NTRU in Constrained Devices

Daniel V. Bailey[1,2], Daniel Coffin[2], Adam Elbirt[2,4], Joseph H. Silverman[3,2], and Adam D. Woodbury[4,2]

[1] Computer Science Department, Brown University
[2] NTRU Cryptosystems, Inc.
[3] Mathematics Department, Brown University
[4] Electrical and Computer Engineering Department, Worcester Polytechnic Institute

Abstract. The growing connectivity offered by constrained computing devices signals a critical need for public-key cryptography in such environments. By their nature, however, public-key systems have been difficult to implement in systems with limited computational power. The NTRU public-key cryptosystem addresses this problem by offering better computational performance than previous practical systems. The efficiency of NTRU is applied to a wide variety of constrained devices in this paper, including the Palm Computing Platform, Advanced RISC Machines ARM7TDMI, the Research in Motion pager, and finally, the Xilinx Virtex 1000 family of FPGAs. On each of these platforms, NTRU offers exceptional performance, enabling a new range of applications to make use of the power of public-key cryptography.

1 Motivation

Since their introduction in the 1970s, the development of microprocessors and public-key cryptosystems has been intervolved. Ever faster, cheaper, better microprocessors have allowed the use of public-key cryptosystems in a dizzying array of applications.

One of the more popular of these is the use of desktop personal computers to mediate the purchase of goods and services on the Internet. For years now, desktop computers have offered adequate performance to make the arduous calculations involved in traditional public-key cryptosystems invisible to the casual user. This performance has resulted in the ubiquitous deployment of crypto-enabled web browsers such as Microsoft's Internet Explorer on desktop PCs.

Conversely, the need for public-key cryptography has led microprocessor vendors to add functionality to their products. Desktop eCommerce led Intel to add random-number generation and unique IDs to the Pentium/III processor. The need for secure authentication in GSM cellular telephone applications has resulted in 8-bit microcontrollers with custom hardware to accelerate modular exponentiation.

The number of embedded systems that require cryptography is about to explode. Just as the ubiquitous PC networking made possible by TCP/IP led to public-key crypto libraries in desktop web browsers, wireless networking is

Ç.K. Koç, D. Naccache, and C. Paar (Eds.): CHES 2001, LNCS 2162, pp. 262–272, 2001.

set to offer universal connectivity to a diverse array of computing devices. From washing machines to cell phones, televisions to automobiles, wireless networking standards such as Bluetooth and IEEE 802.11b will bring networking to devices that previously stood alone. As we've seen on the desktop, with communications comes the need for security, and so for public-key cryptosystems.

In contrast to their desktop-bound, powerful brethren, these embedded devices offer severely constrained computing capacity. Power, memory, and CPU cycles all must be judiciously conserved.

The computational efficiency of NTRU allows implementors to build efficient wirelessly-communicating embedded systems. Furthermore, algorithmic improvements introduced in [4] augment the original construction to allow for greater computational savings.

In this paper, we apply these results in the context of embedded systems. We report on fast NTRU implementations for the Palm Computing Platform, the Research in Motion pager, the Advanced RISC Machines ARM7TDMI, and finally field-programmable gate arrays (FPGAs).

2 The NTRU Public-Key Cryptosystem

NTRU is a public-key cryptosystem based on the Shortest Vector Problem in a lattice. Lattices find application in pure and applied mathematics, computer science, physics, and cryptography. In particular, the SVP has been intensively studied for more than one hundred years for its use in these and other areas of mathematics and science. Theory and experimentation [2] suggest the SVP is difficult in lattices of very high dimension. Such instances of the SVP form the basis of NTRU.

2.1 Basic Setup

NTRU is best described using the ring of polynomials

$$R = Z[X]/(X^N - 1).$$

These are polynomials with integer coefficients

$$a(X) = a_0 + a_1X + a_2X^2 + \cdots + a_{N-1}X^{N-1}$$

that are multiplied together using the extra rule $X^N \equiv 1$. So the product

$$c(X) = a(X) * b(X)$$

is given by

$$c_k = a_0b_k + a_1b_{k-1} + \cdots + a_{N-1}b_{k+1} = \Sigma_{i+j \equiv k \bmod N}\, a_ib_j.$$

In particular, if we write $a(X), b(X)$, and $c(X)$ as vectors

$$a = [a_0, a_1, \cdots, a_{N-1}], \quad b = [b_0, b_1, \cdots, b_{N-1}], \quad c = [c_0, c_1, \cdots, c_{N-1}],$$

then $c = a * b$ is the usual discrete convolution product of two vectors.

To quickly sum up the other relevant basic properties of NTRU:

1. NTRU uses three public parameters (N, p, q) with $\gcd(p, q) = 1$.
2. Typical parameter sets that yield security levels similar to 1024-bit RSA and 4096-bit RSA respectively are $(N, p, q) = (251, 3, 128)$ and $(N, p, q) = (503, 3, 256)$.
3. Coefficients of polynomials are reduced modulo p and/or modulo q.
4. The inverse of $a(X) \bmod q$ is the polynomial $A(X) \in R$ satisfying $a(X) * A(X) \equiv 1 \bmod q$.

The inverse (if it exists) is easily computed using the Extended Euclidean Algorithm. Inverses are only needed for key generation.

2.2 Key Generation

Choose random polynomials $F, g \in R$ with small coefficients and set $f = 1 + pF$. Compute the polynomial

$$h \equiv g * f^{-1} \bmod q.$$

The public key is h and the private key is f.

2.3 Encryption

The plaintext m is a polynomial with coefficients taken $\bmod p$. Choose a random polynomial r with small coefficients. The ciphertext is

$$e \equiv pr * h + m \bmod q.$$

2.4 Decryption

Compute

$$a \equiv e * f \bmod q,$$

choosing the coefficients of a to satisfy $A \leq a_i < A + q$. The value of A is fixed and is determined by a simple formula depending on the other parameters. Then $a \bmod p$ is equal to the plaintext m.

2.5 Why NTRU Works

The decryption process yields the polynomial

$$\begin{aligned} a &\equiv e * f \bmod q \\ &\equiv (pr * h + m) * f \bmod q && \text{(since } e \equiv pr * h + m\text{)} \\ &\equiv pr * g + m * f \bmod q && \text{(since } h * f \equiv gf - 1 * f \equiv g\text{)} \end{aligned}$$

The coefficients of r, g, m, and f are small, so the coefficients of

$$pr * g + m * f$$

will lie in an interval of length less than q. Choosing the appropriate interval, we recover

$$a = pr * g + m * f = pr * g + m * (1 + pF)$$

exactly, not merely modulo q. Then reduction modulo p yields $a \equiv m \bmod p$.

3 NTRU Algorithmic Optimizations

3.1 Choice of p

If the coefficients of the polynomial $pr*g+m*f$ do not lie in an interval of length at most q, then decryption will not work. Appropriate choices of parameters reduce this to a very low probability, which may be reduced even further by the following observation.

The discussion above assumes that p is an integer, where we recall that p and q must be relatively prime. However, as noted in [4], there is no particular reason that p must be an integer. It could instead be a polynomial, provided the ideals generated by p and q are relatively prime in the ring R. Our first choice for such a polynomial would naturally be a binomial $X^k \pm 1$. Unfortunately, the elements

$$X^k \pm 1 \text{ and } X^N - 1 \text{ and } 128$$

are not relatively prime in $Z[X]$.

The next natural candidate is $p = X + 2$. It is simple to verify the relative primality of p and q in this case:

$$X^N - 1 = X^N + 2^N - 2^N - 1 = (X^N + 2^N) - 128 \cdot 2^{N-7} - 1.$$

As noted in [1] and [4], operations modulo binomials are efficiently computed. Thus $p = X + 2$ and $q = 128$ form the basis for a very efficient implemention of NTRU.

3.2 Polynomials of Low Hamming Weight

The most time consuming part of NTRU encryption is computation of the product $r(X) * h(X) \bmod q$. Similarly, the most time consuming part of NTRU decryption is computation of the product $f(X) * e(X) \bmod q$. The polynomials $h(X)$ and $e(X)$ have coefficients that are more or less randomly distributed modulo q, while one normally takes $r(X)$ and $f(X)$ to have binary (i.e., 0 or 1) or ternary (i.e., -1, 0, or 1) coefficients.

Suppose that $r(X)$ is a binary polynomial with d ones. Then computation of the product $r(X) * h(X) \bmod q$ requires approximately dN operations, where one operation is an addition and a remainder modulo q.

A common trick (see [7] for instance) is to choose a polynomial of low Hamming weight. We extend this idea by taking a product of low Hamming weight polynomials as suggested in [5]. To this end, we write $r(X) = r_1(X)r_2(X)$, where r_1 and r_2 are binary polynomials with d_1 and d_2 ones respectively. Then $r(X)$ will have approximately $d_1 d_2$ ones, a few twos, and rarely a three. Rather than computing $r(X) * h(X) \bmod q$ as $(r_1 * r_2) * h$, it is far more efficient to compute it as

$$r(X) * h(X) = r_1(X) * \big(r_2(X) * h(X)\big),$$

which requires only $(d_1 + d_2)N$ operations. Thus the computational complexity is proportional to the sum of d_1 and d_2.

On the other hand, the search space for the pair of polynomials (r_1, r_2) has size approximately $\binom{N-1}{d_1-1}\binom{N-1}{d_2-1}$, so is proportional to the product of the r_1 search space and the r_2 search space. In practice, there are meet-in-the-middle approaches that reduce the size of the search space, see [5] for details and security considerations. Further, the number of nonzero coefficients in $r_1(X)r_2(X)$ is essentially the product d_1d_2. Thus one might say that using a product $r = r_1r_2$ requires computation proportional to the sum $d_1 + d_2$ while giving security proportional to the product d_1d_2. In rough terms, this explains why one obtains significant performance gains without changing the level of security. In the common case of $N = 251$ and $q = 128$, it is common to set $r = r_1 * r_2 + r_3$, where each of r_1, r_2, r_3 is binary with 8 nonzero coefficients.

Given the above, multiplication involving the private key $f(X)$ is aided by writing

$$f(X) = 1 + p * (f_1(X) * f_2(X) + f_3(X)).$$

3.3 A Fast Convolution Algorithm in Software

Under the assumption that the coefficients of f_1, f_2, f_3 are binary, we thus have an efficient algorithm for ring multiplication in software. The central idea is that rather than storing f or the individual f_i polynomials as N-element arrays in memory, it suffices to store those array offsets whose locations correspond to a nonzero entry. Thus, a polynomial $f_i(X) = X^{191} + X^{178} + \cdots + X^{14} + X^2$ would be stored in memory as the array $191, 178, \cdots, 14, 2$. For convenience, arrays representing the $f_i(X)$ polynomials are concatenated into a single array which we denote b.

Recall that the coefficients c_k of the product of $b(X)$ and some general polynomial $a(X)$ have the form

$$c_k = a_0b_k + a_1b_{k-1} + \cdots + a_{N-1}b_{k+1} = \sum_{i+j \equiv k \mod N} a_ib_j.$$

The sparse nature of $f_i(X)$ causes most of these inner product terms to be zero. So rather than employing a traditional polynomial multiplication algorithm that expends a great deal of effort computing zero terms, we take a different approach. Scanning the b array allows us to calculate only those inner product terms which may be non-zero. A particular non-zero coefficient will appear in N inner product terms.

The algorithm begins by zero-initializing an array of coefficients that will hold the result $c(X) = f_i(X)a(X)$. For each entry of the b array we calculate the N inner product terms corresponding to a non-zero coefficient in $f_i(X)$. Since $f_i(X)$ is binary, each non-zero inner product term is simply a coefficient of $a(X)$. These terms are individually accumulated in their corresponding location in the c array. Repeating this process for all non-zero coefficients calculates $f_i(X)a(X)$ at a cost of d_iN additions of $\log_2(q)$-bit numbers.

With this procedure in hand, we may compute the overall $f(X)a(X)$ multiplication with the following steps:

1. $t(X) \leftarrow a(X)f_1(X)$
2. $c(X) \leftarrow t(X) * f_2(X) = a(X) * f_1(X) * f_2(X)$
3. $t(X) \leftarrow f_3(X) * a(X)$
4. $c(X) \leftarrow c_k + t_k \bmod N = f_3(X) * a(X) + f_1(X) * f_2(X) * a(X)$

Thus in the common practical case of $N = 251$ and $q = 128$ with each of f_1, f_2, f_3 having eight nonzero coefficients, the convolution is computed with $251 \times 8 \times 3 = 6024$ seven-bit additions and no multiplications. This algorithm is thus ideally suited for the low-power, low-clockrate, narrow arithmetic architectures found in constrained devices.

Pseudocode for this operation is found as Algorithm 1, where all array offsets are to be taken modulo N for clarity in exposition.

Algorithm 1. Fast Convolution Algorithm

Require: b an array of $d_1 + d_2 + d_3$ nonzero coefficient locations representing the polynomial $f(X) = 1 + p * (f_1(X) * f_2(X) + f_3(X))$, a the array $a(X) = \Sigma a_i$, N the number of coefficients in $f(X), a(X)$.
Ensure: c the array where $c(X) = f(X)a(X)$
  for $0 \leq j < d_1$ **do** {Compute $t(X) \leftarrow a(X) * f_1(X)$}
    for $0 \leq k \leq N - 1$ **do**
      $t_{k+b_j} \leftarrow t_{k+b_j} + a_k$
    end for
  end for
  for $d_1 \leq j < d_2$ **do** {Compute $c(X) \leftarrow t(X) * f_2(X)$}
    for $0 \leq k \leq N - 1$ **do**
      $c_{k+b_j} \leftarrow c_{k+b_j} + t_k$
    end for
  end for
  for $0 \leq k \leq N$ **do** {Zero out t}
    $t_k \leftarrow 0$
  end for
  for $d_2 \leq j < d_3$ **do** {Compute $t(X) \leftarrow f_3(X) * a(X)$}
    for $0 \leq k \leq N - 1$ **do**
      $t_{k+b_j} \leftarrow t_{k+b_j} + a_k$
    end for
  end for
  for $0 \leq k \leq N - 1$ **do** {$c(X) \leftarrow f_3(X) * a(X) + f_1(X) * f_2(X) * a(X)$}
    $c_k \leftarrow c_k + t_k \bmod q$
  end for

For sake of comparison, we implemented Karatsuba-Ofman polynomial multiplication and Algorithm 1 on a variety of embedded systems. These results are found in Table 1.

Table 1. Polynomial Multiplication Algorithm Comparison

Operation	MC68EX328 Dragonball (20 MHz Palm Vx)	Intel 80386 (20 MHz RIM 957)	37MHz ARM7
Karatsuba	25 msec	178 msec	12.75 msec
Algorithm 1	3.2 msec	28 msec	1.62 msec

4 NTRU Embedded Reference Implementation

The NTRU Embedded Reference Implementation package is designed for use in applications where both high performance and small footprint are important considerations. The package contains the NTRU algorithm [3], the NTRU Signature Scheme (NSS) [6], a random number generation utility, and public domain versions of the AES selected Rijndael symmetric cipher and the SHA-1 hash function.

The software library is implemented in ANSI C and is easily ported while maintaining high performance. Two important design choices include the use of an internal memory management scheme and support for 8/16/32/64-bit environments.

Internal Memory Management Scheme. Memory allocation on constrained devices is typically a source of inefficiency and portability problems. While some devices disallow the use of heap management functions (such as malloc, realloc, and free), others significantly restrict the use of stack space. Regardless of which operations are available, there is normally significant CPU overhead associated with native memory management functions. For efficiency and portability, the implementation establishes its own internal memory management scheme. When an application initializes the implementation, a block of memory is created from either the stack or the heap and used to satisfy the application's dynamic memory management needs. Thus, the implementation's memory management is abstracted from the application environment, improving portability, security and performance.

8/16/32/64-bit environments. One of the requirements of the software is to support many different devices. Popular microprocessors have word lengths ranging from 8–64 bits. To provide maximum flexibility, storage of public and private key information as well as intermediate results is generally in arrays of 8-bit types and all operations are 8 bits wide. While this provides a flexible approach supporting operation on different size devices, it may not be the most efficient approach on all devices. For example, on some devices a 16- or 32-bit operation has the same cycle cost as a corresponding 8-bit instruction. This fact can be exploited when tailoring NTRU for a specific platform.

4.1 NTRU C Performance Results

The NTRU design decisions lead to a generic software base that can be run on many different platforms. Outside of good software engineering practices, there

are no platform specific C optimization tricks used in the reference implementation. Even without platform specific optimizations, the performance numbers, as shown in Table 2, are impressive on a variety of popular processors for constrained devices. In these tables, msec is taken to mean milliseconds. In addition, in this and all remaining sections of this paper, we report results for NTRU with parameters $(N, p, q) = (251, X + 2, 128)$.

Table 2. NTRU Performance Results

Operation	MC68EX328 Dragonball (20 MHz Palm Vx)	Intel 80386 (20 MHz RIM 957)	37MHz ARM7
Key Generation	1130 msec	858 msec	80.6 msec
Encryption	47 msec	39 msec	3.25 msec
Decryption	89 msec	72 msec	6.75 msec

4.2 NTRU Optimized for Palm Computing Platforms

The Motorola Dragonball microprocessor is widely used in Palm computing platforms. While the Dragonball supports 8-, 16-, and 32-bit data operations, memory is organized into 16-bit words. Although the NTRU fast convolution algorithm operates on 7-bit polynomial coefficients, each operand fetch actually retrieves a full 16-bit word. Assuming the coefficients are organized in memory along byte boundries, this leads to twice as many memory accesses as should be needed. While an easy choice would be to read the full word and use the two bytes separately, the task of extracting anything but the lowest byte in a register is more expensive than simply fetching the next byte from memory.

Since Algorithm 1 is nothing more than repeated coefficient addition, the arithmetic requirements on the Dragonball are minimal. The result is that most of the time is spent fetching coefficients and storing their sum. A great deal of optimization can be achieved simply by making these memory operations more efficient. Extensive use of the Dragonball's post-increment and pre-decrement pointer operations makes the code much faster than using pointer offsets, the approach taken by the C compiler.

Another performance-limiting factor is Algorithm 1's use of circular array indexing for fast modular reduction. Since the Dragonball has no native support for circular arrays, we can simply place two copies of a in adjacent memory locations and reduce the burden of pointer arithmetic. Figure 1 graphically displays the situation.

By taking the buffering idea one step further, we can exploit the 16-bit architecture of the Dragonball to perform two 7-bit coefficient additions in parallel. To this end, we simply pack two 7-bit coefficients into a word and add. Any overflow from the add operation can be removed by modular reduction via a logical and with $0x7F7F$. This effectively reduces each byte over $q = 128$. The main problem with this scheme is alignment of data on 16-bit boundaries. If the

offset j is odd, then the above scheme with two copies of a will suffice to perform word additions instead of byte additions. If j is even however, none of the words would be aligned to perform the addition. If we make a third copy of a, again adjacent to the other two, we find that if j is odd, $a + (N - j)$ is unaligned, but $a + (2N - j)$ will be. This is shown in figure 2.

The current assembly improvements can be seen in Table 3.

a	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_0	a_1	a_2	a_3	a_4	a_5	a_6
						$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$		
c						c_0	c_1	c_2	c_3	c_4	c_5	c_6		
						$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$		
c'						c'_0	c'_1	c'_2	c'_3	c'_4	c'_5	c'_6		

Fig. 1. Bytewise buffered convolution example; $b_j = 2, N = 7$

a	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_0	a_1	a_2	a_3	a_4	a_5	a_6
													$\oplus$		$\oplus$		$\oplus$		$\oplus$		
c													c_0	c_1	c_2	c_3	c_4	c_5	c_6		
													$\downarrow$		$\downarrow$		$\downarrow$		$\downarrow$		
c'													c'_0	c'_1	c'_2	c'_3	c'_4	c'_5	c'_6		

Fig. 2. Wordwise buffered convolution example; $b_j = 2, N = 7$

Table 3. NTRU Palm Assembly Language Performance Improvements

Operation	Palm C code	Palm Assembly/C code
Key Generation	1130 msec	630 msec
Encryption	47 msec	33 msec
Decryption	89 msec	60 msec

5 NTRU in an FPGA

Due to its low complexity and parallel nature, the NTRU cryptosystem lends itself exremely well to hardware implementation. The primary function in the encryption algorithm is the convolution of the public key $h(X)$, by the random vector, $r(X)$, as described in Algorithm 1. The nature of the construction of r leads to the observation that with overwhelming probability, each coefficient of r is at most 15 (i.e., fits into at most 4 bits), with a limit on the number of non-zero coefficients. This allows the use of repeated coefficient addition as opposed to full coefficient multiplication to implement convolution.

The encryption engine operates in the following steps. First, the operands h, r, and m must be loaded serially, 251 bits at a time. Once the operands

are loaded, the engine begins bit-scanning each r coefficient. For each non-zero coefficient, the engine adds h to the temporary result. This is repeated a number of times corresponding to the value of the current r coefficient. Once this is complete, or if the coefficient is zero, h is rotated left by one coefficent (7 bits) to perform the modular reduction of the result over x^N. The next r coefficient is then scanned, repeating the above process until all r's have been processed. Finally, the engine adds the polynomial m to the result of the convolution and outputs this value as the encrypted message. Because of the expansive nature of encryption, the encrypted message is output serially. Note that h is retained in the encryption engine, and thus successive encryptions only require the loading of r and m, which takes 5 clock cycles.

For the provided implementation, the following tools were used:

- *Synthesis*: Synplicity's Synplify version 6.1.3.
- *Place and Route*: Xilinx's Design Manager version 2.1i_sp6.
- *Simulation*:Viewlogic's Powerview version 6.1 FusionHDL version 1.4 and Viewlogic's Workview Office version 7.53 Speedwave version 6.202.

For the provided implementation, the Xilinx Virtex 1000EFG860 FPGA was chosen as the target device. The package type chosen provides sufficient I/O (656 IOBs) and logic resources to satisfy the design requirements. Further information regarding the Virtex E family may be found in [8]. Note that the VHDL implementation is fully portable to ASIC technology, since no FPGA vendor-specific constructs were used in the provided implementation.

Table 4. FPGA Implementation Results

Encryption Cycles	259
Clock Period	19.975 ns
Clock Frequency	50.063 MHz
Encryption Time	5.174 μs
Encryption Throughput	48.52 Mbps
Slices Used	6373
Logic Resource Utilization	51%
Approximate Gate Count	60,000
Approximate Register Gate Count	40,000
I/O Used	506
I/O Utilization	77%

6 Conclusions

In this paper we have provided practical implementation results for NTRU running on a number of embedded systems including microcontrollers and FPGAs. In addition, we have provided a new fast convolution algorithm which eliminates the need for explicit multiplication in encryption and decryption.

References

1. D. V. Bailey and C. Paar. Efficient arithmetic in finite field extensions with application in elliptic curve cryptography. *Journal of Cryptology*, to appear.
2. D. Coppersmith and A. Shamir. Lattice attacks on NTRU. In *Advances in Cryptography — EUROCRYPT '97*, pages 52–61. Springer-Verlag, 1997. LNCS 1233.
3. J. Hoffstein, J. Pipher, and J. Silverman. NTRU: A new high speed public key cryptosystem. In J. Buhler, editor, *Lecture Notes in Computer Science 1423: Algorithmic Number Theory (ANTS III)*, pages 267–288. Springer-Verlag, Berlin, 1998.
4. J. Hoffstein and J. Silverman. Optimizations for NTRU. In *Proceedings of Public-Key Cryptography and Computational Number Theory*. de Gruyter, Warsaw, September 2000.
5. J. Hoffstein and J. Silverman. Small hamming weight products in cryptography. preprint, September 2000.
6. J. S. J. Hoffstein, J. Pipher. NSS: An NTRU lattice-based signature scheme. In *Advances in Cryptography — EUROCRYPT 2001*. Springer-Verlag, 2001. to appear.
7. A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, Boston, 1993.
8. Xilinx Inc. *Virtex 2.5V Field Programmable Gate Arrays*, 1998.