hgi Lehrstuhl für Embedded Security

Ultra-Compact Reconfigurable NTRUEncrypt Public Key Cryptosystem Core

Elif Bilge Kavun, M.Sc.

Tolga Yalcin, Ph.D.

June 17, 2011

Outline

hgi Lehrstuhl für Embedded Security

- Why Compact NTRUEncrypt Core?
- NTRU Public Key Cryptosystem
- Lightweight NTRU PKC
- NTRUEncrypt Operation
- NTRUEncrypt Data Flow
- NTRU PKC Data Flow (Overall Encryption)
- NTRU PKC Decryption Data Flow
- NTRU Block Diagram
- Performance and Comparison

Why Compact NTRUEncrypt Core?

- Security of lightweight devices, i.e. RFID tags, smart cards, NFC, Internet of things
- A lightweight cryptographic core should
  - Be **compact**: 1K-3K GE,
  - Consume **low-power**: 5-10 uW @ 100 KHz,
  - Provide **64-80 bit** security,
  - Be **flexible** via reconfiguration,
- Lattice-based NTRU public key cryptosystem can be an alternative to other public key cryptosystems (such as RSA and ECC)

NTRU Public Key Cryptosystem

- Based on shortest vector problem (SVP).
- Parameterized by three integers (N, p, q), where N is prime, gcd(p, q) = 1 and $p<<q$.
- Moderate security: N=167, p=3, q=128.
- Encryption: $e(x) \equiv pr(x) * h(x) + m(x) \bmod q$
- Decryption:

$$a(x) \equiv f(x) * e(x) \equiv pr(x) * g(x) + f(x) * m(x) \bmod q$$

$$b(x) \equiv a \bmod p \equiv f(x) * m(x)$$

$$c(x) \equiv b(x) * f_p(x) \bmod p \equiv m(x)$$

- h is the public key and (f,f_p) pair is the private key.

Lightweight NTRU PKC (1)

- Message and key already stored inside the memory (RAM, flash, OTP, etc).
- Required throughput already low (w.r.t. block ciphers and hash functions).
- Can we exploit these two facts and turn them into an advantage?
- Answer is YES:
  - Just use existing memory (with additional temporary memory space and a few operational registers).
  - Minimize core area (just computational logic, and word based at that)!

Lightweight NTRU PKC (2)

- Main impact on the addressing scheme:
  - Several pointers utilized to determine the start points of different coefficients (blinding value, public key, and private keys) and data (plaintext, ciphertext, temporary data).
- In the current design, pointers are given as constants:
  - Possible to store the pointers in registers (or even in the RAM) allowing the NTRU core to work with various parameter sets (reconfigurable security levels).
- Optimization of the number of multiplications:
  - Zero coefficient multiplications are detected by the control logic, and skipped (less memory accesses → improved throughput).

NTRUEncrypt Operation (1)

- Operation explained by means of a (well) known encryption example, where $N = 11$:

$$e = r \times h + m = (r_0 h_0 + r_1 h_{10} + \ldots + r_9 h_2 + r_{10} h_1 + m_0)$$
$$+ (r_0 h_1 + .. + r_{10} h_2 + m_1) x$$
$$+ \ \ldots$$
$$+ (r_0 h_{10} + r_1 h_9 + .. + r_{10} h_0 + m_{10}) x^{10}$$

- h, r, m given as:

$$h = 8 + 25x + 22x^2 + 20x^3 + 12x^4 + 24x^5 + 15x^6 + 19x^7 + 12x^8 + 19x^9 + 16x^{10}$$
$$r = -1 + x^2 + x^3 + x^4 - x^5 - x^7$$
$$m = -1 + x^3 - x^4 - x^8 + x^9 + x^{10}.$$

NTRUEncrypt Operation (2)

In summary, blinding value, *r*, can be represented as:

r = -1,0,1,1,1,-1,0,-1,0,0,0

r = N,Z,N,N,N,N,Z,N,Z,Z,Z

r = 11 , 00 , 01 , 01 , 01 , 11 , 00 , 11 , 00 , 00 , 00

And stored inside the 8-bit wide memory as:

r = *r00,r01,r02,r03, r10,r11,r12,r13, r20,r21,r22(,r23)*

r = *r0 , r1 , r2*

Each of *hi* stored inside one memory location:

h = h0 , h1 , h2 , h3 , h4 , h5 , h6 , h7 , h8 , h9 , h10

NTRUEncrypt Operation (3)

- Then how do we calculate *e0*, e.g. *e0*

  $e0 = r00 \ h0 + r01 \ h10 + \ldots + r21 \ h2 + r22 \ h1 + m0$

- Normally, we need to:
  - Read all *r*'s (11 words), all *h*'s (11 words), *m0* (1 word),
  - Write *e0* (1 word)

  → 24 memory accesses in total (assuming SP memories)
- However:
  - All *r*'s are stored in 3 words → 3 memory accesses for *r*'s.
  - Only, 6 of r's are non-zero → 6 memory accesses for *h*'s.
  - Only 11 memory accesses in total → 2.18 times the regular throughput.

NTRUEncrypt Data Flow (1)

- *Encryption starts: Initializations*
  - coef_ptr ← r_ptr
  - data_ptr ← h_ptr
  - read_ptr ← ptxt_ptr
  - write_ptr ← ctxt_ptr
- *Read r0*
  - ram_adr = coef_ptr, ram_ren = 1
  - coef_ptr ← coef_ptr + 1
- *Backup r01,r02,r03 in coef_reg, r00 = 0 → Read h0*
  - ram_adr = data_ptr, ram_ren = 1
  - data_ptr ← data_ptr + 1

NTRUEncrypt Data Flow (2)

- *Accumulate r00 h0, r01=0 → r02í 0 → Read h9*
  - ram_adr = data_ptr ï 1, ram_ren = 1
  - data_ptr ← data_ptr ï 2
  - acc ← *r00 h0 = -h0*
- *Accumulate r02 h9, r03í 0 → Read h8*
  - ram_adr = data_ptr, ram_ren = 1
  - data_ptr ← data_ptr ï 1
  - acc ← acc + *r02 h9 = -h0+h9*
- *Accumulate r03 h8, Read r1*
  - ram_adr = coef_ptr, ram_ren = 1
  - coef_ptr ← coef_ptr + 1
  - acc ← acc + *r03 h8 = -h0+h9+h8*

NTRUEncrypt Data Flow (3)

- *Backup r11,r12,r13 in coef_reg, r10 = 0 → Read h7*
  - ram_adr = data_ptr, ram_ren = 1
  - data_ptr ← data_ptr − 1
- *Accumulate r10 × h7, r11 = 0 → Read h6*
  - ram_adr = data_ptr, ram_ren = 1
  - data_ptr ← data_ptr − 1
  - acc ← acc + *r10 × h7 = -h0+h9+h8+h7*
- *Accumulate r11 × h6, r12=0 → r13 = 0 → Read h4*
  - ram_adr = data_ptr − 1, ram_ren = 1
  - data_ptr ← data_ptr − 2
  - acc ← acc + *r11 × h6 = -h0+h9+h8+h7-h6*

NTRUEncrypt Data Flow (4)

- *Accumulate r13 ⋆ h4, Read r2*
  - ram_adr = coef_ptr, ram_ren = 1
  - coef_ptr ← coef_ptr + 1
  - acc ← acc + *r13 ⋆ h4 = -h0+h9+h8+h7-h6-h4*
- *Backup r21,r22,r23 in coef_reg, r20=0 → r21=0 → r22=0 → r23=0 → Read m0*
  - ram_adr = read_ptr, ram_ren = 1
  - read_ptr ← read_ptr + 1
- *Accumulate m0, Write e0*
  - ram_adr = write_ptr, ram_wen = 1, ram_inp = acc + *m0*
  - write_ptr ← write_ptr + 1
- *Continue with e1!!!*

Initialization:	state ← 0000000 (idle state)
Start:	coef_pointer ← coef_init (blinding value for encryption) data_pointer ← data_init (public key for encryption) read_pointer ← plaintext_pointer write_pointer ← ciphertext_pointer msg_counter ← 1 state ← 0000001 (coef read state)
state = 0000001:	read_adr = coef_ptr (RAM read address) coef_pointer ← coef_pointer + 1 state ← 0000010 (first data read state)
state = 0000010:	**if coef_pointer[0] Í 0:** read_adr = data_ptr data_pointer ← data_pointer - 1 state ← 0000100 (second data read state) **else if coef_pointer[1] Í 0: (act like state = 0000100)** read_adr = data_ptr - 1 data_pointer ← data_pointer - 2 state ← 0001000 (third data read state) **else if coef_pointer[2] Í 0: (act like state = 0001000)** read_adr = data_ptr - 2 data_pointer ← data_pointer - 3 state ← 0010000 (fourth data read state) **else if coef_pointer[3] Í 0: (act like state = 0010000)** read_adr = data_ptr - 3 data_pointer ← data_pointer - 4 if msg_counter == N state ← 0100000 (plaintext read state) else state ← 0000001 (coef read state) **else if msg_counter == N: (act like state = 0100000)** read_adr = read_pointer data_pointer ← data_pointer - 4 state ← 1000000 (ciphertext write state) **else: (act like state = 0000001)** read_adr = coef_pointer coef_pointer ← coef_pointer + 1 data_pointer ← data_pointer - 4 state ← 0000010 (read first data state)

hgi Lehrstuhl für Embedded Security

NTRU PKC Data Flow (Overall Encryption) (1)

```
state = 0000100:   if  cf_pnt[1] Í 0:
                       rd_adr = dt_ptr
                       dt_pnt ← dt_pnt - 1
                       state ← 0001000  (third data read state)
                   else if  cf_pnt[2] Í 0:  (act like state = 0001000)
                       rd_adr = dt_ptr - 1
                       dt_pnt ← dt_pnt - 2
                       state ← 0010000  (fourth data read state)
                   else if  cf_pnt[3] Í 0:  (act like state = 0010000)
                       rd_adr = dt_ptr - 2
                       dt_pnt ← dt_pnt - 3
                       if  msg_cnt == N        state ← 0100000  (plaintext read state)
                       else                    state ← 0000001  (coef read state)
                   else if  msg_cnt == N:  (act like state = 0100000)
                       rd_adr = rd_pnt
                       dt_pnt ← dt_pnt - 3
                       state ← 1000000  (ciphertext write state)
                   else:  (act like state = 0000001)
                       rd_adr = cf_pnt
                       cf_pnt ← cf_pnt + 1
                       dt_pnt ← dt_pnt - 3
                       state ← 0000010  (read first data state)
```

```
state = 0001000:   if  cf_pnt[2] Í 1:
                       rd_adr = dt_ptr
                       dt_pnt ← dt_pnt - 1
                       state ← 0010000  (fourth data read state)
                   else if  cf_pnt[3] Í 0:  (act like state = 0010000)
                       rd_adr = dt_ptr - 1
                       dt_pnt ← dt_pnt - 2
                       if  msg_cnt == N        state ← 0100000  (plaintext read state)
                       else                    state ← 0000001  (coef read state)
                   else if  msg_cnt == N:  (act like state = 0100000)
                       rd_adr = rd_pnt
                       dt_pnt ← dt_pnt - 2
                       state ← 1000000  (ciphertext write state)
                   else:  (act like state = 0000001)
                       rd_adr = cf_pnt
                       cf_pnt ← cf_pnt + 1
                       dt_pnt ← dt_pnt - 2
                       state ← 0000010  (read first data state)
```

hgi Lehrstuhl für Embedded Security

NTRU PKC Data Flow (Overall Encryption) (2)

Lehrstuhl für Embedded Security

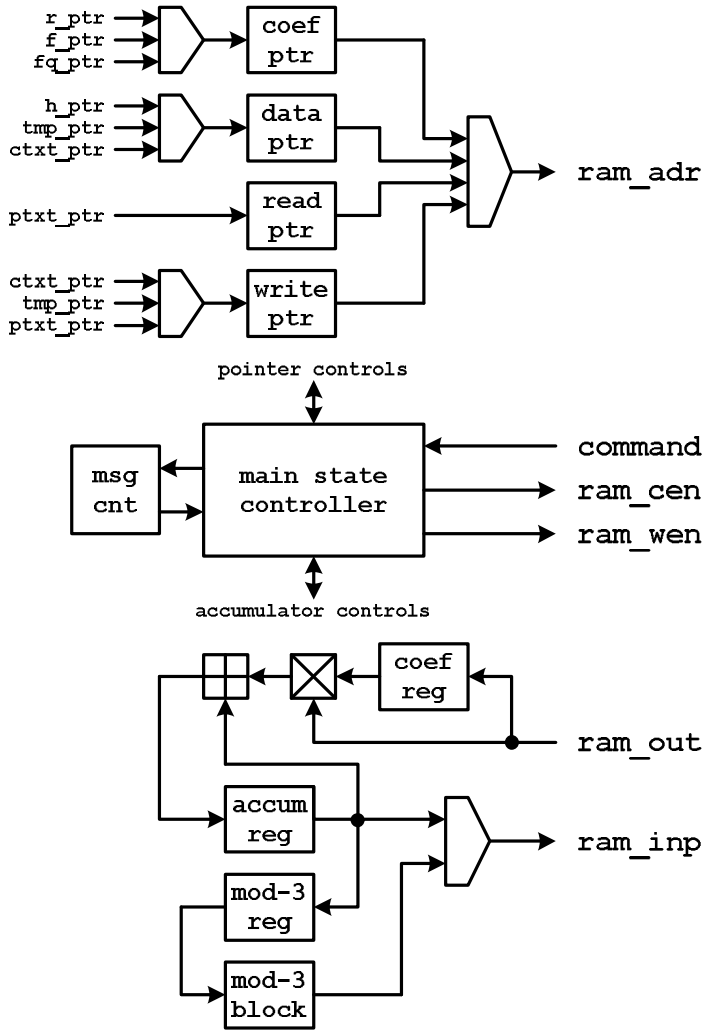
NTRU PKC Data Flow (Overall Encryption) (3)

state = 0010000:	**if cf_pnt[3] Í 0:** rd_adr = dt_ptr dt_pnt ← dt_pnt - 1 if msg_cnt == N state ← 0100000 (plaintext read state) else state ← 0000001 (coef read state) **else if msg_cnt == N: (act like state = 0100000)** rd_adr = rd_pnt dt_pnt ← dt_pnt - 1 state ← 1000000 (ciphertext write state) **else: (act like state = 0000001)** rd_adr = cf_pnt cf_pnt ← cf_pnt + 1 dt_pnt ← dt_pnt - 1 state ← 0000010 (read first data state)
state = 0100000:	rd_adr = rd_pnt state ← 1000000 (ciphertext write state)
state = 1000000:	wr_ptr = wr_pnt cf_pnt ← coef_init dt_pnt ← dt_pnt + 2 rd_pnt ← rd_pnt + 1 wr_pnt ← wr_pnt + 1 msg_cnt ← msg_cnt + 1 if msg_cnt == N state ← 0000000 (idle state) else state ← 0000001 (coef read state)

NTRU PKC Decryption Data Flow

- Decryption performed using the same idea, but multiplication is done in two phases.
- An additional mod-3 register is used.
- Except this difference and initialization of address pointers, all the operations performed during encryption and decryption are the same.
- To simplify the circuit and minimize gate count, the same address change and coefficient read logic are used (with different pointers) for encryption and the both phases of decryption.

NTRU Block Diagram



Performance

N=167

Å # of cycles (enc)

$$N \times (\lceil 2N / W \rceil + 2 \times L_r + 2) \quad \Rightarrow \text{ 13,360 cycles}$$

Å # of cycles (dec)

$$2 \times N \times (\lceil 2N / W \rceil + 2 \times L_f + 2) \quad \Rightarrow \text{ 54,776 cycles}$$

Performance

Å Implemented on a Xilinx-Virtex5 VLX20T

Number of Slices	204
Max. Speed (MHz)	136.5
Number of LUTS	457
BRAM	1

Performance

Å a Xilinx Spartan3-S50

Number of Slices	307
Max. Speed (MHz)	57.2
Number of LUTS	553
BRAM	1

Performance

Å an Actel APA075

Number of Tiles	1157
Max. Speed (MHz)	15.1
BRAM	4

Performance

- Implemented on 0.13u Faraday low-leakage CMOS std-cell library for minimum area

Gate Count (KGE)	1.4
Max. Speed (MHz)	254
SPRAM	1 KB
Power (uW/MHz)	1.6

Comparison

Atici et al.[9]

Å Implemented on 0.13u Faraday low-leakage library

	Number of encryption cycles	Number of decryption cycles	Area (KGE)	$P_{tot.}$ (enc) @ 500KHz (µW)	$P_{tot.}$ (dec) @ 500KHz (µW)
[9]	28,390	59,619	10,500	5.98	6.11
Ours (exc. RAM)	**13,360**	**< 54,776**	**1,400**	**< 1.00**	
Ours (inc. RAM)	**13,360**	**< 54,776**	**8,900**	**< 4.90**	
Ours (1KB RAM) (N=251)	**13,360**	**< 54,776**	**10,500**	**< 6.90**	

Thanks for listeningé

Questions?