Performance Evaluation of the NTRU Encryption Scheme on an 8-bit AVR Micro-controller

December 28, 2018

Authors:
Adriano Franci
Thibault Simonetto

Academic Supervisor:
Prof. Dr. Jean-Sébastien Coron

Project Supervisor:
Dr. Johann Groszschädl

1 Introduction

Today's cryptography for modern communication systems are mostly based on asymmetric cryptography algorithm, known as public-key cryptography.
One of the widely used public-key cryptography algorithm is RSA[3]. It is based on the computational hardness assumption of integer factorisation. However, in the early 2000s, it has been proven that such systems are vulnerable to quantum computer. Thus a new mathematical model should be found that could be proved quantum resistant.
The NTRU crypto-system[5] (described in section 2) is such an alternative based on the closest lattice vector problem.
Asymmetric cryptography is also an important part of the concept of Internet of Things (IoT). In fact the challenge for such hardware is the possibility for secure communication while keeping the cost of the hardware low. Symmetric cryptography is already widely used and proven to be performant and cost efficient for such small controller and limited hardware. But symmetric cryptography do provide a low security scheme.
The NTRU public-key crypto-system main characteristics are the low memory and computational requirements while keeping a high security level in communication thanks to the asymmetric scheme of encryption. This makes the NTRU crypto-system a good candidates for being implemented on IoT hardware (e.g. 8 bit AVR controller).

In this work we will go trough the different steps of implementation and performance measurement of the NTRUencrypt algorithm on an 8 bit AVR controller. Specially for the mask generating utility function used to provide of a variable length primitive output. And if such implementation can approximate the performance of a standard 64 bit computer.

2 Background

This sections presents the concepts, methodologies and technologies used to produce the implementation and the performance evaluation.

2.1 NTRU

NTRU algorithm which is the abbreviation for $N-th$ degree truncated polynomial ring, is the group of lattice based public key encryption schemes.
Such algorithm relies on the hardness of the *Closest Vector Problem*[4] which in turns is a generalisation of the *Shortest Vector Problem*.

2.1.1 Algebraic structure and operation

NTRU crypto-system operation is using polynomial of degree $N-1$ with Integer coefficient. Elements are on the form

$$F = f_0 + f_1X + f_2X^2 + ... + f_{N-1}X^{N-1}$$

We will denote by R the set of all those polynomials.
Let $a, b \in R$

$$a + b = (a_0 + b_0) + (a_1 + b_1)X + ... + (a_{N-1} + b_{N-1})X^{N-1}$$

$$a \cdot b = c_0 + c_1X + c_2X^2 + ... + c_{N-1}X^{N-1},$$

We have that $+, \cdot$ are defined in R.
Thus R defines a ring , such ring is called *Truncated polynomial Ring* and is isomorphic to the quotient ring $\mathbb{Z}[X]/(X^N - 1)$.
An important operation that is used by `NTRUEncrypt` during encryption and decryption is the modular arithmetic of those polynomials.
Such operation is defined as the reduction of the coefficient of a polynomial a by an integer q.
Another notion that will be used in the `NTRUEncrypt` operation is the inverse modulo q of an element in the truncated polynomial ring.
This element denoted by a_q is defined by :

$$a \cdot a_q = 1 \text{ mod q}$$

To be used, the NTRU crypto-system should be set up with a set of parameters (N, p, q) which defines the underlying algebraic structure. With N the *degree parameter* , p and q two integers with $q >> p$ and gcd(p,q)=1.
Currently a value $N = 439$ allows to ensure 128 bits of security [1]. The degree parameter N will determine the quotient ring $R = \mathbb{Z}[X]/(X^N - 1)$.

2.1.2 Public key creation

A person that is willing to create a private/public key pair for use with `NTRUEncrypt` will follow the following procedure.

1. Randomly choose 2 polynomials $f, g \in R$.
2. Compute the inverse f_q of f (modulo q) and f_p of f (modulo p) [1]. If the inverses don't exists then go to step 1.

[1] The inverse exists such that $f_q * f = 1 \text{ mod } q$

3. Compute $h = f_q * g \bmod q$. [2]

4. The output h is the public key and f, f_p are the private parameters.

2.1.3 Encryption

The encryption takes the parameters set (N, q, p) the message M and the public key h as input. As for other public key encryption scheme, `NTRUEncrypt` requires additional padding and formatting to avoid certain attack and break determinism of such algorithm. (Such utility function will be emphasised latter).
To encrypt a message M a person will have to go through the following steps:

1. Translate M into a polynomial $m \in R$. [3]

2. Generate a random blinding polynomial $r \in R$.

3. Compute $c = ph * r + m \bmod q$

4. The output c is the cipher text

2.1.4 Decryption

The decryption will be executed by the public owner, hence the parameters (N, q, p), the private parameters f, f_p and finally the cipher text c are the input.
To recover the original message M the person will have to go through the following steps :

1. Compute $a = f * c \bmod q$

2. Compute $b = a \bmod p$. (Reduce each coefficient of $a \bmod p$).

3. Compute $m = fp * b \bmod p$.

4. Output m is the decrypted polynomial representing the message M.

This group of algorithm has couple of advantages over the well known RSA algorithm. First, the main operation of NTRU relies on the multiplication of polynomials with constant degree (*e.g.* 438 for 128 bit security) on small integer. Which in turns is much less costly that operation for the same level of security using RSA algorithm that relies on modular exponentiation (for 128 bit security level, operation performed on 3072-bit integers). Second, NTRU is more robust with regards to quantum computing than RSA.
Furthermore RSA operation are very costly in terms of resource consumption which make it very unlikely to be implemented on resource limited hardware. Thanks to the possibility of efficiency increase of the NTRU algorithm, an implementation of such high security scheme on small and resource critical devices can be considered.

2.2 Mask Generating Function

Our work will be focused on one of the high resource consuming part of the execution of NTRU algorithm which is the padding performed during encryption. Such padding is accomplished using a mask generating function (MGF). Such function is similar to a hash function. The difference is that hash function produce fixed length output while the MGF can produce variable length output.

[2] The operation * refers to the multiplication in R and is defined by the cyclic convolution product.
[3] Please note that the resulting m represent the padded and formatted message M

2.3 IoT & Performance Evaluation

One of the main motivation for the optimisation of high-security level crypto-system is the need of such high security for on low powered small device that have spread out on our networks for the past 10 years. In fact those Internet of Things (IoT) devices already started to take a important part of information gathering as well as processing.
Those devices take the form of very small controller often equipped with low calculation power coupled with limited battery resource.
Recent news have shown that the security in communication on those devices was very low and sometimes not even considered. Because of the difficulty of combining correct security level with limitation on hardware that such device impose.
Thus, progress in the area of efficient implementation of NTRU algorithm for low-power consumption could be a major advancement on the security of IoT devices.
A good performance testing scheme could be a first estimation of the power consumption (which will be approximated with clock cycle) of the optimised time consuming operation (*e.g.* mask generating function) over existing RSA high resource consuming operation. Such benchmark will give us material to argue on the possibility of optimisation of power consumption as well as enhancement of security level for such device.

3 Implementation & Testing

This section presents the motivations to implement the MGF-1 function, its implementation and the testing of the implementation.

As described in the background section, the part that we have worked on concerns the utility function of mask generating function. Again this function is used to apply some padding during the encryption to improve security.

Function	Total	Polynomial arithmetics		Hash functions		Rest	
		k cycles	Percentage	k cycles	Percentage	k cycles	Percentage
Key genration	5424	5082	93.6%	247	4.5%	95	1.7%
Encryption	1008	780	77.4%	160	15.9%	68	6.7%
Decryption	1757	1560	88.8%	104	5.9%	93	5.3%

Table 1: A Cost Breakdown [2] of Reference Code of NTRUEncrypt [1]

The table 1 described the clock cycles needed for different part of the algorithm depending on the operation type used (polynomial arithmetics, hash function, others).
It clearly emphasis the fact that the hash function takes up to 15.9% of the total computation cycle number. It is the second most expensive operation after the polynomial arithmetics (that takes up to 77.4% of the total cycle number). Plus, the rest is negligible as it takes only up to 6.7%.
Hence the polynomial arithmetics and the hash function $MGF-1$ are good candidates to be optimised. We decided to work on the mask generating function.
After having pointed out which function was the most consuming one [4], we could start with the implementation and optimisation of the $MGF-1$ function in C according to the description of J. Schanck [5].

Algorithm 2 MGF-TP-1

Input: $seed \in \{0,1\}^*, minCallsMask, N$

1: $Z = \mathsf{Hash}(seed)$
2: $buf = \mathsf{Hash}(Z|0) \mid \mathsf{Hash}(Z|1) \mid \ldots \mid \mathsf{Hash}(Z|minCallsMask)$
3: $\boldsymbol{v} = 0 \in \mathbb{Z}[x]$
4: $i = 0$
5: **for each** octet O in buf **do**
6: **if** $O < 3^5$ **then**
7: Choose $(c_0, \ldots, c_4) \in \{0,1,2\}^5$ such that $\sum_{j=0}^{4} c_j \cdot 3^j = O$
8: Let $c'_j \in \{-1,0,1\}$ be the minimal integer representative of $c_j \bmod 3$.
9: $\boldsymbol{v} = \boldsymbol{v} + c'_0\boldsymbol{x}^i + \cdots + c'_4\boldsymbol{x}^{i+4}$
10: $i = i + 5$
11: **end if**
12: **end for**

Output: $\boldsymbol{v} \bmod \boldsymbol{x}^N$ {Truncate $\boldsymbol{v}$ to its first N coefficients}

Figure 1: Algorithm MGF-TP-1 [5]

The Figure 1 shows the description of the algorithm MGF-TP-1 provided by J. Schanck [5]. It describes the algorithm from a pure mathematical point of view.

[4] Remember that the goal is to implement such algorithm on a small controller, hence power consumption is critical and should be optimised.

The first part of the algorithm (line 1 to 4) calls a hash-function multiple times with different inputs and concatenates the outputs in a $buffer$ to increase the size of the input *seed*.

The second part (line 5 to 12), is used to find a polynomial representation of degree 5 for each bytes of the $buffer$ if the byte is bigger than 3^5. Else, the byte is dropped to guarantee a uniform distribution of the output. In the description, the polynomial is then formatted to be added to another buffer called v.

In the last part, the buffer v is truncated to the N coefficients.

The goal is to implement a version of the algorithm that will run on an 8-bit AVR Microcontroller. The chosen programming language is C as the code is portable and easier to write compare to assembly. Moreover, C allows the programmer to manipulate the memory directly and in efficient way. The performance are typically better with C than other programming languages that require virtual machines like Java. The challenge is to translate this mathematical description of the algorithm into an optimised running version of the algorithm written in C.

As the implementation follows the mathematical description, only the optimisation are described in this section. The first optimisation is to calculate only the necessary polynomial representations instead of calculating all of them and truncate the result to the first ones. This reduces the computation time, as less computations are necessary. It also reduces the memory usage as it does not store all the polynomial representations.
In practice, we don't calculate the polynomial representation of every bytes in the $buffer$ bigger than 3^5, but only for the N first bytes in the $buffer$ bigger than 3^5. This implies not calculating v and not performing the modulo operation on v.
The second optimisation has been to remove the formatting of line 8. In fact, this formatting is only necessary to compute a correct v , which is not required according to the first optimisation. Moreover, certain memory slot are reused to store different variables to limit the memory usage with the help of pointers in C. Finally, the compiled assembly code has been optimised by Dr. Groszschädl.

This function is implemented following a Test-driven development methodology. Thanks to the already existing implementation of NTRUOpenSourceProject [1], it has been possible to compute the output with the predefined inputs and therefore to create test cases.
The NTRUOpenSourceProject provides a sample execution of the complete NTRU algorithm. However, the NTRUOpenSourceProject does not provide the MGF-1 algorithm as a single C function which makes it harder to determine what correspond to the MGF-1 algorithm, its input and output.
Using the C debugger on the NTRUOpenSourceProject implementation, it is possible to retrieve the inputs and output of a correct execution of MGF-1 to create the following test case.

```
{Test case for MGF-1 implementation}

INPUT
seed = {0xd0, 0x4c, 0x38, 0xca, 0x20, 0xa1, 0xb5, 0x5f,
        0x3e, 0x95, 0x47, 0x1f, 0x2b, 0xb1, 0xc0, 0x6e,
        0x70, 0xd0, 0xf0, 0x97, 0x52, 0xf4, 0x60, 0x3b,
        0xf5, 0x52, 0xc3, 0x8b, 0x76, 0x7a, 0x32, 0x62,
        0x55, 0x3c, 0xa5, 0xf4, 0x35, 0x06, 0xec, 0xcc,
        0x7d, 0x52, 0x84, 0xe3, 0x3d, 0x24, 0x6e, 0x90,
        0x09, 0xe9, 0xe3, 0xaa, 0x68, 0x71, 0x00, 0x5b,
        0xc8, 0x53, 0x02, 0x80, 0x78, 0xe3, 0x86, 0x2c,
        0xed, 0x91, 0x63, 0x56, 0x5a, 0x3d, 0xc5, 0x7d,
        0xd5, 0x91, 0xaf, 0x9b, 0x96, 0x3c, 0xc7, 0xb5,
        0x6d, 0x5f, 0xe6, 0x6d, 0x8a, 0x06, 0xf5, 0xee,
        0xc6, 0xed, 0xc2, 0x3f, 0x59, 0xb2, 0x07, 0x5f,
        0xb5, 0xb9, 0x60, 0xc7, 0x80}
mincallsMask = 6
N = 20

EXPECTED OUTPUT
mask = {0x00, 0x00, 0x00, 0x01, 0x02, 0x01, 0x00, 0x00,
        0x02, 0x02, 0x01, 0x01, 0x02, 0x02, 0x00, 0x00,
        0x01, 0x01, 0x00, 0x01}
```

This test case has been used to determine whether our implementation is correct or not. The implementation and the C version of the test case can be found on the repository of the project [6].

4 Performance evaluation

5 Conclusion

References

[1] *Open Source NTRU Public Key Cryptography and Reference Code: NTRUOpenSourceProject/ntru-crypto*. NTRU Open Source Project, December 2018. URL: `https://github.com/NTRUOpenSourceProject/ntru-crypto`.

[2] W. Dai, W. Whyte, and Z. Zhang. Optimizing Polynomial Convolution for NTRUEncrypt. *IEEE Transactions on Computers*, 67(11):1572–1583, November 2018.

[3] Jakob Jonsson, Kathleen Moriarty, Burt Kaliski, and Andreas Rusch. PKCS #1: RSA Cryptography Specifications Version 2.2. URL: `https://tools.ietf.org/html/rfc8017`.

[4] Daniele Micciancio. Closest Vector Problem. August 2003. URL: `http://cseweb.ucsd.edu/~daniele/papers/CryptoEncyclopediaCVP.pdf`.

[5] John Schanck. Practical Lattice Cryptosystems: NTRUEncrypt and NTRUMLS. December 2015. URL: `https://uwspace.uwaterloo.ca/handle/10012/10074`.

[6] Thibault Simonetto and Adriano Franci. NTRU implementation for 8-bit ARV Microcontroller, December 2018. original-date: 2018-11-29T15:20:25Z. URL: `https://github.com/thibaultsmnt/lu.uni.mics.ntru`.