Lightweight Post-Quantum Key Encapsulation for 8-bit AVR Microcontrollers

Hao Cheng, Johann Großschädl, Peter B. Rønne, and Peter Y. A. Ryan

DCS and SnT, University of Luxembourg
6, Avenue de la Fonte, L–4364 Esch-sur-Alzette, Luxembourg
{hao.cheng,johann.groszschaedl,peter.roenne,peter.ryan}@uni.lu

Abstract. Recent progress in quantum computing has increased interest in the question of how well the existing proposals for post-quantum cryptosystems are suited to replace RSA and ECC. While some aspects of this question have already been researched in detail (e.g. the relative computational cost of pre- and post-quantum algorithms), very little is known about the RAM footprint of the proposals and what execution time they can reach when low memory consumption rather than speed is the main optimization goal. This question is particularly important in the context of the Internet of Things (IoT) since many IoT devices are extremely constrained and possess only a few kB of RAM. We aim to contribute to answering this question by exploring the software design space of the lattice-based key-encapsulation scheme ThreeBears on an 8-bit AVR microcontroller. More concretely, we provide new techniques for the optimization of the ring arithmetic of ThreeBears (which is, in essence, a 3120-bit modular multiplication) to achieve either high speed or low RAM footprint, and we analyze in detail the trade-offs between these two metrics. A low-memory implementation of BabyBear that is secure against Chosen Plaintext Attacks (CPA) needs just about 1.7 kB RAM, which is significantly below the RAM footprint of other lattice-based cryptosystems reported in the literature. Yet, the encapsulation time of this RAM-optimized BabyBear version is below 12.5 million cycles, which is less than the execution time of scalar multiplication on Curve25519. The decapsulation is more than four times faster and takes roughly 3.4 million cycles on an ATmega1284 microcontroller.

Keywords: Post-quantum cryptography · Key encapsulation mechanism · AVR architecture · Efficient implementation · Low RAM footprint

1 Introduction

In 2016, the U.S. National Institute of Standards and Technology (NIST) initiated a process to evaluate and standardize quantum-resistant public-key cryptographic algorithms and published a call for proposals [14]. This call, whose submission deadline passed at the end of November 2017, covered the complete spectrum of public-key functionalities: encryption, key agreement, and digital signatures. A total of 72 candidates were submitted, of which 69 satisfied the

minimum requirements for acceptability and entered the first round of a multi-year evaluation process. In early 2019, the NIST selected 26 of the submissions as candidates for the second round; among these are 17 public-key encryption or key-encapsulation algorithms and nine signature schemes. The 17 algorithms for encryption (resp. key encapsulation) include nine that are based on certain hard problems in lattices, seven whose security rests upon classical problems in coding theory, and one that claims security from the presumed hardness of the (supersingular) isogeny walk problem on elliptic curves [15]. This second round focuses on evaluating the candidates' performance across a variety of systems and platforms, including "not only big computers and smart phones, but also devices that have limited processor power" [15].

Lattice-based cryptosystems are considered the most promising candidates for deployment in constrained devices due to their relatively low computational cost and reasonably small keys and ciphertexts (resp. signatures). Indeed, the benchmarking results collected in the course of the `pqm4` project[1], which uses a 32-bit ARM Cortex-M4 as target device, show that most of the lattice-based Key-Encapsulation Mechanisms (KEMs) in the second round of the evaluation process are faster than ECDH key exchange based on Curve25519, and some candidates are even notably faster than Curve25519 [10]. However, the results of `pqm4` also indicate that lattice-based cryptosystems generally require a large amount of run-time memory since most of the benchmarked lattice KEMs have a RAM footprint of between 5 kB and 30 kB. For comparison, a variable-base scalar multiplication on Curve25519 can have a RAM footprint of less than 500 bytes [4]. One could argue that the `pqm4` implementations have been optimized to reach high speed rather than low memory consumption, but this argument is not convincing since even a conventional implementation of Curve25519 (i.e. an implementation without any specific measures for RAM reduction) still needs only little more than 500 bytes RAM. Therefore, the existing implementation results in the literature lead to the conclusion that lattice-based KEMs require an order of magnitude more RAM than ECDH key exchange.

The high RAM requirements of lattice-based cryptosystems (in relation to Curve25519) pose a serious problem for the emerging Internet of Things (IoT) since many IoT devices feature only a few kB of RAM. For example, a typical wireless sensor node like the MICAz mote [3] is equipped with an 8-bit micro-controller (e.g. ATmega128L) and comes with only 4 kB internal SRAM. These 4 kB are easily sufficient for Curve25519 (since there would still be 7/8 of the RAM available for system and application software), but not for lattice-based KEMs. Thus, there is a clear need to research how lattice-based cryptosystems can be optimized to reduce their memory consumption and what performance such low-memory implementations can reach. The present paper addresses this research need and introduces various software optimization techniques for the ThreeBears KEM [8], a lattice-based cryptosystem that was selected for the second round of NIST's standardization project. The security of ThreeBears is based on a special version of the Learning With Errors (LWE) problem, the

[1] See `https://www.github.com/mupq/pqm4`.

so-called Integer Module Learning with Errors (I-MLWE) problem [5]. ThreeBears is unique among the lattice-based second-round candidates since it uses an integer ring instead of a polynomial ring as algebraic structure. Hence, the major operation of ThreeBears is integer arithmetic (namely multiplication modulo a 3120-bit prime) and not polynomial arithmetic.

The conventional way to speed up the polynomial multiplication that forms part of lattice-based cryptosystems like classical NTRU or NTRU Prime is to use a multiplication technique with sub-quadratic complexity, e.g. Karatsuba's method [11] or the so-called Toom-Cook algorithm. However, the performance gain due to these techniques comes at the expense of a massive increase of the RAM requirements. For integer multiplication, on the other hand, there exists a highly effective approach for performance optimization that does not increase the memory footprint, namely the so-called hybrid multiplication method from CHES 2004 [6] or one of its variants like the Reverse Product Scanning (RPS) method [13]. In essence, the hybrid technique can be viewed as a combination of classical operand scanning and product scanning with the goal to reduce the number of load instructions by processing several bytes of the two operands in each iteration of the inner loop. Even though the hybrid technique can also be applied to polynomial multiplication, it is, in general, less effective because the bit-length of the polynomial coefficients of most lattice-based cryptosystems is not a multiple of eight.

Contributions. This paper analyzes the performance of ThreeBears on an 8-bit AVR microcontroller and studies its flexibility to achieve different trade-offs between execution time and RAM footprint. Furthermore, we describe (to the best of our knowledge) the first highly-optimized software implementations of BabyBear (an instance of ThreeBears with parameters to reach NIST's security category 2) for the AVR platform. We developed four implementations of BabyBear, two of which are optimized for low RAM consumption, and the other two for fast execution times. Our two low-RAM BabyBear versions are the most memory-efficient software implementations of a NIST second-round candidate ever reported in the literature.

Our work is based on the optimized C code contained in the ThreeBears submission package [8], which adopts a "reduced-radix" representation for the ring elements, i.e. the number of bits per limb is less than the word-size of the target architecture. On a 32-bit platform, a 3120-bit integer can be stored in an array of 120 limbs, each consisting of 26 bits. However, our AVR software uses a radix of 2^{32} (i.e. 32 bits of the operands are processed at a time) since this representation enables the RPS method to reach peak performance and it also reduces the RAM footprint. We present two optimizations for the performance-critical Multiply-ACcumulate (MAC) operation of ThreeBears; one aims to minimize the RAM requirements, while the goal of the second is to maximize performance. Our low-memory implementation of the MAC combines one level of Karatsuba with the RPS method [13] to accelerate the so-called tripleMAC operation of the optimized C source code from [8], which is (relatively) light in

terms of stack memory. On the other hand, the speed-optimized MAC consists of three recursive levels of Karatsuba multiplication and uses the RPS method underneath. We implemented both MAC variants in AVR Assembly language to ensure they have constant execution time and can resist timing attacks.

As already mentioned, our software contains four different implementations of the ThreeBears family: two versions of CCA-secure BabyBear, and two versions of CPA-secure BabyBearEphem. For both BabyBear and BabyBearEphem, we developed both a Memory-Efficient (ME) and a High-Speed (HS) implementation, which internally use the corresponding MAC variant. We abbreviate these four versions as ME-BBear, ME-BBear-Eph, HS-BBear, and HS-BBear-Eph. Our results show that ThreeBears provides the flexibility to optimize for low memory footprint *and* still achieves very good execution times compared to the other second-round candidates. In particular, the CCA-secure BabyBear can be optimized to run with only 2.4 kB RAM on AVR, and the CPA-secure version requires even less memory, namely just 1.7 kB.

2 Preliminaries

2.1 8-bit AVR Microcontrollers

8-bit AVR microcontrollers continue to be widely used in the embedded realm (e.g. smart cards, wireless sensor nodes). The AVR architecture is based on the modified Harvard memory model and follows the RISC philosophy. It features 32 general-purpose working registers (i.e. `R0` to `R31`) of 8-bit width, which are directly connected to the Arithmetic Logic Unit (ALU). The current revision of the AVR instruction set supports 129 instructions in total, and each of them has fixed latency. Examples of instructions that are frequently used in our software are addition and subtraction (`ADD`/`ADC` and `SUB`/`SBC`); they take a single cycle. On the other hand, the multiplication (`MUL`) and also the load and store (`LD`/`ST`) instructions are more expensive since they have a latency of two clock cycles. The specific AVR microcontroller on which we simulated the execution time of our software is the ATmega1284; it features 16 kB SRAM and 128 kB flash memory for storing program code.

2.2 Overview of ThreeBears

ThreeBears has three parameter sets called BabyBear, MamaBear, and PapaBear, matching NIST security categories 2, 4, and 5, respectively. Each parameter set comes with two instances, one providing CPA security and the other CCA security. Taking BabyBear as example, the CPA-secure instance is named BabyBearEphem (with the meaning of ephemeral BabyBear), while the CCA-secure one is simply called BabyBear. In the following, we only give a short summary of the CCA-secure instance of ThreeBears. In contrast to encryption schemes with CCA-security, CPA-secure ones, roughly speaking, do not repeat and verify the key generation and encryption (i.e. encapsulation) as part of the decryption (i.e. decapsulation) procedure, see [8] for details.

Notation and Parameters. ThreeBears operates in the field $\mathbb{Z}/N$, where the prime modulus $N = 2^{3120} - 2^{1560} - 1$ is a so-called "golden-ratio" Solinas trinomial prime [7]. N is commonly written as $N = \phi(x) = x^D - x^{D/2} - 1$. The addition and multiplication $(+, *)$ in $\mathbb{Z}/N$ will be explained in Subsect. 3.1. An additional parameter d determines the the module dimension; this dimension is 2 for BabyBear, 3 for MamaBear and 4 for PapaBear, respectively.

Key Generation. To generate a key pair for ThreeBears, the following operations have to be performed:

1. Generate a uniform and random string sk with a fixed length.
2. Generate two noise vectors $(a_0, \ldots, a_{d-1})$ and $(b_0, \ldots, b_{d-1})$, where a_i, $b_i \in \mathbb{Z}/N$ is sampled from a noise sampler using sk.
3. Compute $r = \textsc{Hash}(sk)$.
4. Generate a $d \times d$ matrix $\boldsymbol{M}$, where each element $M_{i,j} \in \mathbb{Z}/N$ is sampled from a uniform sampler using r.
5. Obtain vector $\boldsymbol{z} = (z_0, \ldots, z_{d-1})$ by computing each $z_i = b_i + \Sigma_{j=0}^{d-1} M_{i,j} * a_j \bmod N$
6. Output sk as *private key* and $(r, \boldsymbol{z})$ as *public key*.

Encapsulation. The encapsulation operation gets a public key $(r, \boldsymbol{z})$ as input and produces a ciphertext and shared secret as output:

1. Generate a uniform and random string *seed* with a fixed-length.
2. Generate two noise vectors $(\hat{a}_0, \ldots, \hat{a}_{d-1})$, $(\hat{b}_0, \ldots, \hat{b}_{d-1})$ and a noise c, where $\hat{a}_i$, $\hat{b}_i$, $c \in \mathbb{Z}/N$ is sampled from noise sampler by given r and *seed*.
3. Generate a $d \times d$ matrix $\boldsymbol{M}$, where each element $M_{i,j} \in \mathbb{Z}/N$ is sampled from uniform sampler by given r.
4. Obtain vector $\boldsymbol{y} = (y_0, \ldots, y_{d-1})$ by computing each $y_i = \hat{b}_i + \Sigma_{j=0}^{d-1} M_{j,i} * \hat{a}_j \bmod N$, and compute $x = c + \Sigma_{j=0}^{d-1} z_j * \hat{a}_j \bmod N$.
5. Use Melas FEC encoder to encode *seed*, and use this encoded output together with x to extract a fixed-length string f.
6. Compute $ss = \textsc{Hash}(r, seed)$.
7. Output ss as *shared secret* and $(f, \boldsymbol{y})$ as *ciphertext*.

Decapsulation. The decapsulation gets a private key sk and ciphertext $(f, \boldsymbol{y})$ as input and produces a shared secret as output:

1. Generate a noise vector $(a_0, \ldots, a_{d-1})$, where $a_i \in \mathbb{Z}/N$ is sampled from a noise sampler by given sk.
2. Compute $x = \Sigma_{j=0}^{d-1} y_j * a_j \bmod N$.
3. Derive a string from f together with x, and use Melas FEC decoder to decode this string to obtain the string *seed*.
4. Generate the public key $(r', \boldsymbol{z'})$ through Key Generation by given sk.
5. Repeat Encapsulation to get ss' and $(f', \boldsymbol{y'})$ by using the obtained *seed* and key pair $(sk, (r', \boldsymbol{z'}))$.

6. Check whether $(f', \boldsymbol{y'})$ equals to $(f, \boldsymbol{y})$; if they are equal then output ss' as *shared secret*; if not then output $\textsc{Hash}(sk, f, \boldsymbol{y})$ as *shared secret*.

The three operations described above use a few auxiliary functions such as samplers (noise sampler and uniform sampler), hash functions, and a function for the correction of errors. Both the samplers and hash functions are based on `cSHAKE256` [12], which uses the Keccak permutation [1] at the lowest layer. In addition, ThreeBears adopts Melas BCH code for Forward Error Correction (FEC) since it is very fast, has low RAM consumption and small code size, and can also be easily implemented to have constant execution time.

We determined the execution time of several implementations contained in the ThreeBears NIST submission package on an AVR microcontroller. Like is the case with other lattice-based cryptosystems, the arithmetic computations of ThreeBears determine the memory footprint and have a big impact on the overall execution time. Hence, our work primarily focuses on the optimization of the costly MAC operation of the form $r = r + a * b \bmod N$. Concerning the auxiliary functions, we were able to significantly improve their performance (in relation to the C code of the submission package) thanks to a highly-optimized AVR assembler implementation of the permutation of Keccak[2]. Further details about the auxiliary functions are outside of the scope of this work; we refer the reader to the specification of ThreeBears [8].

3 Optimizations for the MAC Operation

The multiply-accumulate (MAC) operation of ThreeBears, in particular the 3120-bit multiplication that is part of it, is very costly on 8-bit microcontrollers and requires special attention. This section deals with optimization techniques for the MAC operation on the AVR platform. As already stated in Sect. 1, we follow two strategies to optimize the MAC, one whose goal is to minimize the RAM footprint, whereas the other aims to maximize performance. The result is a memory-optimized MAC and a speed-optimized MAC, which are described in Subsect. 3.3 and 3.4, respectively.

3.1 The MAC Operation of ThreeBears

ThreeBears defines its field operations $(+, *)$ as

$$a + b := a + b \bmod N \quad \text{and} \quad a * b := a \cdot b \cdot x^{-D/2} \bmod N$$

where $+$ and $\cdot$ are the conventional integer addition and multiplication, respectively. Note that a so-called clarifier $x^{-D/2}$ is multiplied with the factors in the field multiplication, which serves to reduce the distortion of the noise. As shown in [7], the Solinas prime N enables very fast Karatsuba multiplication [11]. We can write the field multiplication in the following way, where $\lambda = x^{D/2}$ and the subscripts H and L are used to denote the higher/lower half of an integer:

[2] See `https://github.com/XKCP/XKCP/tree/master/lib/low/KeccakP-1600/AVR8`.

$$
\begin{aligned}
z = a * b = a \cdot b \cdot \lambda^{-1} &= (a_L + a_H\lambda)(b_L + b_H\lambda) \cdot \lambda^{-1} \\
&= a_L b_L \lambda^{-1} + (a_L b_H + a_H b_L) + a_H b_H \lambda \\
&= a_L b_L (\lambda - 1) + (a_L b_H + a_H b_L) + a_H b_H \lambda \\
&= (a_L b_H + a_H b_L - a_L b_L) + (a_L b_L + a_H b_H)\lambda \\
&= (a_H b_H - (a_L - a_H)(b_L - b_H)) + (a_L b_L + a_H b_H)\lambda \mod N \qquad (1)
\end{aligned}
$$

Compared to a conventional Karatsuba multiplication, which requires three half-size multiplications and six additions/subtractions, the Karatsuba method for multiplication in $\mathbb{Z}/N$ saves one addition or subtraction. Consequently, the MAC operation can be performed as specified by Eq. (2) and Eq. (3):

$$
\begin{aligned}
r &= r + a * b \bmod N \\
&= (r_L + a_H b_H - (a_L - a_H)(b_L - b_H)) + (r_H + a_L b_L + a_H b_H)\lambda \bmod N \qquad (2) \\
&= (r_L + a_H b_L - a_L(b_L - b_H)) + (r_H + (a_L + a_H)b_H + a_L(b_L - b_H))\lambda \bmod N \qquad (3)
\end{aligned}
$$

3.2 Full-Radix Representation for Field Elements

The NIST submission package of ThreeBears consists of different implementations, including a reference implementation, optimized implementations, and additional implementations (e.g. a low-memory implementation). As mentioned in Sect. 1, they all use a *reduced-radix* representation for the 3120-bit integers (e.g. on a 32-bit platform, each limb is 26 bits long, and an element of the field consists of 120 limbs). Since this representation implies that there are six free bits in a 32-bit word, it is possible to store the carry (or borrow) bits that are generated during a field operation instead of immediately propagating them to the next-higher word, which reduces dependencies and enables instruction-level parallelism. Modern super-scalar processors can execute several instructions in parallel and, in this way, improve the running time of ThreeBears.

Our implementations of BabyBear for AVR use a *full-radix* representation for the field elements for a number of reasons. First, small 8-bit microcontrollers have a single-issue pipeline and can not execute instructions in parallel, even when there are no dependencies among instructions. Furthermore, leaving six bits of "headroom" in a 32-bit word increases the number of limbs (in relation to the full-radix case) and, hence, the number of (32×32)-bit multiplications to be carried out. This is a bigger problem on AVR than on high-end processors where multiplications are a lot faster. Finally, the reduced-radix representation requires more space for a field-element (i.e. larger memory footprint) and more load/store instructions. In our full-radix representation, an element of the field consists of 98 words of a length of 32 bits and consumes $98 \times 4 = 392$ bytes in RAM, while the original representation requires $120 \times 4 = 480$ bytes. The full-radix representation with 32-bit words has also arithmetic advantages since (as mentioned in Sect. 1) it allows one to accelerate the MAC operation using the RPS method [13]. Thus, we fix the number representation radix to 2^{32}, despite the fact that we are working on an 8-bit microcontroller.

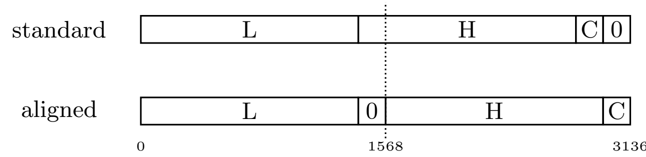

Fig. 1. Standard and aligned form of a field element (AVR uses little-endian)

We define two forms of storage for a full-radix field element: *standard* and *aligned*. Both forms are visually sketched in Fig. 1, where "L" and "H" stands respectively for the lower and higher 1560 bits of a 3120-bit field element. The standard form is basically the straightforward way of storing a multi-precision integer. Since a 3120-bit integer occupies 98 32-bit words, there are 16 unused bits (i.e. two empty bytes) in the most significant word. In our optimized MAC operations, the result is not always strictly in the range $[0, N)$ but can also be in $[0, 2N)$, which means the second most significant byte is either 0 or 1. We call this byte "carry byte" and mark it with a "C" in Fig. 1. Furthermore, we use "0" to indicate the most significant byte because it is 0 all the time.

The reason why we convert a standard integer into aligned form is because it allows us to perform the Karatsuba multiplication more efficiently. From an implementer's viewpoint, the standard form is suboptimal for Karatsuba since it does not place the lower ("L") and upper ("H") 1560 bits into the lower and upper half of the operand space (see Fig. 1). Concretely, the lowest byte of the upper 1560 bits is located at the most significant byte of the lower half in the space, which introduces some extra effort for alignment and addressing. The aligned form splits the lower and upper 1560 bits in such a way that they are ideally located for Karatsuba multiplication.

3.3 Memory-Optimized MAC Operation

The NIST submission package of ThreeBears includes a low-memory implementation for each instance, which aims to minimize stack consumption. These low-memory variants are based on a special memory-efficient MAC operation that uses one level of Karatsuba's technique [11], which follows a modification of Eq. (3), namely Eq. (4) shown below:

$$(r_L + a_H b_L - 2a_L(b_L - b_H)) + (r_H + (a_L + a_H)b_H)\lambda + a_L(b_L - b_H)\lambda^2 \bmod N \quad (4)$$

This MAC implements the multiplications using the product-scanning method and operates on reduced-radix words. Our memory-optimized MAC operation was developed on basis of this original low-memory MAC, but performs all its computations on aligned full-radix words (after some alignment operations).

Algorithm 1 shows our low-RAM one-level Karatsuba MAC, which consists of two major parts: a main MAC loop interleaved with the modular reduction (from line 1 to 26) and a final reduction modulo N (from line 27 to 43). The

Algorithm 1 Memory-optimized MAC operation

Input: Aligned s-word integers $A = (A_{s-1}, \ldots, A_1, A_0)$, $B = (B_{s-1}, \ldots, B_1, B_0)$, and $R = (R_{s-1}, \ldots, R_1, R_0)$, each word contains ω bits; β is a parameter of alignment

Output: Aligned s-word product $R = R + A \cdot B \cdot x^{-D/2} \bmod N = (R_{s-1}, \ldots, R_1, R_0)$

1: $Z_0 \leftarrow 0$, $Z_1 \leftarrow 0$
2: $l \leftarrow s/2$
3: **for** i from 0 to $l-1$ by 1 **do**
4: $\quad Z_2 \leftarrow 0$
5: $\quad k \leftarrow i+1$
6: $\quad$**for** j from 0 to i by 1 **do**
7: $\quad\quad k \leftarrow k-1$
8: $\quad\quad Z_0 \leftarrow Z_0 + A_{j+l} \cdot B_k$
9: $\quad\quad Z_1 \leftarrow Z_1 + (A_j + A_{j+l}) \cdot B_{k+l}$
10: $\quad\quad Z_2 \leftarrow Z_2 + A_j \cdot (B_k - B_{k+l})$
11: $\quad$**end for**
12: $\quad Z_0 \leftarrow Z_0 - 2 \cdot Z_2$
13: $\quad k \leftarrow l$
14: $\quad$**for** j from $i+1$ to $l-1$ by 1 **do**
15: $\quad\quad k \leftarrow k-1$
16: $\quad\quad Z_1 \leftarrow Z_1 + 2^\beta \cdot A_{j+l} \cdot B_k$
17: $\quad\quad Z_2 \leftarrow Z_2 + 2^\beta \cdot (A_j + A_{j+l}) \cdot B_{k+l}$
18: $\quad\quad Z_0 \leftarrow Z_0 + 2^\beta \cdot A_j \cdot (B_k - B_{k+l})$
19: $\quad$**end for**
20: $\quad Z_0 \leftarrow Z_0 + Z_2 + R_i$
21: $\quad Z_1 \leftarrow Z_1 + Z_2 + R_{i+l}$
22: $\quad R_i \leftarrow Z_0 \bmod 2^\omega$
23: $\quad Z_0 \leftarrow Z_0/2^\omega$
24: $\quad R_{i+l} \leftarrow Z_1 \bmod 2^\omega$
25: $\quad Z_1 \leftarrow Z_1/2^\omega$
26: **end for**
27: $Z_0 \leftarrow 2^\beta \cdot Z_0 + R_{l-1}/2^{\omega-\beta}$
28: $Z_1 \leftarrow 2^\beta \cdot Z_1 + R_{s-1}/2^{\omega-\beta}$
29: $R_{l-1} \leftarrow R_{l-1} \bmod 2^{\omega-\beta}$
30: $R_{s-1} \leftarrow R_{s-1} \bmod 2^{\omega-\beta}$
31: $Z_0 \leftarrow Z_0 + Z_1$
32: **for** i from 0 to $l-1$ by 1 **do**
33: $\quad Z_1 \leftarrow Z_1 + R_i$
34: $\quad R_i \leftarrow Z_1 \bmod 2^\omega$
35: $\quad Z_1 \leftarrow Z_1/2^\omega$
36: **end for**
37: $Z_0 \leftarrow 2^\beta \cdot Z_0 + R_{l-1}/2^{\omega-\beta}$
38: $R_{l-1} \leftarrow R_{l-1} \bmod 2^{\omega-\beta}$
39: **for** i from l to $s-1$ by 1 **do**
40: $\quad Z_0 \leftarrow Z_0 + R_i$
41: $\quad R_i \leftarrow Z_0 \bmod 2^\omega$
42: $\quad Z_0 \leftarrow Z_0/2^\omega$
43: **end for**
44: **return** $(R_{s-1}, \ldots, R_1, R_0)$

designer of ThreeBears coined the term "tripleMAC" to refer to the three word-level MACs in the inner loops (line 8 to 10 and 16 to 19). Certainly, this tripleMAC is the most frequent computation carried out by Algorithm 1 and dominates the overall execution time. In order to reach peak performance on AVR, we replace the conventional product-scanning technique by an optimized variant of the hybrid multiplication method [6], namely the so-called Reverse Product-Scanning (RPS) method [13], which processes four bytes (i.e. 32 bits) of the operands per loop-iteration. In addition, we split each of the inner loops containing a tripleMAC up into three separate loops, and each of these three loops computes one word-level MAC. Due to the relatively small register space of AVR, it is not possible to keep all three accumulators (i.e. Z_0, Z_1, and Z_2) in registers,which means executing three word-level MACs in the same inner loop would require a large number of `LD` and `ST` instructions to load and store the accumulators in each iteration. Thanks to our modification, an accumulator has to be loaded/stored only before/after the whole inner loop.

The three inputs and the output of Algorithm 1 are aligned integers, where s is 98 and ω is 32. The parameter β specifies the shift-distance (in bits) when converting an ordinary integer to aligned form ($\beta = 8$ in our case). Each of the three accumulators Z_0, Z_1, and Z_2 in Algorithm 1 is 80 bits long and occupies

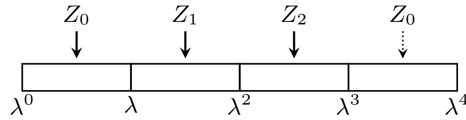

Fig. 2. Three accumulators for coefficients of $\lambda^0 = 1$, λ, and λ^2 of a product R

10 registers. Figure 2 illustrates the relation between the accumulators and the coefficients of λ^0, λ, and λ^2 in an aligned output R. Referring to Eq. (4), we suppose each coefficient can be 3120 bits long, but Z_0, Z_1, and Z_2 accumulate only the lower 1560 bits of the coefficients of λ^0, λ, and λ^2, respectively, in the first tripleMAC (line 8 to 10). After the first inner loop, double of Z_2 must be subtracted from Z_0 (line 12), which corresponds to the operation of the form $a_H b_L - 2a_L(b_L - b_H)$ in Eq. (4). The second inner loop (beginning at line 14) computes the higher half of each coefficient, but this time the word-products are added to different accumulators compared to the first tripleMAC (e.g. the 64-bit word-products of the form $A_{j+l} \cdot B_k$ are added to Z_1 instead of Z_0). In the second tripleMAC, the factor 2^β needs to be considered in order to ensure proper alignment. The third word-level MAC (at line 18) can be regarded as computing (the lower half of) the coefficient of λ^3. Normally, we should use an additional accumulator Z_3 for this third MAC, but it is more efficient to re-use Z_0. This is possible since, after the second inner loop, we would normally have to compute $Z_1 \leftarrow Z_1 - 2 \cdot Z_3$, a similar operation as at line 12. But because

$$\lambda^3 = \lambda^2 \cdot \lambda = (\lambda + 1) \cdot \lambda = \lambda^2 + \lambda = (\lambda + 1) + \lambda = 2\lambda + 1 \bmod N,$$

we also have to compute $Z_0 \leftarrow Z_0 + Z_3$ and $Z_1 \leftarrow Z_1 + 2 \cdot Z_3$. Combining these two computations with $Z_1 \leftarrow Z_1 - 2 \cdot Z_3$ implies that Z_1 can simply keep its present value and only Z_0 accumulates the value of Z_3. Thus, Algorithm 1 does not compute $Z_1 \leftarrow Z_1 - 2 \cdot Z_3$, but instead directly accumulates the sum of the word-products $A_j \cdot (B_k - B_{k+l})$ into Z_0 (which also saves a few load and store instructions). This "shortcut" is indicated in Fig. 2 with a dashed arrow from Z_0 to the coefficient of λ^3. Lines 20 to 25 add the lower 32-bit words of Z_0 and Z_1 to the corresponding words of the result R and right-shift Z_0 and Z_1. The part from line 27 to 30 brings the output of the MAC into a properly aligned form. Thereafter (line 31 to 43), a modulo-N reduction (based on the relation $\lambda^2 \equiv \lambda + 1 \bmod N$) along with a conversion to 32-bit words is performed. The output of Algorithm 1 is an aligned integer in the range of $[0, 2N)$.

We implemented Algorithm 1 completely in AVR Assembly language. Even though each accumulator Z_i consists of 80 bits (ten bytes), we only load and store nine bytes of Z_i in each inner loop. A simple analysis shows that that the accumulator values in the first inner loop can never exceed 2^{72}, which allows us to only load and store the nine least-significant bytes. In the second tripleMAC loop, each word-product is multiplied by 2^β (i.e. shifted left by eight bits) and so it is not necessary to load/store the least-significant accumulator byte.

Algorithm 2 Speed-optimized MAC operation

Input: Aligned field elements $A = (A_H, A_L)$, $B = (B_H, B_L)$ and $R = (R_H, R_L)$
Output: Aligned product $R = R + A \cdot B \cdot x^{-D/2} \bmod N = (R_H, R_L)$

1: $(Z_H, Z_L) \leftarrow (0, 0)$, $(T_H, T_L) \leftarrow (0, 0)$
2: $T_L \leftarrow |A_L - A_H|$
3: **if** $A_L - A_H < 0$ **then** $s_a \leftarrow 1$; **else** $s_a \leftarrow 0$
4: $T_H \leftarrow |B_L - B_H|$
5: **if** $B_L - B_H < 0$ **then** $s_b \leftarrow 1$; **else** $s_b \leftarrow 0$
6: $(Z_H, Z_L) \leftarrow T_L \cdot T_H \cdot (-1)^{1-(s_a \oplus s_b)}$
7: $(R_H, R_L) \leftarrow (R_H, R_L) + (Z_H, Z_L)$
8: $T_L \leftarrow A_H, T_H \leftarrow B_H$
9: $(Z_H, Z_L) \leftarrow T_L \cdot T_H$
10: $R_H \leftarrow R_H + Z_H$
11: $T_L \leftarrow Z_H + Z_L$
12: $R_L \leftarrow R_L + T_L$
13: $R_H \leftarrow R_H + T_L$
14: $T_L \leftarrow A_L, T_H \leftarrow B_L$
15: $(Z_H, Z_L) \leftarrow T_L \cdot T_H$
16: $R_H \leftarrow R_H + Z_L$
17: $R_L \leftarrow R_L + Z_H$
18: $R_H \leftarrow R_H + Z_H$
19: $(R_H, R_L) \leftarrow (R_H, R_L) \bmod N$
20: **return** (R_H, R_L)

3.4 Speed-Optimized MAC Operation

The MAC operations of the implementations in the NIST submission package of ThreeBears are not suitable to reach high speed on AVR. Therefore, we developed our speed-optimized MAC operation from scratch and implemented it according to a variant of Eq. (2), namely Eq. (5) specified below. We divide the three full-size 3120-bit products (e.g. $a_L b_L$) of Eq. (2) into two halves, and use l for representing $a_L b_L$, m for $-(a_L - a_H)(b_L - b_H)$ and h for $a_H b_H$:

$$\begin{aligned} r &= (r_L + h + m) + (r_H + l + h)\lambda \mod N \\ &= (r_L + (h_L + h_H\lambda) + (m_L + m_H\lambda)) + (r_H + (l_L + l_H\lambda) + (h_L + h_H\lambda))\lambda \\ &= (r_L + h_L + m_L) + (r_H + l_L + h_L + m_H + h_H)\lambda + (l_H + h_H)\lambda^2 \\ &= (r_L + m_L + \underline{h_L + h_H + l_H}) + (r_H + m_H + h_H + l_L + \underline{h_L + h_H + l_H})\lambda \quad (5) \end{aligned}$$

The underlined parts in Eq. (5) are common parts of the coefficients of λ^0 and λ. Algorithm 2 specifies our speed-optimized MAC, which operates on half-size (i.e. 1560-bit) parts of the operands A, B and R. We omitted the details of the final step (line 19) in Algorithm 2, i.e. the modulo-N reduction, because it is very similar to lines 27 to 43 in Algorithm 1. Compared with Algorithm 1, the speed-efficient MAC operation is designed in a more straightforward way since it computes each entire half-size multiplication separately to obtain a full-size intermediate product (line 6, 9, and 15). However, it consumes more memory to store the intermediate products (e.g. Z_H, Z_L and T_H, T_L).

We still take advantage of RPS technique to speed up the inner-loop operation, but combine it Karatsuba's method. Our experiments with different levels of Karatsuba multiplication showed that the 2-level Karatsuba approach with the RPS technique underneath (i.e. 2-level KRPS) yields the best performance for a multiplication of 1560-bit operands. Consequently, we execute three levels of Karatsuba (i.e. 3-level KRPS) altogether for the MAC operation. Each level uses the so-called subtractive Karatsuba algorithm described in [9] to achieve

Table 1. Execution time (in clock cycles) of our AVR implementations

Implementation	Security	MAC	KeyGen	Encaps	Decaps
ME-BBear	CCA-secure	1,033,728	8,746,418	12,289,744	18,578,335
ME-BBear-Eph	CPA-secure	1,033,728	8,746,418	12,435,165	3,444,154
HS-BBear	CCA-secure	604,703	6,123,527	7,901,873	12,476,447
HS-BBear-Eph	CPA-secure	604,703	6,123,527	8,047,835	2,586,202

Table 2. RAM usage and code size (both in bytes) of our AVR implementations

Implementation	MAC		KeyGen		Encaps		Decaps		Total	
	RAM	Size	RAM	Size	RAM	Size	RAM	Size	RAM	Size
ME-BBear	82	2,760	1,715	6,432	1,735	7,554	2,368	10,110	2,368	12,264
ME-BBear-Eph	82	2,760	1,715	6,432	1,735	7,640	1,731	8,270	1,735	10,998
HS-BBear	934	3,332	2,733	7,000	2,752	8,140	4,559	10,684	4,559	11,568
HS-BBear-Eph	934	3,332	2,733	7,000	2,752	8,226	2,356	8,846	2,752	10,296

fast and constant execution time. All half-size multiplications performed at the second and third level use space that was initially occupied by input operands to store intermediate values, i.e. we do not allocate additional RAM inside the half-size multiplications. This is also the reason for the two operations at line 8 and 14, where we move the operands to T_H and T_L before the multiplication so that we do not modify the inputs A and B.

4 Performance Evaluation and Comparison

Except of the MAC operations, all components of our ME and HS software are taken from the low-memory and speed-optimized implementation contained in the NIST package of ThreeBears (with minor optimizations). Our software consists of a mix of C and AVR assembly language, i.e. the performance-critical MAC operation and Keccak permutation are written in AVR assembly, and all other functions in C. Atmel Studio v7.0, our development environment, comes with the 8-bit AVR GNU toolchain including avr-gcc version 5.4.0. We used the cycle-accurate instruction set simulator of Atmel Studio to precisely determine the execution times. The source codes were compiled with optimization option `-O2` using the ATmega1284 microcontroller as target device.

Table 1 shows the execution time of a MAC operation, key generation, encapsulation, and decapsulation of our software. A speed-optimized MAC takes only 605 k clock cycles, while the memory-optimized version requires 70% more cycles. The speed difference between these two types of MAC directly impacts the overall running time of ME-BBear(-Eph) versus HS-BBear(-Eph), because there are several MACs in KeyGen, Encaps and Decaps. Taking HS-BBear as example, KeyGen, Encaps, and Decaps needs 6.12 M, 7.90 M, and 12.48 M clock cycles, respectively. Their ME counterparts are roughly 1.5 times slower.

Table 2 specifies both the memory footprint and code size of the four basic operations (MAC, KeyGen, Encaps, and Decaps). The speed-optimized MAC

Table 3. Comparison of our implementation with other key-establishment schemes (all of which target 128-bit security) on the 8-bit AVR platform (the execution times of Encaps and Decaps are in clock cycles; RAM and code size are in bytes).

Implementation	Algorithm	Encaps	Decaps	RAM	Size
This work (ME-CCA)	ThreeBears	12,289,744	18,578,335	2,368	12,264
This work (ME-CPA)	ThreeBears	12,435,165	3,444,154	1,735	10,998
This work (HS-CCA)	ThreeBears	7,901,873	12,476,447	4,559	11,568
This work (HS-CPA)	ThreeBears	8,047,835	2,586,202	2,752	10,296
Cheng et al [2]	NTRU Prime	8,160,665	15,602,748	n/a	11,478
Düll et al [4] (ME)	Curve25519	14,146,844	14,146,844	510	9,912
Düll et al [4] (HS)	Curve25519	13,900,397	13,900,397	494	17,710

consumes 934 bytes of memory, while the memory-optimized MAC requires as little as 82 bytes, which is just 9% of the former. Due to the memory-optimized MAC operation and full-radix representation of field elements, ME-BBear has a RAM footprint of 1.7 kB for each KeyGen and Encaps, while Decaps is more memory-demanding and needs 2.4 kB RAM. However, ME-BBear-Eph requires only 1.7 kB of RAM altogether. On the other hand, the HS implementations need over 1.5 times more RAM than their ME counterparts. In terms of code size, each of the four implementations requires roughly 11 kB.

Table 3 compares implementations of both pre and post-quantum schemes (targeting 128-bit security) on 8-bit AVR microcontrollers. Compared to the CCA-secure version of the second-round NIST candidate NTRU Prime [2], HS-BBear is slightly faster for both Encaps and Decpas. On the other hand, when compared with the optimized implementation of Curve25519 in [4], the Encaps operation of each BabyBear variant in Table 3 is faster than a variable-base scalar multiplication, while the Decaps of ME-BBear is slower, but that of the HS variant still a bit faster. Notably, the Decaps operation of our CPA-secure implementations is respectively 4.0 times (ME) and 5.4 times (HS) faster than Curve25519.

One of the most significant advantages of the ThreeBears cryptosystem is its relatively low RAM consumption, which is important for deployment on constrained devices. Table 4 compares the RAM footprint of implementations of ThreeBears and a few other NIST candidates on microcontrollers. Due to the very small number of state-of-the-art implementations of NIST candidates for the 8-bit AVR platform, we include in Table 4 also some recent results from the `pqm4` library, which targets 32-bit ARM Cortex-M4. In addition, we list the results of the original low-RAM implementations of BabyBear (for both the CCA and CPA variant) from the NIST package. Our memory-optimized BabyBear is the most RAM-efficient implementation among all CCA-secure PQC schemes and needs 5% less RAM than the second most RAM-efficient scheme Kyber. Furthermore, ME-BBear-Eph requires the least amount of RAM of all (CPA-secure) second-round NIST PQC candidates, and improves the original low-memory implementation of the designer by roughly 26.2%.

Table 4. Comparison of RAM consumption (in bytes) of NIST PQC implementations (all of which target NIST security category 1 or 2) on 8-bit AVR and on 32-bit ARM Cortex-M4 microcontrollers.

Implementation	Algorithm	Platform	KeyGen	Encaps	Decaps
CCA-secure schemes					
This work (ME)	ThreeBears	AVR	1,715	1,735	2,368
Hamburg [8]	ThreeBears	Cortex-M4	2,288	2,352	3,024
pqm4 [10]	ThreeBears	Cortex-M4	3,076	2,964	5,092
pqm4 [10]	NewHope	Cortex-M4	3,876	5,044	5,044
pqm4 [10]	Round5	Cortex-M4	4,148	4,596	5,220
pqm4 [10]	Kyber	Cortex-M4	2,388	2,476	2,492
pqm4 [10]	NTRU	Cortex-M4	11,848	6,864	5,144
pqm4 [10]	Saber	Cortex-M4	9,652	11,388	12,132
CPA-secure schemes					
This work (ME)	ThreeBears	AVR	1,715	1,735	1,731
Hamburg [8]	ThreeBears	Cortex-M4	2,288	2,352	2,080
pqm4 [10]	ThreeBears	Cortex-M4	3,076	2,980	2,420
pqm4 [10]	NewHope	Cortex-M4	3,836	4,940	3,200
pqm4 [10]	Round5	Cortex-M4	4,052	4,500	2,308

5 Conclusions

We presented the first highly-optimized Assembler implementation of ThreeBears for the 8-bit AVR architecture. Our simulation results show that, even with a fixed parameter set like BabyBear, many trade-offs between execution time and RAM consumption are possible. The memory-optimized CPA-secure version of BabyBear requires only slightly more than 1.7 kB RAM, which sets a new record for memory efficiency among all known software implementations of second-round candidates. Due to this low memory footprint, BabyBear fits easily into the SRAM of 8-bit AVR ATmega microcontrollers and will even run on severely constrained devices like an ATmega128L with 4 kB SRAM. While a RAM footprint of 1.7 kB is still clearly above the 500 B of Curve25519, the execution times are in favor of BabyBear since a CPA-secure decapsulation is four times faster than a scalar multiplication. ThreeBears is also very well suited to be part of a hybrid pre/post-quantum key agreement protocol since the multiple-precision integer arithmetic can (potentially) be shared with the low-level field arithmetic of Curve25519, thereby reducing the overall code size when implemented in software or the total silicon area in the case of hardware implementation. For all these reasons, ThreeBears is an excellent candidate for a post-quantum cryptosystem to secure the IoT.

Acknowledgements. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 779391 (FutureTPM).

References

1. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Keccak. In T. Johansson and P. Q. Nguyen, editors, *Advances in Cryptology — EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 313–314. Springer Verlag, 2013.
2. H. Cheng, D. Dinu, J. Großschädl, P. B. Rønne, and P. Y. A. Ryan. A lightweight implementation of NTRU Prime for the post-quantum internet of things. In M. Laurent and T. Giannetsos, editors, *Information Security Theory and Practice — WISTP 2019*, volume 12024 of *Lecture Notes in Computer Science*, pages 103–119. Springer Verlag, 2020.
3. Crossbow Technology, Inc. MICAz Wireless Measurement System. Data sheet, available for download at `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf`, Jan. 2006.
4. M. Düll, B. Haase, G. Hinterwälder, M. Hutter, C. Paar, A. H. Sánchez, and P. Schwabe. High-speed Curve25519 on 8-bit, 16-bit and 32-bit microcontrollers. *Designs, Codes and Cryptography*, 77(2–3):493–514, Dec. 2015.
5. C. Gu. Integer version of Ring-LWE and its applications. Cryptology ePrint Archive, Report 2017/641, 2017. `http://eprint.iacr.org/2017/641`.
6. N. Gura, A. Patel, A. S. Wander, H. Eberle, and S. Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer Verlag, 2004.
7. M. Hamburg. Ed448-Goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625, 2015. `http://eprint.iacr.org/2015/625`.
8. M. Hamburg. ThreeBears: Round 2 specification. Available for download at `http://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions`, 2019.
9. M. Hutter and P. Schwabe. Multiprecision multiplication on AVR revisited. *Journal of Cryptographic Engineering*, 5(3):201–214, Sept. 2015.
10. M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen. `pqm4`: Testing and benchmarking NIST PQC on ARM Cortex-M4. Cryptology ePrint Archive, Report 2019/844, 2019. `http://eprint.iacr.org/2019/844`.
11. A. A. Karatsuba and Y. P. Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 145(2):293–294, ?? 1962.
12. J. M. Kelsey, S.-J. H. Chang, and R. A. Perlner. SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash. NIST Special Publication 800-185, available for download at `http://doi.org/10.6028/NIST.SP.800-185`, 2016.
13. Z. Liu, H. Seo, J. Großschädl, and H. Kim. Reverse product-scanning multiplication and squaring on 8-bit AVR processors. In L. C.-K. Hui, S. Qing, E. Shi, and S.-M. Yiu, editors, *Information and Communications Security — ICICS 2014*, volume 8958 of *Lecture Notes in Computer Science*, pages 158–175. Springer Verlag, 2015.
14. National Institute of Standards and Technology (NIST). Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. Available for download at `http://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf`, 2016.
15. National Institute of Standards and Technology (NIST). NIST reveals 26 algorithms advancing to the post-quantum crypto 'semifinals'. Press release, available online at `http://www.nist.gov/news-events/news/2019/01/nist-reveals-26-algorithms-advancing-post-quantum-crypto-semifinals`, 2019.