DESIGN, AUTOMATION & TEST IN EUROPE
01 – 05 February 2021 · virtual conference
The European Event for Electronic System Design & Test

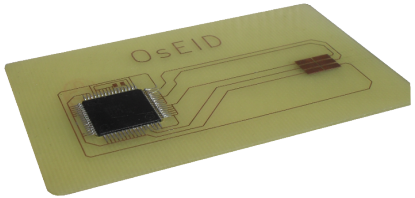
AVRNTRU: Lightweight NTRU-Based Post-Quantum Cryptography for 8-bit AVR Microcontrollers

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

University of Luxembourg

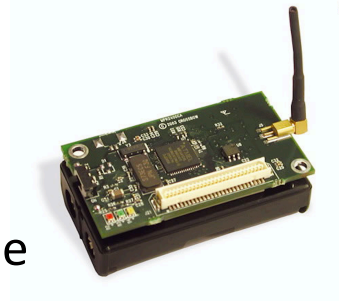
8-bit AVR Devices

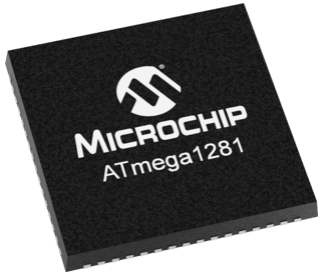
- **Memsic Iris Sensor Node**
  - 8-bit ATmega1281 microcontroller
  - 8 kB RAM, 128 kB flash memory



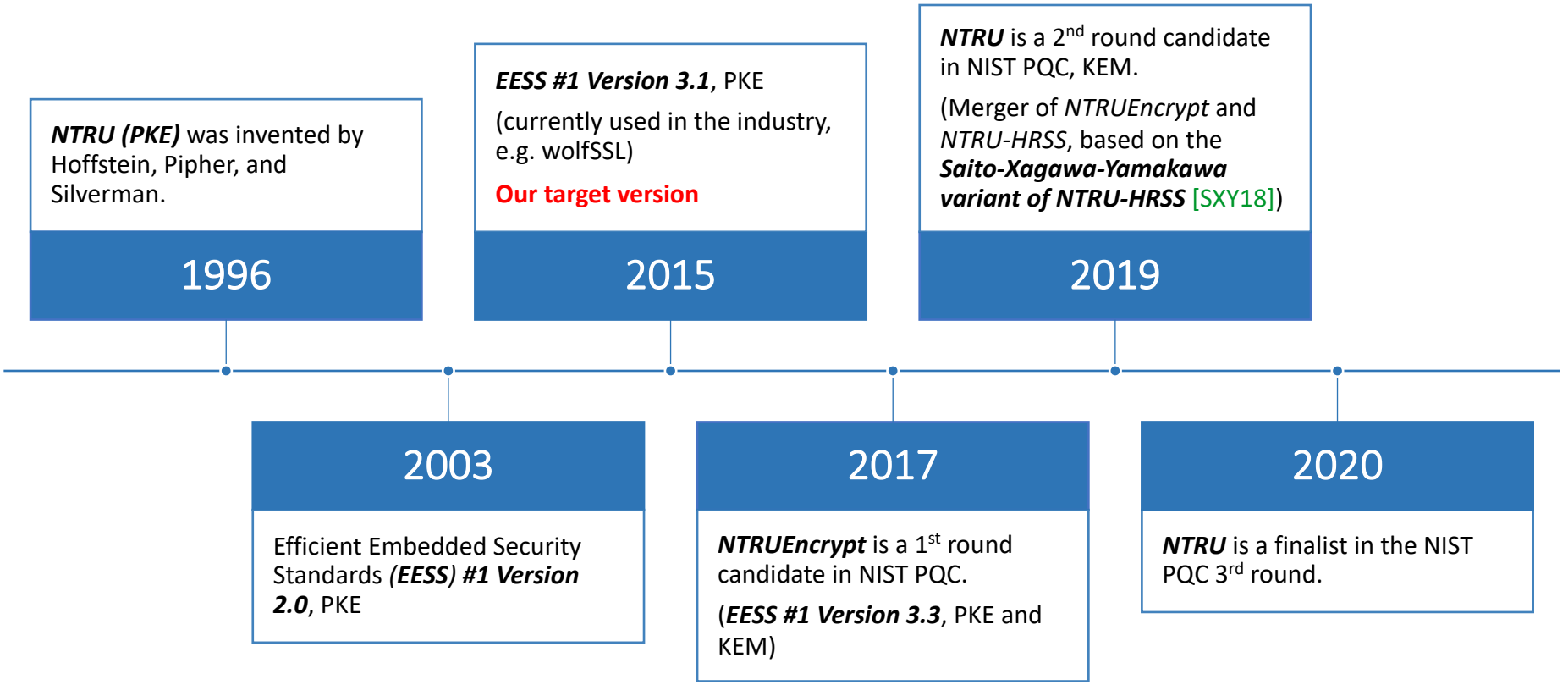
Atmel®

- **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



[SXY18] T. Saito, K. Xagawa, and T. Yamakawa, "Tightly-Secure Key-Encapsulation Mechanism in the Quantum Random Oracle Model.", in *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2018.

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 $\boldsymbol{r(x) = u(x) * v(x)}$)**
  - Product-form polynomials
  - "Hybrid" multiplication method from CHES 2004 [GPW+04]
  - Record-setting execution time

Ring Multiplication $r(x) = u(x) * v(x)$

- **Underlying ring of NTRUEncrypt**
  - Truncated polynomial ring $R = Z_q[x]/(x^N - 1)$
  - 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 $r(x) = u(x) * v(x)$

- **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]

[HS01] J. Hoffstein and J. H. Silverman, “Optimizations for NTRU”, in *Public-Key Cryptography and Computational Number Theory*, 2001.

Product-Form Multiplication

- **Product-form polynomials**
  - $v(x) = v_1(x) * v_2(x) + v_3(x)$
  - $v_1(x), v_2(x)$ 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**
  - $r(x) = u(x) * v(x) = u(x) * v_1(x) * v_2(x) + u(x) * v_3(x)$
  - Consists of three sparse multiplications $u(x) * v_i(x)$

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 $u(x) * v_1(x)$ $u'(x) * v_2(x)$ $u(x) * v_3(x)$

$u'(x) = u(x) * v_1(x)$

Product-Form Multiplication

- **Sparse multiplicaton $\boldsymbol{w(x) = u(x) * v_i(x)}$**
  - $v_i(x)$ is sparse ternary polyomial, i.e. each coefficient is in $\{-1, 0, 1\}$
  - Contains only the addition and subtraction of coeffcients (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 coeffcients of $v_i(x)$

Problem: Timing Attacks

> "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.

[DWZ18] W. Dai, W. Whyte, and Z. Zhang, "Optimizing Polynomial Convolution for NTRUEncrypt", in *IEEE Transactions on Computers*, 2018.

Towards Timing-Attack Resistance

- **Sources of timing leakage**
  - Calculation of indices (i.e. pointer arithmetic) for accessing the coeffcients 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 $u(x) * v_1(x)$ $u'(x) * v_2(x)$ $u(x) * v_3(x)$

$u'(x) = u(x) * v_1(x)$

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 index[j] = k+1-(INTMASK(k+1>=N)&N) costs 13 clock cycles

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

void mul_tern_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;
  }
}
```

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 index[j] = k+1-(INTMASK(k+1>=N)&N) costs 13 clock cycles
- Our idea: **reduce address corrections!**

```
1  #define INTMASK(x) (~((x) - 1))
2
3  void mul_tern_sparse(uint16_t *w, const uint16_t *u, const
       uint16_t *v, int vlen, int N)
4  {
5    int index[vlen], i, j, k;
6    register uint16_t z;
7
8    for (i = 0; i < vlen; i ++)
9      index[i] = INTMASK(v[i] != 0) & (N - v[i]);
10
11   for (i = 0; i < N; i++) {
12     z = w[i];
13     for (j = 0; j < vlen/2; j ++) {
14       k = index[j];
15       z += u[k];
16       index[j] = k + 1 - (INTMASK(k + 1 >= N) & N);
17     }
18     for (j = vlen/2; j < vlen; j ++) {
19       k = index[j];
20       z -= u[k];
21       index[j] = k + 1 - (INTMASK(k + 1 >= N) & N);
22     }
23     w[i] = z;
24   }
25 }
```

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)

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

void mul_tern_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 w0, w1, w2, w3, w4, w5, w6, w7;

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

  for (i = 0; i < N; i += 8) {
    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 (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;
  }
}
```

[GPW+04] N. Gura, A. Patel, A. S. Wander, H. Eberle, and S. Chang Shantz, “Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs”, in *Cryptographic Hardware and Embedded Systems (CHES)*, 2004.

Outline of Our Ring Multiplication

Ring Multiplication $r(x) = u(x) * v(x)$

Product-form Multiplication $r(x) = u(x) * v_1(x) * v_2(x) + u(x) * v_3(x)$

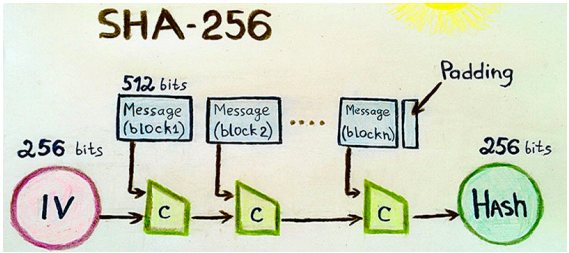
Sparse Multiplications

- $u(x) * v_1(x)$ — Hybrid Method
- $u'(x) * v_2(x)$ — Hybrid Method
- $u(x) * v_3(x)$ — Hybrid Method

$u'(x) = u(x) * v_1(x)$

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]



[CDG19] H. Cheng, D. Dinu, and Johann Großschädl, “Efficient Implementation of the SHA-512 Hash Function for 8-bit AVR Microcontrollers”, in *Innovative Security Solutions for Information Technology and Communications*, 2019.

Timings on 8-bit ATmega1281 (clock cycles)

Operation	EES443EP1(128-bit)	EES743EP1(256-bit)
Ring Multiplication	192,577	519,746
Encryption	847,973	1,550,538
Decryption	1,051,871	2,080,078

- 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

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

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

Comparison

Implementation	Algorithm	Security	Platform	Encryption	Decryption
This work	NTRU	128-bit	ATmega1281	847,973	1,051,871
This work	NTRU	256-bit	ATmega1281	1,550,538	2,080,078
[GPW+04]	RSA	80-bit	ATmega128	3,440,000	87,920,000
[DHH+15]	ECC	128-bit	ATmega2560	13,900,397	13,900,397
[LPO+17]	RLWE	128-bit	ATxmega128	796,872	215,031
[LPO+17]	RLWE	256-bit	ATxmega128	1,975,806	553,536

- Outperforms the scalar multiplication on Curve25519 over an order of magnitude
- AVRNTRU 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 !