AVRNTRU: Lightweight NTRU-Based Post-Quantum Cryptography for 8-bit AVR Microcontrollers

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

University of Luxembourg
8-bit AVR Devices

• Memsic Iris Sensor Node
  - 8-bit ATmega1281 microcontroller
  - 8 kB RAM, 128 kB flash memory

• 8-bit AVR Architecture
  - 8-bit RISC, 133 instructions
  - 32 general-purpose registers
  - Most arithmetic/logic instructions 1 cycle
  - RAM accessing and mul instructions 2 cycles
**NTRUEncrypt Timeline**

- **1996**
  - **NTRUEncrypt** (PKE) was invented by Hoffstein, Pipher, and Silverman.

- **2003**
  - Efficient Embedded Security Standards (*EESS*) #1 Version 2.0, PKE

- **2015**
  - **EESS #1 Version 3.1**, PKE
    - (currently used in the industry, e.g. wolfSSL)
    - **Our target version**

- **2019**
  - **NTRUEncrypt** is a 2nd round candidate in NIST PQC, KEM.
    - (Merger of **NTRUEncrypt** and **NTRU-HRSS**, based on the **Saito-Xagawa-Yamakawa variant of NTRU-HRSS** [SXY18])

- **2017**
  - **NTRUEncrypt** is a 1st round candidate in NIST PQC.
    - (**EESS #1 Version 3.3**, PKE and KEM)

- **2020**
  - **NTRUEncrypt** is a finalist in the NIST PQC 3rd round.

---

Our Contribution

• **AVRNTRU: NTRUEncrypt for 8-bit AVR microcontrollers**
  - Compliant with EESS #1 version 3.1 (Sept. 2015)
  - Supports product-form parameter sets with SHA-256, e.g. EES443EP1 (128-bit) and EES743EP1 (256-bit)
  - Scalable: change parameter set w/o re-compilation
  - Resistance against timing attacks

• **Ring arithmetic (multiplication** \( r(x) = u(x) \ast v(x) \)**
  - Product-form polynomials
  - “Hybrid” multiplication method from CHES 2004 [GPW+04]
  - Record-setting execution time
Ring Multiplication $r(x) = u(x) \ast v(x)$

• Underlying ring of NTRUEncrypt
  - Truncated polynomial ring $R = \mathbb{Z}_q[x]/(x^N - 1)$
  - Typical instantiation (128-bit): $N = 443, q = 2^{11} = 2048$

• Polynomial multiplication with reductions
  - Operand $u(x)$ is ring element; $v(x)$ is a ternary polynomial
  - Polynomial-level reduction: modulo the quotient polynomial $x^N - 1$ to get result of degree $N - 1$ (operands $u(x)$ and $v(x)$ of degree $N - 1$; product has degree $2N - 2$)
  - Coefficient-level reduction: modulo the modulus $q$ (bitwise logical AND)
Ring Multiplication \( r(x) = u(x) \ast v(x) \)

- **Implementation options**
  - Operand scanning, product scanning \( O(N^2) \)
  - Karatsuba \( O(N^{\log_2 3}) \), Toom-Cook \( O(N^{1.46}) \)

- **Our approach**
  - Based on product-form polynomial \( O(N^{1.5}) \)
  - Was first proposed in [HS01]

Product-Form Multiplication

• Product-form polynomials
  - \( v(x) = v_1(x) \times v_2(x) + v_3(x) \)
  - \( v_1(x), v_2(x) \) and \( v_3(x) \) can be sparse (i.e. have few non-0 coefficients) since coefficients cross-multiply
  - Non-0 indices stored instead of coefficients for each \( v_i(x) \)

• Ring Multiplication
  - \( r(x) = u(x) \times v(x) = u(x) \times v_1(x) \times v_2(x) + u(x) \times v_3(x) \)
  - Consists of three sparse multiplications \( u(x) \times v_i(x) \)
Outline of Our Ring Multiplication

Ring Multiplication

\[ r(x) = u(x) \ast v(x) \]

\[ r(x) = u(x) \ast \nu_1(x) \ast \nu_2(x) + u(x) \ast \nu_3(x) \]

Product-form Multiplication

Sparse Multiplications

\[ u(x) \ast \nu_1(x) \quad u'(x) \ast \nu_2(x) \quad u(x) \ast \nu_3(x) \]

\[ u'(x) = u(x) \ast \nu_1(x) \]
Sparse multiplication $w(x) = u(x) \ast v_i(x)$

- $v_i(x)$ is sparse ternary polynomial, i.e. each coefficient is in $\{-1, 0, 1\}$
- Contains only the addition and subtraction of coefficients (each addition or subtraction instruction takes 1 clock cycle; while each multiplication instruction takes 2 clock cycles on AVR)
- Execution time depends on the number of non-0 coefficients of $v_i(x)$
“The use of product-form parameter sets was originally intended to provide improved performance by allowing a specialized multiplication algorithm that used knowledge of the indices of the non-zero coefficients […]. However, this index-based multiplication proves to be very hard to implement in a constant-time fashion without losing the speed benefits, so in this paper we concentrate on other approaches of multiplication.”

- [DWZ18] states sparse multiplication is hard to implement in constant-time without losing the speed benefits.
- The straightforward implementation of sparse multiplication is vulnerable to timing attacks.

Towards Timing-Attack Resistance

• **Sources of timing leakage**
  - Calculation of indices (i.e. pointer arithmetic) for accessing the coefficients $u_i$ of polynomial $u(x)$
  - Data-dependent RAM accesses (cache hits/misses)

• **Constant-time implementation**
  - No cache in AVR Microcontrollers
  - Remove conditional statements (e.g. if-else branches)
Outline of Our Ring Multiplication

Ring Multiplication

\[ r(x) = u(x) * v(x) \]

Product-form Multiplication

\[ r(x) = u(x) * v_1(x) * v_2(x) + u(x) * v_3(x) \]

Sparse Multiplications

\[ u(x) * v_1(x) \quad u'(x) * v_2(x) \quad u(x) * v_3(x) \]

\[ u'(x) = u(x) * v_1(x) \]
Sparse Multiplication (Product Scanning)

• Each coefficient addition/subtraction $z += u[k], z -= u[k]$ (incl. required load and store instructions) costs **10 clock cycles**

• Each address correction $\text{index}[j] = k+1-(\text{INTMASK}(k+1\geq N)\&N)$ costs **13 clock cycles**

```c
#define INTMASK(x) "((x) - 1))

void mul_term_sparse(uint16_t *w, const uint16_t *u, const uint16_t *v, int vlen, int N)
{
    int index[vlen], i, j, k;
    register uint16_t z;

    for (i = 0; i < vlen; i ++)
        index[i] = INTMASK(v[i] != 0) & (N - v[i]);

    for (i = 0; i < N; i++) {
        z = w[i];
        for (j = 0; j < vlen/2; j++) {
            k = index[j];
            z += u[k];
            index[j] = k + 1 - (INTMASK(k + 1 >= N) & N);
        }
        for (j = vlen/2; j < vlen; j++) {
            k = index[j];
            z -= u[k];
            index[j] = k + 1 - (INTMASK(k + 1 >= N) & N);
        }
        v[i] = z;
    }
}
Sparse Multiplication (Product Scanning)

• Each coefficient addition/subtraction $z += u[k]$, $z -= u[k]$ (incl. required load and store instructions) costs **10 clock cycles**

• Each address correction $\text{index}[j] = k + 1 - (\text{INTMASK}(k+1>=N) \& N)$ costs **13 clock cycles**

• Our idea: reduce **address corrections**!

```c
#define INTMASK(x) (~(x - 1))

void mul_term_sparse(uint16_t *w, const uint16_t *u, const uint16_t *v, int vlen, int N)
{
  int index[vlen], i, j, k;
  register uint16_t z;

  for (i = 0; i < vlen; i++)
    index[i] = INTMASK(v[i] != 0) & (N - v[i]);

  for (i = 0; i < N; i++)
  {
    z = w[i];
    for (j = 0; j < vlen/2; j++)
    {
      k = index[j];
      z += u[k];
      index[j] = k + 1 - (INTMASK(k + 1 >= N) & N);
    }
    for (j = vlen/2; j < vlen; j++)
    {
      k = index[j];
      z -= u[k];
      index[j] = k + 1 - (INTMASK(k + 1 >= N) & N);
    }
    w[i] = z;
  }
}
Our Sparse Multiplication (Hybrid Method)

- Hybrid multiplication method from CHES 2004 [GPW+04]

- Perform 8 coefficient additions or subtractions in each iteration (of inner loops)

```c
#include <stdint.h>

#define INTMASK(x) (~(x - 1))

void mul_term_sparse(uint16_t *v, const uint16_t *u, const
                    uint16_t *w, int vlen, int N)
{
    int index[vlen], i, j, k;
    register uint16_t w0, w1, w2, w3, w4, w5, w6, w7;

    for (i = 0; i < vlen; i++)
        w0 = w[i]; w1 = w[i+1]; w2 = w[i+2]; w3 = w[i+3];
        w4 = w[i+4]; w5 = w[i+5]; w6 = w[i+6]; w7 = w[i+7];

    for (i = 0; i < N; i += 8) {
        for (j = 0; j < vlen/2; j++) {
            k = index[j];
            w0 = u[k]; w1 = u[k+1]; w2 = u[k+2]; w3 = u[k+3];
            w4 = u[k+4]; w5 = u[k+5]; w6 = u[k+6]; w7 = u[k+7];
            index[j] = k + 8 - (INTMASK(k + 8 >= N) & N);
        }
        for (j = vlen/2; j < vlen; j++) {
            k = index[j];
            w0 = u[k]; w1 = u[k+1]; w2 = u[k+2]; w3 = u[k+3];
            w4 = u[k+4]; w5 = u[k+5]; w6 = u[k+6]; w7 = u[k+7];
            index[j] = k + 8 - (INTMASK(k + 8 >= N) & N);
        }
        w[i] = w0; w[i+1] = w1; w[i+2] = w2; w[i+3] = w3;
        w[i+4] = w4; w[i+5] = w5; w[i+6] = w6; w[i+7] = w7;
    }
}
```

Outline of Our Ring Multiplication

Ring Multiplication

\[ r(x) = u(x) \times v(x) \]

Product-form Multiplication

\[ r(x) = u(x) \times v_1(x) \times v_2(x) + u(x) \times v_3(x) \]

Sparse Multiplications

- \[ u(x) \times v_1(x) \] (Hybrid Method)
- \[ u'(x) \times v_2(x) \] (Hybrid Method)
- \[ u(x) \times v_3(x) \] (Hybrid Method)

\[ u'(x) = u(x) \times v_1(x) \]
Auxiliary Functions

• Performance depends on SHA-256
  - Index Generation Function (IGF)
  - Blinding Polynomial Generation Method (BPGM)
  - Mask Generation Function (MGF)

• Optimization for SHA-256
  - Adopt the techniques in [CDG19]

Timings on 8-bit ATmega1281 (clock cycles)

<table>
<thead>
<tr>
<th>Operation</th>
<th>EES443EP1 (128-bit)</th>
<th>EES743EP1 (256-bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ring Multiplication</td>
<td>192,577</td>
<td>519,746</td>
</tr>
<tr>
<td>Encryption</td>
<td>847,973</td>
<td>1,550,538</td>
</tr>
<tr>
<td>Decryption</td>
<td>1,051,871</td>
<td>2,080,078</td>
</tr>
</tbody>
</table>

- For comparison, optimized multi-level Karatsuba ring mul 1.1 M clock cycles
- Our ring mul 5.7x faster, only 22.7%-33.5% of total encryption time
- Auxiliary functions (SHA-256) dominate execution time
- Code size: 8.9 kB (incl. two parameter sets)
- RAM footprint: 2.9 kB (128-bit Enc) – 6.4 kB (256-bit Dec)
## Comparison

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Algorithm</th>
<th>Security</th>
<th>Platform</th>
<th>Encryption</th>
<th>Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>This work</td>
<td>NTRU</td>
<td>128-bit</td>
<td>ATmega1281</td>
<td>847,973</td>
<td>1,051,871</td>
</tr>
<tr>
<td>This work</td>
<td>NTRU</td>
<td>256-bit</td>
<td>ATmega1281</td>
<td>1,550,538</td>
<td>2,080,078</td>
</tr>
<tr>
<td>[BBJ15]</td>
<td>NTRU</td>
<td>128-bit</td>
<td>ATmega64</td>
<td>1,390,713</td>
<td>2,008,678</td>
</tr>
<tr>
<td>[GPB+17]</td>
<td>NTRU</td>
<td>128-bit</td>
<td>Cortex-M0</td>
<td>588,044</td>
<td>950,371</td>
</tr>
<tr>
<td>[GPB+17]</td>
<td>NTRU</td>
<td>256-bit</td>
<td>Cortex-M0</td>
<td>1,411,557</td>
<td>2,377,054</td>
</tr>
</tbody>
</table>

- 1.6x faster compared to the state of the art on AVR
- A bit slower than ARM Cortex-M0 implementations
### Comparison

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Algorithm</th>
<th>Security</th>
<th>Platform</th>
<th>Encryption</th>
<th>Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>This work</td>
<td>NTRU</td>
<td>128-bit</td>
<td>ATmega1281</td>
<td>847,973</td>
<td>1,051,871</td>
</tr>
<tr>
<td>This work</td>
<td>NTRU</td>
<td>256-bit</td>
<td>ATmega1281</td>
<td>1,550,538</td>
<td>2,080,078</td>
</tr>
<tr>
<td>[GPW+04]</td>
<td>RSA</td>
<td>80-bit</td>
<td>ATmega128</td>
<td>3,440,000</td>
<td>87,920,000</td>
</tr>
<tr>
<td>[LPO+17]</td>
<td>RLWE</td>
<td>128-bit</td>
<td>ATxmega128</td>
<td>796,872</td>
<td>215,031</td>
</tr>
<tr>
<td>[LPO+17]</td>
<td>RLWE</td>
<td>256-bit</td>
<td>ATxmega128</td>
<td>1,975,806</td>
<td>553,536</td>
</tr>
</tbody>
</table>

- Outperforms the scalar multiplication on Curve25519 over an order of magnitude
- AVRNRTRU is faster than RLWE when only ring arithmetic is considered
Concluding Remarks

- Product-form parameters are useful in practice
- A new speed record for the arithmetic part of a lattice-based cryptosystem on an 8-bit device
- AVRNTRU achieves fastest execution time of all known NTRUEncrypt software implementations for AVR
- AVRNTRU is well suited for deployment on resource-limited devices in the post-quantum era
Thank you for your attention !