Towards Post-quantum Security for IoT Endpoints with NTRU

Oscar M. Guillen*, Thomas Pöppelmann†, Jose M. Bermudo Mera*,
Elena Fuentes Bongenaar‡, Georg Sigl*§ and Johanna Sepulveda*
*Technische Universität München, Germany
oscar.guillen@tum.de, jmb.mera@tum.de, sigl@tum.de, johanna.sepulveda@tum.de
†Infineon Technologies AG, Germany, thomas.poeppelmann@infineon.com
‡Radboud University, E.Fuentes.Bongenaar@protonmail.com
§Fraunhofer Institute AISEC, Germany, georg.sigl@aisec.fraunhofer.de

***Abstract*—The NTRU cryptosystem is one of the main alternatives for practical implementations of post-quantum, public-key cryptography. In this work, we analyze the feasibility of employing the NTRU encryption scheme, NTRUEncrypt, in resource constrained devices such as those used for Internet-of-Things endpoints. We present an analysis of NTRUEncrypt's advantages over other cryptosystems for use in such devices. We describe four different NTRUEncrypt implementations on an ARM Cortex M0-based microcontroller, compare their results, and show that NTRUEncrypt is suitable for use in battery-operated devices. We present performance and memory footprint figures for different security parameters, as well as energy consumption in a resource constrained microcontroller to backup these claims. Furthermore, to the best of our knowledge, in this work we present the first time-independent implementation of NTRUEncrypt.**

Keywords*—IoT, post-quantum, security, NTRUEncrypt, embedded devices.

I. Introduction

Sensors and actuator devices, commonly known as endpoints, are one of the most widespread and pervasive types of Internet-of-Things (IoT) devices. The engineering requirements (form factor, long battery life, and deployability) for this type of devices are challenging. Additionally, security also becomes a critical concern, as these devices may collect, store, or use sensitive data for extended periods. This is especially true in an interconnected IoT infrastructure where remote attackers present a great risk.

Public-key cryptography (PKC) is the foundation for establishing secured communication channels between multiple parties. It is a critical technology for highly interconnected networks and is therefore vital for the IoT. However, the fear of quantum computers' arrival is beginning to cause doubt that traditional PKC algorithms such as RSA and ECC can provide security in the long term [1]. Indeed, if capable quantum computers become available, cryptographic systems based on the integer factorization problem and the (elliptic curve) discrete logarithm problem could be attacked in polynomial time due to the work of Shor [2]. In reaction, the National Security Agency (NSA) has announced its intention to transition toward quantum-resistant, meaning post-quantum, cryptography for governmental usage in the foreseeable future [3]. Concurrently, the National Institute of Standards and Technology (NIST) in the United States has already started moving forward with standardization efforts [4]. As a consequence, the choice of a suitable PKC cipher-suite is vital for applications and devices with long life-cycle, especially if they are deployed in the field and hard to update. Among the classes of post-quantum algorithms that exist today, lattice-based cryptography, which is based on computationally hard problems in certain lattices, offers a very good trade-off between security and efficiency. One example of an efficient post-quantum lattice-based cryptosystem is NTRU [5], which has already been around for nearly 20 years. It supports encryption and decryption operations that are considerably faster than RSA. It also requires relatively short keys, when compared with other post-quantum approaches. Computational efficiency, low memory requirements, and the recent push toward post-quantum cryptography and long-term security make NTRUEncrypt an attractive alternative for providing post-quantum security to IoT devices. Although there are patents on the cryptosystem itself as well as on the product-form efficiency optimization [6], they expire in 2017 and 2021 respectively. In addition, NTRUEncrypt is free for non-commercial use when used with software under the GNU GPL. Previous works have dealt with embedded software implementations of NTRUEncrypt [7], [8], [9], [10]. Despite showing good results, such implementations are based on outdated microprocessor/microcontroller architectures or use a set of parameters that is unable to meet current security requirements, especially when considering quantum adversaries.

Our contribution: In this work, we perform a feasibility study of using NTRUEncrypt as a public-key encryption algorithm for IoT devices, using up-to-date parameters for better protection against classical and quantum adversaries. We analyze the influence of different, standardized NTRUEncrypt parameters in terms of runtime and memory footprint, on a modern CPU architecture for constrained IoT devices, namely ARM Cortex-M0. We report the performance and cost results for four NTRUEncrypt implementations: i) the standard NTRUEncrypt, ii) NTRUEncrypt using products of low Hamming-weight polynomials, iii) an *time-independent* implementation, and iv) an optimization using pattern searches. To the best of our knowledge, we are presenting the first time-independent implementation of NTRUEncrypt that prevents leakage of timing information on computations involving secret values.

II. Related Work

For IoT endpoints, high-performance and low-overhead software implementations of cryptographic algorithms are very important, since low-cost microcontrollers (MCUs) that lack crypto accelerators are typically used. There are a couple of works that deal with the implementation of NTRUEncrypt in software for embedded devices. The work of [8] describes the implementation of NTRUEncrypt on an ARM7 32-bit embedded processor. It made use of the Chinese Remainder

978-3-9815370-8-6/17/$31.00

Theorem (CRT) to improve the convolution's performance. However, the results showed no significant improvement over a straightforward implementation. Other optimized NTRUEncrypt implementations have been proposed in [7], [9], and [11]. Firstly, the work of [7] evaluated NTRUEncrypt implementations in three architectures: MC68EX328, Intel 80386 and ARM7. Their work focused on the choice of parameters and the use of polynomials with low Hamming weights for efficient convolution as algorithmic optimizations. Their work also describes the NTRU Embedded Reference Implementation (NERI). A software library, implemented in ANSI C, that could be ported to other platforms without significant performance loss. The work of [11] focused as well on the polynomial convolution. In this case a sliding-window approach was used. This method finds patterns of ones within the binary representation of polynomials in order to create a pre-computation table that can later be reused. The drawback of this work is the selection of the NTRUEncrypt parameters, which cause inefficient reductions modulo q. Lastly, in [9] a memory-efficient version of [7] was presented. Their implementation of NTRUEncrypt stores the ternary polynomials as an array of indexes of non-zero coefficients, instead of an N array coefficient, drastically reducing the memory consumption. This implementation was evaluated in 8-bit resource-constrained MCUs, namely the ATMega128 and ATMega163. Despite showing good results, all of the aforementioned works present at least one of the following drawbacks: i) NTRUEncrypt parameters are already deprecated; ii) Protection against chosen ciphertext attacks (CCA) is not considered; and iii) the implementations are based on outdated microprocessor/microcontroller architectures. Additionally, to the best of our knowledge, there is no previous work that reports an time-independent NTRUEncryt implementation. Our work aims to overcome the discussed drawbacks of the previous works.

III. NTRUEncrypt advantages for IoT

A. Performance

NTRUEncrypt usually exhibits better *performance* than RSA and ECC when compared at similar security levels. Due to the low computational complexity of polynomial convolution, NTRUEncrypt's core operation, it requires only $\mathcal{O}(n^2)$ operations for encrypting or decrypting a message of size n. RSA and ECC, which are based on modular exponentiation (RSA) and repeated squaring and doubling (ECC) operations, would require $\mathcal{O}(n^3)$ operations to encrypt or decrypt the same message [5]. The NTRUEncrypt version using product-form polynomials further reduces encryption to $\mathcal{O}(n \log n)$ operations. Its strong performance makes it an excellent candidate for use in combination with pre-quantum public-key encryption. This is an approach that has recently been used in an experiment announced by Google [12] to limit the risk of transitioning to a less understood post-quantum crypto suite.

B. CCA Security

The standardized version of NTRUEncrypt [13] provides security against chosen ciphertext attacks (CCA), using a padding scheme that limits ciphertext malleability and thus a large range of attacks (e.g., manipulation of encrypted message or extraction of secret keys from a decryption oracle). So far, these transformations have not been implemented for other lattice-based schemes, such as public-key encryption based on Ring-LWE [14], as they present some challenges [15], [16] due to small decryption-error probabilities in common parameter sets [17], [18].

C. Composability with TLS

NTRUEncrypt may be integrated into the Transfer Layer Security (TLS) protocol. The TLS Working Group published an Internet-Draft in 2001 proposing integrating NTRUEncrypt, along with lightweight signature algorithms into a new TLS cipher suite [19]. The primary purpose would be to provide post-quantum cryptography for resource-constrained, embedded devices. Proposals for NTRU-based key-exchange [20] and the integration of NTRUEncrypt into Tor [21] have recently been published.

IV. NTRUEncrypt

NTRUEncrypt is a public-key-based cryptosystem based on the Shortest Vector Problem (SVP), which roughly establishes that given basis vectors of a lattice, it is hard to find a shortest vector in that lattice. The SVP is especially difficult in large dimension lattices. In this section, we describe the mathematical basis of NTRUEncrypt and the three major operations: encryption, decryption, and key-generation.

A. Mathematical background

NTRUEncrypt works in the truncated polynomial ring modular q presented in (1). Where, $Z_q[X]$ is a polynomial with integer coefficients reduced $\bmod q$ and degree of at most $N-1$.

$$R_{N,q} = \frac{Z_q[X]}{X^N - 1} \tag{1}$$

Let $a(x)$ and $b(x)$ be truncated polynomials in R. Where, $a(x)$ is defined as in (2).

$$a(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{N-1} x^{N-1} \tag{2}$$

Addition and multiplication within R are defined as in (3), (4) and (5). The multiplication is defined by the discrete convolution product of two vectors.

Let polynomial multiplication $a(x) * b(x)$ be defined as in (4) and (5). It corresponds to the circular convolution.

$$a(x) + b(x) = (a_0 + b_0) + \ldots + (a_{N-1} + b_{N-1}) x^{N-1} \tag{3}$$

$$a(x) * b(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_{N-1} x^{N-1} \tag{4}$$

where

$$c_k = \sum_{i+j \equiv k \mod N} a_i b_j \tag{5}$$

NTRUEncrypt is defined by three public parameters (N, p, q), where N is a prime number that defines the degree of R, and $gcd(p, q) = 1$, where q should be larger than p. Usually p is set to 3 (ternary polynomials), and q for efficiency reasons, is generally set to an integer number power of two $q = 2^k$. The recommended set of parameters can be found in [13].

B. Key-generation

Key-generation creates the private key f and the public key h. It is composed by three steps. The first step generates the random ternary polynomials $F(x)$ and $g(x)$. To improve the performance of NTRUEncrypt, the private key is defined as

$f = 1 + pF(x)$. This characteristic dispenses with the need of computing $f_p(x)$. The second step computes only $f_q(x)$ as the inverse of the polynomial $f(x) \mod q$. Finally, the third step computes the public key $h(x)$ as in (6).

$$h(x) \equiv p f_q(x) * g(x) \mod q \tag{6}$$

C. Encryption

Encryption transforms any plaintext, m, into a cyphered message, e. To protect against Chosen Ciphertext Attacks (CCA), usually the Shortest Vector Encryption Scheme, third revision (SVES-3) [13] is used, which is a variant of the NTRUEncrypt Asymmetric Encryption Padding (NAEP) scheme [22]. Encryption is then composed of four steps. The first step encodes the message m into a ternary polynomial, $m(x)$. The second step generates a random polynomial $r(x)$ from the ring R based on the Blinding Polynomial Generation Method $(BPGM)$ [13] and $m(x)$. The third step computes a mask by means on the Mask Generation Function (MGF) and $r(x) * h(x)$. By XORing $m(x)$ with the mask, $m'(x)$ is generated. Finally, the ciphertext $e(x)$ is computed as in (7).

$$e(x) \equiv r(x) * h(x) + m'(x) \mod q \tag{7}$$

D. Decryption

Decryption retrieves the original plaintext m from e by means of the private key f. It also verifies that the ciphertext is valid. Four steps are performed to retrieve m. The first step retrieves $m'(x)$ by quantifying the expressions shown in (8) and (9). The second step quantifies the received $r(x) * h(x)$ as in (10). The third step quantifies the mask by using MGF and $r(x) * h(x)$. Finally, $m(x)$ is retrieved as in (11).

$$a(x) = e(x) * f(x) \mod q \tag{8}$$
$$m'(x) = a(x) \mod p \tag{9}$$
$$r(x) * h(x) = e(x) - m'(x) \mod q \tag{10}$$
$$m(x) = m'(x) - MGF(r(x) * h(x)) \mod p \tag{11}$$

To verify the validity of the ciphertext, $BPGM(m)$ is computed and multiplied by h. A valid ciphertext must match the value computed in (10).

V. NTRUEncrypt Implementation

In this section, we present our implementations of NTRUEncrypt using SVES-3 padding [13] on the Infineon XMC1100 ARM Cortex-M0 32-bit microcontroller. The code was written in C and preprocessor directives were used to modify the code's characteristics according to the chosen parameter set. Parameters have to be chosen at compile time, thereby reducing possible overhead from unused code. The implementation supports the three main cryptographic operations: i) key-generation, ii) encryption, and iii) decryption.

We implemented four different versions of NTRUEncrypt, which will be presented in the following subsections. The first two versions are based on the standard [13]. The first version corresponds to the straightforward implementation. The second is the implementation that uses product-form optimization. The third is an time-independent implementation. The fourth version uses a fast polynomial convolution algorithm based on pattern multiplication, thus enhancing NTRUEncrypt's performance. Fig. 1 shows a block diagram with the different functions required to perform NTRUEncrypt's three cryptographic operations. The functions that were modified or included for each of the four implementations are identified in the figure.

A. Standard NTRUEncrypt (Model A)

This version is the implementation of the standard NTRUEncrypt cryptosystem [13], with optimizations which will be detailed in the following. The pseudo-random number generator (PRNG) integrated into the XMC1100 MCU was used to enable fast key-generation. It generates a 16-bit value in only 18 cycles. For our purposes, the random seed used to initialize the PRNG was generated on a PC and sent to the MCU through the serial interface. Note that for a real-world application, the seed could also be generated within the MCU, e.g. dual oscillator architecture [23]. The numbers obtained were analyzed using the NIST Statistical Test Suite [24]. According to this test the PRNG produces statistically independent numbers when a random seed is used. The PRNG is used to generate both random polynomials, $F(x)$ and $g(x)$, described in Section IV-B. Integer polynomials are stored as an array of size N containing all of the coefficient values since ternary polynomials in NTRUEncrypt are sparse and the number of ones and minus ones is less than one third for all parameter sets. To reduce memory overhead, we store the indexes of the coefficients equal to one and minus one respectively, instead of storing the N coefficients. Given that the target microcontroller does not provide a hardware divider, modular operations were implemented as iterative functions based on binary shifts, adds, and masking operations. Computing the inverse of $f \mod q$ is based on the Almost Inverse Algorithm and Newton iteration [25]. The standard version of NTRUEncrypt uses supporting algorithms to maintain CCA security, namely the previously mentioned masking generation function (MGF) and a blinding polynomial generation method (BPGM). Since both the MGF and the BPGM make repetitive calls to an underlying hash function, we employed speed-optimized versions of SHA-1 and SHA-256 where used. The MGF requires a transformation from octets to coefficients. This was done using a lookup table for storing all coefficient values. The optimized hash function and the MGF lookup table implementation trade memory for performance.

B. Product-form multiplication (Model B)

The most time consuming operation is the convolution of two truncated polynomials of degree $N-1$ over an integer ring. This operation is performed in all cryptographic functions. The purpose of this NTRUEncrypt implementation is to speed up computation of the products $r(X) * h(X) \mod q$ (cf. equation 7) and $f(X) * e(X) \mod q$ (cf. equation 8) during the encryption and decryption operations respectively. The coefficients of the polynomials $h(X)$ and $e(X)$ are randomly distributed modulo q, whereas $r(X)$ and $f(x)$ are binary or ternary coefficients. The speed increase is achieved using the product-form polynomial multiplication introduced in [6]. The idea behind this technique is to form a multiplier as a product of factors with low Hamming-weight exponents to reduce the convolution's computational complexity. For this, binary polynomials $F(x)$ and $r(x)$ are replaced by product-form polynomials, which are of the form $f_1 * f_2 + f_3$, where f_1, f_2, and f_3 are sparse polynomials as described in [6]. Fig. 1 shows the modifications required for product-form multiplication. It includes the `mult_prod` function and the modification of

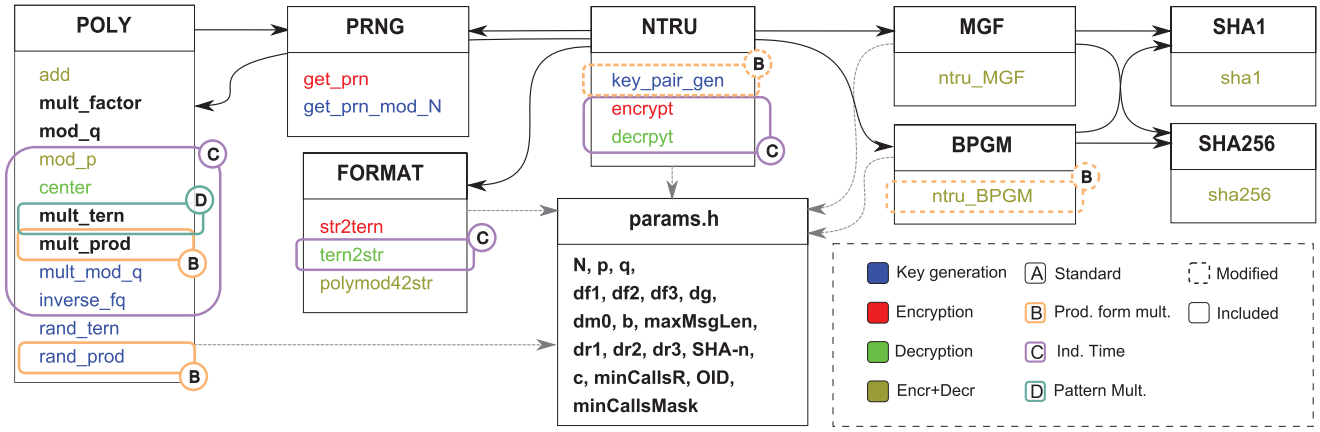


Fig. 1. Block diagram of NTRUEncrypt describing the differences between implementations.

ntru_BPGM. The number of coefficients, used to represent the ternary polynomial, is smaller in the product-form version than it is in the standard version. To employ a single structure type, array lengths equal the maximum number of coefficients that will be used based on the parameter set.

C. Time-independent implementation (Model C)

This implementation modifies the functions that make use of secret values during encryption and decryption to make them time constant. Figure 1 shows the functions that exhibit time dependency with respect to the secret values. The term2str and center functions were protected by executing them until completion. If errors are encountered, they are reported upon completion. The mod_p function was protected by using lookup tables. The remaining functions were rewritten to execute the same number of operations. Please note that this implementation is not time-constant, but rather independent of the cryptographic secrets. Although timing variations can still occur during execution, they are not related to the secrets and thus cannot be exploited to retrieve any secret information. This approach reduces overhead, relative to a completely time-constant implementation, due to the exclusive modification of operations handling secret values.

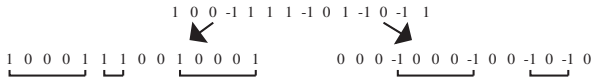
D. Pattern multiplication (Model D)

We investigated binary and ternary multiplication for non-product-form polynomials as alternative to product-form polynomials. Lee et al. [11] proposed and Buchmann et al. [26] improved this approach. The general idea is that a polynomial, in this example $f = x^2 + x^3 + x^4 + x^6 + x^8 + x^{10} + x^{11} + x^{13}$ for $N = 14$, can be represented as vector of bits in which patterns are already visible:

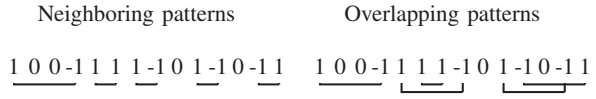
$$0\,0\,\underbrace{1\,1}\,\underbrace{1\,0\,1}\,0\,\underbrace{1\,0\,1}\,\underbrace{1\,0\,1}$$

The pattern polynomials can then be used to compute $f * g$ in the following way as $x^2g + x^3g + x^4(g + x^2g) + x^8(g + x^2g) + x^{11}(g + x^2g)$, where three times only rotations of $g + x^2g$ are added. This is more efficient than using normal convolution multiplication.

We have extended this approach for the ternary use-case. One option is to use unmixed patterns in which a pattern search is first performed only over all positive positions and then over all negative positions as depicted bellow:



Another option is using *mixed patterns*. Here a pattern can have two different nonzero coefficients, which leads to *homogeneous* patterns, 10^*1 or $-10^* - 1$, and new *heterogeneous* patterns of the form $10^* - 1$ and -10^*1. We also explored the creating *overlapping patterns* to obtain more frequent occurrences of a few patterns, rather than single occurrences of many different ones:



The different pattern-multiplication variants have been implemented for polynomial multiplication in encryption and decryption. However, they are also interesting for key-management as the pattern can already be computed and stored along with the secret key.

VI. Experiments and Results

A. Experimental setup

The implementation target is the Infineon XMC 2Go Evaluation Board for the XMC1100, an ARM Cortex-M0 microcontroller family. The XMC1100 provides 64KB of flash memory, 16KB of RAM, and an internal clock running at a maximum of 32MHz. The XMC1100 was programmed in C using the GNU ARM Embedded Toolchain. Code size and memory footprint measurements are obtained from the linker-generated map-file. We use the microcontroller's system timer (SysTick) to measure timing (corroborated using an oscilloscope). The stack's dynamic memory usage was measured by first loading the RAM with a known pattern and, after the function to be profiled was executed, measuring how much memory was overwritten. A script was written in Python on the PC side to automatically compile and load different versions of the implementation and retrieve the measurements through the USB interface.

TABLE I. SECURITY LEVEL FOR DIFFERENT NTRUENCRYPT PARAMETER SETS.

Criteria	Security level (bits)			
	112	128	192	256
Key-size	EES401EP1	EES449EP1	EES677EP1	EES1087EP2⋆
Trade-off	EES541EP1	EES613EP1	EES887EP1⋆	EES1171EP1
Speed	EES659EP1	EES761EP1	EES1087EP1⋆	EES1499EP1
Product-form	EES401EP2	EES443EP1	EES587EP1	EES743EP1

Table I shows the parameters evaluated as well as their respective security level as taken from [13], which corresponds to the revised version of [27]. The criteria for ordering the parameter sets corresponds to their optimization goals, namely: short key-size, encryption/decryption speed, a trade-off between key-size and speed, and the optimization using product-form polynomials. Please note that insufficient RAM prevented evaluation using parameters EES1499EP1 and EES1171EP1. Starred parameter sets were evaluated for encryption and decryption, but not key-generation for the same reason.

B. Performance evaluation

For the performance evaluation, we used the optimization flag $-$O3, which proved to deliver the best performance results. Fig. 2 shows the runtime differences between the parameter sets for each of the implementations described in Section V. Table II shows the separate operations involved in encryption and decryption for the performance-optimized parameter sets with a 128-bit security level.

We observe that the fastest parameter sets for key-generation are EES401EP1 and EES401EP2. They take an average of 25.32 and 21.58 million cycles respectively, whereas parameter sets EES743EP1 and EES761EP1 take the longest time with 71.18 and 74.61 million cycles respectively. The overhead incurred by the time-independent implementation for key-generation ranged between 30% and 40% for product-form parameter sets, and between 53% and 60% for the non-product-form parameter sets. However, since keys can be generated off-line for most applications, instantiating only encryption and decryption functions is recommended in a timing attack resistant implementation. Our results also show that SVES-3 padding incurs noticeable overhead and should not be neglected when evaluating the performance of NTRU. As an example, 222,557 cycles were spent on MGF and BPGM for our EES443EP1-B implementation compared to 325,349 spent on the polynomial arithmetic to compute $r * h$.

For encryption and decryption, we observe that the fastest parameter sets are those corresponding to the product-form parameter sets ordered by key-length, whereas for the non-product-form parameter sets, we observe that the order is related to the set's optimization criteria as well as to the security level. The time-independent implementation's overhead was found to be between 20% and 35% for encryption and 25.5% and 38.5% for decryption. Table III shows a performance comparison between the results obtained for our implementations and the software implementations known in literature.

Our results also show that pattern multiplication cannot beat product-form polynomials and that the efficiency naturally depends on the distribution of the polynomial on which the pattern search is conducted. For the computation of $f * e$ roughly 625,740 cycles(EES401EP1), 1,401,420 cycles (EES677EP1), 1,194,520 cycles (EES887EP1), or 696,190 cycles (EES761EP1) are required (including pattern finding) when using overlapping mixed patterns. However, the performance is still better than straightforward ternary polynomial multiplication.

TABLE II. PERFORMANCE COMPARISON OF OUR DIFFERENT IMPLEMENTATIONS FOR A 128−bit SECURITY LEVEL.

Operation	Parameter Set / Implementation (cycles)			
	EES761EP1-A	EES761EP1-C	EES443EP1-B	EES443EP1-C
Encryption				
RNG	1,389	1,389	859	859
str2tern	5,433	5,433	3,200	3,200
BPGM	155,200	155,200	140,794	140,794
$r * h$	994,571	1,408,005	325,349	449,453
MGF	120,278	120,278	81,763	81,763
Mask Add	42,864	39,772	23,470	21,671
e	13,789	13,789	8,058	8,058
Decryption				
$f * e$	1,005,885	1,424,804	328,995	457,328
a	38,394	33,570	22,425	19,574
b	137,86	24,537	14,354	8,058
MGF	120,275	120,275	81,762	81,762
pt	32,142	28,991	17,187	15,355
tern2str	10,526	6,900	6,121	4,008
BPGM	169,512	170,262	147,195	147,623
$r * h$	994,571	1,408,005	325,349	449,453

TABLE III. PERFORMANCE COMPARISON WITH RESPECT TO PREVIOUS IMPLEMENTATIONS OF NTRUENCRYPT IN SMALL DEVICES. RESULTS ARE GIVEN IN NUMBER OF CYCLES.

Platform	Work	Security	(N,p,q)	Key-gen.	Encryption	Decryption
Cortex M0	our B	128-bit	(443,3,2048)	26,114,818	588,044	950,371
Cortex M0	our B	192-bit	(587,3,2048)	45,659,315	1,040,538	1,634,821
Cortex M0	our B	256-bit	(743,3,2048)	71,186,959	1,411,557	2,377,054
ARM7	[7]	deprecated	(251,X+2,128)	2,982,200	120,250	249,750
ARM7TDMI	[8]	deprecated	(107,3,64)	4,570,000	245,000	285,000
ARM7TDMI	[8]	deprecated	(503,3,256)	120,605,000	5,545,000	8,155,000
ATMega128	[9]	deprecated	(167,3,128)	9,088,000	208,000	496,000
ATMega128	[9]	deprecated	(251,3,128)	17,260,000	256,000	628,000
ATMega163	[9]	deprecated	(167,3,128)	9,228,000	213,200	524,000
ARM7TDMI	[10]	128-bit	(439,3,2048)	26,469,600	693,720	998,760
ATMega64	[10]	128-bit	(439,3,2048)	76,800,000	2,008,000	1,392,000

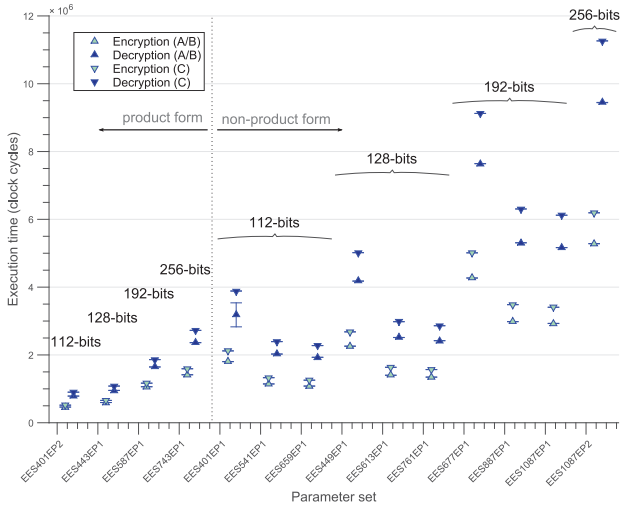


Fig. 2. Runtime for different security levels of our NTRUEncrypt implementations.

C. Cost Evaluation

Fig. 3 shows the code's memory footprint of the different parameter sets for all our implementations. The time-independent and standard versions of NTRUEncrypt require

roughly the same amount of program memory. In some cases, the compiler was able to obtain a modestly smaller size for the independent-time implementation. The overhead incurred by encryption and decryption operations is also negligible.

To estimate energy consumption, we used the numbers for power-supply current during active mode, from the XMC1100 data sheet (with f_{mclk} and $f_{pclk} = 1\text{MHz}$), and the results presented in Fig. 2. For parameter EES1087EP2 (worst case), a 10% decrease in battery life is seen when executing over three thousand encryption or decryption operations per day and only 1% using parameter set EES443EP1 (best case).

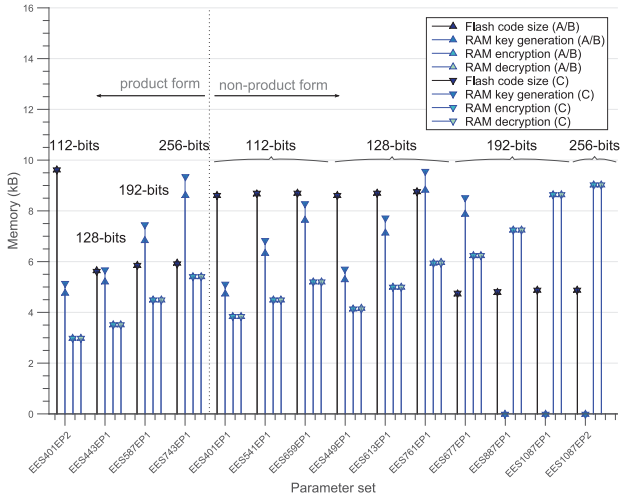


Fig. 3. Code and memory footprint for different security levels of our NTRUEncrypt implementations.

VII. Conclusion

In this work, we presented a feasibility analysis of the NTRUEncrypt cryptosystem for IoT-endpoints. The implementation results show good performance with a relatively low memory footprint. For example, parameter set EES761EP1, which provides 128-bit security, executes encryption in 1.3×10^6 cycles and decryption in 2.4×10^6 cycles, while occupying 9 KB of Flash and 6 KB of RAM (i.e. 14% and 37.5% of the resources of the target microcontroller). This shows that NTRUEncrypt is a viable alternative or possible additional layort on top of current PKC for providing long-term, public-key encryption on resource-constrained devices.

Acknowledgments. We thank the anonymous reviewers for their valuable comments and suggestions. This work was partly funded by the German Federal Ministry of Education and Research (BMBF), grant number 01IS160253 (ARAMiS II). This work was done while Elena Fuentes Bongenaar was an intern at Infineon Technologies and studying at Radboud University.

References

[1] J. Buchmann, A. May, and U. Vollmer, "Perspectives for Cryptographic Long-term Security," *Commun. ACM*, vol. 49, no. 9, pp. 50–55, Sep. 2006.

[2] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM J. Comput.*, vol. 26, no. 5, pp. 1484–1509, Oct. 1997.

[3] NSA Information Assurance Directorate. (2015) Commercial national security algorithm suite. [Online]. Available: https://www.iad.gov/iad/programs/iad-initiatives/cnsa-suite.cfm

[4] L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone, "Report on Post-quantum Cryptography," *National Institute of Standards and Technology Internal Report*, vol. 8105, 2016. [Online]. Available: http://dx.doi.org/10.6028/NIST.IR.8105

[5] J. Hoffstein, J. Pipher, and J. H. Silverman, "NTRU: A ring-based public key cryptosystem," in *ANTS*, 1998, pp. 267–288.

[6] J. Hoffstein and J. H. Silverman, "Random small Hamming weight products with applications to cryptography," *Discrete Applied Mathematics*, vol. 130, no. 1, pp. 37 – 49, 2003, the 2000 Com2MaC Workshop on Cryptography.

[7] D. V. Bailey, D. Coffin, A. Elbirt, J. H. Silverman, and A. D. Woodbury, "NTRU in Constrained Devices," in *CHES*, 2001, pp. 262–272.

[8] O. Collen Marie, "Efficient NTRU implementation," Master's thesis, Worcester Polytechnic Institute, 2002. [Online]. Available: https://www.wpi.edu/Pubs/ETD/Available/etd-0430102-111906/unrestricted/corourke.pdf

[9] M. Monteverde, "NTRU Software Implementation for Constrained Devices," Master's thesis, Katholieke Universiteit Leuven, 2008.

[10] A. Boorghany, S. B. Sarmadi, and R. Jalili, "On Constrained Implementation of Lattice-Based Cryptographic Primitives and Schemes on Smart Cards," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 3, pp. 42:1–42:25, Apr. 2015.

[11] M.-K. Lee, J. W. Kim, J. E. Song, and K. Park, "Sliding window method for NTRU," in *ACNS*, 2007, pp. 432–442.

[12] M. Braithwaite. (2015) Experimenting with post-quantum cryptography. [Online]. Available: https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html

[13] W. Whyte, "EESS 1: Implementation Aspects of NTRUEncrypt, Version 3.1," Consortium for Efficient Embedded Security, Tech. Rep., September 2015. [Online]. Available: https://www.securityinnovation.com/products/ntru-crypto/ntru-resources#ntrustandards

[14] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," *J. ACM*, vol. 60, no. 6, p. 43, 2013.

[15] C. Peikert, "Lattice cryptography for the internet," in *PQCrypto*, 2014, pp. 197–219.

[16] C. Dwork, M. Naor, and O. Reingold, "Immunizing encryption schemes from decryption errors," in *EUROCRYPT*, 2004, pp. 342–360.

[17] R. Lindner and C. Peikert, "Better key sizes (and attacks) for LWE-based encryption," in *CT-RSA*, 2011, pp. 319–339.

[18] N. Göttert, T. Feller, M. Schneider, J. A. Buchmann, and S. A. Huss, "On the design of hardware building blocks for modern lattice-based encryption schemes," in *CHES 2012*, 2012, pp. 512–529.

[19] A. Singer, "NTRU Cipher Suites for TLS," IETF, Internet-Draft draft-ietf-tls-ntru-00, Jul. 2001, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-ietf-tls-ntru-00

[20] J. M. Schanck, W. Whyte, and Z. Zhang, "Quantum-Safe Hybrid (QSH) Ciphersuite for Transport Layer Security (TLS) version 1.3," IETF, Internet-Draft draft-whyte-qsh-tls13-02, Apr. 2016, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-whyte-qsh-tls13-02

[21] ——, "A quantum-safe circuit-extension handshake for Tor," *IACR ePrint*, vol. 2015, p. 287, 2015, to appear in PETS'16.

[22] N. Howgrave-Graham, J. H. Silverman, A. Singer, and W. Whyte, "NAEP: provable security in the presence of decryption failures," *IACR ePrint*, vol. 2003, p. 172, 2003.

[23] B. Jun and P. Kocher, "The Intel Random Number Generator," Cryptography Research Inc. white paper, Tech. Rep., April 1999.

[24] NIST Statistical Test Suite. [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html

[25] J. H. Silverman, "Almost inverses and fast NTRU key creation," *NTRU Cryptosystems,(Technical Note# 014): http://www. ntru. com/cryptolab/pdf/NTRU Tech014. pdf*, 1999.

[26] J. A. Buchmann, M. Döring, and R. Lindner, "Efficiency improvement for NTRU," *IACR ePrint*, vol. 2007, p. 263, 2007. [Online]. Available: http://eprint.iacr.org/2007/263

[27] J. Hoffstein, J. Pipher, J. M. Schanck, J. H. Silverman, W. Whyte, and Z. Zhang, "Choosing parameters for NTRUEncrypt," *IACR ePrint*, vol. 2015, p. 708, 2015. [Online]. Available: http://eprint.iacr.org/2015/708