Noname manuscript No.
(will be inserted by the editor)

Arithmetic Coding and Blinding Countermeasures for Lattice Signatures

Engineering a Side-Channel Resistant Post-Quantum Signature Scheme with Compact Signatures

Markku-Juhani O. Saarinen

Manuscript version of Tuesday 11th October, 2016

Abstract We describe new arithmetic coding techniques and side-channel blinding countermeasures for lattice-based cryptography. Using these techniques, we develop a practical, compact, and more quantum-resistant variant of the BLISS Ideal Lattice Signature Scheme. We first show how the BLISS parameters and hash-based random oracle can be modified to be more secure against quantum pre-image attacks while optimizing signature size. Arithmetic Coding offers an information theoretically optimal compression for stationary and memoryless sources, such as the discrete Gaussian distributions often present in lattice-based cryptography. We show that this technique gives better signature sizes than the previously proposed advanced Huffman-based signature compressors. We further demonstrate that arithmetic decoding from an uniform source to target distribution is also an optimal non-uniform sampling method in the sense that a minimal amount of true random bits is required. Performance of this new Binary Arithmetic Coding (BAC) sampler is comparable to other practical samplers. The same code, tables, or circuitry can be utilized for both tasks, eliminating the need for separate sampling and compression components. We then describe simple randomized blinding techniques that can be applied to anti-cyclic polynomial multiplication to mask timing- and power consumption side-channels in ring arithmetic. We further show that the Gaussian sampling process can also be blinded by a split-and-permute techniques as an effective countermeasure against side-channel attacks.

Keywords Lattice Signatures · Arithmetic coding · Side-Channel Countermeasures · Quantum Resistant Cryptography · BLISS

Markku-Juhani O. Saarinen, Dark Matter LLC, Abu Dhabi.
E-mail: markku.saarinen@darkmatter.ae
Part of this work was funded by the European Union H2020 SAFEcrypto project (grant no. 644729) while the Author was visiting CSIT, Queen's University Belfast, UK.

1 Introduction

Recent years have seen an increased focus on Lattice-based and other "quantum-resistant" public key cryptography for classical computers, which has also been fueled by official governmental warnings and endorsements [3, 4, 6, 26].

However, standardization efforts for these new algorithms are only starting [5] and post-quantum algorithms clearly have not yet reached the level of maturity that is expected from traditional cryptosystems (RSA or Elliptic Curve). Lattice-based public key algorithms have been criticized for their key and signature / ciphertext sizes, and for their lack of resistance against side-channel attacks. This paper offers a number of novel implementation techniques that address these issues.

The BLISS (Bimodal Lattice Signature Scheme) was proposed in CRYPTO 2013 by Ducas, Durmus, Lepoint, and Lyubashevsky [8] and offers some of the most compact and efficient lattice-based signatures currently available. A recent survey of practical lattice-based signature schemes [16] found BLISS to offer state-of-the-art performance and recommended it for practical use. The scheme has been implemented in FPGA hardware [29], on an 8-bit AVR target [23], and has been distributed as a part of the strongSwan IPsec VPN suite. Relevant hardware implementation techniques are also considered in [32]. We use BLISS as basis for our new lattice-based signature implementation, dubbed BLZZRD.

After describing the notation (Section 2.1) and the original BLISS scheme (Section 2.2), we move to our new discoveries:

1. **Grover's Attack on Random Oracle** can be mounted with a Quantum Computer against many signature algorithms. The attack can be applied against the random oracle component of BLISS (Section 3). We show how to modify the algorithm and its security parameters to counter this attack with small overhead (Section 3.1).

2. **Arithmetic Coding** is ideally suited for compression of stationary distributions (Section 4.1) arising in Lattice cryptography, achieving near-optimal efficiency. We show how a Binary Arithmetic Coder (BAC) can be implemented with limited precision (Section 5), and used to compress lattice signatures (Section 5.2).

3. **Gaussian sampling can be implemented with a BAC.** The same arithmetic coding tables, code, and circuitry can also be used to efficiently implement Gaussian sampling, another major operation required in Lattice cryptosystems (Section 5.4). The resulting sampler is optimal in the sense that it requires a minimum number of true random bits.

4. **Polynomial Blinding** is an efficient ring-arithmetic side-channel countermeasure. It hides the details of time or energy consumption of individual arithmetic operations via randomization. Polynomial blinding can be implemented with a minimal impact on overall performance or footprint (Section 6.1).

5. **Split Shuffled Sampling** is a general side-channel countermeasure for Gaussian samplers. Here two or more sample vectors are randomized and combined with each other, masking the properties of individual samples during the random sampling process (Section 6.2). This is an effective countermeasure against attacks such as the cache attack presented against BLISS in CHES 2016 [1].

Some efficient implementation tricks for tasks such as computation of the discrete Gaussian distribution are also described. We further take advantage of recent advances in security proof techniques, indicating a smaller precision requirement which results leads to smaller, faster implementations.

2 BLISS and BLZZRD

BLISS is an lattice public key signature algorithm proposed by Ducas, Durmus, Lepoint, and Lyubashevsky [8]. We use the more recent BLISS-B variant [7] as basis of our work. We refer the reader to these works for security analysis and other design rationale of the original proposals. Since BLZZRD is an improvement with quantum resistance, compressed signatures, and side-channel resistant countermeasures, we concentrate on these specific implementation techniques in present work.

2.1 Conventions and Notation

Arithmetic is performed in a cyclotomic ring $\mathbf{R} = \mathbb{Z}_q[x]/(x^n + 1)$ where q is a small prime. Such a ring is anti-circulant as $x^n \equiv -1 \pmod{x^n + 1}$. Arithmetic in this ring can be performed efficiently via Number Theoretic Transforms (NTT, finite field FFT) when n divides $q - 1$ and n is a power of 2.

Non-boldface f_i denotes individual coefficients of polynomial $\mathbf{f} = \sum_{i=0}^{n-1} f_i x^i$. Hence $\mathbf{f} \in \mathbf{R}$ is a ring element, while $f_i \in \mathbb{Z}_q$. We use $\mathbf{a} * \mathbf{b} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i b_j x^{i+j}$ to denote the product of polynomials $\mathbf{a}$ and $\mathbf{b}$. For anti-circulant rings of degree n we always have $f_{i+n} = -f_i$. We may therefore reduce $f_i = (-1)^{\lfloor i/n \rfloor} f_{i \bmod n}$ for all $i \in \mathbb{Z}$.

We interchangeably use polynomials as zero-indexed vectors; dot product is defined as $\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n-1} a_i b_j$, the Euclidean norms are $\|\mathbf{a}\|^2 = \mathbf{a} \cdot \mathbf{a}$, $\|\mathbf{a}\|_2 = \sqrt{\mathbf{a} \cdot \mathbf{a}}$, and sup norm $\|\mathbf{a}\|_\infty$ is the largest absolute value $\max\{|a_0|, |a_1|, \ldots, |a_{n-1}|\}$. $\lfloor x \rceil = \lfloor x + \frac{1}{2} \rfloor$ denotes the closest integer to x.

Table 1 Original parameter sets and security targets for BLISS-B from [7,8,29].

Scheme Optimized for Security target	BLISS-B0 Fun $\leq$ 60 bits	BLISS-BI Speed 128 bits	BLISS-BII Size 128 bits	BLISS-BIII Security 160 bits	BLISS-BIV Security 192 bits
Degree n	256	512	512	512	512
Modulus q	7681	12289	12289	12289	12289
Secret key density δ_1, δ_2	0.55, 0.15	0.3, 0.0	0.3, 0.0	0.42, 0.03	0.45, 0.06
Deviation σ	100	215	107	250	271
Ratio α	0.748	1.610	0.801	1.216	1.027
Repetition rate M	2.44	1.21	2.18	1.40	1.61
Challenge weight κ	12	23	23	30	39
Truncation params d, p	5, 480	10, 24	10, 24	9, 48	8, 96
Verification bound B_2	2492	12878	11074	10206	9901
Verification bound B_∞	530	2100	1563	1760	1613

A discrete Gaussian distribution with mean μ, deviation σ and variance σ^2 is denoted $\mathcal{N}_{\mathbb{Z}}(\mu, \sigma^2)$. We use exclusively zero-centered ($\mu = 0$) distributions. See Section 4.1 for definitions and further discussion.

In pseudocode we use $\wedge, \vee, \neg$ symbols to denote bitwise Boolean manipulation of two's complement integers. A binary modular reduction operator $\bmod n$ returns an integer in range $[0, n-1]$.

2.2 Description of BLISS

Our description differs somewhat from the original description (which is even more dense). We used the original reference implementation[1] and the strongSwan "production grade" implementation[2] to verify the correctness of our interpretation.

Various parameters, symbols, and security claims used in descriptions are given in Table 1. These are the parameters used by both the publications and can also be found in the implementations. These are unmodified for BLZZRD apart from κ. See Section 2.2for further information.

2.3 Key Generation

The BLISS-B key generation is perhaps the simplest part of the scheme. Algorithm 1 describes the process. The BLISS-B key generation procedure differs from original BLISS in that keys are not rejected based on an additional norm.

2.4 Creating Signatures

Algorithm 2 describes the signature generation method. Here we note that the purpose of the random oracle H is to take the input $(\mathbf{w}, \mathbf{M})$ and hash it into a vector of length n that has exactly κ ones, with other entries being zero (the vector can also be interpreted as a polynomial in the ring). The reference implementation is an ad hoc construction based on SHA-512 [13]; the strongSwan implementation now uses the MGF1 construction [17]. We use a two-stage random oracle described in Section 3.1. The BLISS-B signature method differs from BLISS in steps 8-15 of Algorithm 2; this is the Greedy Sign Choices algorithm for minimizing the norm $\|\mathbf{c} \cdot \mathbf{t}\| + \|\mathbf{c} \cdot \mathbf{u}\|$. Otherwise the algorithms are equivalent and BLISS can even be used to verify BLISS-B signatures without modification.

2.5 Verifying Signatures

The BLISS signature verification process is described in Algorithm 3. We note that verification is very fast; only two quick checks for norm bounds and a single ring multiplication is required in addition to a call to the random oracle.

[1] Original BLISS reference implementations are available from: `http://bliss.di.ens.fr/`

[2] strongSwan: `https://wiki.strongswan.org/projects/strongswan/wiki/BLISS`

Algorithm 1 BLISS-B [7,8] Key Generation. $\mathbf{f}$ and $\mathbf{g}$ are polynomials in cyclotomic ring $\mathbb{Z}_q[x]/(x^n+1)$.

$(\mathbf{f},\mathbf{g}) \leftarrow$ Uniform polynomials with exactly $\lceil \delta_1 n \rceil$ entries in $\{\pm 1\}$ and $\lceil \delta_2 n \rceil$ entries in $\{\pm 2\}$.
$\mathbf{g} \leftarrow 2\mathbf{g}+1$ — *All but g_0 are even ($\{\pm 2\}$ or $\{\pm 4\}$).*
$\mathbf{a} \leftarrow \mathbf{g}/\mathbf{f}$. Restart if $\mathbf{f}$ is not invertible. — *Quick division via Number Theoretic Transform.*

Output: Private key $(\mathbf{f},\mathbf{g})$ and public key $\mathbf{a}$.

Algorithm 2 BLISS-B [7,8] Signature algorithm. $H(\mathbf{w},\mathbf{M})$ is a deterministic random oracle that produces a vector with κ ones from input. Note the ring polynomial anti-circulant wraparound; $f_{i+n} = -f_i$.

Input: Private key $(\mathbf{f},\mathbf{g})$ and public key $\mathbf{a}=\mathbf{g}/\mathbf{f}$.
Input: Message to be signed $\mathbf{M}$.

$(\mathbf{t},\mathbf{u}) \leftarrow (\mathcal{N}_{\mathbb{Z}}^n(0,\sigma^2), \mathcal{N}_{\mathbb{Z}}^n(0,\sigma^2))$ — *All coefficients are discrete Gaussians.*
$\mathbf{v} \leftarrow \mathbf{t} * \mathbf{a}$ — *Multiplication in the ring using NTT.*
for $i = 0,1,\ldots,n-1$ **do**
  $v_i \leftarrow \big((q+1)v_i + u_i\big) \mod 2q$ — *Equivalent to adding q to v_i if odd, then u_i.*
  $w_i = \lfloor \frac{v_i}{2^d} \rceil \mod p$ — *Truncation to nearest integer, limit by $p = \lfloor 2^{d-1} q \rfloor$.*
end for
$\mathbf{c} \leftarrow H(\mathbf{w},\mathbf{M})$ — *Using $\mathbf{w}$ and $\mathbf{M}$, create a vector with κ ones.*
$(\mathbf{a},\mathbf{b}) \leftarrow (\mathbf{0},\mathbf{0})$ — *Init the "GreedySC" sign choices algorithm.*
for all $c_i = 1$ **do**
  if $\left(\sum_{j=0}^{n-1} f_j a_{i+j} + g_j b_{i+j}\right) \geq 0$ **then**
    $(\mathbf{a},\mathbf{b}) \leftarrow (\mathbf{a},\mathbf{b}) - x^i(\mathbf{f},\mathbf{g})$ — *Shift both $\mathbf{f}$ and $\mathbf{g}$ by i positions for subtraction.*
  else
    $(\mathbf{a},\mathbf{b}) \leftarrow (\mathbf{a},\mathbf{b}) + x^i(\mathbf{f},\mathbf{g})$ — *Add. Anti-circulant shift for norm minimization.*
  end if
end for
$s \leftarrow$ Random bit. — *Randomize sign.*
$(\mathbf{t},\mathbf{u}) \leftarrow (\mathbf{t},\mathbf{u}) + (-1)^s(\mathbf{a},\mathbf{b})$
Continue with probability $1/\left(M \exp\left(-\frac{\|\mathbf{a}\|^2+\|\mathbf{b}\|^2}{2\sigma^2}\right)\cosh\left(\frac{\mathbf{t}\cdot\mathbf{a}+\mathbf{u}\cdot\mathbf{b}}{\sigma^2}\right)\right)$, otherwise restart.
for $i = 0,1,\ldots,n-1$ **do**
  $z_i \leftarrow w_i - \lfloor \frac{v_i - u_i}{2^d} \rceil \mod p$ — *Create "rounding correction" for the signature.*
end for

Output: Message signature is $(\mathbf{t},\mathbf{z},\mathbf{c})$.

Algorithm 3 BLISS-B [7,8] Signature verification.

Input: Public key $\mathbf{a}$.
Input: Signature $(\mathbf{t},\mathbf{z},\mathbf{c})$ and message $\mathbf{M}$.

1: Reject if $\sqrt{\|\mathbf{t}\|^2 + 2^{2d}\|\mathbf{z}\|^2} > B_2$ — *Euclidean norm bound check.*
2: Reject if $\max(\|\mathbf{t}\|_\infty, 2^d\|\mathbf{z}\|_\infty) > B_\infty$ — *Suprenum norm bound check.*
3: $\mathbf{v} \leftarrow \mathbf{t} * \mathbf{a}$ — *Ring arithmetic using NTT* mod q.
4: **for** $i = 0,1,\ldots,n-1$ **do**
5:   $v_i \leftarrow (q+1)v_i + c_i q \mod 2q$ — *Adding q to v_i if v_i is odd, and / or $c_i = 1$.*
6:   $w_i \leftarrow \lfloor \frac{v_i}{2^d} \rceil + z_i \mod p$ — *Truncated value "corrected" with signature* $\mathbf{z}$.
7: **end for**
8: $\mathbf{c}^\star \leftarrow H(\mathbf{w},\mathbf{M})$ — *Run the oracle on $\mathbf{w}$ and $\mathbf{M}$, compare.*

Output: Accept signature if $\mathbf{c} = \mathbf{c}^\star$.

3 Security of the Random Oracle

Is there a trivial way to forge a signature $(\mathbf{t},\mathbf{z},\mathbf{c})$ that will pass signature verification (Algorithm 3) for some public key $\mathbf{a}$ and message $\mathbf{M}$?

We can easily choose arbitrary $\mathbf{t}$ and $\mathbf{z}$ that pass steps 1 and 2 (norm bounds). What is left is the matching of the oracle outputs $\mathbf{c} \stackrel{?}{=} \mathbf{c}^\star$. There are $\binom{n}{\kappa}$ possible ways for $H(\mathbf{w},\mathbf{M})$ to construct a vector with κ entries as 1, and remaining $n - \kappa$ entries being zero.

Examining the BLISS reference implementations, we find that $\mathbf{c}$ may not be transmitted or used as a vector (or a sorted list) but as an unsorted output of the oracle; in this case there are $\frac{n!}{(n-\kappa)!}$ possibilities. To get from list (ordered set) matching to vector (unordered set) matching complexity, one can simply rearrange the input $\mathbf{c}$ values in the signature to match the output from the oracle for the forged message $\mathbf{M}$ (note that this is not as easy with our revised Oracle described in Section 3.1.) These entropies (given in Table 2) appear to be consistent with security claims.

This is reasonable in classical computing where $O(2^H)$ pre-image search is required where H is the entropy of target range; however the signer herself can create a signature that matches two different messages with relative ease. BLISS signatures are therefore not collision resistant.

We note that pre-image search is one of the things that quantum computers do well. This leads us the following simple theorem:

Theorem 1 (Exhaustive Quantum Forgery) *A quantum adversary can forge an arbitrary BLISS signature with* $O\left(\sqrt{\binom{n}{\kappa}}\right)$ *complexity.*

Proof With fixed $\mathbf{t}$ and $\mathbf{c}$ in Algorithm 3 one can mount an exhaustive pre-image search to find some message $\mathbf{M}$ or small-norm vector $\Delta\mathbf{z} = \mathbf{z} - \mathbf{z}'$ that satisfies $\mathbf{c} = H(\mathbf{w} + \Delta\mathbf{z}, \mathbf{M})$. This search with Grover's Algorithm [14,15] will require $\frac{\pi}{4}\sqrt{\binom{n}{\kappa}}$ lookups. This is optimal for a quantum search [38]. □

We observe that with the suggested parameters and security levels in Tables 1 and 2, the security level against a quantum adversary falls short of the target; for example, BLISS-I and BLISS-II with claimed 128-bit security would fall with complexity of roughly 2^{66}.

3.1 New Random Oracle

With $n = 512$, a trivial encoding of $\mathbf{c}$ requires 9κ bits for transmitting the indexes. In BLZZRD the $\mathbf{c}$ vector itself (or the nonzero index set) is not transmitted. An intermediate function $H_i(\mathbf{w}, \mathbf{M})$ is first used to compute a θ-bit random hash c_θ. The intermediate then can be converted to the $\mathbf{c}$ vector with ones at exactly κ positions via H_o oracle function:

$$\mathbf{c} = H(\mathbf{w}, \mathbf{M}) = H_o\left(H_i(\mathbf{w}, \mathbf{M})\right). \tag{1}$$

Related random Oracle construction was also proposed in [36], were hash function output was extended using a PRNG. Fortunately we now have a hash standard that directly supports XOFs (eXtendable-Output Functions).

BLZZRD uses the SHA3 hash function and SHAKE XOF [12,13]. H_i is implemented with SHA3-256 or SHA3-384, depending on corresponding θ value:

$$c_\theta = H_i(\mathbf{w}, \mathbf{M}) = \mathsf{SHA3} - \theta(\ \mathbf{w} \mid \mathbf{M}\ ). \tag{2}$$

Contents of $\mathbf{w} = (\ w_0 \mid w_1 \mid \cdots \mid w_{n-1}\ )$ are encoded as network byte order 16-bit integers. To produce the $\mathbf{c}$ indexes, SHA3's SHAKE256 extendable-output function (XOF) is used. The c_θ intermediate hash is used to seed the XOF, from which an arbitrary number of 16-bit big-endian values can be extracted. These are masked to index range $[\ 0, n-1\ ]$ and rejected if already contained in $\mathbf{c}$. This process is repeated until κ ones are found for the $\mathbf{c}$ vector.

Note that SHAKE256 accepts an arbitrary-sized input "key" and has an internal chaining capacity of 512 bits. Its collision resistance as a hash function is at 256-bit security level, and this is also its security level against a quantum preimage attack; more than adequate for 128-, 160-, and 192-bit security goals.

When transmitting or comparing signatures (Algorithms 2 and 3) we may use c_θ instead of the full $\mathbf{c}$ vector or index set. This leads to more compact signatures. See Table 2 for numerical values for θ; it has been chosen to be the double of the of the security parameter to counter the Quantum Forgery Attack of Theorem 1. The new κ parameter is chosen so that $\binom{n}{\kappa} > 2^\theta$.

The increased κ has an impact on both signature creation and verification speed, and also the

Table 2 Complexity of exhaustive forgery attack on the hash-based random oracle H, based on entropy. The parameters used by new variant given below. The new variant does not communicate the $\mathbf{c}$ vector itself, but an intermediate hash that can be used to deterministically generate it. BLZZRD does not contain an equivalent of "toy" BLISS-0.

Variant / Attack	I	II	III	IV
Original BLISS n, κ	512, 23	512, 23	512, 30	512, 39
Orig. $\mathbf{c}$ entropy, bits	131.82	131.82	161.04	195.02
Orig. bits on wire ($\kappa \log_2 n$)	207	207	270	351
Orig. Grover's search security	66	66	80	96
BLZZRD n, κ	512, 58	512, 58	512, 82	512, 113
New $\mathbf{c}$ entropy, bits	256.81	256.81	320.58	385.30
New intermediate hash θ, bits	256	256	384	384
New Grover's search security	128	128	160	192

rejection ratio (Step 18 in Algorithm 2.) Experimentally the slowdown is less than 30% in all cases. This is somewhat arbitrary due to the fashion that the constant M is determined in [8].

Resistance against general Quantum attacks. Even though our modification of κ helps to secure the scheme against this particular quantum attack, it does not secure the scheme against others. We suggest other authors to suggest revised parameters for other parameters.

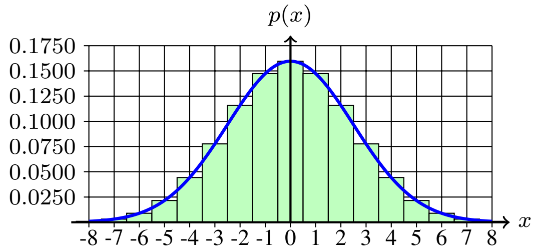

Fig. 1 The green bars illustrate the probability mass for integers $x \in \mathbb{Z}$ with a $\sigma = 2.5$ discrete Gaussian distribution (Equation 3). Blue line is the corresponding continuous probability density function.

4 Discrete Gaussians

Obtaining random numbers from a Gaussian (Normal) or other non-uniform distributions is called sampling. Cryptographically secure sampling is required by many Lattice-based cryptographic algorithms; see [9] for an overview.

4.1 Gaussian Sampling

A random sampler from a zero-centered discrete Gaussian distribution $\mathcal{N}_{\mathbb{Z}}(0, \sigma^2)$ returns integer $x \in \mathbb{Z}$ with probability given by density function $\rho_\sigma(x)$. This probability mass of discrete Gaussian distribution at x is exactly proportional to $\rho_\sigma(x) \propto \exp(-\frac{x^2}{2\sigma^2})$, where σ is a deviation parameter (See Figure 1). For $\sigma \gtrapprox 2$ we can approximate it to high precision with

$$\rho_\sigma(x) \approx \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}. \tag{3}$$

4.2 Tailcut

As $|x|$ grows, $\rho_\sigma(x)$ eventually diminishes into statistical insignificance. We may choose a cryptographic threshold parameter such as $\epsilon = 2^{-128}$ and simply ignore all $|x| > \tau$ when $2 \sum_{x>\tau} \rho_\sigma(x) \leq \epsilon$. Therefore the probability that a random x from the distribution satisfies $-\tau < x < \tau$ is greater than $1 - \epsilon$. We call τ the "tailcut parameter". It's typical numerical value for $\epsilon = 2^{-128}$ bound is related to deviation by roughly $\tau = 13.2\sigma$.

4.3 Efficient computation of density tables

Essentially all sampling algorithms require distribution density tables. Evaluation of transcendental functions (such as exp) is slow if generic algorithms are used.

We derive two parameters $b = \exp(-\frac{1}{2\sigma^2})$ and $c = 1/\sigma\sqrt{2\pi}$ from deviation σ. Equation 3 can now be written as $\rho_\sigma(x) = cb^{(x^2)}$. Since $x^2 \in \mathbb{Z}$, square-and-multiply exponentiation algorithms (analogous to those used for modular exponentiation) may be used to efficiently compute the power $b^{(x^2)}$ at arbitrary $x \in \mathbb{Z}$. Real values can be kept at fixed point range $[0,1]$ if floating point arithmetic is not available.

When tabulating the density, due to symmetry $\rho_\sigma(x) = \rho_\sigma(-x)$ we consider the sequence $t_0, t_1, t_2, ..$ with $t_i = \rho_\sigma(i)$. By observing that the ratio of consecutive values satisfies $\frac{t_{i+1}}{t_i} = u_i = b^{2i+1}$ we arrive at the following recurrence:

$$\begin{array}{lll} t_0 = c & u_0 = b & \text{Initialize.} \\ t_i = t_{i-1}u_{i-1} & u_i = b^2 u_{i-1} & \text{For } i \geq 1. \end{array} \tag{4}$$

The algorithm of Equation 4 computes consecutive discrete Gaussian density function values with only two multiplications per point (b^2 stays constant). This new technique makes the table initialization process much faster, a great advantage for limited-resource implementations.

4.4 Gaussian sampling algorithms

We first analyze the "inversion sampler" which is one way of using uniform random bits to select an element from the target distribution by inverting its cumulative distribution. Since $\mathcal{N}_\mathbb{Z}(0, \sigma^2)$ is symmetric we can define a cumulative sequence $s_i = \sum_{x=-i}^{i} \rho_\sigma(x)$. It can be computed as an extension of sequences of Equation 4 via $s_0 = t_0$ and $s_i = s_{i-1} + 2t_i$. Clearly the sum of all probabilities converges to $s_\infty = 1$.

For cryptographic applications we can assume a source of unbiased random bits $z_i \in \{0, 1\}$. A sequence of n random bits can be viewed as real-valued $z \in [0, 1]$ via binary fraction $z = 0.z_1z_2z_3 \ldots z_n$. When n is finite, $z_1, z_2, ..z_n$ only defines a range of size 2^{-n}. For uniformly ($n = \infty$) random $z \in [0, 1]$ the corresponding sampled integer x can be derived with additional random sign bit via

$$x = \begin{cases} 0 & \text{if } z < s_0, \\ \pm i & \text{if } s_{i-1} \leq z < s_i \text{ for } i \geq 1. \end{cases} \tag{5}$$

This corresponds to "inversion sampling", and can be implemented via a binary search into monotonically increasing table s_i by first randomizing a large number of bits to create a high-precision real number z.

A wide variety of sampling methods such as Inversion Sampling [27], Knuth-Yao Sampling [19], The Ziggurat Method [2,10,24,25], Kahn-Karney Sampling [18], and "Bernoulli" sampling [8] have been proposed for lattice cryptography.

5 Arithmetic Coding

Arithmetic Coding is a classical data compression technique [30], notable for its optimality under certain conditions. Information theory tells us that the average entropy (in bits/symbol) of a stationary and memoryless discrete source such as $\Omega = \mathcal{N}_\mathbb{Z}(0, \sigma^2)$ is

$$H(\Omega) = -\sum_{x=-\infty}^{\infty} \rho_\sigma(x) \log_2 \rho_\sigma(x). \tag{6}$$

Equation 6 gives us a lower bound for bits required to represent any discrete Gaussian sequence; this is also the expected number of random entropy bits required in sampling.

Theorem 2 *Arithmetic coding is optimal for a static and memoryless discrete source Ω in the sense that it is able to encode n symbols from such a source into $nH(\Omega) + O(1)$ data bits.*

Proof See Section 1.5 of [34]. □

5.1 Implementing a Binary Arithmetic Coder

Arithmetic Coding and Decoding can be implemented in many ways. Binary Arithmetic Code (BAC) [22] is one such approach. For BLZZRD reference code, we implemented a BAC encoder and decoder for a static (non-dynamic) distribution. Range scaling details of the new codec were inspired by [34], even though that implementation isn't a BAC. See [37] for a classic implementation of arithmetic coding.

Table 3 The internal variables used by main BAC routines. Note that interpretation of "input" and "output" differs for encoding and decoding, as does the size of a word.

n	Size of the decoded vector of words.
b, l	The interval is $[b, b + l - 1]$. Can be implemented with P-bit arithmetic.
c	Scaled binary division point, P bits.
x	Current input byte or word.
ibit	Input bit counter.
ibuf[ ]	Input vector. Indexed by iptr as ibuf[iptr].
owrd	Stores an output word and carry bits; must have a type large enough for additional carry bits.
obit	Output bit counter.
obuf[ ]	Output vector. Indexed by olen as obuf[olen].

Our implementation is in C language, uses 64-bit fixed point precision, and is highly portable. Importantly, we pose no specific limits on buffer carry bit back-propagation, such as "bit stuffing" used in Q-Coder hardware architecture [28]. Full source is distributed together with the reference implementation (see Section 7). The implementation is only about 250 lines.

Table 3 gives the variable conventions used in the implementation. Constants used to define the BLZZRD BAC are as follows:

- P Bit precision used by the implementation. We used $P = 64$.
- D Alphabet size in bits. For discrete Gaussians, we need $2^D > 2\tau$.

With a BAC, an alphabet of size 2^D can be viewed as a binary tree of depth D. For each one of the D binary decisions in encoding or decoding, a single comparison is required.

The input frequencies can be arbitrarily scaled "counts". The probability of occurrence of x is

$$\mathsf{Pr}(x) = \frac{\mathsf{freq}[x]}{\sum_{i=0}^{2^D-1} \mathsf{freq}[i]}. \quad (7)$$

Given a table of frequencies $\mathsf{freq}[2^D]$, function BuildDist (Algorithm 4) creates a corresponding scaled BAC distribution table "tree" $\mathsf{dist}[2^D]$ that is used by both encoding and decoding functions.

The encoding routine AriEncode (Algorithm 6) requires a helper function StoreWithCarry (Algorithm 5) to propagate carries. Arithmetic coding routines generally require an output buffer as these carries cannot be reliably predicted. The decoding routine AriDecode (Algorithm 8) utilizes a helper function ShiftGetBit (Algorithm 7) to get new bits into the working variable x.

The coding and decoding procedure is illustrated by Figure 2. Exhaustive testing with various distributions was performed to verify the correct operation of the codec.

5.2 Compressing Signatures

We have implemented arithmetic coding compression and decompression for the $\mathbf{t}$ component of signatures generated with new BLZZRD parameters. The $\mathbf{z}$ vector was also modelled as discrete Gaussian with a small σ. See Table 4 for our parameters and experimental results. Rather than transmitting the $\mathbf{c}$ vector indexes, an intermediate hash of θ bits is transmitted. The last line gives the experimental average signature sizes, including overhead required to encode parameter and array sizes.

5.3 Comparison with a Huffman Code

The extended version of [29] describes an advanced "block" Huffman encoding technique for the BLISS-1 parameter set. A similar technique is also implemented in the strongSwan project.

The codec achieves good performance (for a Huffman code) by assigning codes to quadruplets $\left(\lfloor\frac{\mathrm{abs}(t_{2i})}{2^8}\rfloor, \lfloor\frac{\mathrm{abs}(t_{2i+1})}{2^8}\rfloor, z_{2i}, z_{2i+1}\right)$ rather than individual t_i and z_i values. The lower 8 bits of t_i values is stored as they are, and up to four sign bits are stored for nonzero entries.

We note that the 64-entry Huffman tree given in Appendix B of [29] for BLISS-I can only encode values in range $-1023 \leq t_i \leq 1023$. Since t_i comes from Gaussian distribution with $\sigma = 215$, there is a probability of $P = 1 - \sum_{x=-1023}^{1023} \rho_\sigma(x) \approx 2^{-18.98}$ that individual t_i values will overflow and a probability of $1 - (1 - P)^n \approx 2^{-9.98}$ (or roughly one in thousand) that an individual $\mathbf{t}$ signature vector cannot be compressed using the proposed code.

Algorithm 4 BuildDist. Create a scaled BAC distribution tree from alphabet frequency counts.

Input: $\mathsf{freq}[2^D]$, a frequency count for alphabet.

- **for** $i = 2^{D-1}, \ldots 8, 4, 2, 1$ **do**
  - $j \leftarrow 0$
  - **while** $j < 2^D$ **do**
    - $c_0 \leftarrow \sum_{k=j}^{j+i-1} \mathsf{freq}[k]$ — *"Left" tree branch count for 0.*
    - $c_1 \leftarrow \sum_{k=i+j}^{j+2i-1} \mathsf{freq}[k]$ — *"Right" branch count for 1.*
    - $\mathsf{dist}[i+j] \leftarrow \left\lfloor \frac{2^P c_0}{c_0+c_1} \right\rfloor$ — *Division point scaled to* $[0, 2^P - 1]$.
    - $j \leftarrow j + 2i$ — *Big step.*
  - **end while**
- **end for**

Output: $\mathsf{dist}[2^D]$, a BAC distribution tree.

Algorithm 5 StoreWithCarry(b). Stores the highest bit of b to variable owrd. If the byte is full, it is stored to obuf[olen], while also adjusting the carry if necessary.

- $\mathsf{owrd} \leftarrow 2\,\mathsf{owrd} + \left\lfloor \frac{b}{2^{P-1}} \right\rfloor$ — *Add highest bit of b to output, shift left*
- $\mathsf{obit} \leftarrow \mathsf{obit} + 1$ — *Output bit counter.*
- **if** $\mathsf{obit} \geq 8$ **then**
  - $\mathsf{obuf}[\mathsf{olen}] \leftarrow \mathsf{owrd} \bmod 2^8$
  - $i \leftarrow \mathsf{olen}$ — *Index for carry propagation.*
  - **while** $(\mathsf{owrd} \geq 2^8)$ **and** $(i > 0)$ **do**
    - $i \leftarrow i - 1$ — *Proceed left.*
    - $\mathsf{owrd} \leftarrow \left\lfloor \frac{\mathsf{owrd}}{2^8} \right\rfloor + \mathsf{obuf}[i]$
    - $\mathsf{obuf}[i] \leftarrow \mathsf{owrd} \bmod 2^8$ — *Add carry to bytes until done.*
  - **end while**
  - $\mathsf{obit} \leftarrow 0,\ \mathsf{owrd} \leftarrow 0,\ \mathsf{olen} \leftarrow \mathsf{olen} + 1$ — *Full byte output.*
- **end if**

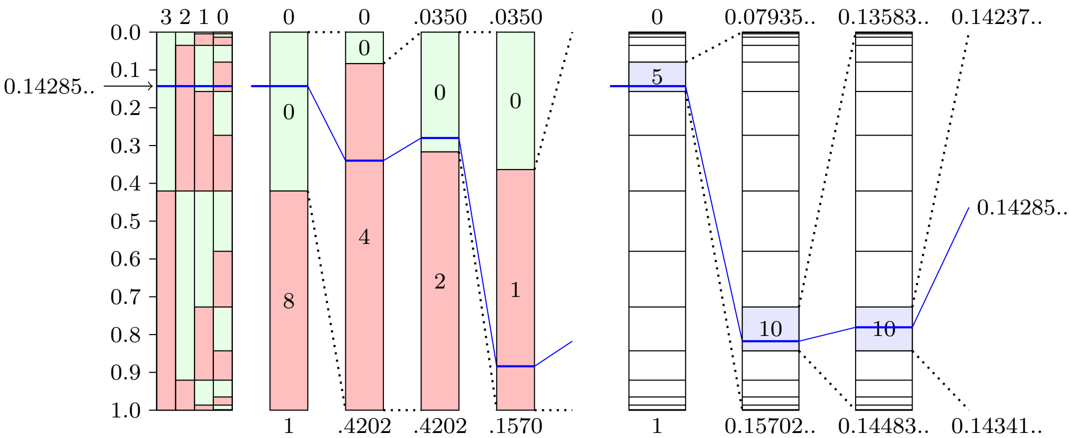

Fig. 2 In this toy example we are encoding discrete Gaussian with $\sigma = 2.5$ with tail cutting applied at ≈ 0.001 level; the range is $-7 \ldots 7$ and the integers fit 4 bits. For convenience the distribution is centered at 8 average (high bit flip). Left side shows the operation of binary sample decoding; this corresponds to binary search within the cumulative distribution. On each step either min or max bound is set to divisor value. Only one scaling multiplication and comparison is required. Four bits yields the output $0 + 4 + 0 + 1 = 5$. Right side illustrates decoding of multiple samples. Output $(5, 10, 10)$ corresponds to $(-3, 2, 2)$ when adjusted to zero centre. After decoding 3 samples the bounds $\mathsf{min} = 0.14237..{}_{10} = 0.0010010001110..{}_2$ and $\mathsf{max} = 0.14285..{}_{10} = 0.0010010010010..{}_2$ already match in 8 binary fraction bits. Both bounds and the input fraction can be shifted left, discarding the integer bits.

Algorithm 6 AriEncode. Perform Arithmetic Coding on a sequence of input words.

Input: $\mathsf{ibuf}[n]$: A sequence of n input words from the alphabet.
Input: $\mathsf{dist}[2^D]$: A BAC distribution tree table constructed with BuildDist.

$b \leftarrow 0,\ l \leftarrow 2^P - 1$ — *Initial base b and range l.*
$\mathsf{owrd} \leftarrow 0,\ \mathsf{obit} \leftarrow 0,\ \mathsf{olen} \leftarrow 0$ — *Initialize* StoreWithCarry *output variables.*
for $\mathsf{iptr} = 0, 1, 2, \ldots n-1$ **do**
  $x = \mathsf{ibuf}[\mathsf{iptr}]$. — *Get a new input word x.*
  for $\mathsf{ibit} = D-1 \ldots, 2, 1, 0$ **do**
    $c \leftarrow \mathsf{dist}[(x \wedge (2^P - 2^{\mathsf{ibit}})) \vee 2^{\mathsf{ibit}}]$ — *Get centre point from masked BAC freq. table.*
    $c \leftarrow \lfloor \frac{lc}{2^P} \rfloor$ — *Scale c to $[0, l[$.*
    if $(x \wedge 2^{\mathsf{ibit}}) = 0$ **then**
      $l \leftarrow c$ — *Input bit is 0: select lower part.*
    else
      $b \leftarrow b + c,\ l \leftarrow l - c$ — *Input bit is 1: select higher part.*
      if $b > 2^P$ **then**
        $b \leftarrow b - 2^P,\ \mathsf{owrd} \leftarrow \mathsf{owrd} + 1$ — *Carry. Note:* owrd*'s type should not overflow.*
      end if
    end if
    while $l < 2^{P-1}$ **do**
      StoreWithCarry(b) — *Store the highest bit of b.*
      $b \leftarrow 2b \bmod 2^P,\ l \leftarrow 2l \bmod 2^P$ — *Shift bounds left.*
    end while
  end for
end for
while $b > 0$ **or** $\mathsf{owrd} \neq 0$ **do**
  StoreWithCarry(b) — *Purge remaining nonzero bits from b.*
  $b \leftarrow 2b \bmod 2^P$ — *Shift b bound left.*
end while

Output: $\mathsf{obuf}[\mathsf{olen}]$, a byte vector with the encoded data.

Algorithm 7 ShiftGetBit(x). Shifts x left and fetches a new bit from ibuf byte array.

$\mathsf{ibit} \leftarrow \mathsf{ibit} - 1$ — *Decrease number of available bits.*
if $\mathsf{ibit} < 0$ **then**
  if $\mathsf{iptr} < \mathsf{ilen}$ **then**
    $\mathsf{iwrd} \leftarrow \mathsf{ibuf}[\mathsf{iptr}]$ — *Get a new byte.*
    $\mathsf{iptr} \leftarrow \mathsf{iptr} + 1$
  else
    $\mathsf{iwrd} \leftarrow 0$ — *Assume zeros.*
  end if
  $\mathsf{ibit} \leftarrow 7$ — *Manage input pointers.*
end if
$x \leftarrow (2x \mod 2^P) + (\lfloor \frac{\mathsf{iwrd}}{2^{\mathsf{ibit}}} \rfloor \mod 2)$ — *Shift x left, add a bit from* iwrd *at the bottom.*

Ignoring these frequent overflow conditions, our exhaustive testing has showed that the block Huffman codec compresses $(\mathbf{t}, \mathbf{z})$ to an average of 5650.1 bits/signature. Our arithmetic coder achieves a slightly better result, 5565.2 bits/signature. A non-customized Huffman coding technique would yield significantly inferior results.

Furthermore, [29] states that "$\mathbf{c}$ does not contain any easily removable redundancy", and therefore $\kappa \log_2(n) = 207$ bits are used. This is not strictly accurate as our new oracle (Section 3.1) requires a smaller number of bits. In case of the original BLISS-I security parameters only $\theta = 128$ bits would be required, a 38% drop. For BLZZRD-I, we have $\kappa = 58$ and $\theta = 256$, indicating a re-

Algorithm 8 AriDecode. Perform Arithmetic Decoding on a sequence of input bytes.

Input: ibuf[ilen]: A sequence of input bytes.
Input: dist[2^D]: A BAC distribution tree table constructed with BuildDist.

```
b ← 0, l ← 2^P − 1, x ← 0                    Initial base b, range l, and input variable x.
iwrd ← 0, iptr ← 0, ibit ← 0                 Initialize input pointers.
for i = 1, 2, ... P do
    ShiftGetBit(x)                           Fill up x with P input bits.
end for
for optr = 0, 1, 2, ... n − 1 do
    owrd ← 0                                 Zero this output word.
    for obit = D − 1 ..., 2, 1, 0 do
        c ← dist[(owrd ∧ (2^P − 2^obit)) ∨ 2^obit]   Get centre point from masked BAC freq. table.
        c ← ⌊lc / 2^P⌋                       Scale c to [0, l[.
        if x − b < c then
            l ← c                            Output bit 0: Select lower part, reduce range.
        else
            b ← b + c                        Output bit 1: Select upper part.
            l ← l − c                        Reduce range.
            owrd ← owrd ∨ 2^obit             Store bit.
        end if
        while l < 2^(P−1) do
            ShiftGetBit(x)                   Add an input bit to x.
            b ← 2b mod 2^P, l ← 2l           Shift bounds left.
        end while
    end for
    obuf[optr] ← owrd                        Store the output word.
end for
```

Output: obuf[n], a word vector with the decoded data.

Table 4 BLZZRD signature compression using Binary Arithmetic Coding.

Parameter / Variant	BLZZRD-I	BLZZRD-II	BLZZRD-III	BLZZRD-IV
Security target, bits	128	128	160	192
Revised κ	58	58	82	113
Arith. code σ for $\mathbf{t}$	215	107	250	271
Arith. code σ for $\mathbf{z}$	0.4792	0.4352	0.646	1.136
Intermediate hash θ	256	256	384	384
Avg. compressed $\mathbf{t}$	5079.2	4563.8	5190.6	5250.2
Avg. compressed $\mathbf{z}$	486.0	326.9	779.3	1206.1
Encoding overhead	32	32	32	32
Signature average, bits	**5843.2**	**5178.7**	**6385.9**	**6872.3**

duction to less than half of the corresponding plain BLISS encoding size.

Arithmetic Coding is somewhat slower than Huffman coding, but as can be observed in Table 7, the compression is not a bottleneck in the operation. As indicated by Theorem 2, Arithmetic Coding is the most efficient way to compress signatures with elements from non-uniform but static distribution.

5.4 Sampling with an Arithmetic Decoder

An interesting consequence of the information theoretic optimality of an arithmetic coder is that when the decoder is fed uniform random bits, it will output random samples in the desired target distribution. Therefore the same code, circuitry, and tables that are used to compress signatures can double as a random sampler as well. We have utilized Algorithm 8 as a Gaussian sampler in this fashion. With precision $P = 64$ we may reasonably expect to reach a security level close to 128-bit level, based on the "Valiant-Valiant" sampling Theorem and conjectures [33].

Table 5 gives performance characteristics of our initial implementation. The binary search CDF sampler used tables of size 2^{12} and an additional sign bit, whereas the BAC sampler used tables of size 2^{13} without special coding for sign bits. The random usage was calculated from average number of random bytes used to sample vectors of length $n = 512$. The table also includes numbers for a blinded sampler (Section 6.2). Precision of $P = 64$ was used in all cases and the speeds are comparable. We observe that the main advantage of the new BAC sampler is that it uses a very small number of true random bits; the actual random usage is typically within 1.3% of the theoretical minimum for vector sizes used on Lattice cryptography in experiments with our implementation.

By comparison, The Knuth-Yao sampler can be also be expected to use a small number of bits, shown to average $H+2$ bits per sample [19]. However, Knuth-Yao always uses a *whole number of bits per sample* whereas a BAC sampler can be utilized to create vectors of samples where bits of entropy are "shared" between individual samples. This elementary property directly follows from Theorem 2. In many ways the Knuth-Yao sampler compares to an BAC sampler like Huffman Codes compare to Arithmetic Compression.

For BLZZRD parameters the actual randomness usage of Knuth-Yao is about 16% higher than the BAC sampler for vectors of length $n = 512$. Knuth-Yao of [31] is suitable primarily for small deviations used in Ring-LWE encryption ($\sigma \approx 3$), not for larger σ required by BLISS / BLZZRD.

6 Randomized Side-Channel Countermeasures for Ideal Lattices

Blinding is a standard countermeasure against both the timing attack [20] and emissions-based attacks such as Differential Power Analysis [21] for traditional public key cryptosystems. Blinding countermeasures add randomness to private key operations, making determination of secrets from observations more difficult for the attacker. In case of RSA, there are two kinds of blinding, base blinding and exponent blinding. In case of ECC, scalar blinding can be used.

Examining private key operations in lattice cryptosystems, we note that there are two main components that can introduce side-channel vulnerabilities to the implementation; ring polynomial arithmetic and Gaussian sampling. We have implemented analogous blinding countermeasures against both. For BLISS/BLZZRD (Algorithm 2), special note should be made to implement the GreedySC component using side-channel resistant techniques as well. An advanced cache attack against a BLISS implementation was described in CHES 2016 [1], which attacked the samplers of a reference implementation.

6.1 Blinded Polynomial Multiplication

Basic arithmetic in $\mathbb{Z}_q$ and especially the modular reduction by small prime q may easily introduce emissions to an ideal lattice cryptography implementation. For example, the arithmetic operations described for a low-resource Ring-LWE implementation in [23] contain a number of data-conditional execution cases.

The simplest form of polynomial blinding is to multiply polynomials with random constants $a, b \in \mathbf{Z}_q$. This can be done in regular or NTT domain; the results are equivalent. One can set $a = b^{-1}$ or multiply the result by $c = (ab)^{-1}$:

$$\mathbf{h} = a\mathbf{f} * b\mathbf{g}$$
$$\mathbf{f} * \mathbf{g} = (ab)^{-1}\mathbf{h}.$$

Note that anti-cyclic NTT multiplication requires each polynomial to be "pre-processed" by multiplying entries with tabulated roots of unity ω^i anyway. Therefore by choosing $a = \omega^i, a = \omega^j$, this type of blinding can be done with virtually no extra cost. The normalization constant becomes $c = \omega^{-i-j}$.

An another type of blinding of anti-cyclic polynomial multiplication can be achieved via circularly "shifting" the polynomials. As noted in Section 2.1, we may write a polynomial as $\mathbf{f} =$

Table 5 Sampling with an Arithmetic Coder vs a simple inverse CDF sampler on a Core i7-5600U @ 2.6 GHz. Note that $P = 64$ precision was also used for BLZZRD-III and BLZZRD-IV, even though it is really not sufficient above 128-bit security level. The same precision was used for all samplers.

Parameter / Variant	BLZZRD-I	BLZZRD-II	BLZZRD-III	BLZZRD-IV
Deviation σ	215	107	250	271
Entropy $H\left(\mathcal{N}_{\mathbb{Z}}(0,\sigma^2)\right)$	9.7953	8.7886	10.013	10.129
BAC Sampler				
Random bits / Sample	9.9261	8.9194	10.144	10.260
Samples / Second	8,900,000	9,600,000	8,800,000	8,800,000
CDF Sampler				
Random bits / Sample	64	64	64	64
Samples / Second	8,672,000	8,408,000	8,704,000	8,440,000
Blinded CDF ($m = 2$).				
Random bits / Sample	128	128	128	128
Samples / Second	3,400,000	3,400,000	3.400,000	3,400,000

Algorithm 9 $\mathbf{PolyBlind}(\mathbf{v}, s, c)$ returns vector $\mathbf{v}$ of length n shifted by $s, 0 \le s < n$ positions and multiplied with a constant c.

```
1: for i = 0,1,... n − s − 1 do
2:     v'_i ← (c v_(i+s)) mod q
3: end for
4: for i = n − s,... n − 1 do
5:     v'_i ← (q − c v_(i+s−n)) mod q
6: end for
Output: Blinded vector v'.
```

$\sum_{i=0}^{n-1} f_i x^i$. Shifting by j positions is equivalent to computing

$$x^j \mathbf{f} = \sum_{i=0}^{n-1} f_i x^{i+j} = \sum_{i=0}^{n-1} f_{i-j} x^i. \tag{8}$$

Here the coefficients are therefore simply rotated in anti-cyclic fashion. Both constant multiplication and shifting by c are captured by function PolyBlind (Algorithm 9).

Algorithm 9, PolyBlind is very fast. The inverse operation $\mathsf{PolyBlind}'(\mathbf{v}, -s, c^{-1})$, distinguished by a negative shift value s, is equally easy to construct. With that function, we have

$$\mathsf{PolyBlind}'(\mathsf{PolyBlind}(\mathbf{v}, s, c), -s, c^{-1}) = \mathbf{v} \tag{9}$$

Due to isometries of the anti-circulant ring, we can use a total of four blinding parameters: a, b (constants) and r, s (shift values) in the blinded scheme to compute the polynomial product $\mathbf{f} * \mathbf{g}$:

$$\mathbf{f}' = \mathsf{PolyBlind}(\mathbf{f}, r, a) \;\; 0 \le r < n, 0 < a < q$$
$$\mathbf{g}' = \mathsf{PolyBlind}(\mathbf{g}, s, b) \;\; 0 \le s < n, 0 < b < q$$
$$\mathbf{h}' = \mathbf{f}' * \mathbf{g}'$$
$$\mathbf{f} * \mathbf{g} = \mathsf{PolyBlind}'\left(\mathbf{h}', -(r+s), (ab)^{-1}\right).$$

One may choose a and b from tabulated roots of unity; $a = \omega^i, a = \omega^j$ and avoid computing the inverse since $(ab)^{-1} = \omega^{-(i+j)}$. This type of blinding has a relatively small performance penalty. If roots of unity are used as constants, the total amount of "noise" entropy introduced is $4\log_2(n) = 36$ bits.

6.2 Blinded Gaussian Sampling

We note that the BLISS/BLZZRD algorithm always samples vectors of n variables at once. We define a function $\mathsf{VectorSample}(n, \sigma) = \mathcal{N}_{\mathbb{Z}}^n(0, \sigma^2)$ that produces a vector of n samples from discrete Gaussian with deviation parameter σ. Step 1 of signing process (Algorithm 2) can be written as

$$(\mathbf{t}, \mathbf{u}) \leftarrow (\mathsf{VectorSample}(n, \sigma), \mathsf{VectorSample}(n, \sigma)).$$

If VectorSample is implemented naively as a loop, emissions from the implementation may reveal information about the random numbers being sampled and attacker may determine which elements of the random vector have specific features, as is done in the cache attack presented in CHES

Table 6 Parameters for $Z \approx X + kY$ split samplers, where the target distribution is $\mathcal{N}_{\mathbb{Z}}(0, \sigma)$, k is a small integer constant, and X and Y have distribution $\mathcal{N}_{\mathbb{Z}}(0, \sigma')$ with $\sigma' = \frac{1}{\sigma}\sqrt{1+k^2}$. This is a convolution approximation and ϵ gives the statistical distance (total variation distance) to the target distribution. Here we want $\epsilon^2 < \frac{1}{n}$ where n is a security parameter [33, 35].

Parameter / Variant	BLZZRD-I	BLZZRD-II	BLZZRD-III	BLZZRD-IV
Deviation σ	215	107	250	271
Constant k	11	8	12	12
Split deviation σ'	17.8548	13.2717	20.7614	22.5053
Distance ϵ	$2^{-89.1}$	$2^{-76.8}$	$2^{-85.3}$	$2^{-100.1}$

2016 [1]. This may be alleviated to some degree by using $\mathsf{VectorShuffle}(\mathsf{VectorSample}(n, \sigma))$, which is just a random shuffle of the Gaussian sample vector.

From probability theory we know that the sum of any two discrete Gaussian distributions is a discrete Gaussian distribution. More precisely, variances (which is the square of deviation) are additive. Let X and Y have distributions $\mathcal{N}_{\mathbb{Z}}(\mu_X, \sigma_X^2)$ and $\mathcal{N}_{\mathbb{Z}}(\mu_Y, \sigma_Y^2)$, respectively. Then their sum $X + Y$ has distribution $\mathcal{N}_{\mathbb{Z}}(\mu_X + \mu_Y, \sigma_X^2 + \sigma_Y^2)$. With zero-centered distributions the average does not change, but the resulting deviation will be $\sigma_{X+Y} = \sqrt{\sigma_X^2 + \sigma_Y^2}$. By induction, this can be generalized to more variables. We use this property to create a more secure Gaussian vector sampler. Algorithm 10 constructs the target distribution from a sum of random samples from $\sigma' = \frac{1}{\sqrt{m}}\sigma$.

Algorithm 10 $\mathbf{VectorBlindSample}(n, m, \sigma)$ returns $\mathcal{N}_{\mathbb{Z}}^n(0, \sigma^2)$ using m iterations of random blinding.

1: $\mathbf{x} \leftarrow \mathbf{0}$
2: **for** $i = 1, 2, \ldots, m$ **do**
3: $\quad \mathbf{x} \leftarrow \mathbf{x} + \mathcal{N}_{\mathbb{Z}}^n\left(0, (\frac{1}{\sqrt{m}}\sigma)^2\right)$
4: $\quad \mathbf{x} \leftarrow \mathsf{VectorShuffle}(\mathbf{x})$
5: **end for**

Output: Sampled vector $\mathbf{x}$.

As pointed out in [29], one can also choose $\sigma' = \frac{\sigma}{\sqrt{1+k^2}}$ and construct the final distribution as $Z = X + kY$. The convolution parameter k must be carefully chosen so that the statistical distance to the actual target distribution is not too great. Table 6 gives some appropriate parameters to use. In the resulting construction all elements of the second vector are simply multiplied by the integer constant k:

$$\mathbf{v} = \mathsf{VectorShuffle}(\mathsf{VectorSample}(n, \sigma')) + \\ k * \mathsf{VectorShuffle}(\mathsf{VectorSample}(n, \sigma')).$$

The algorithm has a performance impediment factor of m or less. However, fast "leakier" algorithms may be used. More significantly, the required tables are much smaller. For example, a CDF table of only 256 entries gives a secure tail-cut at $\tau = 14.34\sigma$ for BLZZRD-I with parameters of Table 6.

The amount of noise entropy introduced by the permutations is technically in thousands of bits. Even though this does not directly translate to attack complexity, this countermeasure makes emissions- and cache-based attacks on the sampler significantly harder, defeating the BLISS attack of CHES 2016 [1]. It may be possible to adopt the attack to counter some blinding parameters, but probably not all. The countermeasure parameter m can be increased without significantly modifying the implementation or breaking compatibility.

7 Implementation

An experimental self-contained reference implementation that contains most of the features described in this work is available at `https://github.com/mjosaarinen/blzzrd`. Table 7 summarizes its performance on a 2015 laptop computer (Intel Core i7-4870HQ @ 2.50GHz).

Table 7 Breakdown of time consumption by different BLZZRD elementary operations in the plain C reference implementation. We observe that blinding countermeasures for sampling and arithmetic approximately double the time required for creating signatures. CDF sampler was used for these tests.

Primitive	BLZZRD-I	BLZZRD-II	BLZZRD-III	BLZZRD-IV
Generate a Key Pair	0.251 ms	0.250 ms	0.274 ms	0.285 ms
Sign a Message	0.254 ms	0.533 ms	0.326 ms	0.409 ms
.. with Countermeasures	0.498 ms	1.042 ms	0.613 ms	0.757 ms
Verify a Signature	0.064 ms	0.063 ms	0.068 ms	0.069 ms
Compress a Signature	0.044 ms	0.041 ms	0.046 ms	0.049 ms
Uncompress a Signature	0.048 ms	0.042 ms	0.050 ms	0.053 ms

On the same system, OpenSSL's (version 1.0.2g) optimized implementation of ECDSA with the NIST P-256 curve [11] required 0.039ms for creating a signature and 0.097ms for verification. All variants of Reference BLZZRD have about 30% faster signature verification when compared to this implementation of NIST P-256 ECDSA.

NIST P-256 is the fastest curve implemented by OpenSSL; BLZZRD outperforms ECDSA in signature verification with a high margin. For larger curves (409 bits or more), BLZZRD signature generation was also faster.

We note that (unlike OpenSSL benchmark) our BLZZRD reference implementation has no assembly language optimizations and was generally written for readability and maintainability rather than for raw speed. It is also the first implementation with side-channel countermeasures, so direct performance comparisons to other Lattice-based schemes are difficult.

8 Conclusions

We have offered a number of techniques that can be used to improve the security and performance of lattice public key algorithms. Using these techniques, we constructed BLZZRD, a practical, evolutionary variant of the BLISS-B signature scheme.

We have described a direct Grover's attack on the random oracle component of the BLISS signature scheme. In order to counter this attack, a new hash-based random oracle is proposed, with adjusted κ parameter and an "intermediate" hash which reduces signature size. It is currently not easy to estimate the security of all components of BLISS/BLZZRD against quantum attacks, but the new random oracle parameters are consistent with suggested quantum-resistant key sizes for symmetric ciphers [6].

Many algorithms in Lattice cryptography utilize discrete Gaussian variables in signatures and ciphertexts. We show that arithmetic coding can be used to compress these quantities in essentially optimal fashion. We describe an efficient Binary Arithmetic Coder (BAC) that produces smaller signatures than previous compressors. A somewhat surprising finding is that arithmetic coders can be adopted as non-uniform (Gaussian) Samplers that have comparable performance characteristics (millions of samples per second) to other sampling algorithms, but require only a (optimally) small amount of true random bits.

Standard "blinding" side-channel countermeasures for public key algorithms introduce randomness (noise) into private key operations, thus making determination of secrets more difficult. In ideal lattice cryptography, the main components used by private key operations typically are ring arithmetic and non-uniform (Gaussian) random sampling.

For ring arithmetic, we introduce "blinded polynomial multiplication", a simple randomization technique based of constant multiplication and rotation of polynomials. This technique is cheap to implement and utilizes the specific isometries of the types of anti-circulant rings often used in lattice cryptography.

For Gaussian sampling, we note that lattice cryptography algorithms typically require vectors rather than individual samples. We show that sampling processes can be blinded by shuffling these vectors and combining multiple Gaussian distributions with each other. This also results in a smaller implementation footprint since the size of required CDF tables becomes smaller. The general countermeasure is effective against side-channel attacks such as the cache attack recently described against BLISS.

References

1. Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, Gauss, and reload – a cache attack on the BLISS lattice-based signature scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 323–345. Springer, 2016. URL: https://eprint.iacr.org/2016/300, doi:10.1007/978-3-662-53140-2_16.
2. Johannes Buchmann, Daniel Cabarcas, Florian Göpfert, Andreas Hülsing, and Patrick Weiden. Discrete ziggurat: A time-memory trade-off for sampling from a Gaussian distribution over the integers. In Tanja Lange, Kristin Lauter, and Petr Lisoněk, editors, *SAC 2013*, volume 8282 of *LNCS*, pages 402–417. Springer, 2014. Extended version available as IACR ePrint 2014/510. URL: https://eprint.iacr.org/2013/510, doi:10.1007/978-3-662-43414-7_20.
3. Peter Campbell, Michael Groves, and Dan Shepherd. Soliloquy: A cautionary tale. ETSI 2nd Quantum-Safe Crypto Workshop in partnership with the IQC, October 2014. URL: https://docbox.etsi.org/Workshop/2014/201410_CRYPTO/S07_Systems_and_Attacks/S07_Groves_Annex.pdf.
4. CESG. Quantum key distribution: A CESG white paper, February 2016. URL: https://www.cesg.gov.uk/white-papers/quantum-key-distribution.
5. Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. Report on post-quantum cryptography. NISTIR 8105, April 2016. URL: http://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8105.pdf, doi:10.6028/NIST.IR.8105.
6. CNSS. Use of public standards for the secure sharing of information among national security systems. Committee on National Security Systems: CNSS Advisory Memorandum, Information Assurance 02-15, July 2015.
7. Léo Ducas. Accelerating Bliss: the geometry of ternary polynomials. IACR ePrint 2014/874, 2014. URL: https://eprint.iacr.org/2014/874.
8. Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal Gaussians. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013*, pages 40–56. Springer, 2013. Extended version available as IACR ePrint 2013/383. URL: https://eprint.iacr.org/2013/383, doi:10.1007/978-3-642-40041-4_3.
9. Nagarjun C. Dwarakanath and Steven D. Galbraith. Sampling from discrete Gaussians for lattice-based cryptography on a constrained device. *Applicable Algebra in Engineering, Communication and Computing*, 25(3):159–180, June 2014. doi:10.1007/s00200-014-0218-3.
10. Hassan Edrees, Brian Cheung, McCullen Sandora, David B. Nummey, and Deian Stefan. Hardware-optimized ziggurat algorithm for high-speed Gaussian random number generators. In Toomas P. Plaks, editor, *ERSA 2009*, pages 254–260. CSREA Press, 2009. URL: http://sprocom.cooper.edu/sprocom2/pubs/conference/ecsns2009ersa.pdf.
11. FIPS. (FIPS) 186-4, digital signature standard (DSS). Federal Information Processing Standards Publication, July 2013. doi:10.6028/NIST.FIPS.186-4.
12. FIPS. Secure Hash Standard (SHS). Federal Information Processing Standards Publication 180-4, August 2015. doi:10.6028/NIST.FIPS.180-4.
13. FIPS. SHA-3 standard: Permutation-based hash and extendable-output functions. Federal Information Processing Standards Publication 202, August 2015. doi:10.6028/NIST.FIPS.202.
14. Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 212–219. ACM, 1996. URL: http://arxiv.org/abs/quant-ph/9605043, doi:10.1145/237814.237866.
15. Lov K. Grover. From Schrödinger's equation to the quantum search algorithm. *American Journal of Physics*, 69(7):769–777, 2001. URL: http://arxiv.org/abs/quant-ph/0109116, doi:10.1119/1.1359518.
16. James Howe, Thomas Pöppelmann, Máire O'Neill, Elizabeth O'Sullivan, and Tim Güneysu. Practical lattice-based digital signature schemes. *ACM Trans. Embed. Comput. Syst.*, 14(3):41:1–24, April 2015. doi:10.1145/2724713.
17. Jakob Jonsson and Burt Kaliski. Public-key cryptography standards (PKCS) #1: RSA cryptography specifications version 2.1. IETF RFC 3447, February 2003. URL: https://tools.ietf.org/html/rfc3447, doi:10.17487/RFC3447.
18. Charles F. F. Karney. Sampling exactly from the normal distribution, 2014. Preprint arXiv:1303.6257, Version 2. URL: http://arxiv.org/abs/1303.6257.
19. Donald E. Knuth and Andrew C. Yao. The complexity of nonuniform random number generation. In

Joseph F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 357–428, New York, 1976. Academic Press.

20. Paul Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *CRYPTO '96*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996. doi:10.1007/3-540-68697-5_9.
21. Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *CRYPTO '99*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999. doi:10.1007/3-540-48405-1_25.
22. Glen G. Langdon, Jr. An introduction to arithmetic coding. *IBM Journal of Research and Development*, 28(2):135–149, 1984.
23. Zhe Liu, Hwajeong Seo, Sujoy Sinha Roy, Johann Großschädl, Howon Kim, and Ingrid Verbauwhede. Efficient Ring-LWE encryption on 8-bit AVR processors. In Tim Güneysu and Helena Handschuh, editors, *CHES 2015*, volume 9293 of *LNCS*, pages 663–682. Springer, 2015. URL: https://eprint.iacr.org/2015/410, doi:10.1007/978-3-662-48324-4_33.
24. George Marsaglia and Wai Wan Tsang. A fast, easily implemented method for sampling from decreasing or symmetric unimodal density functions. *SIAM Journal on Scientific and Statistical Computing*, 5(2):349–359, 1984. doi:10.1137/0905026.
25. George Marsaglia and Wai Wan Tsang. The ziggurat method for generating random variables. *Journal of Statistical Software*, 5(8):1–7, October 2000. URL: http://www.jstatsoft.org/v05/i08.
26. NSA/CSS. Information assurance directorate: Commercial national security algorithm suite and quantum computing FAQ, January 2016. URL: https://www.iad.gov/iad/library/ia-guidance/ia-solutions-for-classified/algorithm-guidance/cnsa-suite-and-quantum-computing-faq.cfm.
27. Chris Peikert. An efficient and parallel Gaussian sampler for lattices. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 80–97. Springer, 2010. doi:10.1007/978-3-642-14623-7_5.
28. William B. Pennebaker, Joan L. Mitchell, Glen G. Langdon, Jr, and Ronald B. Arps. An overview of the basic principles of the Q-coder adaptive binary arithmetic coder. *IBM Journal of Research and Development*, 32(6):717–726, 1988.
29. Thomas Pöppelmann, Léo Ducas, and Tim Güneysu. Enhanced lattice-based signatures on reconfigurable hardware. In Lejla Batina and Matthew Robshaw, editors, *CHES 2014*, volume 8731 of *LNCS*, pages 353–370. Springer, 2014. Extended version available as IACR ePrint 2014/254. URL: https://eprint.iacr.org/2014/254, doi:10.1007/978-3-662-44709-3_20.
30. Jorma J. Rissanen. Generalized kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20:198–203, May 1976. doi:10.1147/rd.203.0198.
31. Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, and Ingrid Verbauwhede. Compact and side channel secure discrete Gaussian sampling. IACR ePrint 2014/591, 2014. URL: https://eprint.iacr.org/2014/591.
32. Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact Ring-LWE cryptoprocessor. In Lejla Batina and Matthew Robshaw, editors, *CHES 2014*, volume 8731 of *LNCS*, pages 371–391. Springer, 2014. URL: https://eprint.iacr.org/2013/866, doi:10.1007/978-3-662-44709-3_21.
33. Markku-Juhani O. Saarinen. Gaussian sampling precision in lattice cryptography. IACR ePrint 2015/953, October 2015. URL: https://eprint.iacr.org/2015/953.
34. Amir Said. Introduction to arithmetic coding - theory and practice. In Khalid Sayood, editor, *Lossless Compression Handbook*. Academic Press, 2002. Chapter also published as HP Technical report HPL-2004-76. URL: http://www.hpl.hp.com/techreports/2004/HPL-2004-76.pdf.
35. Gregory Valiant and Paul Valiant. An automatic inequality prover and instance optimal identity testing. In *FOCS 2014*, pages 51–60. IEEE Computer Society, 2014. Full version available as http://theory.stanford.edu/~valiant/papers/instanceOptFull.pdf. doi:10.1109/FOCS.2014.14.
36. Patrick Weiden, Andreas Hülsing, Daniel Cabarcas, and Johannes Buchmann. Instantiating treeless signature schemes. IACR ePrint 2013/065, February 2013. URL: https://eprint.iacr.org/2013/065.
37. Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–240, June 1987. doi:10.1145/214762.214771.
38. Christof Zalka. Grover's quantum searching algorithm is optimal. *Physical Review A*, 60:2746–2751, October 1999. URL: http://arxiv.org/abs/quant-ph/9711070, doi:10.1103/PhysRevA.60.2746.