A Lattice-based AKE on ARM Cortex-M4

Julian Speith[1], Tobias Oder[1], Marcel Kneib[2], and Tim Güneysu[1,3]

[1]Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Germany
{julian.speith,tobias.oder,tim.gueneysu}@rub.de
[2]Bosch Engineering GmbH, Germany marcel.kneib@de.bosch.com
[3]DFKI, Germany

Abstract. Lattice-based cryptography is one of the most promising alternatives to currently deployed public key algorithms. Plenty of novel key exchange schemes based on lattices have been proposed. The majority of these proposals however focuses on unauthenticated key exchange schemes which need to be combined with a digital signature scheme for authentication. In this work we analyze the cost of the authentication overhead. We instantiate the generic construction by del Pino et al. [8] with the digital signature scheme BLISS-B and the key exchange scheme JarJar-Simple and implement the authenticated key exchange on a ARM Cortex-M4F to show its practical performance on a constrained device. Our implementation takes a total of 62ms on average for one execution of the protocol. Those speeds are achieved by improving previous work on BLISS by a factor of 100 in terms of cycle counts for an entire run of the signature scheme and the use of optimized arithmetic functions for integer reductions, polynomial multiplications and polynomial inversions. Our lightweight implementation only needs 32 KB of Flash memory and 17 KB of RAM.

Keywords: ARM Cortex-M4 · Post-Quantum Cryptography · Authenticated Key Exchange · Lattice-based Cryptography.

1 Introduction

The ubiquitous threat posed by recent advances in quantum computing to currently employed public key schemes has caused a massive rise in research activities in the area of *post-quantum cryptography* (PQC) in the last couple of years. It is already known that given a fairly powerful quantum computer, one can break cryptographic schemes based on the discrete logarithm problem or the prime factorization problem using Shor's algorithm [18]. As almost all of todays public key schemes employ one of the latter two problems to guarantee their security, the consequences of a powerful real world quantum computer would be devastating to the entire digital infrastructure as implemented today. These research efforts culminated in the NIST requesting candidates for quantum-secure cryptographic algorithms in December 2016. Motivated by those advances in PQC and the rise of public attention to that area of cryptography, we want to show and improve upon the practical applications of such algorithms on constrained devices.

1.1 Related Work

The ring variant of the Learning with Errors problem (ring-LWE) brought forward several new proposals of post-quantum primitives such as NewHope [2], which has already seen multiple optimized implementations for different platforms [3, 19]. NewHope is parametrized in a very conservative way, therefore the authors also propose a lightweight parameter set called JarJar that is expected to have a better performance and still achieves 118 bits of post-quantum security. Other lattice-based constructions include the BLISS signature scheme proposed in [10] and implemented in [15] and its improvement BLISS-B [9], as well as the *key encapsulation mechanism* (KEM) Kyber [7] to only name a few. The latter is one of the first lattice-based key exchange algorithms that also considers a native approach to authentication without requiring additional signatures. A way to combine an arbitrary KEM and a digital signature to build an *authenticated key exchange* (AKE) has been presented in [8]. Furthermore, dozens of new algorithms have been proposed in the context of the NIST post-quantum standardization call and only time will tell which of those proof worthy enough to be considered for implementation in real-world scenarios.

1.2 Contribution

A lot of recent work has only focused on the key exchange part of AKE and authentication has been widely neglected. In this work, we close this gap with the following contributions. Our source code is available online[1].

- To the best of our knowledge we present the first ARM Cortex-M4 implementation of the BLISS-B digital signature algorithm and the JarJar-Simple KEM. In [15], Oder et al. present an implementation of BLISS. Since their work does not consider the algorithmic improvements of BLISS-B, the results give a misleading picture of the actual performance of the scheme on embedded devices. By applying further optimizations we achieve a speed-up that is even higher than what could be expected from the algorithmic improvements of BLISS-B. Our implementation of the key generation is three orders of magnitude faster than the implementation of [15].
- We combine both algorithms to construct a fully authenticated AKE protocol. We choose JarJar-Simple and BLISS-B to instantiate the generic AKE construction from [8] to benefit from synergies between both schemes. In particular, the lattice dimension and the modulus of both schemes are the same.
- We also implement the resulting construction on an ARM Cortex-M4 microcontroller to evaluate its practical performance. Our implementation of the AKE protocol is the first of its kind on a constrained device. We managed to improve the performance and reduce the memory requirements by exploiting the aforementioned synergies between JarJar and BLISS-B.

Our results quantify the often ignored overhead cost for applying authentication to otherwise non-authenticated key exchange schemes.

[1] https://www.seceng.rub.de/research/publications/ake-m4/

2 Preliminaries

2.1 Notation

A single value x is always given in regular font, but can be written in lowercase as well as uppercase. Polynomials in $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$ are denoted by a lowercase bold letter like $\mathbf{a}$. Their elements are written as $a_i \in \mathbf{a}$ with i starting at $i = 0$. A polynomial in NTT domain is denoted using a tilde, e.g., $\tilde{\mathbf{a}}$. A matrix is given as a bold uppercase letter like $\mathbf{A}$ with its elements being given as $a_{i,j} \in \mathbf{A}$. The zero-centered Gaussian distribution with standard deviation σ is denoted by $D_\sigma(x)$.

2.2 Lattice-based Cryptography

We instantiate the AKE from [8] with the JarJar-Simple key exchange scheme and the BLISS-B signature scheme.

JarJar-Simple. JarJar-Simple is a specific parameter set of the NewHope-Simple [1] key exchange scheme proposed in 2016. The *Simple* suffix refers to a simplified reconciliation algorithm compared to the original NewHope variant published in [2]. It is build upon ideal lattices and the ring-LWE problem. A CCA-secure variant of NewHope-Simple has also been submitted to the NIST call for post-quantum proposals. We restrain our description to the JarJar-Simple instantiation, which employs polynomials of length $n = 512$ and a prime $q = 12289$. We also chose to follow the notation of a KEM to simplify its use within the AKE, which results in the three algorithms KeyGen_{KEM} (Algorithm 10), Enc (Algorithm 11), and Dec (Algorithm 12) in Appendix B.

The basic idea is that one party chooses a random bit string $v \in \{0,1\}^{256}$ that is later used as the base for the session key. Each bit of the string is encoded into two coefficients of the polynomial to reduce the possibility of decrypting failures. Details on the encoding and compression functions can be found in Appendix A. It is then sent to the other party hidden by the public key and polynomials with binomial distributed coefficients. The number of required input bits to the binomial sampler is $m = 48$ for JarJar-Simple.

BLISS-B. BLISS [10] is based on [13] and comes with a practical instantiation employing a NTRU version of the ring-SIS problem. It was proposed in 2013 and still is one of the best performing lattice-based Fiat-Shamir signature schemes. Our description of the signature scheme focuses on its NTRU instantiation and does not discuss the general case. Furthermore, we employ the more advanced BLISS-B presented in [9]. The signature scheme comes with five different parameter sets described in Appendix C in more detail. Our implementations focuses on the BLISS-B3 parameter set, though it supports most remaining sets as well.

Algorithm 1: BLISS-B KeyGen_{Sig}

```
Result: public verification key $k_v = \mathbf{A}$ and secret signing key
        $k_s = (\mathbf{A}, \mathbf{S})$
1 begin
2     Choose $\mathbf{f}, \mathbf{g}$ as uniform polynomials with exactly $d_1 = \lceil \delta_1 n \rceil$ entries
        in $\{\pm 1\}$ and $d_2 = \lceil \delta_2 n \rceil$ entries in $\{\pm 2\}$
3     if $\mathbf{f}$ is not invertible then restart
4     $\mathbf{a}_q = (2\mathbf{g} + 1)/\mathbf{f} \bmod q$
5     $\mathbf{A} = (\mathbf{a}_1, q-2) \leftarrow (2\mathbf{a}_q, q-2) \bmod 2q$
6     $\mathbf{S} = (\mathbf{s}_1, \mathbf{s}_2)^t \leftarrow (\mathbf{f}, 2\mathbf{g}+1)^t$
```

Algorithm 2: BLISS-B Sig

```
Data: message $m$, secret signing key $k_s = (\mathbf{A}, \mathbf{S})$
Result: signature $s = (\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$ of message $m$
1 begin
2     $\mathbf{y}_1, \mathbf{y}_2 \overset{\$}{\leftarrow} D_\sigma^n$
3     $\mathbf{u} \leftarrow \zeta \cdot \mathbf{a}_1 \cdot \mathbf{y}_1 + \mathbf{y}_2 \bmod 2q$
4     $\mathbf{c} \leftarrow \mathsf{H}(\lfloor \mathbf{u} \rceil_d \bmod p, m)$
5     $(\mathbf{v}_1, \mathbf{v}_2) \leftarrow \mathsf{GreedySC}(\mathbf{S}, \mathbf{c})$
6     $b \overset{\$}{\leftarrow} \{0, 1\}$
7     $(\mathbf{z}_1, \mathbf{z}_2) \leftarrow (\mathbf{y}_1, \mathbf{y}_2) + (-1)^b \cdot (\mathbf{v}_1, \mathbf{v}_2)$
8     continue with certain probability else restart (rejection step)
9     $\mathbf{z}_2^\dagger \leftarrow (\lfloor \mathbf{u} \rceil_d - \lfloor \mathbf{u} - \mathbf{z}_2 \rceil_d) \bmod p$
```

Key generation, as depicted in Algorithm 1, is strongly related to the NTRU key generation. The signing process shown in Algorithm 2 needs to make sure that $\|\mathbf{Sc}\|_2$ stays below a certain limit. In the original BLISS this has been done by checking a special bound during KeyGen_{Sig}. BLISS-B forgoes this check and instead uses $\mathsf{GreedySC}$ given in Appendix D to take that limit into account. This leads to a massive speedup during KeyGen_{Sig}, which is the main reason why we chose BLISS-B in the first place. Sig then goes on to perform rejection sampling on $\mathbf{z}$ ensuring that $\mathbf{z}$ does not leak any information about the secret key. Verification is depicted in Algorithm 3 and is similar for BLISS as well as BLISS-B. It simply verifies two bounds and computes a $\mathbf{c}'$ to compare it to the $\mathbf{c}$ it receives from the signer.

Algorithm 3: BLISS-B Ver

```
Data: message $m$, signature $s = (\mathbf{z}_1, \mathbf{z}_2^\dagger, \mathbf{c})$, public verification key
      $k_v = \mathbf{A}$
1 begin
2     if $\|(\mathbf{z}_1 | 2^d \cdot \mathbf{z}_2^\dagger)\|_2 > B_2$ then reject
3     if $\|(\mathbf{z}_1 | 2^d \cdot \mathbf{z}_2^\dagger)\|_\infty > B_\infty$ then reject
4     if $\mathbf{c} = \mathsf{H}(\lfloor \zeta \cdot \mathbf{a}_1 \cdot \mathbf{z}_1 + \zeta \cdot q \cdot \mathbf{c} \rceil_d + \mathbf{z}_2^\dagger \bmod p, m)$ then accept
```

AKE The AKE used in our implementation is based on the work of del Pino et al. [8], who originally proposed their scheme in combination with an NTRU-based KEM and a hash-and-sign signature. For their special choice of algorithms the size of the transmissions would decline by around 20% since the message could be reconstructed from the signature and could therefore be omitted during transmission. However, the selected digital signature scheme requires between 300Kb and 700Kb of memory, which is not feasible on constrained devices like ours. Since their AKE is a generic construction, as we can see in Figure 1, it allows to plug in any two post-quantum algorithms fitting the description of a KEM and a digital signature. We therefore choose to employ the schemes presented at the beginning of this section, namely JarJar-Simple and BLISS-B. The advantage of those schemes is that they both employ polynomials of length $n = 512$ and modulus $q = 12289$, making their implementation much smaller in terms of code-size.

$$\begin{array}{rcl}
\textbf{Party A} & & \textbf{Party B}\\
sk^{(1)} \leftarrow \perp & & sk^{(2)} \leftarrow \perp \\
(k_d, k_e) \leftarrow \mathsf{KeyGen}_{KEM}(1^{\lambda}) & & \\
s_1 \leftarrow \mathsf{Sig}\left(k_s^{(1)};\ k_e\right) & \xrightarrow{(k_e, s_1)} & \texttt{if } \mathsf{Ver}\left(k_v^{(1)};\ s_1\right) = 1 \texttt{ then} \\
 & & \quad (c, k) \leftarrow \mathsf{Enc}(k_e) \\
 & & \quad a \leftarrow \mathsf{H}_2(s_1, k_e, c, k) \\
 & & \quad s_2 \leftarrow \mathsf{Sig}\left(k_s^{(2)};\ (c, a)\right) \\
\texttt{if } \mathsf{Ver}\left(k_v^{(2)};\ s_2\right) = 1 \texttt{ then} & \xleftarrow{((c, a), s_2)} & \texttt{else ABORT} \\
k' \leftarrow \mathsf{Dec}(k_d;\ c) \quad & & \\
\texttt{else ABORT} & & \\
 & & \\
\texttt{if } a = \mathsf{H}_2(s_1, k_e, c, k') \texttt{ then} & & \\
sk^{(1)} \leftarrow \mathsf{H}_1(s_1, k_e, s_2, (c, a), k') \quad & & sk^{(2)} \leftarrow \mathsf{H}_1(s_1, k_e, s_2, (c, a), k)
\end{array}$$

Fig. 1. Generic 2-round AKE protocol.

3 Implementation

Our implementation targets the *STM32F4DISCOVERY* platform, which is based on the 32-bit ARM Cortex-M4F architecture. It is clocked at up to 168MHz and features 196KB of RAM in addition to 1MB of Flash memory. The most notable features of the board in respect to our work are the on-board *true random number generator* (TRNG) based on an analog circuit, as well as the DSP extension of the Thumb-1 and Thumb-2 instruction sets.

3.1 Randomness and Sampling

All of the schemes presented in our work at some point require a polynomial to be sampled according to some distribution. While JarJar-Simple employs a binomial sampler, BLISS-B samples polynomials according to a Gaussian distribution or in a special way that ensures a uniform distribution of $d_1 = \lceil \delta_1 n \rceil$ entries in $\{\pm 1\}$ and $d_2 = \lceil \delta_2 n \rceil$ entries in $\{\pm 2\}$. The latter one is called *delta sampler* in the following. Each of those samplers awaits uniform randomness as an input to generate the respective output. Our target platform presents us with two approaches to provide that uniform randomness: Using the TRNG or employing a PRNG seeded by the TRNG. We restrain ourselves to ChaCha20 [5] as a PRNG in this work and make use of the optimized implementation from [3].

For each sampler we evaluated the performance of the sampler using input either from a TRNG or a PRNG. The naive expectation is that the TRNG is slower than the PRNG. The results of our experiments in Table 1 show that this is only true for the binomial sampler. This is because the TRNG is an independent hardware module on the microcontroller platform that runs in parallel to other computations on the Cortex-M4. It indeed requires 40 cycles of a 48 MHz clock to sample 32 true random bits. But assuming a sufficiently large amount of time between two calls to the TRNG, the 32-bit sample can be accessed immediately. Therefore samplers requiring a large amount of randomness at once perform better when using a PRNG as a source for the uniform bits, while others with sporadic calls to the RNG profit from the use of a TRNG. To maximize the performance of our implementation we use a TRNG input for the discrete Gaussian sampler and a PRNG input for binomial and delta sampling.

Table 1. Number of cycles it took to sample a full polynomial using a TRNG or PRNG respectively for $m = 48$, $\sigma = 215$, $d_1 = 154$, and $d_2 = 0$.

Sampler	**TRNG**	**ChaCha20**
Binomial	112 806	104 638
Gaussian	935 608	1 042 453
Delta	13 984	20 761

3.2 Integer Arithmetic

For integer additions in $\mathbb{Z}_q$ we use the Barret reduction [4] given in Algorithm 4, while we employ the Montgomery reduction [14] given in Algorithm 5 only in the context of multiplications. Both algorithms are used in their short variants, e.g., returning results in $[0, 2q)$. Since a full reduction into $[0, q)$ is not always required, this approach saves the cost of an additional conditional subtraction for each execution of one of the algorithms.

Barret Reduction Setting $k = 16$ and $m = 5$ for the Barret reduction ensures correct results for all values below $1/(1/q - m/2^k) = 196864$, which always holds for the 16-bit integers used throughout our implementation.

Montgomery Reduction For the Montgomery reduction we choose $R = 2^{18}$ and $q' \equiv -q^{-1} \bmod R$ for which correctness has already been shown in [2] in the context of NewHope. Since JarJar-Simple and BLISS-B employ the same prime q as NewHope and do not differ in their application of the Montgomery reduction, those results can easily be transferred to our work.

Algorithm 4: short BRed

```
  Data: $a$
  Result: $r$
1 begin
2     $r \leftarrow (a \cdot m) >> k$
3     $r \leftarrow a - (q \cdot r)$
```

Algorithm 5: short MRed

```
  Data: $T$
  Result: $t = T \cdot R^{-1} \bmod q$
1 begin
2     $U \leftarrow T \cdot q' \bmod R$
3     $t \leftarrow \frac{T + U \cdot q}{R}$
```

Reduction mod 2q BLISS-B requires some operations to be performed modulo $2q$. One of those is the equation $\zeta \cdot \mathbf{a}_1 \cdot \mathbf{x}$ with $\mathbf{a}_1$ being $\mathbf{a}_q$ lifted into $\mathcal{R}_{2q}$ by a multiplication with 2 during KeyGen_{Sig} and $\mathbf{x}$ being any arbitrary polynomial in $\mathcal{R}_q$. By omitting the multiplication with 2 during key generation, we get $\mathbf{a}_1 = \mathbf{a}_q$ and need to compute $2\zeta \cdot \mathbf{a}_1 \cdot \mathbf{x}$. While $\mathbf{a}_1 \cdot \mathbf{x}$ can easily be done in $\mathcal{R}_q$ using the NTT, Theorem 1 shows an easy way to compute the remaining multiplications for each coefficient. The whole computation can thus be brought down to a single application of the NTT and a conditional addition.

Theorem 1. *For $\zeta = 6145$ and $2q = 24578$ it holds that*

$$2 \cdot \zeta \cdot x \equiv \begin{cases} x \ \mathit{mod}\ 2q & \mathit{if}\ x\ \mathit{is\ even} \\ x + 12289\ \mathit{mod}\ 2q & \mathit{if}\ x\ \mathit{is\ odd} \end{cases}$$

Correctness. For any even x we can write $x = 2k$ with $k \in \mathbb{Z}_0$ and therefore $2 \cdot 6145 \cdot x = 12290 \cdot 2k = 24580 \cdot k \equiv 2k = x \bmod 24578$.
For any odd x we can write $x = 2k + 1$ with $k \in \mathbb{Z}_0$ and therefore $2 \cdot 6145 \cdot x = 12290 \cdot (2k + 1) = 24580 \cdot k + 12290 \equiv 2k + 1 + 12289 = x + 12289 \bmod 24578$.

Note that x, which represents one coefficient of a polynomial, does not need to be fully reduced modulo q. The theorem still holds for $x \in [0, 2q)$ as output by the NTT.

3.3 Polynomial Arithmetic

Multiplication The state-of-the-art technique for polynomial multiplication in ideal lattice-based cryptography is the number-theoretic transform (NTT) [3, 7, 15–17, 19]. The NTT consists of $\log n$ layers and in each layer the coefficients of the input polynomial are combined pair-wise. Internally, the NTT uses powers of the two parameters ψ and $\omega = \psi^2$ that can be either precomputed or generated on-the-fly. Our implementation of the NTT is based on [3] and has been adapted for $n = 512$ instead of $n = 1024$. The number of layers is therefore reduced to $\log_2(512) = 9$. The authors of [3] are able to achieve a significant speedup by merging three of those layers at a time, resulting in three segments of three layers each in our case. We use precomputed constants for all powers of ψ, ψ^{-1}, ω and ω^{-1} and store them in Montgomery domain, since the Montgomery reduction is used for all multiplications within the NTT. In total those precomputed powers require 3072B of memory, 1024B for the powers of ψ and ψ^{-1} respectively and two times 512B for the powers of ω and ω^{-1}.

Inversion Our inversion is based on Fermat's little theorem, which states that the inverse of an element can be computed as $\mathbf{a}^{-1} \equiv \mathbf{a}^{q-2}$ in $\mathcal{R}_q$. For performance reasons we opt to perform exponentiation in NTT domain such that the theorem can be applied to each coefficient individually. It is therefore sufficient to compute $\tilde{a}_i^{-1} = \tilde{a}_i^{q-2} \bmod q$ for $i = 1, \ldots, n$. Since our exponent is fixed, we are able to apply addition chain exponentiation to compute the inverse. Finding an optimal addition chain is easy for small exponents like ours. We stick with the addition chain given in [15] and verify that it is indeed optimal using the data from [12]. Additionally, Montgomery reductions are used to speed up the computations.

3.4 JarJar-Simple

Large parts of our JarJar-Simple implementation are taken from the NewHope implementation of [3] and [19]. Since both implementations focus on the parameters of NewHope(-Simple), we have adapted them for JarJar-Simple whenever necessary. The authors of [3] have already shown how to reduce the memory footprint of NewHope by reordering some operations to enable the reuse of several memory locations. We transfer their ideas to JarJar-Simple and our implementation shows that their results still hold in our case. Internal memory for three polynomials is still sufficient during encapsulation, although the structure of the algorithm has changed. The same holds for decapsulation, which only requires space for two polynomials in total.

3.5 BLISS-B

We base our work on the implementations of [15] and [11]. One of the biggest changes we apply to all three algorithms of BLISS-B is the use of 16-bit arrays instead of 32-bit ones to store the polynomials internally. This is unproblematic since both q and $2q$ easily fit into 16 bits and all reductions make sure to generate respective results. We therefore halve the memory required to store those polynomials. This also enables us to make use of the synergies between JarJar-Simple and BLISS-B by, e.g., using the same implementation of the NTT and multiplication functions for both schemes.

The new NTT implementation expects inputs within $[0, 2q)$ although sampled polynomials are in $(-q, q)$. For some polynomials we are therefore required to store two versions. However, this can be done by reusing already existing arrays and therefore does not lead to a higher memory consumption. We also improve the overall application of the NTT by, e.g., performing the negation of a polynomial in NTT domain as well. Negation can be seen as the multiplication with a polynomial for which the first coefficient is set to -1 and all the remaining ones are being set to 0. The NTT representation of that polynomial is $(-1, -1, \ldots, -1)$, resulting in each coefficient in NTT domain being multiplied by -1. While this does not decrease the number of multiplications, it enables us to keep the respective polynomial in NTT domain for further multiplications and avoid the use of additional transforms.

Switching to a representation in $[0, 2q)$ does also enable us to make use of the Barret and Montgomery reduction routines, which brings along a significant speedup.

3.6 Building the AKE

The implementation of the AKE itself is rather straight forward and follows Figure 1. With the two main components of the AKE, namely JarJar-Simple and BLISS-B, being discussed in the previous section, the only remaining building block is the hash function used to generate the authentication string a and the session keys $sk^{(1)}$ and $sk^{(2)}$.

We first observe that the H_1 and H_2 have partial overlaps in their inputs. We set $\mathsf{H}_1 = \mathsf{H}_2 = \mathsf{SHA3} - \mathsf{256}$ using the Keccak implementation from [6]. By reordering the inputs of both hash functions to $\mathsf{SHA3} - \mathsf{256}(k, k_e, s_1, c)$ and $\mathsf{SHA3} - \mathsf{256}(k, k_e, s_1, (c, a), s_2)$ we can make use of those overlaps to avoid hashing the same inputs twice. We furthermore modify the Keccak implementation such that it is split into four phases: Initialization, absorption, final absorption and squeezing. We now proceed by running the initialization resulting in a fresh Keccak state called `instance_auth`. We then call the absorption function on the inputs k, k_e, s_1 and c. After those values have been absorbed, we copy `instance_auth` into a second state called `instance_key` and apply final absorption and squeezing to `instance_auth` to generate the authentication string a. The second Keccak state can then be used to additionally absorb a and s_2 before again performing the final absorption and squeezing to obtain the session key $sk^{(1)}$ or $sk^{(2)}$ respectively. This has the obvious advantage of performing faster compared to the naive approach and does additionally avoid us being required to copy all inputs to the hash function into a single array, which outweighs the additional 208B occupied by the second Keccak state.

4 Results and Comparison

The results in this section have been obtained using the *System Workbench for STM32* toolchain in version 2.4. While the cycle counts are measured using the on-board *Data Watchpoint and Trace Unit* (DWT), we estimate the RAM consumption by counting the allocated memory space of each function.

Performance. The results for the major building blocks of JarJar-Simple and BLISS-B are given in Table 2. We focus on the building blocks that are affected by changes we made to the reference implementations and that have the most impact on the total cycle count.

The transforms do also include a bit-reversal step as well as the multiplication with the respective powers of ψ and ψ^{-1}. Looking at the polynomial multiplication and inversion we want to stress that both functions require their inputs to already be transformed into NTT domain.

Table 2. Number of cycles of the main building blocks of the AKE averaging 1000 measurements.

Routine	Cycles	Routine	Cycles
NTT	45 494	Sample binomial	104 638
NTT^{-1}	45 494	Sample Gaussian	1 091 387
Bit-Reversal	5 312	Sample delta	23 268
Polynomial Multiplication	7 470	Generate $\mathbf{c}$	146 866
Polynomial Inversion	83 210	$\mathsf{GreedySC}$	578 593

Table 3 compares our BLISS-B implementation to the results of [15] using regular BLISS. Since [15] only describes the BLISS-1 parameter set we choose BLISS-B1 for a fair comparison, though our AKE implementation still employs BLISS-B3. The huge difference in cycles for the KeyGen_{Sig} algorithms stems from algorithmic differences between BLISS and BLISS-B and to a lesser extent from our changes to the NTT and the polynomial inversion. The speedup due to our optimizations is most prominent in the verification algorithm Ver. Although the BLISS-B Sig algorithm is more complex than the BLISS one, it is still faster than [15] due to a smaller repetition rate M and the use of our improvements. Running all three algorithms of BLISS-B takes roughly 22ms in total.

We compare our JarJar-Simple implementation to the NewHope one given in [3] to show the factor of speedup that can be achieved by simply choosing a smaller dimension. The results for both ciphers can be found in Table 4. At a clock frequency of 168MHz a full key exchange using JarJar-Simple would only take about 8.2ms on our hardware, which is faster than the NewHope implementation by a factor of 1.78. One would expect the cycle count to drop by more than a factor of 1/2 due to the NTT's complexity of $n \log n$. Due to the larger standard deviation used in the sampler and more randomness being required per coefficient as a result, this is not the case in our implementation.

We compare our AKE construction to the Module-LWE based Kyber key exchange protocol presented in [7]. The results claimed for Kyber stem from

Table 3. Cycle count for BLISS-1 [15] and BLISS-B1 (our work).

Algorithm	**BLISS-B**	**BLISS**
KeyGen_{Sig}	256 535	367 859 092
Sig	3 171 944	5 927 441
Ver	300 201	1 002 299

Table 4. Cycle count for NewHope [3] and JarJar-Simple (our work).

Algorithm	**JarJar-Simple**	**NewHope**
KeyGen_{KEM}	507 744	906 902
Enc	744 367	1 312 000
Dec	89 758	173 000

a ported and slightly adapted version of the Kyber reference implementation using full authentication for both parties. As one can see from Table 5 our AKE implementation compares rather favorable to the Kyber one in terms of speed, although we want to stress that Kyber comes along with greater security claims. The cycle count of the precomputations is not critical since they could in most scenarios also be done external. However, considering the remaining three algorithms our AKE implementation sees a speedup of a factor of about 2 for KeyGen_A and Shared_B as well as a speedup of almost a factor of 7 for Shared_A. A full run of the AKE requires 62ms on average excluding precomputations and when being clocked at 168MHz, which makes it applicable in real world scenarios.

Table 5. Cycle count for our AKE implementation compared to a ported but unoptimized version of Kyber [7].

Algorithm	**AKE**	**Kyber**
Precomputations	517 377	6 590 440
KeyGen_A	3 900 854	7 354 193
Shared_B	5 333 723	11 940 641
Shared_A	1 124 200	7 598 468

Memory. We only focus on the functions with the largest memory footprint. Table 6 shows the memory consumed by those functions internally and excludes inputs such as keys, messages or signatures.

All arithmetic functions work exclusively on their inputs and require very little internal memory such that they are not being discussed any further in this section. The same applies for Gaussian and delta sampling, since they employ the TRNG and do not require to store randomness in a buffer.

Table 7 shows the memory consumed be keys and signatures. Since they are omitted in Table 6, they need to be considered for each of the functions separately. The AKE precomputations total 1024B and operate on two public and two secret BLISS-B keys resulting in an additional 6144B. Note that each secret key does also contain its public key and it is sufficient to store the public key once. KeyGen_A requires 6992B in addition to 6620B for a signature, k_s, k_d,

Table 6. Estimates of RAM memory footprints by building block in bytes.

Routine	Size
Sample uniform	880
Sample binomial	3 216
Generate **c**	1 360
BLISS-B KeyGen_{Sig}	1 024
BLISS-B Sig	6 992
BLISS-B Ver	2 444

Routine	Size
JarJar-Simple KeyGen_{KEM}	5 296
JarJar-Simple Enc	6 320
JarJar-Simple Dec	2 256
AKE precomputations	1 024
AKE KeyGen_A	6 992
AKE Shared_B	7 232
AKE Shared_A	2 716

and k_e. The memory footprint of Shared_B totals 7232B excluding 9560B for keys, signatures, and messages. On the other side of the key exchange Shared_A adds up to 2716B additionally to 7512B used for its inputs.

Table 7. Memory footprints of keys and signatures in bytes.

Element	Size
BLISS-B k_s	3 072
BLISS-B k_v	1 024
BLISS-B s	1 596

Element	Size
JarJar-Simple k_e	928
JarJar-Simple k_d	1 024
JarJar-Simple c	1 280
JarJar-Simple k	32

5 Conclusion

In this work, we presented the first microcontroller implementation of a lattice-based authenticated key exchange scheme. Our ARM Cortex-M4 implementation achieves a performance of 62ms on average for an entire run of the protocol. By our careful choice of lightweight parameters, we managed to keep the memory requirement as low as 32 KB of Flash and 17 KB of RAM memory. Our results reveal the cost of transforming an unauthenticated key exchange scheme into an authenticated one on embedded devices. For future work we plan to also incorporate side-channel countermeasures.

Acknowledgements

The research in this work was supported in part by the European Unions Horizon 2020 program under project number 644729 SAFEcrypto.

References

1. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Newhope without reconciliation. Cryptology ePrint Archive, Report 2016/1157 (2016), https://eprint.iacr.org/2016/1157
2. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key exchange–a new hope. In: 25th USENIX Security Symposium, August 10-12, 2016, Austin, TX, USA. pp. 327–343. USENIX (2016)
3. Alkim, E., Jakubeit, P., Schwabe, P.: Newhope on ARM Cortex-M. In: International Conference on Security, Privacy, and Applied Cryptography Engineering. pp. 332–349. Springer (2016)
4. Barrett, P.: Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In: Conference on the Theory and Application of Cryptographic Techniques. pp. 311–323. Springer (1986)
5. Bernstein, D.J.: ChaCha, a variant of Salsa20. In: Workshop Record of SASC. vol. 8, pp. 3–5 (2008)
6. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., Van Keer, R.: Keccak code package. https://github.com/gvanas/KeccakCodePackage
7. Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Stehlé, D.: CRYSTALS–Kyber: a CCA-secure module-lattice-based KEM. IACR Cryptology ePrint Archive **2017**, 634 (2017)
8. Del Pino, R., Lyubashevsky, V., Pointcheval, D.: The whole is less than the sum of its parts: Constructing more efficient lattice-based AKEs. In: International Conference on Security and Cryptography for Networks. pp. 273–291. Springer (2016)
9. Ducas, L.: Accelerating BLISS: the geometry of ternary polynomials. IACR Cryptology ePrint Archive **2014**, 874 (2014)
10. Ducas, L., Durmus, A., Lepoint, T., Lyubashevsky, V.: Lattice signatures and bimodal gaussians. In: Advances in Cryptology–CRYPTO 2013, pp. 40–56. Springer (2013)
11. Dutertre, B., A. Mason, I., Lepoint, T.: Bliss: Bimodal lattice signature schemes. https://github.com/SRI-CSL/Bliss
12. Flammenkamp, A.: Shortest addition chains. http://wwwhomes.uni-bielefeld.de/achim/addition_chain.html, accessed: 2018-01-02
13. Lyubashevsky, V.: Lattice signatures without trapdoors. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 738–755. Springer (2012)
14. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of computation **44**(170), 519–521 (1985)
15. Oder, T., Pöppelmann, T., Güneysu, T.: Beyond ECDSA and RSA: Lattice-based digital signatures on constrained devices. In: Proceedings of the 51st Annual Design Automation Conference. pp. 1–6. ACM (2014)
16. Pöppelmann, T., Ducas, L., Güneysu, T.: Enhanced lattice-based signatures on reconfigurable hardware. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 353–370. Springer (2014)
17. Saarinen, M.J.O.: Arithmetic coding and blinding countermeasures for lattice signatures. Journal of Cryptographic Engineering pp. 1–14 (2017)
18. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM review **41**(2), 303–332 (1999)
19. Streit, S., De Santis, F.: Post-quantum key exchange on ARMv8-A: A new hope for NEON made simple. IEEE Transactions on Computers (2017)

A JarJar-Simple Encoding and Compression

JarJar-Simple requires a bit-string to be encoded within a polynomial of ring $\mathcal{R}_q$. Since the algorithm works on polynomials of length 512 by default, one can encode each bit into two coefficients to reduce the failure rate of the algorithm. The encoding function is given in Algorithm 6, while the decoding is given in Algorithm 7. While encoding is rather straight-forward and only consists of two multiplications, decoding is not that intuitive. Due to the introduction of error polynomials, the encoded and transmitted polynomial might be slightly off compared to the original encoded polynomial. Therefore decoding is build upon a threshold decision that mitigates the chances of decoding errors.

Algorithm 6: JJSEncode

```
Data: bitstring $v \in \{0,1\}^{256}$
Result: polynomial $\mathbf{k} \in \mathcal{R}_q$
1 begin
2     for $i \leftarrow 0$ to $n/2$ do
3         $k_i \leftarrow v_i \cdot \lfloor q/2 \rfloor$
4         $k_{i+256} \leftarrow v_i \cdot \lfloor q/2 \rfloor$
```

Algorithm 7: JJSDecode

```
Data: polynomial $\mathbf{k}' \in \mathcal{R}_q$
Result: bitstring $v \in \{0,1\}^{256}$
1 begin
2     for $i \leftarrow 0$ to $n/2$ do
3         $t \leftarrow |(k'_i - \lfloor q/2 \rfloor) + (k'_{i+256} - \lfloor q/2 \rfloor)|$
4         if $t < q/2$ then
5             $v_i \leftarrow 1$
6         else
7             $v_i \leftarrow 0$
```

When sending the encoded and encrypted key we can compress the polynomial containing the key to only three bits, since decoding can take care of the resulting inaccuracies. Compression and decompression are done by simply switching between the moduli q and 8 as specified in the routines given in Algorithm 8 and Algorithm 9.

Algorithm 8: JJSCompress

```
Data: polynomial $\mathbf{c} \in \mathcal{R}_q$
Result: compressed polynomial $\bar{\mathbf{c}} \in \{0,\dots,7\}^n$
1 begin
2     for $i \leftarrow 0$ to $n$ do
3         $\bar{c}_i \leftarrow \lfloor (c_i \cdot 8)/q \rceil \bmod 8$
```

```
Algorithm 9: JJSDecompress
  Data: compressed polynomial $\bar{\mathbf{c}} \in \{0, \dots, 7\}^n$
  Result: polynomial $\mathbf{c} \in \mathcal{R}_q$
1 begin
2     for $i \leftarrow 0$ to $n$ do
3         $c_i \leftarrow \lfloor(\bar{c}_i \cdot q)/8\rceil$
```

B JarJar

```
Algorithm 10: JarJar-Simple KeyGen$_{KEM}$
  Result: decapsulation key $k_d = \mathbf{s}$ and encapsulation key $k_e = (\mathbf{b}, seed)$
1 begin
2     $seed \xleftarrow{\$} \{0,1\}^{256}$
3     $\mathbf{a} \leftarrow \mathsf{Parse}(\mathsf{SHAKE-128}(seed))$
4     $\mathbf{s}, \mathbf{e} \xleftarrow{\$} B_m^n$
5     $\mathbf{b} \leftarrow \mathbf{as} + \mathbf{e}$
```

```
Algorithm 11: JarJar-Simple Enc
  Data: encapsulation key $k_e = (\mathbf{b}, seed)$
  Result: pair $(c, k)$ with $k = \mu$ being the key and $c = (\mathbf{u}, \bar{\mathbf{c}})$ being the
          ciphertext encapsulating $k$
1 begin
2     $\mathbf{s}', \mathbf{e}', \mathbf{e}'' \xleftarrow{\$} B_m^n$
3     $\mathbf{a} \leftarrow \mathsf{Parse}(\mathsf{SHAKE-128}(seed))$
4     $\mathbf{u} \leftarrow \mathbf{as}' + \mathbf{e}'$
5     $v \xleftarrow{\$} \{0,1\}^{256}$
6     $v' \leftarrow \mathsf{SHA3-256}(v)$
7     $\mathbf{k} \leftarrow \mathsf{JJSEncode}(v')$
8     $\mathbf{c} \leftarrow \mathbf{bs}' + \mathbf{e}'' + \mathbf{k}$
9     $\bar{\mathbf{c}} \leftarrow \mathsf{JJSCompress}(\mathbf{c})$
10    $\mu \leftarrow \mathsf{SHA3-256}(v')$
```

```
Algorithm 12: JarJar-Simple Dec
  Data: decapsulation key $k_d = \mathbf{s}$, ciphertext $c = (\mathbf{u}, \bar{\mathbf{c}})$
  Result: key $k = \mu$
1 begin
2     $\mathbf{c}' \leftarrow \mathsf{JJSDecompress}(\bar{\mathbf{c}})$
3     $\mathbf{k}' \leftarrow \mathbf{c}' - \mathbf{us}$
4     $v' \leftarrow \mathsf{JJSDecode}(\mathbf{k}')$
5     $\mu \leftarrow \mathsf{SHA3-256}(v')$
```

C BLISS-B Parameter Sets

BLISS-B comes with five different parameter sets, the first of which is called BLISS-B0 and is merely just a set of parameters that shall only be used for experimental purpose. Since it is irrelevant in the real world we omit it in our description entirely. Table 8 shows the four remaining parameter sets. All of them employ a common dimension of $n = 512$ as well as the same prime $q = 12289$.

Table 8. BLISS-B parameter sets.

Parameter	**BLISS-I**	**BLISS-II**	**BLISS-III**	**BLISS-IV**
security (classical)	128 bits	128 bits	160 bits	192 bits
Grover's search ([17])	66 bits	66 bits	80 bits	96 bits
δ_1, δ_2	$0.3, 0$	$0.3, 0$	$0.42, 0.03$	$0.45, 0.06$
σ	215	107	250	271
α	1.61	0.801	1.216	1.027
M	1.21	2.18	1.4	1.61
κ	23	23	30	39
d	10	10	9	8
B_2, B_∞	$12872, 2100$	$11074, 1563$	$10206, 1760$	$9901, 1613$

D BLISS-B GreedySC

```
Algorithm 13: GreedySC
   Data: matrix $\mathbf{S} = [\mathbf{s}_0, \ldots, \mathbf{s}_{n-1}] \in \mathbb{Z}^{2n \times n}$, binary vector $\mathbf{c}$
   Result: vector $\mathbf{v} = \mathbf{S}\mathbf{c}'$ for $\mathbf{c}' = \mathbf{c} \bmod 2$
 1 begin
 2     $\mathbf{v} \leftarrow \mathbf{0} \in \mathbb{Z}^{2n}$
 3     for $i \in \mathcal{I}_c$ do
 4         $\varsigma_i \leftarrow \mathrm{sgn}(\langle \mathbf{v}, \mathbf{s}_i \rangle)$
 5         $\mathbf{v} \leftarrow \mathbf{v} - \varsigma_i \cdot \mathbf{s}_i$
```