Beyond ECDSA and RSA: Lattice-based Digital Signatures on Constrained Devices

Tobias Oder
Horst Görtz Institute for IT-Security
Ruhr University Bochum
tobias.oder@rub.de

Thomas Pöppelmann
Horst Görtz Institute for IT-Security
Ruhr University Bochum
thomas.poeppelmann@rub.de

Tim Güneysu
Horst Görtz Institute for IT-Security
Ruhr University Bochum
tim.gueneysu@rub.de

ABSTRACT

All currently deployed asymmetric cryptography is broken with the advent of powerful quantum computers. We thus have to consider alternative solutions for systems with long-term security requirements (e.g., for long-lasting vehicular and avionic communication infrastructures). In this work we present an efficient implementation of BLISS, a recently proposed, post-quantum secure, and formally analyzed novel lattice-based signature scheme. We show that we can achieve a significant performance of 35.3 and 6 ms for signing and verification, respectively, at a 128-bit security level on an ARM Cortex-M4F microcontroller. This shows that lattice-based cryptography can be efficiently deployed on today's hardware and provides security solutions for many use cases that can even withstand future threats.

Categories and Subject Descriptors

SEC2.3 [**Hardware and Embedded Systems Security**]: Embedded and Cross-Layer Security—*Embedded Security case studies (Medical, Automotive, smartgrid, industrial control etc.)*

1. INTRODUCTION

Security has become a crucial aspect of many recent hardware and software systems, in particular for those being potentially exposed to attacks for the next 10-20 years. A major challenge is the design of corresponding cryptographic and security functions that guarantee protection over the entire lifespan, such as in the case of long-lasting vehicular or avionic systems. In vehicular communication infrastructures (V2G/V2V), for example, Elliptic Curve Cryptography (ECC) using 224-bit and 256-bit NIST curves was standardized for message authentication in IEEE 1609.2. The relatively large security parameters for ECC were considered a conservative choice to achieve long-term security against attackers using conventional computers. However, with the availability of powerful quantum computers that IBM announced to become available within the next 15 years [3], all currently used asymmetric cryptosystems – including ECC – are known to be broken due to Shor's algorithm [13]. Referring not only to the specific situation for V2G/V2V but also to other use-cases in the Internet of Things, we see an urgent demand for *alternative* cryptosystems in general to achieve long-term security that can also withstand such future attacks. More precisely, we need novel cryptosystems that rely on completely different security assumptions but still enable efficient implementations on contemporary and constrained hardware.

A solution to this problem could be provided by recently proposed lattice-based digital signatures that combine efficiency with small key and signature sizes. Their underlying security assumptions not only allow reductions to hard lattice problems but also involve polynomial multiplication as core function what can be computed and scaled efficiently using FFT-like arithmetic. A very promising candidate for message authentication and digital signatures is the family of Bimodal Lattice Signature Schemes (BLISS) that was introduced at CRYPTO'13 [5] and perfectly combines all aforementioned properties.

Contribution. In this work we present the first implementation of the BLISS signature scheme tailored to a 32-bit ARM Cortex-M4F RISC (1024 KB flash/192 KB SRAM) microcontroller that provides an equivalent level of security compared to 256-bit ECC or 128-bit of symmetric security. This common series of ARM microcontrollers is not only deployed in vehicular environments but also in many other embedded devices, such as smart meter gateways, medical instrumentation or industrial control systems so that our results are also likewise applicable to many other applications. For this platform we investigate the optimal implementation of polynomial multiplication using the Number-Theoretic Transform (NTT) as well as an analysis of the efficiency of several Gaussian samplers which is another crucial operation of BLISS. Finally, our implementation on this low-cost platform achieves a significant performance, namely 28 signing and 167 verification operations per second, outperforming classical cryptosystems such as RSA and ECC.

Outline. This work is structured as follows: We provide background implementation on BLISS in Section 2 and details about our implementation in Section 3. Performance results and a comparison with other asymmetric cryptosystems are part of Section 4, before we conclude with some remarks on future work in Section 5.

DAC '14, June 01 - 05 2014, San Francisco, CA, USA

2. BIMODAL LATTICE-BASED SIGNATURE SCHEMES

In this section we provide background and a description of the BLISS variant that we target for implementation. Further details such as the reasoning of the construction and associated security proofs are given in the original work and its full version [6, 5].

Lattices have various applications in asymmetric cryptography and are currently an emerging research topic with much potential. New schemes allow the construction of efficient and formally verifiable signature schemes which are presumably not protected by patents. Those schemes are supposed to offer much better security guarantees than ad-hoc designs like the broken NTRUSign [7] signatures. The principle of the BLISS signature scheme is based on ideas given in [11, 9]. The general approach is to create the signature using the Fiat-Shamir transformation in a way that it does not leak information about the secret key. This is achieved by rejection sampling which ensures that the signature is distributed according to a discrete Gaussian distribution D_σ and independent of the secret key. The distribution D_σ is defined such that a value $x \in \mathbb{Z}$ is sampled from D_σ with the probability $\rho_\sigma(x)/\rho_\sigma(\mathbb{Z})$ where $\rho_\sigma(x) = \exp(\frac{-x^2}{2\sigma^2})$ and $\rho_\sigma(\mathbb{Z}) = \sum_{k=-\infty}^{\infty} \rho_\sigma(k)$. All arithmetic for the presented concrete instantiation of BLISS is performed in the ring $R = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$ represented as polynomials with n coefficients where each coefficient is reduced modulo q.

Algorithm 1: BLISS Key Generation

Result: Key pair $(\mathbf{A}, \mathbf{S})$ such that $\mathbf{AS} = q \bmod 2q$

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

Key Generation. The key generation algorithm given in Algorithm 1 is similar to the key generation algorithm of NTRU. First polynomials $\mathbf{f}$ and $\mathbf{g}$ with densities δ_1 and δ_2 are sampled such that they have $d_1 = \lceil \delta_1 n \rceil$ coefficients in $\{\pm 1\}$ and $d_2 = \lceil \delta_2 n \rceil$ coefficients in $\{\pm 2\}$. The secret key is computed as $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2)^t = (\mathbf{f}, 2\mathbf{g} + 1)^t$ and the public key is $\mathbf{A} = (2\mathbf{a}_q, q - 2) = (\frac{2 \cdot (2\mathbf{g}+1)}{\mathbf{f}}, q - 2)$. For successful key generation, it is necessary to find a polynomial $\mathbf{f}$ that is invertible. If this is not the case, key generation is restarted. The size of the signature depends on the maximum possible norm of the vector $\mathbf{Sc}$, which is defined as $N_\kappa(\mathbf{S})$ and computed as

$$N_\kappa(\mathbf{S}) = \max_{\substack{I \subset \{1,\ldots,n\} \\ \#I = \kappa}} \sum_{i \in I} \left(\max_{\substack{J \subset \{1,\ldots,n\} \\ \#J = \kappa}} \sum_{j \in J} T_{i,j} \right)$$

where $\mathbf{T} = \mathbf{S}^t \cdot \mathbf{S} \in \mathbb{R}^{n \times n}$. A private key $\mathbf{S}$ whose N_κ value is too big is rejected, to keep the signature size small.

Signature Generation. The generation of the signature is described in Algorithm 2. First two masking polynomials $\mathbf{y}_1, \mathbf{y}_2$ are sampled where each coefficient is randomly chosen according to the Gaussian distribution D_σ. Then the polynomial $\mathbf{u}$ is computed as $\mathbf{u} = \zeta \cdot \mathbf{a}_1 \cdot \mathbf{y}_1 + \mathbf{y}_2 \bmod 2q$ where multiplication by $\zeta = \frac{1}{q-2} \bmod 2q$ and reduction modulo $2q$ are necessary in order to achieve a low rejection rate. Note, that it is still possible to compute $\mathbf{a}_1 \cdot \mathbf{y}_1$ using FFT-techniques which require a prime modulus for maximum efficiency. Then only the higher-order bits[1] $\lfloor \mathbf{u} \rceil_d$ of the polynomial $\mathbf{u}$ are hashed together with the message μ. This is done using a standard cryptographic hash function which is a practical instantiation of a random oracle – a theoretical construction used in the formal security analysis. The entropy provided by the output of the (pseudo-random) hash function is then used to construct the sparse polynomial $\mathbf{c}$ with κ coefficients equal to one and the remaining coefficients set to zero. In Line 6 the secret key is multiplied with $\mathbf{c}$ (which depends on the message and the random $\mathbf{y}_1, \mathbf{y}_2$) and masked using the Gaussian distributed masking polynomials $\mathbf{y}_1, \mathbf{y}_2$. The rejection step is performed in Line 7 to make the signature independent of the secret key. Due to this rejection step the signature generation restarts with a certain probability. Finally the size of the signature $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$ is reduced by compressing $\mathbf{z}_2$.

Algorithm 2: BLISS Signature Algorithm

Data: Message μ, public key $\mathbf{A} = (\mathbf{a}_1, q - 2) \in \mathcal{R}_{2q}^{1\times 2}$, secret key $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2)^t \in \mathcal{R}_{2q}^{2\times 1}$

Result: Signature $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$ of the message μ

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

Signature Verification. In order to verify a signature it is checked if $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$ is a valid signature of the message μ. First the l_2 and l_∞ norms of $\mathbf{z}$ are computed and the signature is rejected if $\mathbf{z}$ is too large (depending on the parameters B_2 and B_∞). The actual verification is done by computing $H(\lfloor \zeta \cdot \mathbf{a}_1 \cdot \mathbf{z}_1 + \zeta \cdot q \cdot \mathbf{c} \rceil_d + \mathbf{z}_2^\dagger \bmod p, \mu)$ and testing whether the result is equal to $\mathbf{c}$. The verification process is explained in detail in Algorithm 3.

Parameters Selection. The designers of BLISS provide four parameter sets suitable for practical applications [5]. The parameter set BLISS-I provides 128 bits of equivalent symmetric security which is comparable to ECC with 256 bits and sufficient to provide security beyond 15-20 years. In Table 1, we provide detailed information for BLISS-I and show that it can be used even on constrained devices due to its moderate resource consumption. Note that all BLISS parameter sets have in common the choice of $n = 512$ and $q = 12289$ for the polynomial ring. Therefore, it is pos-

[1]For any integer x, the d high-order bits of x are denoted by $\lfloor x \rceil_d$ so that x can be written as $x = \lfloor x \rceil_d \cdot 2^d + [x \bmod 2^d]$.

Algorithm 3: BLISS Verification Algorithm

Data: Message μ, public key $\mathbf{A} = (\mathbf{a}_1, q-2) \in \mathcal{R}_{2q}^{1\times 2}$, Signature $(\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$
Result: Accept (valid) or Reject (invalid)

1 **begin**
2 **if** $||(\boldsymbol{z}_1|2^d \cdot \boldsymbol{z}_2^\dagger)||_\infty > B_\infty$ **then** Reject
3 **if** $||(\boldsymbol{z}_1|2^d \cdot \boldsymbol{z}_2^\dagger)||_2 > B_2$ **then** Reject
4 Accept iff $\mathbf{c} = H(\lfloor \zeta \cdot \mathbf{a}_1 \cdot \mathbf{z}_1 + \zeta \cdot q \cdot \mathbf{c} \rceil_d + \mathbf{z}_2^\dagger \bmod p, \mu)$

sible to use the same core functions for polynomial arithmetic when switching to even higher security levels (BLISS-III/IV) or slightly smaller signatures (BLISS-II).

Table 1: BLISS-I parameter set providing 128 bit of equivalent symmetric security. Signature and key sizes are the theoretical optimal values and given in kilobits (kb).

n	q	δ_1, δ_2	σ	α	κ
512	12289	0.3, 0	215	1	23
N_κ	B_2, B_∞	Repetition rate	|sk|	|pk|	|sig|
1.62	12872, 2100	1.6	2 kb	7 kb	5.6 kb

3. IMPLEMENTATION

In this section we provide implementation details for our ARM-based target device. More precisely, we describe how to implement different methods to sample from a discrete Gaussian distribution, the polynomial arithmetic, and further procedures required for key generation on a constrained device.

3.1 Target Device

The ARM Cortex-M series consists of five 32-bit RISC microcontrollers where the Cortex-M4F is the most powerful one in the series. It has 21 core registers, made up of 13 general purpose registers and 5 special registers. A further core feature is the floating point unit (FPU) that makes the difference between Cortex-M4 and Cortex-M4F and allows floating point addition and multiplication in one cycle. The Cortex-M4 features a digital signal processor that is capable of performing instructions like multiplication with subsequent addition in a single cycle. For generation of randomness we rely on the on-board TRNG which is equipped with a fault detector and employs ring oscillators and an LFSR. The RNG runs with 48 MHz and can output 32 random bits every 40 periods.

3.2 Gaussian Sampling

In this section, we investigate potential approaches to sample from a discrete Gaussian distribution (cf. Line 2 of Algorithm 2) that matches the security requirements of BLISS. For each signing attempt it is necessary to generate $2n = 1024$ Gaussian distributed coefficients for a standard deviation $\sigma = 215$ with high precision matching the assumptions of the security proof. As best candidates for this purpose, we evaluate the Knuth-Yao [14, 8] and Ziggurat [2] sampling algorithms as well as an approach using Bernoulli distributed variables [5]. The conceptually simplest algorithm to sample from a Gaussian distribution is to choose a uniformly random $u \in \{-\tau\sigma, ..., \tau\sigma\}$ (in this case τ is denoted as tail-cut) and to accept it with a probability proportional to $\exp(-x^2/2\sigma^2)$. However, this requires costly computation of the exp() function with high precision, a large number of random bits, and leads to ≈ 10 trials per sample.

3.2.1 Bernoulli Sampler

The authors of [5] proposed a promising sampling algorithm that makes use of Bernoulli distributed variables. A Bernoulli distributed variable $\mathcal{B}_c$ outputs one with a probability of $c \in [0, 1]$ and zero otherwise. Sampling from this distribution is easy by lazily evaluating if $y < c$ for a uniformly random $y \in [0, 1)$ and precomputed c. The general idea (for more details we refer to [6]) of the proposed sampler is to optimize rejection sampling by reducing the probability of rejections. This is done by sampling first from an intermediate distribution, called binary Gaussian distribution, and then from the target distribution. The rejection rate is thus reduced to ≈ 1.47 (compared to 10 for classical rejection sampling) and no computations of the exponentiation function exp() or large precomputed tables are necessary any more.

The implementation of the Bernoulli sampler is straightforward. The algorithm requires to store 336 bytes of precomputed data and we use loop unrolling to speed up certain parts of the algorithm. This reduces the average required cycles from 2049 to 1835 cycles per Gaussian distributed coefficient.

3.2.2 Knuth-Yao Algorithm

The Knuth-Yao algorithm allows to sample from a Gaussian distribution [8, 14] by constructing a binary tree from the probability matrix and performing a random walk to sample an element. The probability matrix consists of the binary expansion of the probabilities of all $x \in [0, \tau\sigma]$ ignoring leading zero digits. The matrix determines a rooted binary tree with internal nodes that always have two successors, as well as terminal leaves. The leaves are labeled with the value that is returned if this leaf is reached during the random walk through the tree. The number of leaves at level n is equal to the number of 1's in column n of the probability matrix (starting with column 0). The row in which a one appears is used as label for one of the leaves. All remaining nodes become internal nodes with two successors that get labeled the same way.

Our implementation of the Knuth-Yao algorithm is inspired by [14] which has been adapted to software. Necessary tables were computed using a computer algebra system (SAGE) from which we remove a large number of leading zeros that occur in the first columns. The overall memory consumption is 19064 bytes which is still large but favorable compared to a naïve cumulative distribution table approach, which would require 41280 bytes. However, the detection of leading zeros leads to some overhead so that we need on average 2404 cycles per sample.

3.2.3 Ziggurat Algorithm

The discrete Ziggurat algorithm [2] is similar to the Bernoulli sampler and an approach to optimize rejection sampling. For this purpose m rectangles with the left corners on

the y-axis and the right corners on the graph of the probability distribution function are computed such that all rectangles have the same size. The entire area under the graph is then covered by rectangles and a rectangle R_i can efficiently be stored by just storing the coordinates (x_i, y_i) of the lower right corner.

To sample a value, first a rectangle R_i is uniformly random sampled. The next step is to uniformly choose an x value within the sampled rectangle. If this x value is smaller or equal to the x coordinate of the previous rectangle, x gets accepted, because all points $(x_j, y_j) \in R_i$ with $x_j \leq x_{i-1}$ definitively lie within the area covered by the graph. Otherwise, one has to sample a value y and compute the exp() function to determine whether a value gets rejected or accepted.

The main challenge of an implementation of the Ziggurat algorithm is its computational complexity and the infrequent high precision rejection sampling. Thus we had to implement expensive multi-precision arithmetic and compute the exponential function using limit representation (see [8] for a short survey on methods to compute exp()).

3.3 Polynomial Arithmetic

Polynomial arithmetic is one of the most time-consuming components of BLISS. In this section we concentrate on the costly multiplication and inversion in $\mathbb{Z}_p/\langle x+1\rangle$.

3.4 Number Theoretic Transform

For polynomial multiplication we use the Number Theoretic Transform (NTT) which is a discrete Fourier transform over a finite field. It allows us to achieve asymptotic complexity of $\mathcal{O}(n \log n)$ and also efficient implementations (e.g., see [12]). To speed up the computation of the NTT, we precompute all necessary twiddle factors which are powers and inverse powers of the primitive n-th root ω. Moreover, we have unrolled the first two stages of the implemented radix-2 decimation-in-time algorithm. The core of the algorithm is implemented in assembler and listed in Table 2.

Another useful feature of our target microcontroller is a bit reversal instruction (`RBIT`). This is necessary as the input to the transform has to be reordered such that coefficients are exchanged with their bit-reversed counterpart. With the help of the `RBIT` instruction, we can implement this step efficiently using inline assembly.

3.5 Polynomial Inversion

During the key generation, we have to compute the multiplicative inverse of $\mathbf{f}$ (Algorithm 1, Line 5). For this task we use Fermat's little theorem to compute the multiplicative inverse as $\mathbf{f}^{-1} = \mathbf{f}^{q-2}$ in R. To speed this exponentiation up, we use an addition chain [10] that requires 18 polynomial multiplications[2]. Processing the exponent $12289 - 2 = 12287 = 10111111111111_2$ bit-wise via the the square-and-multiply algorithm algorithm would need 25 polynomial multiplications what makes the addition chain approach preferable. Also an implementation using Montgomery reduction turned out to be slower than using addition chains. All operations during the inversion are performed in the frequency domain and it suffices to compute the NTT transformation at the beginning and the end of the exponentiation. This provides the possibility to apply an early test for invertibility of the input, because we can simply test whether there are coefficients that are equal to 0 after transforming the input into the frequency domain. Another advantage of the frequency domain is that polynomial multiplication is just coefficient-wise multiplication. We can exploit this to minimize the memory consumption by computing the addition chain iteratively for all coefficients. Thus we do not have to store whole polynomials as intermediate results but just one coefficient.

[2]See Achain-All `http://www-cs-faculty.stanford.edu/~uno/programs.html`

Table 2: Implementation of the NTT butterfly operation in C (on the left) and assembly (on the right). The ARM Cortex-M4F supports conditional execution and powerful DSP instructions, like Multiply-Subtract (MLS), which speed up our implementation.

C Code	Assembler
// omega[m] in LR	ldr.W LR, [R6]
// out[b] in R9	ldr.W R9, [R0,R12,LSL #2]
r =(omega[m]*out[b])%q;	mul LR, R9, LR
// q in R11 and R8	mov R11, R8
// out[a] in R9	ldr.W R9, [R0,R7,LSL #2]
// out[b] in R10	ldr.W R10, [R0,R12,LSL #2]
	sdiv R11, LR, R11
// result mod q in R11	mls R11, R8, LR
out[b]=out[a]+(q-r)%q;	subS.W R10, R9, R11
	IT MI
	addMI R10, R10, R8
out[a]=out[a]+r % q;	add R9, R9, R11
	cmp R9, R8
	IT GE
	subGE.W R9, R9, R8
// write back out[b]	str.W R10, [R0,R12,LSL #2]
// write back out[a]	str.W R9, [R0,R7,LSL #2]

3.6 Sparse Multiplication

In Line 6 of Algorithm 2, computation of $\mathbf{s}_{1,2}\mathbf{c}$ is required. Since $\mathbf{c}$ is only a sparse polynomial where κ coefficients are set to one, applying the NTT is not the optimal solution. Moreover, we do not need to reduce modulo $2q$ as the polynomials $\mathbf{s}_{1,2}$ have only small coefficients. For efficiency reasons we therefore only store the index of the coefficients of $\mathbf{c}$ that are one and computing those with runtime $\mathcal{O}(n\kappa)$. We further decreased the running time of the sparse schoolbook multiplication by 36,6% (from 354,419 to 224,626 cycles) by unrolling the inner loop at the cost of a slightly increased code size by 602 bytes.

4. RESULTS AND EVALUATION

In this section, we present performance results for our implementation of BLISS-I on the Cortex-M4F. The Cortex-M4F microcontroller operates at 168 MHz and our code is compiled using IAR Embedded Workbench for ARM in version 6.60.1.5104. For precise benchmarks, we determined average cycle counts of a subroutine with random inputs from 1000 runs and used a data watchpoint trigger to exactly evaluate the clock cycle counter.

Table 3: Measurement results of the major building blocks of our BLISS-I implementation. The NTT transformation is applied on polynomials with $n = 512$ coefficients. Gaussian distributed values are sampled from $D_{\mathbb{Z}^n,\sigma}$ for $n = 512$ and $\sigma = 215$. We denote by g/s/v if a routine is used in key (g)eneration, (s)igning, or (v)erification.

Routine	Cycle	Application
NTT Trans.	122,619	g/s/v
NTT Multiplication	508,624	g/s/v
Polynomial Inversion	470,606	g
Computation of $\mathrm{N}_\kappa(\mathbf{S})$	1,043,447	g
Generate $\mathbf{c}$	220,022	s/v
Drop bits	8,225	s/v
Sparse Multiplication	224,626	s
Huffman Encoding	78,927	s
Huffman Decoding	115,943	v
Sampling Bernoulli	935,925	s
Sampling Knuth-Yao	1,231,326	s
Sampling $\mathrm{Ziggurat}_{\mathrm{speed}}$	1,057,814	s
Sampling $\mathrm{Ziggurat}_{\mathrm{average}}$	1,729,098	s
Sampling $\mathrm{Ziggurat}_{\mathrm{size}}$	3,378,909	s

4.1 Performance Results

Cycle counts for major building blocks of BLISS are given in Table 3. The results show that computations required during key generation are expensive in terms of running time and RAM consumption. However, the polynomial inversion is even 8.5% faster than a single NTT multiplication since we preserve the NTT-transformed representation between different functions and use addition chains to speed up the inversion. By using the sparse multiplication, we save about 54.9% of the runtime compared to a NTT multiplication on the same values. We evaluated three instantiations of the Ziggurat algorithm, one for a rather small precomputed table with a size of 2560 bytes, one as a time-memory-tradeoff with a precomputed table of 5120 bytes, and one for a rather large precomputed table with a size of 10240 bytes for time-critical applications. The trade-off sampler is 61,2% faster than the size-optimized sampler whereas the speed-optimized sampler only has a 51,2% better performance than the trade-off sampler. But even the speed-optimized variant of the Ziggurat sampler is not able to outperform the Bernoulli sampler. The generation of the signature component $\mathbf{c}$, that also includes the SHA-3 hash function, performs well compared to other subroutines. The removal of lower bits of $\mathbf{z}_2$ is negligible with about 8000 cycles. Merging all components together, the number of clock cycles and memory consumption for key generation, signing and verification are given in Table 4.

Key generation is a rather slow process since it needs to be restarted frequently and requires time-consuming computations (see Table 3). However, it is usually done once per device. The performance of the signing operation is directly determined by the chosen sampler. We observe that the RAM consumption of the signing operation is independent from the sampling algorithm since all samplers need a comparatively small amount of memory.

The table used for the Bernoulli sampler is also used for the computation of the hyperbolic cosine function. The total amount of flash memory includes 900 bytes for the Bernoulli sampler. Additional flash memory is also consumed by code that initializes peripherals of the STM32F4 and is responsible for debug and profiling output.

Table 4: Results for an implementation of the signing algorithm with different samplers, verification and key generation. Note that the flash consumption for the signing algorithms already includes the code and tables for verification and key generation.

Operation	Cycle	RAM	Flash
$\mathrm{Signing}_{\mathrm{Ber}}$	5,927,441	18,580	24,648
$\mathrm{Signing}_{\mathrm{KY}}$	6,865,089	18,580	44,036
$\mathrm{Signing}_{\mathrm{Zig\text{-}Speed}}$	5,984,686	18,580	36,028
$\mathrm{Signing}_{\mathrm{Zig\text{-}Average}}$	8,335,711	18,580	30,908
$\mathrm{Signing}_{\mathrm{Zig\text{-}Size}}$	16,396,414	18,580	28,420
Verification	1,002,299	11,520	-
Key Generation	367,859,092	21,272	-

All in all our results show that the Bernoulli sampler is clearly the best choice compared to other evaluated samplers. It provides lowest runtime and needs the smallest precomputed table compared to the other two evaluated samplers. It would be possible to speed-up the Ziggurat with even larger tables but for most constrained devices this is not an option. A major issue with the Ziggurat sampler is the requirement for multi-precision arithmetic. The Knuth-Yao sampler is the least favorable choice with respect to performance and memory consumption compared to the Bernoulli and the speed-optimized Ziggurat sampler. Furthermore, a major drawback of this algorithm is the large table and the high amount of memory accesses that slow down the implementation. Therefore, we do not recommend the Knuth-Yao sampler for the evaluated target platform. These results are also counter-intuitive, since we expected the algorithm with the *largest* table to outperform the others. But instead, the Bernoulli sampler as the one with the *smallest* table is the preferable solution according to our results.

4.2 Performance Comparison

In Table 5 we provide results obtained from the documentation of the STM32 Cryptographic Library [15] which is evaluated on a STM32F4xx family (Cortex-M4) microcontroller. There are also other results for ARM-based microcontrollers in the literature, but many implementations run on outdated hardware (e.g., [1] on ARM7TDMI) and do not allow a fair comparison. A recent work evaluates ECC on a much less powerful Cortex-M0, with a special focus on energy consumption [4]. A comparison with the CPU implementation of BLISS given in [5] is certainly not fair as well. Besides architectural differences the biggest advantage of desktop CPUs is that much more memory is available so that Gaussian sampling can be implemented using the Cumulative Distribution Table (CDT) which is faster than the storage efficient algorithms used in this work.

In terms of security, the implemented signature scheme was designed to provide a level of 128 bits of equivalent symmetric security. It can be compared to RSA-2048 (or maybe even RSA-4096) and ECC-256. In terms of speed our verification routine outperforms RSA and ECC for all common

security parameters. Moreover, our implementation is twice as fast compared to ECC-256 regarding signature generation. It becomes also obvious that RSA gets impractical for parameter sets larger than RSA-2048 on constrained devices, especially due to the very slow signing. For minimal signature size, our implementation applies Huffman encoding to obtain the signature size of 5600 bits. The public key requires 1024 bytes and the private key requires 384 bytes. For n-RSA signatures the size of the signature is $\frac{n}{8}$ bytes and for n-ECC signatures (ECDSA) the signature size is $2\frac{n}{8}$ bytes.

Table 5: Comparison of the most efficient instantiation of our implementation with the RSA and ECC implementation of the STM32 Cryptographic Library (target device: STM32F4xx family) [15].

Operation	Cycles (key gen)	Cycles (sign)	Cycles (verify)
$\text{BLISS}_{\text{Ber}}$	367,859,092	5,927,441	1,002,299
RSA-1024	-	30,627,432	1,573,079
RSA-2048	-	228,068,226	6,195,481
ECC-192	7,400,421	7,720,020	14,716,374
ECC-224	9,849,334	10,414,487	19,558,528
ECC-256	12,713,277	13,102,239	24,702,099

5. CONCLUSION AND FUTURE WORK

In this work we have shown that it is possible to implement a post-quantum, lattice-based signature scheme on a Cortex-M4F microcontroller with a reasonable flash and memory consumption. The most optimal variant takes $6 \cdot 10^6$ cycles for signing, $1 \cdot 10^6$ cycles for verification and $368 \cdot 10^6$ cycles for key generation. Running the target device at 168 MHz, this corresponds to 28 signing, 167 verification and 0.46 key generation operations per second. As a consequence, our implementation is faster than the reference implementation of RSA and ECC for comparable security levels (see Table 5) and suitable for many embedded applications, e.g., V2V/V2G infrastructures.

For future work we plan to evaluate the resistance against side-channel attacks and the implementation of countermeasures. Additionally, we plan to optimize the scheme for arithmetic using vector extensions (e.g., ARM NEON). We especially expect verification to profit, as the algorithm basically consists of an NTT multiplication which is very amendable to vectorization. Another direction for future work are new techniques for Gaussian sampling and general cryptanalysis to increase confidence in the chosen parameter set.

6. REFERENCES

[1] M. Aydos, T. Yanik, and Çetin Kaya Koç. A high-speed ECC-based wireless authentication on an ARM microprocessor. In *ACSAC*, pages 401–410. IEEE Computer Society, 2000.

[2] 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. *IACR Cryptology ePrint Archive*, 2013:510, 2013.

[3] K. Chang. I.B.M. researchers inch toward quantum computer. New York Times Article, February 28, 2012. `http://www.nytimes.com/2012/02/28/technology/ibm-inch-closer-on-quantum-computer.html?_r=1&hpw`.

[4] R. de Clercq, L. Uhsadel, A. V. Herrewege, and I. Verbauwhede. Ultra low-power implementation of ECC on the ARM Cortex-M0+. *IACR Cryptology ePrint Archive*, 2013:609, 2013.

[5] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky. Lattice signatures and bimodal Gaussians. In R. Canetti and J. A. Garay, editors, *CRYPTO (1)*, volume 8042 of *Lecture Notes in Computer Science*, pages 40–56. Springer, 2013. Proceedings version of [6].

[6] L. Ducas, A. Durmus, T. Lepoint, and V. Lyubashevsky. Lattice signatures and bimodal Gaussians. *IACR Cryptology ePrint Archive*, 2013:383, 2013. Full version of [5].

[7] L. Ducas and P. Q. Nguyen. Learning a zonotope and more: Cryptanalysis of NTRUSign countermeasures. In X. Wang and K. Sako, editors, *ASIACRYPT*, volume 7658 of *Lecture Notes in Computer Science*, pages 433–450. Springer, 2012.

[8] 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.

[9] 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*, volume 7428 of *Lecture Notes in Computer Science*, pages 530–547. Springer, 2012.

[10] D. E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[11] V. Lyubashevsky. Lattice signatures without trapdoors. In D. Pointcheval and T. Johansson, editors, *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 738–755. Springer, 2012.

[12] T. Pöppelmann and T. Güneysu. Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware. In A. Hevia and G. Neven, editors, *LATINCRYPT*, volume 7533 of *Lecture Notes in Computer Science*, pages 139–158. Springer, 2012.

[13] P. W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *FOCS*, pages 124–134, 1994.

[14] S. Sinha Roy, F. Vercauteren, and I. Verbauwhede. High precision discrete Gaussian sampling on FPGAs. *Selected Areas in Cryptography, SAC 2013, Burnaby, British Columbia, Canada, August 14-16, to appear*, 2013. `http://www.cosic.esat.kuleuven.be/publications/article-2372.pdf`.

[15] STMicroelectronics. UM0586 STM32 Cryptographic Library. `http://www.st.com/st-web-ui/static/active/en/resource/technical/document/user_manual/CD00208802.pdf`.