frac{z}{y}\right) - \ln\left(\frac{1-z}{1-y}\right) = 0 = a_1 \quad \text{ at } z = y \\ \frac{d^2}{dz^2} D_{\mathrm{KL}}(\mathcal{B}_z, \mathcal{B}_y) &= \frac{1}{z(1-z)} \geq 4 = a_2 \quad \text{ when } z \in (0,1) \end{aligned}$$

we obtain that

$$D_{\mathrm{KL}}(\mathcal{B}_x, \mathcal{B}_y) \geq \frac{a_0}{0!} + \frac{a_1}{1!}(x-y) + \frac{a_2}{2!}(x-y)^2 = 2(x-y)^2$$

and conclude $2(x-y)^2 \leq q\epsilon$. □