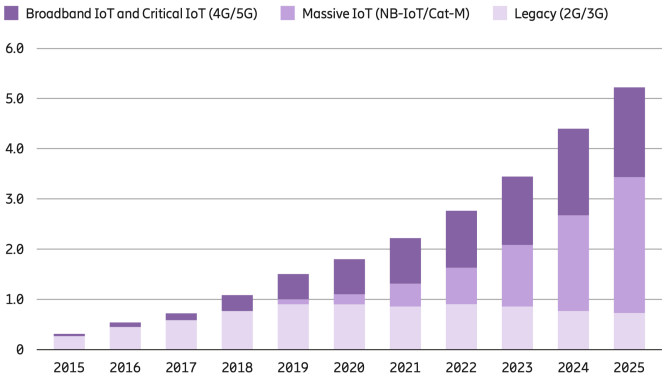
Lightweight Post-Quantum Key Encapsulation for 8-bit AVR Microcontrollers

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

University of Luxembourg

CARDIS2020

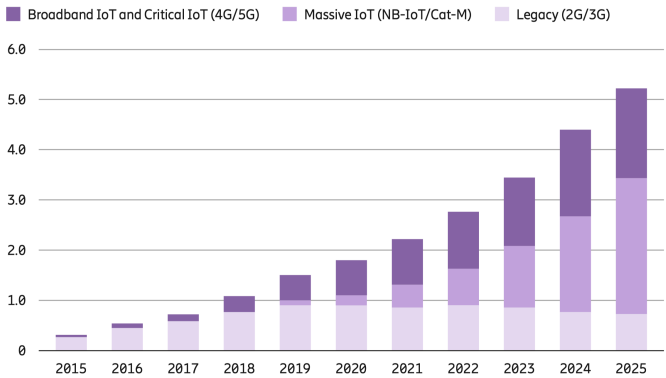
IoT on the Rise



IoT	2019	2025	CAGR
Wide-area IoT	1.6	5.5	23%
Cellular IoT[3]	1.5	5.2	23%
Short-range IoT	9.1	19.1	13%
Total	**10.7**	**24.6**	**15%**

Source: Ericsson Mobility Report (June 2020)

IoT on the Rise



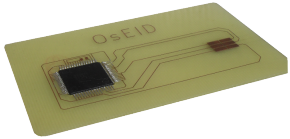
IoT	2019	2025	CAGR
Wide-area IoT	1.6	5.5	23%
Cellular IoT[3]	1.5	5.2	23%
Short-range IoT	9.1	19.1	13%
Total	**10.7**	**24.6**	**15%**

Source: Ericsson Mobility Report (June 2020)

IoT needs lightweight cryptographic schemes and protocols

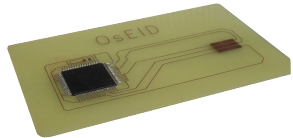
8-bit AVR Microcontrollers

Atmel®



8-bit AVR Microcontrollers

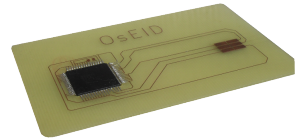
Atmel®



8-bit AVR Architecture

- RISC philosophy and modified Harvard memory model
- 32 general-purpose working registers of 8-bit width

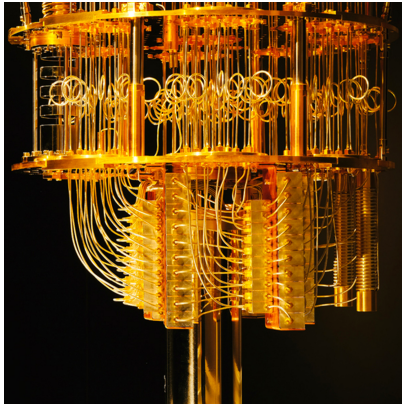
8-bit AVR Microcontrollers



8-bit AVR Architecture

- RISC philosophy and modified Harvard memory model
- 32 general-purpose working registers of 8-bit width
- Bitwise logical and most arithmetic instructions take 1 clock cycle
- Multiplication and RAM accessing instructions take 2 clock cycles

Quantum Cryptanalyses



Quantum Computing

exploits quantum-mechanical phenomena

Algorithms for Quantum Computation:
Discrete Logarithms and Factoring

Peter W. Shor
AT&T Bell Labs
Room 2D-149
600 Mountain Ave.
Murray Hill, NJ 07974, USA

Abstract

A computer is generally considered to be a universal computational device; i.e., it is believed able to simulate any physical computational device with a cost in computation time of at most a polynomial factor. It is not clear whether this is still true when quantum mechanics is taken into consideration. Several researchers, starting with David Deutsch, have developed models for quantum mechanical computers and have investigated their computational properties. This paper gives Las Vegas algorithms for finding discrete logarithms and factoring integers on a quantum computer that take a number of steps which is polynomial in the input size, e.g., the number of digits of the integer to be factored. These two problems are generally considered hard on a classical computer and have been used as the basis of several proposed cryptosystems. (We thus give the first examples of quantum cryptanalysis.)

1 Introduction

Since the discovery of quantum mechanics, people have found the behavior of the laws of probability in quantum mechanics counterintuitive. Because of this behavior, quantum mechanical phenomena behave quite differently than the phenomena of classical physics that we are used to. Feynman seems to have been the first to ask what effect this has on computation [13, 14]. He gave arguments as [1, 2]. Although he did not ask whether quantum mechanics conferred extra power to computation, he did show that a Turing machine could be simulated by the reversible unitary evolution of a quantum process, which is a necessary prerequisite for quantum computation. Deutsch [9, 10] was the first to give an explicit model of quantum computation. He defined both quantum Turing machines and quantum circuits and investigated some of their properties.

The next part of this paper discusses how quantum computation relates to classical complexity classes. We will thus first give a brief intuitive discussion of complexity classes for those readers who do not have this background. There are generally two resources which limit the ability of computers to solve large problems: time and space (i.e., memory). The field of analysis of algorithms considers the asymptotic demands that algorithms make for these resources as a function of the problem size. Theoretical computer scientists generally classify algorithms as efficient when the number of steps of the algorithms grows as a polynomial in the size of the input. The class of problems which can be solved by efficient algorithms is known as P. This classification has several nice properties. For one thing, it does a reasonable job of reflecting the performance of algorithms in practice (although an algorithm whose running time is the tenth power of the input size, say, is not truly efficient). For another, this classification is nice theoretically, as different reasonable machine models produce the same class P. We will see this behavior reappear in quantum computation, where different models for

Shor's Algorithm

solves IFP and DLP in polynomial time

NIST PQC Standardization


NIST

Information Technology Laboratory

COMPUTER SECURITY RESOURCE CENTER

PROJECTS | POST-QUANTUM CRYPTOGRAPHY

Post-Quantum Cryptography PQC

Post-Quantum Cryptography Standardization

The Round 3 candidates were announced July 22, 2020. NISTIR 8309, Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process is now available. NIST has developed Guidelines for Submitting Tweaks for Third Round Finalists and Candidates.

- Solicit, evaluate and standardize one or more quantum-resistant PKC algorithms
- Evaluate candidates' performance also on resource-constrained devices
- Now is Round 3

Lattice-Based KEMs in IoT

- Benchmarking results collected in `pqm4` [1] (ARM Cortex-M4)
  - faster than Curve25519
  - RAM footprint often between 5 kB ~ 30 kB (vs 500 bytes of Curve25519)

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

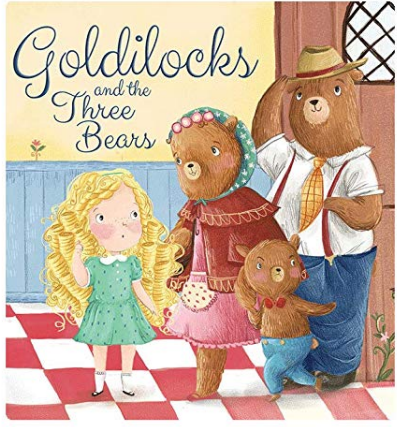
Lattice-Based KEMs in IoT

- Benchmarking results collected in `pqm4` [1] (ARM Cortex-M4)
  - faster than Curve25519
  - RAM footprint often between 5 kB ~ 30 kB (vs 500 bytes of Curve25519)
- Deployment in AVR devices
  - AVR devices feature only a few kB of RAM (e.g. MICAz mote has only 4 kB RAM)
  - need low-memory implementations

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

A Fairy Tale

- Designer
  - Mike Hamburg
- Ed448-Goldilocks [Ham15]
  - RFC7748 and TLS 1.3
  - "Golden-ratio" Solinas prime $2^{448} - 2^{224} - 1$ (Goldilocks)
- ThreeBears
  - NIST PQC Round 2 candidate (KEM)
  - Integer Module Learning With Errors (I-MLWE) [Gu17]
  - BabyBear (II), MamaBear (IV), PapaBear (V)
  - has both CCA and CPA instances



This Work

- Analyzes the performance of ThreeBears on AVR
- Studies its flexibility to achieve different trade-offs between RAM footprint and execution time

Our Implementation

- First highly-optimized software implementations of BabyBear for AVR platform (constant-time)

Our Implementation

- First highly-optimized software implementations of BabyBear for AVR platform (constant-time)
  - **M**emory-**E**fficient ME-BBear (CCA) ME-BBear-Eph (CPA)
    based on low-memory implementation in NIST package
    most memory-efficient software implementation of Round 2 candidate

Our Implementation

- First highly-optimized software implementations of BabyBear for AVR platform (constant-time)
  - **M**emory-**E**fficient ME-BBear (CCA) ME-BBear-Eph (CPA)
    based on low-memory implementation in NIST package
    most memory-efficient software implementation of Round 2 candidate
  - **H**igh-**S**peed HS-BBear (CCA) HS-BBear-Eph (CPA)
    based on optimized implementation in NIST package

Our Implementation

- First highly-optimized software implementations of BabyBear for AVR platform (constant-time)
  - **M**emory-**E**fficient ME-BBear (CCA) ME-BBear-Eph (CPA)
    based on low-memory implementation in NIST package
    most memory-efficient software implementation of Round 2 candidate
  - **H**igh-**S**peed HS-BBear (CCA) HS-BBear-Eph (CPA)
    based on optimized implementation in NIST package
- Memory-optimized and speed-optimized
  Multiply-**AC**cumulate (MAC) operations $r = r + a * b$

ThreeBears KEM

- The underlying field
  - $\mathbb{Z}/N$
- Prime (“golden-ratio” Solinas prime [Ham15])
  - $N = 2^{3120} - 2^{1560} - 1$
  - $N = \phi(x) = x^D - x^{D/2} - 1$
  - $N = \lambda^2 - \lambda - 1$

ThreeBears KEM

- The underlying field
  - $\mathbb{Z}/N$
- Prime (“golden-ratio” Solinas prime [Ham15])
  - $N = 2^{3120} - 2^{1560} - 1$
  - $N = \phi(x) = x^D - x^{D/2} - 1$
  - $N = \lambda^2 - \lambda - 1$
- Field operations
  - $(+, \cdot)$ are conventional integer addition and multiplication
  - addition $(+)$ $\quad a + b := a + b \bmod N$
  - multiplication $(*)$ $\quad a * b := a \cdot b \cdot \lambda^{-1} \bmod N$

ThreeBears KEM (CCA)

Key Generation

$sk \leftarrow random()$

$\boldsymbol{a}, \boldsymbol{b} \leftarrow noise\_sampler(sk)$

$r \leftarrow hash(sk)$

$\boldsymbol{M} \leftarrow uniform\_sampler(r)$

$\boldsymbol{z} \leftarrow z_i = b_i + \Sigma_{j=0}^{d-1} \boldsymbol{M}_{i,j} * a_j$

- - - - - - - - - - - - - - - - - - - -

private key sk

public key $(r, \boldsymbol{z})$

Dimension d is 2 for BabyBear, 3 for MamaBear, 4 for PapaBear

ThreeBears KEM (CCA)

Key Generation

$sk \leftarrow random()$

$\boldsymbol{a}, \boldsymbol{b} \leftarrow noise\_sampler(sk)$

$r \leftarrow hash(sk)$

$\boldsymbol{M} \leftarrow uniform\_sampler(r)$

$\boldsymbol{z} \leftarrow z_i = b_i + \Sigma_{j=0}^{d-1} M_{i,j} * a_j$

---

private key sk

public key $(r, \boldsymbol{z})$

Encapsulation

$g \leftarrow random()$

$\hat{\boldsymbol{a}}, \hat{\boldsymbol{b}}, c \leftarrow noise\_sampler(r, g)$

$\boldsymbol{M} \leftarrow uniform\_sampler(r)$

$\boldsymbol{y} \leftarrow y_i = \hat{b}_i + \Sigma_{j=0}^{d-1} M_{j,i} * \hat{a}_j$

$x = c + \Sigma_{j=0}^{d-1} z_j * \hat{a}_j$

$f \leftarrow (FEC\_encode(g),\ x)$

$ss \leftarrow hash(r, g)$

---

shared secret ss

ciphertext $(f, \boldsymbol{y})$

Dimension d is 2 for BabyBear, 3 for MamaBear, 4 for PapaBear

ThreeBears KEM (CCA)

Key Generation

$sk \leftarrow random()$

$\boldsymbol{a}, \boldsymbol{b} \leftarrow noise\_sampler(sk)$

$r \leftarrow hash(sk)$

$\boldsymbol{M} \leftarrow uniform\_sampler(r)$

$\boldsymbol{z} \leftarrow z_i = b_i + \Sigma_{j=0}^{d-1} M_{i,j} * a_j$

---

private key sk

public key $(r, \boldsymbol{z})$

Encapsulation

$g \leftarrow random()$

$\hat{\boldsymbol{a}}, \hat{\boldsymbol{b}}, c \leftarrow noise\_sampler(r, g)$

$\boldsymbol{M} \leftarrow uniform\_sampler(r)$

$\boldsymbol{y} \leftarrow y_i = \hat{b}_i + \Sigma_{j=0}^{d-1} M_{j,i} * \hat{a}_j$

$x = c + \Sigma_{j=0}^{d-1} z_j * \hat{a}_j$

$f \leftarrow (FEC\_encode(g), x)$

$ss \leftarrow hash(r, g)$

---

shared secret ss

ciphertext $(f, \boldsymbol{y})$

Decapsulation

$\boldsymbol{a} \leftarrow noise\_sampler(sk)$

$x = \Sigma_{j=0}^{d-1} y_j * a_j$

$g \leftarrow FEC\_decode(f, x)$

$(r', \boldsymbol{z'}) \leftarrow KeyGen(sk)$

$ss', (f', \boldsymbol{y'}) \leftarrow Encaps(g, (r', \boldsymbol{z'}))$

$(f', \boldsymbol{y'}) \stackrel{?}{=} (f, \boldsymbol{y})$

---

shared secret

$ss' \leftarrow$ ✓

$hash(sk, f, \boldsymbol{y}) \leftarrow$ ✗

Dimension d is 2 for BabyBear, 3 for MamaBear, 4 for PapaBear

ThreeBears KEM (CPA)

Key Generation

$sk \leftarrow random()$

$\boldsymbol{a}, \boldsymbol{b} \leftarrow noise\_sampler(sk)$

$r \leftarrow hash(sk)$

$\boldsymbol{M} \leftarrow uniform\_sampler(r)$

$\boldsymbol{z} \leftarrow z_i = b_i + \Sigma_{j=0}^{d-1} M_{i,j} * a_j$

---

private key sk

public key $(r, \boldsymbol{z})$

Encapsulation

$g \leftarrow random()$

$\hat{\boldsymbol{a}}, \hat{\boldsymbol{b}}, c \leftarrow noise\_sampler(r, g)$

$\boldsymbol{M} \leftarrow uniform\_sampler(r)$

$\boldsymbol{y} \leftarrow y_i = \hat{b}_i + \Sigma_{j=0}^{d-1} M_{j,i} * \hat{a}_j$

$x = c + \Sigma_{j=0}^{d-1} z_j * \hat{a}_j$

$t \leftarrow hash(r, g)$

$f \leftarrow$ (FEC_encode(t), x)

$ss \leftarrow hash(r, t)$

---

shared secret ss

ciphertext $(f, \boldsymbol{y})$

Decapsulation

$\boldsymbol{a} \leftarrow noise\_sampler(sk)$

$x = \Sigma_{j=0}^{d-1} y_j * a_j$

$t \leftarrow FEC\_decode(f, x)$

$r \leftarrow hash(sk)$

$ss \leftarrow hash(r, t)$

---

shared secret ss

Dimension d is 2 for BabyBear, 3 for MamaBear, 4 for PapaBear

Implementation View Point

- Auxiliary functions
  - samplers: noise/uniform sampler → cSHAKE256 → Keccak permutation
  - forward error correction (FEC): Melas BCH code
- Arithmetic components
  - MAC operation: $r = r + a * b \mod N$

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

Implementation View Point

- Auxiliary functions
  - samplers: noise/uniform sampler → cSHAKE256 → Keccak permutation
    open-source highly-optimized AVR Assembler[2]
  - forward error correction (FEC): Melas BCH code

- Arithmetic components
  - MAC operation: $r = r + a * b \bmod N$

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

Implementation View Point

- Auxiliary functions
  - samplers: noise/uniform sampler → cSHAKE256 → Keccak permutation

    open-source highly-optimized AVR Assembler[2]
  - forward error correction (FEC): Melas BCH code

    small memory/code requirements, constant time and runtime is almost negligible
- Arithmetic components
  - MAC operation: $r = r + a * b \bmod N$

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

Implementation View Point

- Auxiliary functions
  - samplers: noise/uniform sampler → cSHAKE256 → Keccak permutation
    open-source highly-optimized AVR Assembler[2]
  - forward error correction (FEC): Melas BCH code
    small memory/code requirements, constant time and runtime is almost negligible
- Arithmetic components
  - MAC operation: $r = r + a * b \mod N$
    dominate both the RAM footprint and the execution time !!

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

Field Element Representation

$$N = 2^{3120} - 2^{1560} - 1 \quad \rightarrow \quad \text{3120-bit integer}$$

Field Element Representation

$$N = 2^{3120} - 2^{1560} - 1 \quad \rightarrow \quad \text{3120-bit integer}$$

Reduced-radix representation ($w = 32$)

- Format 120 × 26 bits = 3120 bits
- RAM usage 120 × 4 bytes = 480 bytes

Field Element Representation

$$N = 2^{3120} - 2^{1560} - 1 \quad \rightarrow \quad \text{3120-bit integer}$$

Reduced-radix representation ($w = 32$)

- Format 120 × 26 bits = 3120 bits
- RAM usage 120 × 4 bytes = 480 bytes

Full-radix representation ($w = 32$)

- Format 97.5 × 32 bits = 3120 bits
- RAM usage 98 × 4 bytes = 392 bytes
- Why?
  - Sequential order in AVR
  - Reduce RAM consumption

MAC Operation $r = r + a * b \bmod N$

$$N = \lambda^2 - \lambda - 1 \quad \rightarrow \quad \lambda^{-1} = \lambda - 1$$

MAC Operation $r = r + a * b \bmod N$

$$N = \lambda^2 - \lambda - 1 \quad \rightarrow \quad \lambda^{-1} = \lambda - 1$$

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

e_L/e_H stands for the lower/higher half of element e

MAC Operation $r = r + a * b \bmod N$

$$N = \lambda^2 - \lambda - 1 \quad \rightarrow \quad \lambda^{-1} = \lambda - 1$$

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

e_L/e_H stands for the lower/higher half of element e

MAC Operation $r = r + a * b \bmod N$

$$
\begin{aligned}
r &:= r + a * b \mod N \\
&= (r_L + a_H b_H - (a_L - a_H)(b_L - b_H)) + (r_H + a_L b_L + a_H b_H)\lambda \mod N && (2) \\
&= (r_L + a_H b_L - a_L(b_L - b_H)) + (r_H + (a_L + a_H) b_H + a_L(b_L - b_H))\lambda \mod N && (3) \\
&= (r_L + a_H b_L - 2a_L(b_L - b_H)) + (r_H + (a_L + a_H) b_H)\lambda + a_L(b_L - b_H)\lambda^2 \mod N && (4)
\end{aligned}
$$

Memory-Optimized MAC

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

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 $R = R + A \cdot B \cdot \lambda \bmod N = (R_{s-1},\ldots,R_1,R_0)$

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

RAM consumption

- Three 80-bit accumulators
- some local variables
- no more levels of Karatsuba (reduce RAM usage)

Memory-Optimized MAC

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

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 $R = R + A \cdot B \cdot \lambda \bmod N = (R_{s-1}, \ldots, R_1, R_0)$

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

Main MAC loop

- Product-scanning
- Interleaved with modular reductions $\lambda^2 = \lambda + 1$ (lines from 19 to 24)

Memory-Optimized MAC

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

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 $R = R + A \cdot B \cdot \lambda \bmod N = (R_{s-1}, \ldots, R_1, R_0)$

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

Final reduction modulo N
Carry propagation

Memory-Optimized MAC

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

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 $R = R + A \cdot B \cdot \lambda \bmod N = (R_{s-1}, \ldots, R_1, R_0)$

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

Inner loop operations

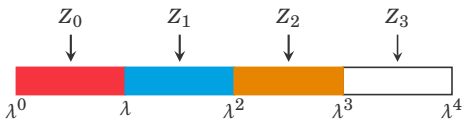
Triple MAC

Memory-Optimized MAC

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

Algorithm 2 First triple MAC loop

1: $Z_2 \leftarrow 0$, $k \leftarrow i + 1$
2: **for** j from 0 to i by 1 **do**
3: $\quad k \leftarrow k - 1$
4: $\quad Z_0 \leftarrow Z_0 + A_{j+l} \cdot B_k$
5: $\quad Z_1 \leftarrow Z_1 + (A_j + A_{j+l}) \cdot B_{k+l}$
6: $\quad Z_2 \leftarrow Z_2 + A_j \cdot (B_k - B_{k+l})$
7: **end for**
8: $Z_0 \leftarrow Z_0 - 2 \cdot Z_2$



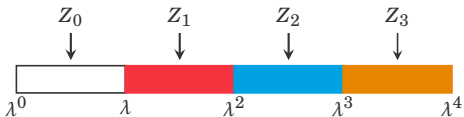
product of $A \cdot B$

Memory-Optimized MAC

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

Algorithm 3 Second triple MAC loop

1: $k \leftarrow l$
2: **for** j from $i+1$ to $l-1$ by 1 **do**
3: $\quad k \leftarrow k - 1$
4: $\quad Z_1 \leftarrow Z_1 + 2^\beta \cdot A_{j+l} \cdot B_k$
5: $\quad Z_2 \leftarrow Z_2 + 2^\beta \cdot (A_j + A_{j+l}) \cdot B_{k+l}$
6: $\quad Z_3 \leftarrow Z_3 + 2^\beta \cdot A_j \cdot (B_k - B_{k+l})$
7: **end for**
8: $Z_1 \leftarrow Z_1 - 2 \cdot Z_3$



product of $A \cdot B$

Memory-Optimized MAC

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

Algorithm 3 Second triple MAC loop

1: $k \leftarrow l$
2: **for** j from $i+1$ to $l-1$ by 1 **do**
3: $\quad k \leftarrow k - 1$
4: $\quad Z_1 \leftarrow Z_1 + 2^\beta \cdot A_{j+l} \cdot B_k$
5: $\quad Z_2 \leftarrow Z_2 + 2^\beta \cdot (A_j + A_{j+l}) \cdot B_{k+l}$
6: $\quad Z_3 \leftarrow Z_3 + 2^\beta \cdot A_j \cdot (B_k - B_{k+l})$
7: **end for**
8: ~~$Z_1 \leftarrow Z_1 - 2 \cdot Z_3$~~

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

$$Z_0 \leftarrow Z_0 + Z_3$$
$$Z_1 \leftarrow Z_1 - 2 \cdot Z_3 + 2 \cdot Z_3 = Z_1$$

Memory-Optimized MAC

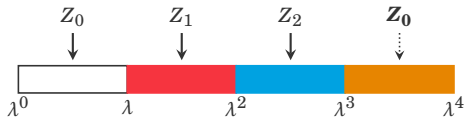
$$r := (r_L + a_H b_L - 2a_L(b_L - b_H)) + (r_H + (a_L + a_H)b_H)\lambda + a_L(b_L - b_H)\lambda^2 \bmod N$$

Algorithm 3 Second triple MAC loop

1: $k \leftarrow l$
2: **for** j from $i+1$ to $l-1$ by 1 **do**
3: $\quad k \leftarrow k - 1$
4: $\quad Z_1 \leftarrow Z_1 + 2^\beta \cdot A_{j+l} \cdot B_k$
5: $\quad Z_2 \leftarrow Z_2 + 2^\beta \cdot (A_j + A_{j+l}) \cdot B_{k+l}$
6: $\quad \mathbf{Z_0} \leftarrow \mathbf{Z_0} + 2^\beta \cdot A_j \cdot (B_k - B_{k+l})$
7: **end for**

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

$$Z_0 \leftarrow Z_0 + Z_3$$
$$Z_1 \leftarrow Z_1 - 2 \cdot Z_3 + 2 \cdot Z_3 = Z_1$$

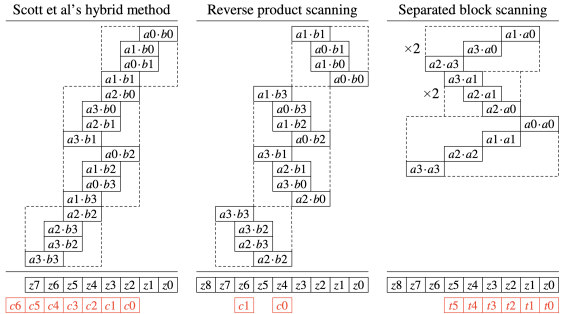


product of $A \cdot B$

(4×4)-byte Multiplication

Reverse Product Scanning (RPS) multiplication [LSGK14]

- Enhanced variant of conventional hybrid multiplication [GPW+04]
- Fast, small-code-size, fewer-registers and parameterized



Inner-loop operations of RPS multiplication (middle)

Speed-Optimized MAC

$$
\begin{aligned}
r :&= (r_L + a_H b_H - (a_L - a_H)(b_L - b_H)) + (r_H + a_L b_L + a_H b_H)\lambda \mod N \\
&= (r_L + h + m) + (r_H + l + h)\lambda \mod N \\
&= (r_L + (h_L + h_H\lambda) + (m_L + m_H\lambda)) + (r_H + (l_L + l_H\lambda) + (h_L + h_H\lambda))\lambda \mod N \\
&= (r_L + h_L + m_L) + (r_H + l_L + h_L + m_H + h_H)\lambda + (l_H + h_H)\lambda^2 \mod N \\
&= (r_L + m_L + \underline{h_L + h_H + l_H}) + (r_H + m_H + h_H + l_L + \underline{h_L + h_H + l_H})\lambda \mod N \qquad (5)
\end{aligned}
$$

$\boldsymbol{l}$ $a_L b_L$

$\boldsymbol{m}$ $-(a_L - a_H)(b_L - b_H)$

$\boldsymbol{h}$ $a_H b_H$

Speed-Optimized MAC

$$r := (r_L + m_L + \underline{h_L + h_H + l_H}) + (r_H + m_H + h_H + l_L + \underline{h_L + h_H + l_H})\lambda \mod N$$

Experiments for speed-optimized MAC

- Combination
  - Subtractive Karatsuba method [HS14] $\Theta(n^{\log_2 3})$
  - RPS multiplication [LSGK14] $\Theta(n^2)$
- Result
  - 3-level Karatsuba with (390×390)-bit RPS multiplication underneath for the entire MAC

MAC Optimization Strategies

- Memory-optimized MAC operation
  - Equation (4)
  - one-level Karatsuba multiplication (product-scanning)
  - RPS technique for inner-loop operation
- Speed-optimized MAC operation
  - Equation (5)
  - three-level Karatsuba multiplication
  - RPS multiplication

Measurement Environment

Experiment Setup

- Target MCU: ATmega1284 (16 kB RAM; 128 kB flash memory)
- Development tool: Atmel Studio v7.0
- Compiler: avr-gcc 5.4.0

Our source code

- AVR Assembler: MAC operation; Keccak permutation
- C code: other components

Performance Evaluation

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

HS version is 1.5x faster compared to ME

Performance Evaluation

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

ME version is 1.5x RAM-efficient compared to HS

Comparison – AVR Implementations

Comparison with other key-establishment algorithms (all of which target 128-bit security)

on 8-bit AVR (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
[CDG+19]	NTRU Prime	8,160,665	15,602,748	n/a	11,478
[DHH+15] (ME)	Curve25519	14,146,844	14,146,844	510	9,912
[DHH+15] (HS)	Curve25519	13,900,397	13,900,397	**494**	17,710

Encaps 1.12x faster; Decaps 4.0x faster; RAM 3.5x more; compared to Curve25519

Comparison – RAM Footprint

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

Implementation	Algorithm	Platform	KeyGen	Encaps	Decaps
CCA-secure schemes					
This work (ME)	ThreeBears	AVR	1,715	1,735	**2,368**
[Ham19]	ThreeBears	Cortex-M4	2,288	2,352	3,024
pqm4	ThreeBears	Cortex-M4	3,076	2,964	5,092
pqm4	Kyber	Cortex-M4	2,388	2,476	2,492
pqm4	NTRU	Cortex-M4	11,848	6,864	5,144
pqm4	Saber	Cortex-M4	9,652	11,388	12,132
CPA-secure schemes					
This work (ME)	ThreeBears	AVR	1,715	**1,735**	1,731
[Ham19]	ThreeBears	Cortex-M4	2,288	2,352	2,080
pqm4	ThreeBears	Cortex-M4	3,076	2,980	2,420
pqm4	NewHope	Cortex-M4	3,836	4,940	3,200
pqm4	Round5	Cortex-M4	4,052	4,500	2,308

The most RAM-efficient software implementation of Round 2 candidates

Conclusion

- The first highly-optimized Assembler implementation of ThreeBears for AVR
- Many trade-offs between execution time and RAM consumption are possible
- A new record for memory efficiency among second-round candidates
- Very well suited for a hybrid pre/post-quantum key agreement protocol
- An excellent candidate for a post-quantum cryptosystem to secure the IoT

Thank you for your attention!