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 just about 12 million cycles, which is less than the execution time of a 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 [15]. 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 [16]. 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" [16].

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 [11]. 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 [5]. 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 that 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 [4] is equipped with an 8-bit microcontroller (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 [9], 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` (accessed on 2020–06–30).

so-called Integer Module Learning with Errors (I-MLWE) problem [6]. 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 [12] 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 [7] or one of its variants like the Reverse Product Scanning (RPS) method [14]. 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.

We used the optimized C code contained in the ThreeBears submission package [9] as starting point of our research. This optimized C implementation adopts a reduced-radix representation for the ring elements, which means the 3120-bit integers are represented as arrays of 120 limbs, whereby each limb has a length of 26 bits. Our AVR implementations, on the other hand, are based on 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 operation consists of one level of Karatsuba and uses the RPS approach [14] underneath, which is (relatively) cheap in terms of allocated stack memory. On the other hand, the

speed-optimized MAC combines 3-level Karatsuba multiplication with the RPS method to accelerate the tripleMAC operation of the optimized C source code from [9]. 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, one of the most resource-constrained devices, are widely used in current IoT markets (e.g. smart cards, wireless sensor nodes). The AVR architecture is based on the RISC philosophy and a modified Harvard memory model, and comes with 32 general-purpose working registers (namely `R0` to `R31`) of 8-bit width that are directly connected to the Arithmetic Logic Unit (ALU). The latest revision of AVR instruction set supports 129 instructions altogether, where each instruction has fixed latency. As the example, some instructions that are frequently used in our software are addition/subtraction (`ADD`/`ADC`/`SUB`/`SBC`) which take one clock cycle. In comparison, both the multiplication (`MUL`) and load/store (`LD`/`ST`) instruction are more "expensive" and need two clock cycles. The specific AVR microcontroller on which we simulated the performance of our software is the ATmega1284 that 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 providing respectively CPA and CCA security. Taking BabyBear as an example, the CPA-secure instance is named BabyBearEphem (with the meaning of ephemeral BabyBear) while the CCA-secure one is simply called BabyBear. We here only give a brief overview of the CCA-secure instance of ThreeBears. In contrast with a scheme of CCA-security, the CPA-secure one, roughly speaking, does not repeat and test the key generation and encapsulation during the decapsulation procedure (see [9] for details).

Notation and Parameters. ThreeBears is performed in a field $\mathbb{Z}/N$, where the prime modulus $N = 2^{3120} - 2^{1560} - 1$ is a so-called "golden-ratio" Solinas trinomial prime [8]. N is usually written in a form of $N = \phi(x) = x^D - x^{D/2} - 1$. The field addition and multiplication operations ($+$, $*$) will be explained in Sect. 3.1. Further, a parameter d decides the module dimension, which 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 $r = \text{Hash}(sk)$.
4. Generate a $d \times d$ matrix $\boldsymbol{M}$, where each element $M_{i,j} \in \mathbb{Z}/N$ is sampled from 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 $ss = \text{Hash}(r, seed)$.
7. Output ss as *shared secret* and $(f, \boldsymbol{y})$ as *ciphertext*.

Decapsulation. The decapsulation gets a private key sk and a 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 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 equal then output ss' as *shared secret*; if not then output $\textsc{Hash}(sk, f, \boldsymbol{y})$ as *shared secret*.

In above-described algorithms, there exist some so-called "auxiliary" functions such as samplers (noise sampler and uniform sampler), hash functions and error correction. Both the samplers and hash functions take advantage of the `cSHAKE256` [13], which relies on the Keccak permutation [1] at the lowest layer. Besides, the designer uses a Melas BCH code for forward error correction (FEC) in ThreeBears. His Melas FEC implementation has small code and memory requirements, runs in constant time, and its runtime is almost negligible.

We measured various implementations of the NIST package of ThreeBears on the AVR processor. Like most of the other post-quantum cryptographic schemes, the arithmetic computations dominate the performance (both RAM footprint and execution time) of ThreeBears. Hence, our work principally focuses on the optimization of the most costly MAC operation thereof ($r = r + a * b \bmod N$). Concerning the auxiliary functions, thanks to an open-source highly-optimized AVR Assembler[2] of Keccak permutation, they gained significant speed improvements. Other details regarding auxiliary functions are out of the scope of this work, and we advise readers to refer the specification of ThreeBears [9].

3 Optimizations for MAC Operation

The multiply-accumulate (MAC) operation in ThreeBears, $r = r + a * b \bmod N$, particularly the field multiplication thereof, is very costly on AVR devices and deserves special care. This section deals with the optimization approaches of MAC operation on the AVR platform. As mentioned in Sect. 1, we applied two strategies for MAC optimizations, i.e. memory-optimized MAC and speed-optimized MAC, which are illustrated in Sect. 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 operations ($+$ and $\cdot$) are the conventional integer addition and multiplication. Notably, a clarifier $x^{-D/2}$ is multiplied with factors during the field multiplication, which is used for reducing distortion of the noise. As pointed out in [8], this golden-ratio Solinas prime N contributes to a fast Karatsuba multiplication [12]. Considering the multiplication in ThreeBears (we let $\lambda = x^{D/2}$, and e_L/e_H stands for the lower/higher half of element e hereafter), it is:

[2] `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 (six additions or subtractions and three multiplications), the Karatsuba multiplication in $\mathbb{Z}/N$ saves one addition/subtraction. Consequently, the MAC operation can be performed as Eq. (2) and transformed as Eq. (3):

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

3.2 Full-Radix Representation for Field Elements

In the NIST PQC submission package of ThreeBears, the designer offered multiple implementations, such as reference implementations, optimized implementations and additional implementations (e.g. low-memory implementations). All of them take advantage of a so-called *reduced-radix* representation for the 3120-bit field elements (due to modulus $N = 2^{3120} - 2^{1560} - 1$). For example, on the 32-bit platform, each "limb" is 26 bits long, and a 3120-bit integer consists of 120 limbs. Because there are six unoccupied bits in each 32-bit word, it is possible to store the carry (or borrow) bits during arithmetic computations and not urgent to propagate carry (or borrow) bits instantly, whereby many computations eliminated the dependency with others. High-end processors could beneficially carry out a few computations in parallel and so that saves running time.

However in our work, we come up with a *full-radix* representation for field elements on AVR. First of all, the AVR microprocessor carries out instructions in sequential order. It cannot execute instructions in parallel even though there is no dependency among instructions. More importantly, the machine words of AVR have only 8 bits, and keeping a few bits on top for carries massively increases the number of limbs required to represent big integers, which leads to a high RAM consumption for each element and many more instructions. As introduced before, we take advantage of the RPS method to speed up our MAC operations, and the RPS method operates on the unit of four bytes. Therefore, we set the word-size to 32 for our full-radix representation despite we are working on the 8-bit processor. Each 3120-bit field element consists of 98 32-bit words. Our full-radix method needs only $98 \times 4 = 392$ bytes to represent a field element while the original one takes $120 \times 4 = 480$ bytes. Not only for the multiplication or MAC, but also the whole ThreeBears will get benefits to decrease the consumption of stack memory.

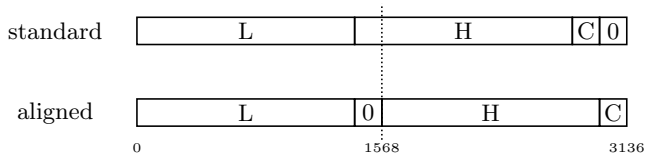

Fig. 1. Standard and aligned form of a field element (AVR uses little-endian)

Furthermore, we define two storage forms for full-radix field elements: *standard* and *aligned*. We illustrate both forms in Fig. 1, where "L" and "H" respectively stands for the lower and higher 1560-bit of the 3120-bit integer. The standard form is, briefly, an ordinary way of storing the multi-precision integer. Since we use 98 32-bit words to store a field element, there are still 16 remaining bits (i.e. two bytes) in the most significant word. In our optimized MAC operations, the output integer is not always strictly in the range $[0, N)$ but in $[0, 2N)$, whereby the second most significant byte is either 0x00 or 0x01. We call this byte as the carry-byte and show it as "C" in Fig. 1. Furthermore, we use "0" to represent the most significant byte because it is 0x00 all the time.

The reason why we convert a standard integer to an aligned form is to perform the Karatsuba multiplication more efficiently. From an implementation viewpoint, the standard form does not split the lower and upper 1560-bit (i.e. "L" and "H") into the lower and the upper half in space (see Fig. 1). Concretely, the lowest byte of the upper 1560-bit ("H") locates at the most significant byte of the lower half in space. This standard form is tricky for Karatsuba multiplication in practice, which needs to pay the extra expense for alignment and addressing. The aligned form splits the lower and upper 1560-bit (i.e. "L" and "H") in space and decreases the above expense.

3.3 Memory-Optimized MAC Operation

The NIST package of ThreeBears includes a series of so-called low-memory implementations, which are designed to minimise the stack usage of each instance. The low-memory variant is equipped with a specialised RAM-efficient MAC operation applied with one-level Karatsuba method, which follows a variant equation of Eq. (3), i.e. Eq. (4) shown below:

$$r := (r_L + a_H b_L - 2a_L(b_L - b_H)) + (r_H + (a_L + a_H)b_H)\lambda + a_L(b_L - b_H)\lambda^2 \bmod N \tag{4}$$

In addition, this MAC makes use of a product-scanning multiplication and operates on the reduced-radix words. Our own memory-optimized MAC is developed on the basis of this original low-memory MAC but performs computations towards the aligned full-radix words (with some necessary alignment operations).

Algorithm 1 explains our one-level Karatsuba memory-efficient MAC, which mainly has two steps: a main MAC loop interleaved with the modular reduction (from lines 1 to 25) and a final reduction modulo N (from lines 26 to 42). 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: $\quad Z_2 \leftarrow 0$, $k \leftarrow i+1$
5: $\quad$ **for** j from 0 to i by 1 **do**
6: $\quad\quad k \leftarrow k-1$
7: $\quad\quad Z_0 \leftarrow Z_0 + A_{j+l} \cdot B_k$
8: $\quad\quad Z_1 \leftarrow Z_1 + (A_j + A_{j+l}) \cdot B_{k+l}$
9: $\quad\quad Z_2 \leftarrow Z_2 + A_j \cdot (B_k - B_{k+l})$
10: $\quad$ **end for**
11: $\quad Z_0 \leftarrow Z_0 - 2 \cdot Z_2$
12: $\quad k \leftarrow l$
13: $\quad$ **for** j from $i+1$ to $l-1$ by 1 **do**
14: $\quad\quad k \leftarrow k-1$
15: $\quad\quad Z_1 \leftarrow Z_1 + 2^\beta \cdot A_{j+l} \cdot B_k$
16: $\quad\quad Z_2 \leftarrow Z_2 + 2^\beta \cdot (A_j + A_{j+l}) \cdot B_{k+l}$
17: $\quad\quad Z_0 \leftarrow Z_0 + 2^\beta \cdot A_j \cdot (B_k - B_{k+l})$
18: $\quad$ **end for**
19: $\quad Z_0 \leftarrow Z_0 + Z_2 + R_i$
20: $\quad Z_1 \leftarrow Z_1 + Z_2 + R_{i+l}$
21: $\quad R_i \leftarrow Z_0 \bmod 2^\omega$
22: $\quad Z_0 \leftarrow Z_0 / 2^\omega$
23: $\quad R_{i+l} \leftarrow Z_1 \bmod 2^\omega$
24: $\quad Z_1 \leftarrow Z_1 / 2^\omega$
25: **end for**
26: $Z_0 \leftarrow 2^\beta \cdot Z_0 + R_{l-1} / 2^{\omega-\beta}$
27: $Z_1 \leftarrow 2^\beta \cdot Z_1 + R_{s-1} / 2^{\omega-\beta}$
28: $R_{l-1} \leftarrow R_{l-1} \bmod 2^{\omega-\beta}$
29: $R_{s-1} \leftarrow R_{s-1} \bmod 2^{\omega-\beta}$
30: $Z_0 \leftarrow Z_0 + Z_1$
31: **for** i from 0 to $l-1$ by 1 **do**
32: $\quad Z_1 \leftarrow Z_1 + R_i$
33: $\quad R_i \leftarrow Z_1 \bmod 2^\omega$
34: $\quad Z_1 \leftarrow Z_1 / 2^\omega$
35: **end for**
36: $Z_0 \leftarrow 2^\beta \cdot Z_0 + R_{l-1} / 2^{\omega-\beta}$
37: $R_{l-1} \leftarrow R_{l-1} \bmod 2^{\omega-\beta}$
38: **for** i from l to $s-1$ by 1 **do**
39: $\quad Z_0 \leftarrow Z_0 + R_i$
40: $\quad R_i \leftarrow Z_0 \bmod 2^\omega$
41: $\quad Z_0 \leftarrow Z_0 / 2^\omega$
42: **end for**
43: **return** $(R_{s-1}, \ldots, R_1, R_0)$

designer gives a name of "tripleMAC" for those three "word-level" MACs in the inner loops (lines 7 to 9 and lines 15 to 17). Certainly, this tripleMAC is the most frequent computation in this MAC. We take advantage of the RPS approach [14] to accelerate it on AVR. In essence, this strategy changes this MAC from using an ordinary product-scanning method to a hybrid method [7]. Notably, in our practical implementation, we divided each inner loop of tripleMAC into three independent loops, and each of these three loops only deals with one word-level MAC. Because of the limited register-space of AVR, if three MACs are in the same inner loop, it takes numerous `LD` and `ST` instructions to load and store three accumulators (i.e. Z_0, Z_1 and Z_2) in each iteration. Through our modification, it only needs to load/store an accumulator before/after the whole inner loop.

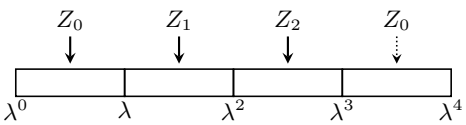

Fig. 2. Three accumulators for coefficients of λ^0, λ and λ^2 of a product R

The input and the output of Algorithm 1 are aligned integers, where s is 98 and ω is 32. β is a parameter of alignment equaling to 8, and it indicates how many bits we moved when converting an integer from standard to aligned. At the beginning of the Algorithm 1, Z_0, Z_1 and Z_2 are three accumulators, and each of them is 80 bits long. Fig. 2 illustrates the relations between 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-bit long. But Z_0, Z_1 and Z_2 only accumulate the lower 1560-bit of coefficients of λ^0, λ and λ^2, respectively. The first tripleMAC (lines 7 to 9) directly reflects this setting. After the first inner loop, Z_0 must subtract the double values of Z_2 (line 11), corresponding to "$a_H b_L - 2a_L(b_L - b_H)$" in Eq. (4). The second inner loop computes the higher half of each coefficient, where this time the tripleMAC (lines 15 to 17) is corresponding to different accumulators. And the second tripleMAC needs to multiply with 2^β because of the alignment. However, the third word-level MAC (at line 17) needs more care, which can be regarded as computing (the lower half of) the coefficient of λ^3. In principle, we should use another accumulator Z_3 to store this output. And after the second inner loop, we are supposed to perform $Z_1 \leftarrow Z_1 - 2 \cdot Z_3$, a similar operation as what we did at line 11. Furthermore, due to

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

we could thereafter perform computations of $Z_0 \leftarrow Z_0 + Z_3$ and $Z_1 \leftarrow Z_1 + 2 \cdot Z_3$. Combined above two computations, Z_1 still keeps its original value while only Z_0 accumulated the value of Z_3. Algorithm 1 could thus save the computations of $Z_1 \leftarrow Z_1 - 2 \cdot Z_3$ and directly accumulate the value of Z_3 to Z_0 (this saves the RAM consumption of Z_3). We also mentioned it in Fig. 2 with a dashed arrow from Z_0 to the coefficient of λ^3. Lines 19 to 24 store one word for each coefficient of λ^0 and λ, and meanwhile update accumulators Z_0 and Z_1. The part from line 26 to 29 makes the output of MAC a strict aligned form. The rest of Algorithm 1, i.e. lines from 30 to 42, executes a modular-N reduction according to $\lambda^2 = 1 + \lambda \bmod N$ and with carry propagation. Finally, the output of Algorithm 1 is an aligned integer in the range of $[0, 2N)$.

We implemented Algorithm 1 in AVR Assembly language. Moreover, although each accumulator Z_i is made up of 80 bits (ten bytes), we only load and store nine bytes of each Z_i for each inner loop. We calculate and confirm that the maximal intermediate value of the first tripleMAC loop is not greater than 2^{72}, which makes it possible to only load and store nine least significant bytes of the accumulator. As for the second tripleMAC loop, each operation needs to multiply with 2^β (i.e. 2^8), which makes sense that no need to load the least significant byte of each accumulator.

3.4 Speed-Optimized MAC Operation

The MAC operations of all the implementations in ThreeBears NIST package are not friendly for AVR to reach high speed. We thus developed our speed-optimized MAC operation from scratch and designed it according to a variation

of Eq. (2) i.e. Eq. (5) shown below. We further divide three full-size products (e.g. $a_L b_L$) of the Eq. (2) into two halves, and use l for indicating $a_L b_L$, 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 the common parts for both coefficients of λ^0 and λ.

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 $A_L - A_H < 0$, $s_a \leftarrow 1$; otherwise $s_a \leftarrow 0$
4: $T_H \leftarrow |B_L - B_H|$
5: if $B_L - B_H < 0$, $s_b \leftarrow 1$; otherwise $s_b \leftarrow 0$
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)

Algorithm 2 describes our speed-optimized MAC, which operates on each half-size (1560-bit) of the elements. We omitted the details of the final step (line 19) in Algorithm 2, i.e. a modulo-N reduction, which is very similar to the lines from 26 to 42 in Algorithm 1. Compared with Algorithm 1, the speed-efficient MAC is designed in a more straightforward way, which separately computes each entire half-size multiplication and obtains a full-size intermediate product (line 6, 9 and 15). But it needs more dynamic 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 accelerate the inner-loop operations. And from our experiments with trying different levels of Karatsuba multiplication, we found that 2-level Karatsuba multiplication combined with RPS technique (i.e. 2-level KRPS) yields the peak performance for a 1560-bit multi-precision multiplication. As a result, we take advantage of a totally 3-level KRPS for the entire MAC operation. For each level Karatsuba multiplication, we employ the subtractive Karatsuba algorithm [10] to avoid the carry bits. In each 2-level KRPS half-size multiplication, we take a trick that utilizes two input variables to store the intermediate values. Consequently, we do not need to

allocate extra memory inside the half-size multiplications. This is also the reason of both 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 change the inputs A and B.

4 Performance Evaluation and Comparison

Except for MAC operations, other components of our ME and HS implementation are developed respectively according to the low-memory implementation and optimized implementation in the NIST package of ThreeBears and with some minor optimizations. Our software is written in a mix of C and AVR assembly language. In detail, only the performance-critical MAC operation and Keccak permutation are developed in AVR Assembler while all of other functions are written in C. Atmel Studio v7.0, our development environment, offers a 8-bit AVR GNU toolchain including avr-gcc version 5.4.0. The cycle-accurate instruction set simulator thereof helps us to determine the accurate execution times of our software. We compiled our source codes with avr-gcc 5.4.0, using the optimization option `-O2`, on the ATmega1284 microcontroller.

Table 1 specifies the execution time of MAC operation, key generation, encapsulation and decapsulation of our software. A speed-optimized MAC costs only 605 k clock cycles while the memory-optimized version requires 70% more of the clock cycles. The speed gap between these two types of MAC directly affects the overall running time of ME- versus HS-BBear(-Eph), because there are several MACs in each of KeyGen, Encaps and Decaps. Taking HS-BBear as an example, KeyGen, Encaps and Decaps respectively needs about 6.12 M, 7.90 M, and 12.48 M clock cycles, which is approximately 1.5 times faster as its ME variant.

Table 2 illustrates both the RAM footprint and code size of MAC, KeyGen, Encaps and Decaps. The speed-optimized MAC takes 934 bytes dynamic memory while the memory-optimized MAC requires 82 bytes which is only 9% of the former one. Thanks to a memory-optimized MAC and a full-radix representation for field elements, ME-BBear takes 1.7 kB RAM for each of KeyGen and Encaps. Decaps is a little bit more costly and needs 2.4 kB RAM. More notably, ME-BBear-Eph requires only about 1.7 kB in total. In contrast, the HS implementations cost more than 1.5 times RAM memory than their ME variants. In terms of code size, each of the four implementations consumes more or less around 11 kB.

Table 3 compares implementations of both pre- and post-quantum schemes (target 128-bit security) on AVR processors. Compared to another NIST candi-

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

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 implementations on AVR

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

Table 3. Comparison of our software with other key-establishment algorithms (all of which target 128-bit security) on 8-bit AVR platform (Encaps and Decaps in clock cycles; RAM and code size 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
Cheng et al [3]	NTRU	847,973	1,051,871	3,895	9,123
Düll et al [5] (ME)	Curve25519	14,146,844	14,146,844	510	9,912
Düll et al [5] (HS)	Curve25519	13,900,397	13,900,397	494	17,710

date NTRU Prime with a CCA-security [2], HS-BBear is faster on both Encaps and Decpas. Although a CCA-secure NTRU software in [3] is faster than BabyBear, yet their target NTRU is not the latest version and is not supported in the 2nd round NIST PQC Standardization. But compared to it, ME-BBear still saves 39.2% of RAM. On the other hand, when compared with a high-speed implementation of Curve25519 in [5], both Encaps of ME- and HS-BBear are faster than a variable-base scalar multiplication on Curve25519, while the Decaps of ME-BBear is slower but that of HS-BBear is still a bit faster. Notably, the Decaps of our CPA-secure implementations are respectively 4.0 times (ME) and 5.4 times (HS) faster than Curve25519.

One of the most significant advantages of the ThreeBears cryptosystem is the pretty cheap RAM consumption, which is very friendly for deployment on constrained devices especially AVR. Table 4 summarises the RAM consumption of microcontroller implementations of ThreeBears and other NIST PQC schemes. Due to the limited number of state-of-the-art implementations of other NIST PQC candidates for 8-bit AVR, we give in Table 4 also some recent results from the `pqm4` library which targets 32-bit ARM Cortex-M4. We also list the original low-memory implementations of BabyBear and BabyBearEphem from the NIST package of ThreeBears. Our memory-efficient BabyBear is the most RAM-efficient implementation among all the CCA-secure NIST PQC schemes, which saves 5% of RAM than the second most RAM-efficient scheme Kyber. Alternatively, ME-BBear-Eph needs the least RAM memory among all the (CPA-secure) NIST PQC schemes, which improved the original low-memory implementation by a factor of 16.6%.

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

Implementation	Algorithm	Platform	KeyGen	Encaps	Decaps
CCA-secure schemes					
This work (ME)	ThreeBears	ATmega1284	1,715	1,735	2,368
Hamburg [9]	ThreeBears	Cortex-M4	2,288	2,352	3,024
pqm4 [11]	ThreeBears	Cortex-M4	3,076	2,964	5,092
pqm4 [11]	NewHope	Cortex-M4	3,876	5,044	5,044
pqm4 [11]	Round5	Cortex-M4	4,148	4,596	5,220
pqm4 [11]	Kyber	Cortex-M4	2,388	2,476	2,492
pqm4 [11]	NTRU	Cortex-M4	11,848	6,864	5,144
pqm4 [11]	Saber	Cortex-M4	9,652	11,388	12,132
CPA-secure schemes					
This work (ME)	ThreeBears	ATmega1284	1,715	1,735	1,731
Hamburg [9]	ThreeBears	Cortex-M4	2,288	2,352	2,080
pqm4 [11]	ThreeBears	Cortex-M4	3,076	2,980	2,420
pqm4 [11]	NewHope	Cortex-M4	3,836	4,940	3,200
pqm4 [11]	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 *Advances in Cryptology — EUROCRYPT 2013*, pp. 313–314. Springer, 2013.
2. H. Cheng, D. Dinu, J. Großschädl, P. B. Rønne, and P. Y. Ryan. A lightweight implementation of NTRU Prime for the post-quantum internet of things. In *Information Security Theory and Practice — WISTP 2019*, pp. 103–119. Springer, 2020.
3. H. Cheng, J. Großschädl, P. B. Rønne, and P. Y. Ryan. A lightweight implementation of NTRUEncrypt for 8-bit AVR microcontrollers. In *Proceedings of the 2nd NIST PQC Standardization Conference*, 2019. Available online at `http://csrc.nist.gov/Events/2019/second-pqc-standardization-conference`.
4. 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.
5. 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.
6. C. Gu. Integer version of Ring-LWE and its applications. Cryptology ePrint Archive, Report 2017/641, 2017.
7. N. Gura, A. Patel, A. S. Wander, H. Eberle, and S. Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *Cryptographic Hardware and Embedded Systems — CHES 2004*, pp. 119–132. Springer, 2004.
8. M. Hamburg. Ed448-Goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625, 2015.
9. M. Hamburg. ThreeBears: Round 2 specification, 2019. `http://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions`.
10. M. Hutter and P. Schwabe. Multiprecision multiplication on AVR revisited. Cryptology ePrint Archive, Report 2014/592, 2014.
11. 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.
12. A. A. Karatsuba and Y. P. Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 145(2):293–294, ?? 1962.
13. J. M. Kelsey, S.-J. H. Chang, and R. A. Perlner. SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash, 2016. NIST Special Publication 800-185, available for download at `http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-185.pdf`.
14. Z. Liu, H. Seo, J. Großschädl, and H. Kim. Reverse product-scanning multiplication and squaring on 8-bit AVR processors. In *Information and Communications Security — ICICS 2014*, pp. 158–175. Springer, 2015.
15. National Institute of Standards and Technology (NIST). Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. `https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf`.
16. 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.