Post-Quantum Key Encapsulation on 8-bit Microcontrollers: A New Hope for the IoT

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

DSC 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 only around 13 million cycles, which is less than the execution time of a scalar multiplication on Curve25519. The decapsulation is over 3.6 times faster and requires roughly 3.7 million cycles on an ATmega1284 microcontroller.

Keywords: Post-quantum cryptography · Key encapsulation mechanism · ThreeBears · AVR microcontroller · Efficient implementation

1 Introduction

In 2016, the U.S. National Institute of Standards and Technology (NIST) announced an initiative to "solicit, 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 considered by the NIST, i.e. public-key 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-establishment algorithms and nine signature schemes. The 17 algorithms for encryption (resp. key establishment) 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. This second round (which is expected to end around mid 2020) focuses on evaluating the candidates' performance across a wide variety of systems and platforms, which includes "not only big computers and smart phones, but also devices that have limited processor power" [16].

Lattice-based cryptosystems seem the most promising candidates for deployment in embedded and mobile devices thanks to their relatively low computational cost combined with reasonably small keys and ciphertexts (resp. signatures). Indeed, benchmarking results collected in the course of the `pqm4` project[1] for a 32-bit ARM Cortex-M4 microcontroller show that most of the lattice-based Key-Encapsluation Mechanisms (KEMs) in the second round of NIST's evaluation process are faster than ECDH key exchange based on Curve25519, and some proposals are even much faster than Curve25519 [12]. Unfortunately, the results of `pqm4` also indicate that lattice-based cryptosystems demand a massive amount of run-time memory (i.e. RAM) since most of the benchmarked lattice KEMs have a RAM footprint of between 5 and 20 kB. For comparison, a variable-base scalar multiplication on Curve25519 can have a RAM footprint of below 500 bytes [6]. One might argue that the `pqm4` implementations were optimized for high speed rather than low memory consumption, but this argument can be countered by the fact that a straightforward implementation of Curve25519 (i.e. an implementation without any specific measures for RAM reduction) still requires just around 500 bytes of RAM. Consequently, the existing implementation results in the literature lead to the conclusion that lattice-based KEMs require (at least) 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 or RAM. For example, a typical wireless sensor node like the MICAz mote [5] is equipped with an 8-bit AVR microcontroller (e.g. ATmega128L) and comes with only 4 kB internal SRAM. These 4 kB are easily sufficient for Curve25519 (which still leaves 7/8 of the RAM capacity for system and application software), but not for lattice-based KEMs. Consequently, there is a pressing 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 presents software optimization techniques for the ThreeBears KEM [9], a lattice-based cryptosystem that made it into the second round of the NIST post-quantum standardization project. The security of

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

ThreeBears relies on a special variant of the well-known Ring Learning With Errors (LWE) problem, namely the so-called Integer Module Learning with Errors (I-MLWE) problem [4]. ThreeBears is unique among the lattice-based second-round candidates since it uses an integer ring instead of a polynomial ring as underlying algebraic structure. Thus, the main operation of ThreeBears is multi-precision integer arithmetic (namely multiplication modulo a 3120-bit prime) and not polynomial arithmetic.

The usual way to speed up the polynomial multiplication that forms part of all lattice-based schemes except ThreeBears is to use a multiplication technique with sub-quadratic complexity, e.g. Karatsuba's method [13] or the Toom-Cook algorithm [17]. Unfortunately, the performance gain of these techniques comes at the expense of increased memory requirements. For integer arithmetic, on the other hand, there exists a highly effective optimization technique that does not increase RAM footprint, namely the so-called hybrid multiplication method from CHES 2004 [7] or one of its improved variants like the Reverse Product Scanning (RPS) method [14]. In essence, the hybrid technique can be seen as a combination of operand scanning and product scanning that reduces the number of load instructions at the expense of a slight increase in code size since, in each iteration of the inner loop, four bytes of the operands are processed at once. Even though the hybrid technique could also be applied to polynomial multiplication, it is less effective because the coefficients of the polynomials in lattice-based cryptography are usually less than 16 bits long, which means only two bytes of each operand can be processed at a time.

Contributions. This paper examines the performance of ThreeBears on 8-bit AVR microcontrollers and evaluates the flexibility of ThreeBears to achieve different trade-offs between execution time and RAM footprint. We present, to the best of our knowledge, the first highly-optimized software implementations of ThreeBears for the AVR platform, which we developed to reach low RAM consumption, high speed and resistance against timing attacks. In particular, our software is the most RAM-efficient among all publicly-known software implementations of NIST second-round candidates for microcontrollers.

In detail, we take advantage of a full-radix representation for each field element, which allows us to decrease both the RAM footprint and running time. In addition, we propose two novel optimizations for the highly performance-critical Multiply-ACcumulate (MAC) operation: one is memory-optimized whereas the other one is speed-optimized. The memory-optimized MAC focuses on minimising the allocated stack memory and uses the RPS method [14] to accelerate the "tripleMAC" operations therein. Alternatively, the speed-optimized MAC splits the field elements up into two halves and takes advantage of a 3-level Karatsubarized RPS approach. Both types of optimized MAC are developed in AVR Assembly language to reach high speed and constant execution time. As a result, our software includes four implementations: two implementations of CCA-secure BabyBear and two implementations of CPA-secure BabyBearEphem. For each BabyBear and BabyBearEphem, our software contains both a Memory-Efficient

(ME) and a High-Speed (HS) variant[2], which use the corresponding types of MAC implementations.

Flexibility is one of the primary evaluation criteria for the PQC cryptosystems mentioned in the NIST Call for Proposals [15]. More precisely, the document defines flexibility as the ability of algorithms to be "implemented securely and efficiently on a wide variety of platforms, including constrained environments, such as smart cards." The 8-bit AVR architecture serves as a good example for a hardware platform for smart cards, which urgently need efficient implementations of quantum-safe cryptosystems. Unfortunately, as mentioned before, many PQC schemes demand relatively high amounts of run-time memory (often more than 10 kB), which exceeds the RAM capacity of most AVR processors and raises doubts whether such PQC schemes can be employed on AVR devices. However, our work shows that ThreeBears provides flexibility to optimize for RAM footprint *and* still achieve reasonably good execution times. Concretely, a CCA-secure ThreeBears KEM of NIST category 2 security can be optimized to run efficiently with 2.4 kB RAM on AVR, while its CPA-secure instance costs only about 1.7 kB run-time memory. This makes ThreeBears a suitable cryptosystem to secure IoT devices in a quantum world.

2 Preliminaries

2.1 8-bit AVR Microcontrollers

8-bit AVR microcontrollers, one of the most resource-constrained and power-efficient 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, equipped with 32 general-purpose working registers (named `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 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, which features 16 kB SRAM and 128 kB flash memory for storing program code.

2.2 ThreeBears KEM

ThreeBears has three parameter sets namely 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

[2] We call these four implementations ME-BabyBear, ME-BabyBearEphem, HS-BabyBear and HS-BabyBearEphem.

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 details in [9]).

Notation and Parameters. ThreeBears is performed in a field $\mathbb{Z}/N$, where the prime modulus $N = 2^{3120} - 2^{1560} - 1$ is a Goldilocks prime [8] which 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 $r = \textsc{Hash}(sk)$.
4. Generate a $d \times d$ matrix M, where each element $M_{i,j} \in \mathbb{Z}/N$ is sampled from uniform sampler using r.
5. Obtain the vector $\tilde{Z} = (Z_0, \ldots, Z_{d-1})$ 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 $(r, \tilde{Z})$ as *public key.*

Encapsulation. The encapsulation operation gets a public key $(r, \tilde{Z})$ as input and produces a ciphertext and session key 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 M, where each element $M_{i,j} \in \mathbb{Z}/N$ is sampled from uniform sampler by given r.
4. Obtain vector $\tilde{Y} = (Y_0, \ldots, Y_{d-1})$ 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 coding to encode *seed* with X and use this encoded output to extract a fixed-length string f.
6. Compute $ss = \textsc{Hash}(r, seed)$.
7. Output ss as *session key* and $(\tilde{Y}, f)$ as *ciphertext.*

Decapsulation. The decapsulation gets a private key sk and a ciphertext $(\tilde{Y}, f)$ as input and produces a session key 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 with X', and use Melas FEC coding to decode this string to obtain the string $seed'$.
4. Generate the public key $(r', \tilde{Z}')$ through Key Generation by given sk.
5. Repeat Encapsulation to get ss' and $(\tilde{Y}', f')$ by using the obtained $seed'$ and key pair $(sk, (r', \tilde{Z}'))$.
6. Check whether $(\tilde{Y}', f')$ equals to $(\tilde{Y}, f)$; if equal then output ss' as *session key*; if not then output $\textsc{Hash}(sk, \tilde{Y}, f)$ as *session key*.

In above-described algorithms, there exist some so-called "auxiliary" functions such as samplers (noise sampler and uniform sampler), hash functions and error-correcting code. Both the samplers and hash functions, take advantage of the `cSHAKE256` [11], which relies on the Keccak permutation [1] at the lowest layer. Besides, the designer provides a Melas-type forward error correction (FEC) as the error-correcting code in ThreeBears, which has small code/memory requirements and runs in constant time.

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[3] 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 operations on the AVR platform. As stated in our contribution, we designed two strategies of 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

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

multiplication, which used for reducing distortion of the noise. As pointed out in [8], the Goldilocks prime contributes to a fast Karatsuba multiplication [13]. 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:

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

Compared to a conventional Karatsuba multiplication (six additions and three multiplications), the Karatsuba multiplication in $\mathbb{Z}/N$ saves one addition. Consequently, the MAC operation can be performed as Eq. (2) and transformed as Eq. (3):

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

$$= (r_L + a_H b_L - a_L(b_L - b_H)) + (r_H + (a_L + a_H)b_H + a_L(b_L - b_H))\lambda \mod N \tag{3}$$

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 $N = 2^{3120} - 2^{1560} - 1$), where each word does not entirely occupy a default data type. For the instance where `uint32_t` is the default data type, each word is 26 bits long, and a 3120-bit integer consists of 120 words. Becasue each `uint32_t` has six unoccupied bits, it can store the carry or borrow bits during arithmetic computations. It is therefore not urgent to propagate carry/borrow bits instantly, whereby many computations eliminated the dependence with others. High-end processors could beneficially carry out a few computations in parallel and so that saves running time.

However, the AVR microprocessor carries out instructions in sequential order, so the advantage of reduced-radix approach does not exist in our situation. We come up with a *full-radix* representation for field elements, making full use of 32 bits of a `uint32_t` data. Each 3120-bit integer consists of 98 32-bit words. In the multi-precision multiplication on AVR platform, the fewer number of words, the fewer multiplication instructions (`MUL`) are performed. Since `MUL` instruction is costly, the full-radix approach saves considerable running time than the original reduced-radix one. Besides, the 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 stack memory consumption.

Furthermore, we define two storage forms for full-radix field elements: *standard* and *aligned*. We illustrate both forms in Fig. 1, where **L**/**H** stands for the lower/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.

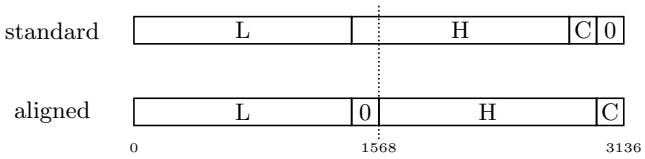

Fig. 1. Standard and aligned form of a field element (AVR uses little-endian)

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 memory of each instance. This low-memory variant is equipped with a specialised RAM-saving MAC operation applied with one-level Karatsuba method, which follows a variant equation of Eq. (3), i.e. Eq. (4) shown below:

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

This low-memory 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. Besides, we add some necessary alignment operations therein.

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** $(R_{s-1}, \ldots, R_1, R_0)$

Algorithm 1 explains our one-level Karatsuba memory-efficient MAC operation, which has two main steps: a product-scanning-based MAC (from line 1 to line 25) and a modular-N reduction (from line 26 to line 42). Both the input and the output of this algorithm are aligned integers, where s is 98 and ω is 32 due to a full-radix representation. β is a parameter of alignment which equals to 8 and it means how many bits we moved when converting an integer from standard to aligned. In addition, the designer gives a name of "tripleMAC" for those three "word-level" MACs in the inner loops (at line 7 to 9 and line 15 to 17). At the beginning of Algorithm 1, Z_0, Z_1 and Z_2 are three 80-bit accumulators, which are sufficient to avoid overflows. 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. In the first inner loop of Algorithm 1, the tripleMAC (lines 7 to 9) directly reflects

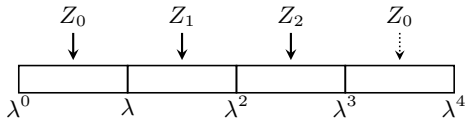

Fig. 2. Three accumulators for coefficients of λ^0, λ and λ^2 of a product R

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 this second loop tripleMAC needs to multiply with 2^β because of the alignment. However, the third operation in the tripleMAC (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. 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 implement the complete Algorithm 1 in AVR Assembler and make use of the RPS multiplication technique to accelerate the most frequently-used tripleMAC computation. 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 during the tripleMAC. We calculate and confirm that the maximal intermediate value of the first inner 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, 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. Consequently, it allows us to save both three `LD`s and `ST`s instructions, totally 12 clock cycles, in each iteration of the inner loop.

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)

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 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 modular-N reduction, which is very similar to the lines from 26 to 42 in Algorithm 1. Unlike the memory-efficient MAC that takes a product-scanning approach to save memory, 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). It is not necessary to load and store three accumulators in each inner loop iteration and therefore saves significant execution time. But consequently, it needs more dynamic memory to store the intermediate products (e.g. Z_H, Z_L and T_H, T_L).

In [14], it is concluded that 2-level Karatsuba multiplication combined with RPS technique (i.e. 2-level KRPS) would yield a peak performance for a 1560-bit

multi-precision multiplication on AVR. As a result, we take advantage of a 2-level KRPS multiplication for each half-size multiplication, and therefore a 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

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. 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. 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 MAC needs, almost the double-time. The speed gap between these two types of MAC directly affects the overall running time of ME- versus HS- BabyBear(Ephem), because there are several MACs in each of KeyGen, Encaps and Decaps. Taking HS-BabyBear as an example, KeyGen, Encaps and Decaps respectively needs about 6.12 M, 7.90 M, and 12.48 M clock cycles, which is more than 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-BabyBear 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-BabyBearEphem 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 1. Execution time (in clock cycles) of our implementations on AVR

Implementation	Security	MAC	KeyGen	Encaps	Decaps
ME-BabyBear	CCA-secure	1,183,453	9,345,332	13,188,102	20,075,571
ME-BabyBearEphem	CPA-secure	1,183,453	9,345,332	13,333,525	3,743,596
HS-BabyBear	CCA-secure	604,703	6,123,527	7,901,873	12,476,447
HS-BabyBearEphem	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-BabyBear	82	2,710	1,715	6,382	1,735	7,504	2,368	10,060	2,368	12,214
ME-BabyBearEphem	82	2,710	1,715	6,382	1,735	7,590	1,731	8,220	1,735	10,948
HS-BabyBear	934	3,332	2,733	7,000	2,752	8,140	4,559	10,684	4,559	11,568
HS-BabyBearEphem	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.

Implementation	Algorithm	Encaps	Decaps	RAM	Size
This work (ME-CCA)	ThreeBears	13,188,102	20,075,571	2,368	12,214
This work (ME-CPA)	ThreeBears	13,333,525	3,743,596	1,735	10,948
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 [6] (ME)	Curve25519	14,146,844	14,146,844	510	9,912
Düll et al [6] (HS)	Curve25519	13,900,397	13,900,397	494	17,710

Table 3 compares implementations of both pre- and post-quantum schemes (target 128-bit security) on AVR processors. Compared to another NIST candidate NTRU Prime with a CCA-security [2], HS-BabyBear 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-BabyBear still saves 39.2% of RAM. On the other hand, when compared with a high-speed implementation of Curve25519 (a widely-used ECC-based KEM) in [6], both Encaps of ME- and HS-BabyBear are faster than a variable-base scalar multiplication on Curve25519, while the Decaps of ME-BabyBear is slower but that of HS-BabyBear is still a bit faster. Notably, the Decaps of our CPA-secure implementations saves respectively 73.1% (ME) and 81.3% (HS) running time compared to Curve25519.

One of the most significant advantages of the ThreeBears cryptosystem is the pretty cheap RAM consumption, which is very friendly for employment 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(Ephem) from the NIST package of ThreeBears. We count both the consumption of stack memory and of heap memory, as well as `.data` and `.bss`, to RAM consumption. 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-

Table 4. Comparison of RAM consumption of NIST PQC implementations (all of which target NIST security category 1 or 2) on various 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` [12]	ThreeBears	Cortex-M4	3,076	2,964	5,092
`pqm4` [12]	NewHope	Cortex-M4	3,876	5,044	5,044
`pqm4` [12]	Round5	Cortex-M4	4,148	4,596	5,220
`pqm4` [12]	Kyber	Cortex-M4	2,388	2,476	2,492
`pqm4` [12]	ROLLO-I	Cortex-M4	3,624	3,540	3,608
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` [12]	ThreeBears	Cortex-M4	3,076	2,980	2,420
`pqm4` [12]	NewHope	Cortex-M4	3,836	4,940	3,200
`pqm4` [12]	Round5	Cortex-M4	4,052	4,500	2,308

efficient scheme Kyber. Alternatively, ME-BabyBearEphem 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%.

5 Conclusions

This paper presented the first highly-optimized and timing-attack-resistant implementations of ThreeBears for the 8-bit AVR architecture. Our simulation results confirm that ThreeBears offers good flexibility and is well suited for implementation on smart cards. Both the two memory-efficient implementations, as well as a CPA-secure high-speed variant, can be used on most AVR microcontrollers, even on an ATmega128L, which features only 4 kB SRAM. For AVR microcontrollers with more than 8 kB SRAM, both high-speed variants can be deployed to shorten the running time. Our implementation sets a new record for memory-efficiency among all known software implementation of second-round candidates on microcontrollers. A comparison with Curve25519-based key exchange shows that our memory-efficient implementation with CPA-security still exceeds the RAM footprint of Curve25519 by more than 1.2 kB, but the encapsulation is slightly faster than a scalar multiplication and the decapsulation is even significantly faster. Furthermore, all proposed optimization techniques are applicable to MamaBear and PapaBear as well. In summary, our work shows that ThreeBears can be well optimized to achieve both low RAM footprint and high speed on resource-constrained microcontrollers, which makes ThreeBears a suitable post-quantum cryptosystem to secure IoT devices.

Acknowledgements This work was supported by the EU 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*, pages 313–314, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
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*, pages 103–119, Cham, 2020. Springer International Publishing.
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. G. Chunsheng. Integer version of ring-lwe and its applications. Cryptology ePrint Archive, Report 2017/641, 2017. `https://eprint.iacr.org/2017/641`.
5. 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.
6. 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.
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 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.
8. M. Hamburg. Ed448-goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625, 2015. `https://eprint.iacr.org/2015/625`.
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. Available for download at `http://eprint.iacr.org/`.
11. John M. Kelsey and Shu-jen H. Chang and Ray A. Perlner. Sha-3 derived functions: cshake, kmac, tuplehash and parallelhash, 2016. NIST Special Publication 800-185, available for download at `https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-185.pdf`.
12. 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. Available for download at `http://eprint.iacr.org`.
13. A. A. Karatsuba and Y. P. Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 145(2):293–294, ?? 1962.
14. 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.

15. National Institute of Standards and Technology. 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.
17. A. L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics - Doklady*, 4(3):714–716, May 1963.