A Lightweight Implementation of NTRU Prime for the Post-Quantum Internet of Things

Hao Cheng[1], Daniel Dinu[2], Johann Großschädl[1], Peter B. Rønne[1], and Peter Y. A. Ryan[1]

[1] SnT and CSC, University of Luxembourg
6, Avenue de la Fonte, L–4364 Esch-sur-Alzette, Luxembourg
{hao.cheng,johann.groszschaedl,peter.roenne,peter.ryan}@uni.lu
[2] IPAS, Intel, Chandler, AZ 85226, USA
daniel.dinu@intel.com

Abstract. The dawning era of quantum computing has initiated various initiatives for the standardization of post-quantum cryptosystems with the goal of (eventually) replacing RSA and ECC. NTRU Prime is a variant of the classical NTRU cryptosystem that comes with a couple of tweaks to minimize the attack surface; most notably, it avoids rings with "worrisome" structure. This paper presents, to our knowledge, the first assembler-optimized implementation of Streamlined NTRU Prime for an 8-bit AVR microcontroller and shows that high-security lattice-based cryptography is feasible for small IoT devices. An encapsulation operation using parameters for 128-bit post-quantum security requires 8.2 million clock cycles when executed on an 8-bit ATmega1284 microcontroller. The decapsulation is approximately twice as costly and has an execution time of 15.6 million cycles. We achieved this performance through (i) new low-level software optimization techniques to accelerate Karatsuba-based polynomial multiplication on the 8-bit AVR platform and (ii) an efficient implementation of the coefficient modular reduction written in assembly language. The execution time of encapsulation and decapsulation is independent of secret data, which makes our software resistant against timing attacks. Finally, we assess the performance one could theoretically gain by using a so-called product-form polynomial as part of the secret key and discuss potential security implications.

Keywords: Lightweight cryptography · Post-quantum cryptography · Key encapsulation mechanism · NTRU Prime · Efficient implementation

1 Introduction

The advent of quantum computing is a technological revolution that will soon have a massive impact on our daily life and may even disrupt whole industries [19]. In short, a quantum computer operates on so-called qubits (the "quantum analog" of bits), which can not only take the two states 0 and 1, but also be in a superposition of both states. A quantum computer with n qubits can be in an arbitrary superposition of up to 2^n states simultaneously, enabling it to process

2^n values in parallel or to store 2^n values in one step. For example, a quantum computer with about 50 logical qubits could solve certain complex optimization problems a lot faster than the most advanced classical supercomputer today. In the not-so-distant future, our daily life will start to get affected by large-scale quantum computers that are powerful enough to aid the discovery of new drugs or materials, organize the routes of millions of self-driving cars in metropolitan areas without introducing traffic jams, and improve the efficiency of national power grids [19]. Unfortunately, quantum computing has also a destructive side because a large-scale quantum computer with a few thousand qubits would be able to break essentially every public-key cryptosystem in use today. This was discovered in the mid-90s by Peter Shor, who also developed a polynomial-time quantum algorithm to factor large integers, which could break the widely-used RSA cryptosystem [25]. Later, it was also found that a generalization of Shor's algorithm would enable one to take discrete logarithms in a large elliptic curve groups, thereby breaking Elliptic Curve Cryptography (ECC).

Estimates as to when the first large-scale quantum computer might become available vary significantly, but optimistic predictions suggest it could happen before the end of the 2020s [21]. Given the real-world threat posed by quantum computing, it is little surprising that research in the domain of *Post-Quantum Cryptography (PQC)*, i.e. cryptography that is able to withstand cryptanalytic attacks carried out using a large quantum computer [3], has gained momentum over the past few years. In 2016, the U.S. National Institute of Standards and Technology (NIST) announced a process to "solicit, evaluate, and standardize quantum-resistant public-key cryptographic algorithms" and published a call to submit proposals [22]. This call, whose submission deadline passed at the end of November 2017, covered the complete spectrum of public-key functionalities considered by the NIST, i.e. public-key encryption, key agreement, and digital signatures. A total of 72 candidates were submitted, of which 69 satisfied the minimum requirements for acceptability and entered the first round of a multi-year evaluation process. In early 2019, the NIST selected 26 of the submissions as candidates for the second round; among these are 17 public-key encryption or key-establishment algorithms and nine signature schemes. The 17 algorithms for encryption (resp. key establishment) include nine that are based on certain hard problems in lattices, seven whose security rests upon classical problems in coding theory, and one that claims security from the presumed hardness of the (supersingular) isogeny walk problem on elliptic curves [22].

NTRU Prime is a family of lattice-based cryptosystems developed by Bernstein, Chuengsatiansup, Lange, and van Vredendaal [4], who drew inspiration from the 20-year old classical NTRU cryptosystem [12]. There are two variants of NTRU Prime; one is the so-called *Streamlined NTRU Prime*, which uses the quotient $h = g/(3f)$ of two secret polynomials g, f as public key (similar to the classical NTRU), while the other, *NTRU LPRime*, has public keys of the form $h = e + Af$, where e, f are secret and A is public (like in cryptosystems based on the Ring Learning With Errors (RLWE) problem [20], e.g. NewHope [1]). In essence, NTRU Prime can be seen as an attempt to improve the security of the

classical NTRU encryption algorithm (and other lattice-based cryptosystems) by avoiding rings with "worrisome" structure and using extension fields of the form $\mathcal{R}/q = (\mathbb{Z}/q)[x]/(x^p - x - 1)$ instead, where p is prime. Multiplication in such fields can be efficiently implemented through several layers of Karatsuba's technique [17], which makes NTRU Prime relatively fast on 64-bit processors with vector instructions. Concretely, the designers of NTRU Prime describe in [4] a highly-optimized implementation of the field multiplication using Intel's AVX2 vector instructions that executes 16 separate multiplications of integers modulo 2^{16} in a SIMD-parallel way. NTRU Prime is among the 26 candidates in the second round of NIST's evaluation process. This second round will focus on evaluating the candidates' performance across a wide variety of systems and platforms, which includes "not only big computers and smart phones, but also devices that have limited processor power" [22].

Research on software optimization techniques that enable fast implementations of (Streamlined) NTRU Prime has, until now, been limited to 64-bit Intel processors with AVX2 vector engine. When using a parameter set for 128 bits of post-quantum security, the AVX2 implementation introduced in [4] requires 59,600 clock cycles for encryption (i.e. "encapsulation" of a 256-bit key) on an Intel Haswell processor, while the decryption ("decapsulation") is 63.5% more costly and takes 97,452 cycles. The only performance figures for NTRU Prime on small platforms (e.g. 8, 16, or 32-bit microcontrollers) we are aware of were reported in a recent paper on `pqm4` [16], a testing and benchmarking toolsuite for NIST PQC candidates on ARM Cortex-M4 devices. Due to the lack of an optimized ARM implementation, the authors of [16] resorted to the reference C code provided by the designers of NTRU Prime, which requires 54.9 million clock cycles for encapsulation and 166.5 million cycles for decapsulation (these cycle counts were determined with Streamlined NTRU Prime and parameters for 128-bit post-quantum security). However, both results do not allow one to reason about the actual performance of NTRU Prime on microcontrollers since the aim of a reference C implementation is to promote the understanding of an algorithm rather than achieving high speed. Therefore, not much is known on how to optimize NTRU Prime for a small microcontroller and what execution time a carefully-tuned assembler implementation could achieve.

In this paper we present a highly-optimized implementation of Streamlined NTRU Prime for 8-bit AVR microcontrollers that we developed from scratch to reach high speed and resistance against timing attacks. We chose 8-bit AVR as evaluation platform for two reasons. First, the 8-bit AVR architecture remains very popular in devices with increased security requirements, e.g. smart cards and (wireless) sensor nodes. Second, 8-bit AVR microcontrollers are among the most resource-limited of all currently used computing platforms, which implies that if NTRU Prime can be implemented to run with acceptable speed on an AVR device, it can also be implemented to run satisfactorily on more powerful 16 and 32-bit microcontrollers (e.g. an ARM Cortex-M), whereas the opposite is not necessarily true. The implementation we describe in the next sections is not purely optimized for speed, but strives for a balance between performance

and other metrics of interest for low-end devices used in the Internet of Things (IoT), in particular binary code size. Therefore, we decided to refrain from full loop unrolling and other optimization techniques that are likely to increase the code size significantly (especially on an 8-bit device) for marginal performance benefits. We also restrict our arsenal of polynomial multiplication algorithms to the basic (i.e. recursive) Karatsuba variant and the schoolbook method for the same reason. Recent results by Kannwischer et al [15] show that a combination of Karatsuba's technique with the asymptotically faster Toom-Cook algorithm [27] can slightly reduce the multiplication time, e.g. by 17.4% for polynomials of degree 701 (excluding the reduction of coefficients), but only at the expense of almost doubled stack usage and significantly increased implementation complexity. On the other hand, our Karatsuba/schoolbook multiplication is simple to implement and has the further advantage of enabling compact code size (see Sect. 4) while remaining competitive in terms of performance.

Instead of potential speed-ups due to the Toom-Cook algorithm, we analyze the performance benefits one could achieve by utilizing so-called *product-form polynomials*, which were first proposed in [13, 14] to reduce the computational cost of the classical NTRU scheme. We show that representing the secret key in product form would cut the decapsulation time by 30%, but we also emphasize that the security implications of product-form secret keys in NTRU Prime are yet to be carefully analyzed. Furthermore, we present efficient implementations of the fast reduction of coefficient products of a length of up to 29 bits modulo a 13-bit prime q. Finally, we demonstrate that, for some 8-bit AVR models like the ATtiny45, the modulo-3 reduction code generated by optimizing compilers may have operand-dependent execution time and enable timing attacks.

2 A Brief Overview of NTRU Prime

NTRU Prime is introduced in [4] as a high-security *prime-degree large-Galois-group inert-modulus* ideal-lattice-based cryptosystem. A distinguishing feature of NTRU Prime is the use of an irreducible non-cyclotomic polynomial P; the designers recommend to choose a polynomial P of prime degree p with a large Galois group. More specifically, they suggest $P = x^p - x - 1$ and recommend to take a prime modulus q such that P is irreducible modulo q, which means q is inert in the ring $\mathcal{R} = \mathbb{Z}[x]/P$ and $\mathcal{R}/q = (\mathbb{Z}/q)[x]/P$ is actually a field. Due to the prime degree of P, the only subfields of $(\mathbb{Z}/q)[x]/P$ are $\mathbb{Z}/q$ and the entire field $(\mathbb{Z}/q)[x]/P$. Furthermore, the requirement of a large Galois group implies that P has, at most, a few roots in any field of reasonable degree, which makes automorphism computations hard. Finally, since q is an inert prime, there are no ring homomorphisms from $(\mathbb{Z}/q)[x]/P$ to any smaller non-0 ring.

The NTRU Prime family of Key Encapsulation Mechanisms (KEMs) specified in [4, 5] consists of Streamlined NTRU Prime and NTRU LPrime, but we only consider the former since it is more implementation-friendly. Streamlined NTRU Prime is similar to classical NTRU, but adopts a rounding technique in the encapsulation and, as explained above, uses a field instead of a ring.

Notation and Parameters. A parameter set for Streamlined NTRU Prime consists of the triple (p, q, w), which defines the main algebraic structures. The parameter p is the degree of the irreducible polynomial $P = x^p - x - 1$ and is prime; the parameter sets given in [5] use 653, 761, and 857. Also the modulus q, which represents the characteristic of the field $\mathcal{R}/q = (\mathbb{Z}/q)[x]/P$, is a prime with typical values of 4621, 4591, and 5167, respectively, for the three degrees considered in [5]. The weight parameter w is a positive integer that defines the number of non-0 coefficients of certain polynomials. A valid parameter set has to satisfy $2p \geq 3w$ and $q \geq 16w + 1$. Reusing the notation of [5], we abbreviate the ring $\mathbb{Z}[x]/P$, the ring $(\mathbb{Z}/3)[x]/P$, and the field $(\mathbb{Z}/q)[x]/P$ as $\mathcal{R}$, $\mathcal{R}/3$, and $\mathcal{R}/q$, respectively. An element of the ring $\mathcal{R}$ is *small* if all its coefficients are in $\{-1, 0, 1\}$. Short is defined as the set of small weight-w elements of $\mathcal{R}$, while Rounded is the set of polynomials $r(x) \in \mathcal{R}$ where each coefficient r_i lies is the range $[-(q-1)/2, (q-1)/2]$ and is rounded to the nearest multiple of 3.

Key Generation. To generate a key pair for Streamlined NTRU Prime, the following operations have to be performed (note that, for brevity, we skip some operations such as the encoding of polynomials to strings).

1. Generate a uniform random *small* polynomial $g(x) \in \mathcal{R}$. Repeat this step until $g(x)$ is invertible in $\mathcal{R}/3$.
2. Compute $v(x) = 1/g(x)$ in $\mathcal{R}/3$.
3. Generate a uniform random polynomial $f(x) \in$ Short.
4. Compute $h(x) = g(x)/(3f(x))$ in $\mathcal{R}/q$.
5. Generate a uniform random polynomial $\rho(x) \in$ Short.
6. Output $h(x)$ as *public key* and $(f(x), v(x), h(x), \rho(x))$ as *private key*.

Encapsulation. The encapsulation operation gets a public key as input and produces a ciphertext and session key as output (again, for brevity, we skip all encoding and decoding operations).

1. Generate a uniform random polynomial $r(x) \in$ Short.
2. Compute $c(x) = h(x)r(x) \in$ Rounded.
3. Compute $C = (c(x), \textsc{Hash}(r(x), h(x)))$.
4. Output C as *ciphertext* and $\textsc{Hash}(1, r(x), C)$ as *session key*.

Decapsulation. The decapsulation gets a key pair and a ciphertext as input and produces a session key as output (encodings and decodings are skipped).

1. Compute $e(x) = 3f(x)c(x) \in \mathcal{R}/q$ and represent each coefficient e_i of $e(x)$ as an integer between $-(q-1)/2$ and $(q-1)/2$.
2. Compute $e(x) = e(x) \bmod 3 \in \mathcal{R}/3$ (i.e. reduce each e_i modulo 3).
3. Compute $r'(x) = e(x)v(x) \in \mathcal{R}/3$.
4. Lift $r'(x) \in \mathcal{R}/3$ to a small polynomial $r'(x) \in \mathcal{R}$.
5. If the weight of $r'(x)$ is not w then set $r'(x) = (1, 1, \ldots, 1, 0, 0, \ldots, 0)$.

6. Compute $c'(x) = h(x)r'(x) \in \mathsf{Rounded}$.
7. Compute $C' = (c'(x), \textsc{Hash}(r'(x), h(x)))$.
8. If C' equals C then output $\textsc{Hash}(1, r'(x), C)$ else output $\textsc{Hash}(0, \rho(x), C)$ as *session key*.

3 Polynomial Multiplication

Since Streamlined NTRU Prime is closely related to the classical NTRU scheme (i.e. NTRUEncrypt), it is not surprising that they share many implementation aspects; in particular, they have in common that their performance depends to a large extent on the polynomial arithmetic. However, the underlying algebraic structures are (slightly) different: NTRUEncrypt is based on the residue class ring $\mathcal{R} = (\mathbb{Z}/q)[x]/(x^N - 1)$ where q is a power of two, while NTRU Prime uses the extension field $(\mathbb{Z}/q)[x]/(x^p - x - 1)$ where q is a prime, e.g. $q = 4621$. The reduction modulo q is basically free in the former case, but relatively expensive for NTRU Prime, especially when constant execution time is required so as to foil timing attacks. Furthermore, the irreducible polynomial P of NTRU Prime contains an additional non-0 coefficient, which makes the reduction operation more costly. Finally, most performance-optimized implementations of classical NTRU for constrained IoT devices use a parameter set with so-called product-form polynomials [14] to minimize the execution time of the ring multiplication (see e.g. [2, 7]). However, product-form parameter sets were not included in the NTRU Prime specification. For all these reasons, one can expect the arithmetic part of NTRU Prime, when implemented for an 8-bit AVR microcontroller, to be significantly slower than that of the classical NTRU cryptosystem.

The encapsulation operation of NTRU Prime includes a single polynomial multiplication where one operand is an element of $\mathcal{R}/q$ (i.e. its coefficients are bounded by q) and the other operand is an element of Short, which means it is a ternary polynomial with exactly w non-0 coefficients. Hence, the polynomial multiplication carried out in NTRU Prime encapsulation is very similar to the ring multiplication in the encryption operation of classical NTRU [12]. On the other hand, the decapsulation of NTRU Prime involves three polynomial multiplications, which is one more than the number of multiplications that have to be executed in classical NTRU decryption. The first polynomial multiplication in the decapsulation gets an element of $\mathsf{Rounded}$ (i.e. an element of $\mathcal{R}/q$) and an element of Short as input. In contrast, the second polynomial multiplication (Step 3 of the decapsulation as presented in the previous section) is performed on two elements of $\mathcal{R}/3$, i.e. two ternary polynomials. The third multiplication of the decapsulation is exactly the same as the polynomial multiplication in the encapsulation, which means the operands are elements of $\mathcal{R}/q$ and Short.

3.1 Karatsuba-Based Polynomial Multiplication

Most algorithms for high-speed polynomial multiplication have their origins in well-known algorithms for multiple-precision multiplication of integers, such as

needed for common public-key cryptosystems like RSA and ECC [8, 11]. From a high-level perspective, polynomial multiplication algorithms can be split into two main categories, namely basic techniques that require n^2 coefficient multiplications to obtain the product of two polynomials consisting of n coefficients each, and advanced techniques with sub-quadratic complexity, e.g. Karatsuba's algorithm [17]. Examples of the former category are the *operand-scanning* and *product-scanning* method, which produce the coefficient-products in a row-wise or column-wise fashion and differ with respect of the number of load and store instructions they need to execute [11]. The so-called *hybrid technique* proposed in [10] is beneficial on microcontrollers with a large number of general-purpose registers (e.g. AVR ATmega) and combines the individual strengths of operand scanning and product scanning. It has a "nested loop" structure and computes $d \geq 2$ coefficient-products in each iteration of the inner loop, which reduces the number of load instructions by a factor of d compared to product scanning.

Multiplication algorithms with sub-quadratic complexity have been known since the 1960s when Karatsuba published his seminal paper [17]. Karatsuba's method reduces a multiplication of two operands consisting of n coefficients to three multiplications of $(n/2)$-coefficient polynomials and a few additions. The half-size multiplications, in turn, can be implemented using any multiplication technique, including conventional operand and product scanning, as well as the hybrid method. Alternatively, it is possible to apply the Karatsuba algorithm recursively until the operands consist of just a single coefficient, in which case the asymptotic complexity becomes $\Theta(n^{\log_2(3)})$. Yet another option is the so-called Arbitrary Degree Karatsuba (ADK) variant described and analyzed in detail in [24]. Also a few multiplication algorithms with even better asymptotic complexity have been studied; an example is the Toom-Cook multiplication we mentioned in Sect. 1 in the context of Kannwischer et al's work on polynomial multiplication for ARM Cortex-M4 processors [15]. An efficient implementation of a 4-way Toom-Cook algorithm for multiplication of degree-256 polynomials on a Cortex-M4 device is described in [18].

Finding the optimal multiplication strategy for the two forms of polynomial multiplication mentioned at the beginning of this section (i.e. $\mathcal{R}/q \times \mathsf{Short}$ and $\mathcal{R}/3 \times \mathcal{R}/3$) is a difficult task. Intuitively, one may assume that a combination of multiplication techniques with sub-quadratic and quadratic complexity will yield peak performance. Yet, the concrete implementation of such a combined strategy raises a few non-trivial questions. Asymptotic complexity bounds are not always meaningful in the real world, especially when the involved operands are relatively short. Therefore, it is necessary to find out which sub-quadratic algorithms are most efficient ones for the multiplications in NTRU Prime (this depends besides the lengths of the polynomials also on certain characteristics of the target architecture). For constrained platforms like 8-bit AVR, it makes sense to base this decision not solely on speed but also on RAM requirements and code size. A second important question is how many recursions of Karatsuba's and/or Toom-Cook's algorithm should be performed before switching to a multiplication method with quadratic complexity, i.e. what operand length is

the "crossover" point? Finally, a third question is which of the basic algorithms should be used: operand scanning, product scanning, or the hybrid method? In order to answer all these questions, we conducted a multitude of experiments with different sub-quadratic algorithms[3], different numbers of recursions of the sub-quadratic algorithms (i.e. different "crossover" points), and different basic multiplication techniques with quadratic complexity.

The results of these experiments show that for a polynomial multiplication of the form $\mathcal{R}/q \times \mathsf{Short}$ (carried out in Step 2 of encapsulation as well as Step 1 and 6 of decapsulation), five recursions of Karatsuba's algorithm provide the best performance across all parameter sets specified in [5]. Below the five levels of Karatsuba, the normal product-scanning technique is used since, due to the bitlength of the coefficient-products and the limited register space, the hybrid multiplication is not efficient. Also alternative Karatsuba variants, such as the ADK algorithm from [24], did not yield superior performance. The situation is different for the polynomial multiplication of the form $\mathcal{R}/3 \times \mathcal{R}/3$, which has to be carried out in Step 3 of the decapsulation. For this multiplication, a combination of the (recursive) Karatsuba algorithm and hybrid method achieves the best results. To be precise, we reached peak performance with four recursions of Karatsuba and using the hybrid method with $d = 4$ at the "lower level" (this is possible because the coefficient-products are relatively small and, thus, more free registers are available). We implemented Karatsuba's algorithm in C and the hybrid multiplication method in both C and AVR assembler, whereby the latter is very similar to the implementations described in [10, 8].

A multiplication of two polynomials of degree $p-1$ through a combination of Karatsuba's algorithm and the hybrid method (or any other multiplication technique) yields a product-polynomial $r(x)$ of degree $2p-2$, which has to be reduced modulo the irreducible polynomial $P = x^p - x - 1$ to get a polynomial of degree $p-1$. Thanks to the relation $x^p \equiv x + 1 \bmod P$, this reduction can be performed by simply substituting each term $r_i x^i$ with $i \geq p$ in $r(x)$ by the sum $r_i x^{i-p+1} + r_i x^{i-p}$ [5]. These substitutions are nothing else than additions of the $p-1$ higher coefficients r_i to r_{i-p+1} and r_{i-p}, which reduces the degree of $r(x)$ to (at most) p so that two further coefficient additions suffice to obtain a result of degree $p-1$. Thus, the cost of the reduction modulo P amounts to $2p$ additions of (unreduced) coefficients. The final step of the multiplication is the reduction of the $p-1$ remaining coefficients modulo q or modulo 3.

Coefficient-Reduction Modulo q. As explained above, we implemented the multiplication of the form $\mathcal{R}/q \times \mathsf{Short}$ using five recursions of Karatsuba as "higher level" algorithm and product scanning at the "lower level." Taking the parameter set `sntrup653` as example, we have $p = 653$, which means the hybrid method is executed with operands of degree $\lceil 653/2^5 \rceil = 21$. Furthermore, since

[3] As stated in Sect. 1, we do not consider the Toom-Cook multiplication algorithm due to its high RAM consumption. The AVR device we use for benchmarking, an ATmega1284 microcontroller, has only 16 kB SRAM, which makes a strong case to take memory requirements into account in the algorithm exploration.

Algorithm 1. Table-based constant-time modular reduction

Input: Integer s of a length of (up to) 29 bits, modulus q of a fixed length of 13 bits
Output: $r = s \bmod q$

1: $b \leftarrow (s_{28}, \ldots, s_{24})$ ▷ extract the five bits $b = (s_{28}, \ldots, s_{24})$ from s
2: $r \leftarrow \mathrm{RT1}[b]$ ▷ reduce $b2^{24}$ modulo q via look-up table RT1
3: $b \leftarrow (s_{23}, \ldots, s_{16})$ ▷ extract the eight bits $b = (s_{23}, \ldots, s_{16})$ from s
4: $r \leftarrow r + \mathrm{RT2}[b]$ ▷ reduce $b2^{16}$ modulo q via look-up table RT2
5: $r \leftarrow r + s \,\&\, 0\mathrm{xffff}$ ▷ add 16 least-significant bits of s to r
6: $b \leftarrow (r_{16}, \ldots, r_{12})$ ▷ extract the five bits $b = (r_{16}, \ldots, r_{12})$ from r
7: $r \leftarrow (r \,\&\, 0\mathrm{xfff}) + \mathrm{RT3}[b]$ ▷ reduce $b2^{12}$ modulo q via look-up table RT3
8: $r \leftarrow r - q \cdot (r \geqslant q)$ ▷ conditionally subtract q from r
9: **return** r

$q = 4621$ and we represent the -1 coefficients of a ternary polynomial (i.e. an element of Short) as $q - 1 = 4620$, a single coefficient-product has a maximum length of 24 bits. The column sum to which the 24-bit coefficient-products are accumulated can become up to 29 bits long, i.e. we need an efficient algorithm for reducing a 29-bit integer modulo a 13-bit integer.

Algorithm 1 shows a generic technique for reducing a 29-bit integer modulo an arbitrary 13-bit integer q using three look-up tables, which we call reduction tables. It is assumed that the input s (representing a column sum of the hybrid method described above) is held in four 8-bit registers, i.e. the individual bytes of s can be conveniently accessed. At first, the five most-significant bits of s are assigned to b and then $b2^{24} \bmod q$ is computed with the help of reduction table RT1, which contains 32 entries. Next, the second-most significant byte of s is processed in a similar way, whereby the 256-entry table RT2 is used to obtain its residue modulo q. The two residues are added up and form the intermediate result r. Then, we extract the 16 least-significant bits from s and add them to r, which has now a length of at most 17 bits. Similar as before, we assign the five most-significant bits of r to b, reduce it using RT3, and add the residue to the 12 least-significant bits of r. Because r is now always less than $2q$, a single subtraction of q is sufficient to have a fully reduced result. However, to ensure constant execution time, we first compare r with the modulus q, which returns 1 if $r \geq q$ and 0 otherwise. This comparison-result is multiplied by q and the product (either q or 0) is then subtracted from r. Note that Algorithm 1 works for any 13-bit modulus q, though each q requires its own set of tables.

Coefficient-Reduction Modulo 3. The reduction modulo 3 can exploit the fact that some multiples of 3 (e.g. 15, 255) have the form $2^k \pm 1$, which allows for a particularly efficient implementation. Thus, the reduction modulo 3 is less costly (in terms of look-up tables) than the modulo-q case, but requires special attention regarding timing attacks. Namely, as described in Sect. 2, one of the operands of the $\mathcal{R}/3 \times \mathcal{R}/3$ multiplication in the decapsulation is $v(x)$, which is a part of the private key. Therefore, an implementer has to take care that this multiplication, including the reduction of all coefficient-products modulo 3, has

Table 1. Execution time (in cycles) of the `__udivmodhi4` function for all 2^{16} possible 16-bit unsigned integers. Columns labeled with "Frequ" and "%" give the frequency (in absolute numbers) and probability (in per cent) of the occurrence of the cycle count.

Cycles	Frequ.	%	Cycles	Frequ.	%	Cycles	Frequ.	%
193	3	0.005	198	7956	12.140	203	3825	5.836
194	45	0.069	199	12243	18.681	204	1323	2.019
195	312	0.476	200	14121	21.547	205	312	0.476
196	1323	2.019	201	12244	18.683	206	45	0.069
197	3825	5.836	202	7956	12.140	207	3	0.005

constant execution time. When using C or C++, a modulo-3 reduction can be implemented by an operation of the form `y = x % 3`, whereby in our case `x` is a 16-bit integer. However, in the course of our work we found out that one can not take it for granted that a C compiler generates constant-time code for this operation. Concretely, we discovered that certain versions of `avr-gcc` generate code with operand-dependent execution time for some AVR models, which can leak information about the secret polynomial $v(x)$.

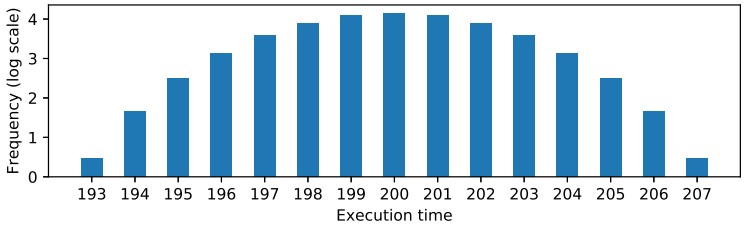

Fig. 1. Frequency of the occurrence (in absolute numbers) of a certain execution time (in cycles) of the `__udivmodhi4` function for all 2^{16} possible 16-bit unsigned integers.

For example, we determined the execution time of the modulo-3 reduction compiled with `avr-gcc` 4.8.2 for an ATtiny45 microcontroller with help of the cycle-accurate simulator Avrora [26]. For target devices that have no hardware multiplier, e.g. ATtiny microcontrollers, `avr-gcc` uses the `__udivmodhi4` function from the runtime library `libgcc` to perform the reduction modulo 3. The same function was also used for devices with hardware multiplier, including the ATmega1284 (our benchmarking device, see Sect. 4), until version 4.7.0 of the `avr-gcc` compiler; thereafter it was replaced with `__umulhisi3` [9]. While the latter function has a constant execution time (i.e. 54 cycles) for all 2^{16} possible inputs, the time required by the former depends on the value of the operand to be reduced. Concretely, the execution time of `__udivmodhi4` varies between 193 clock cycles (for input values 0, 1, and 2) and 207 cycles (for 49149, 49150, and 49151). Thus, the time difference between the longest and shortest execution is 14 cycles. Further details are provided in Table 1 and Figure 1.

In order to ensure that the resistance against timing attacks does not depend on the compiler, we implemented the modulo-3 reduction in assembly language following the approach described in [7].

3.2 Product-Form Polynomial Multiplication

A well-known way to improve the execution time of the original NTRU scheme (i.e. NTRUEncrypt) is to use ternary polynomials in product form, which was originally proposed some 20 years ago [13, 14]. In essence, a ternary polynomial $f(x)$ in product form can be expressed as $f(x) = f_1(x) \star f_2(x) + f_3(x)$, where $f_1(x)$, $f_2(x)$, $f_3(x)$ are three extremely sparsely populated ternary polynomials and $\star$ symbolizes a "convolution," i.e. a polynomial multiplication modulo the irreducible polynomial $P = x^N - 1$ of NTRUEncrypt [12]. For example, when using parameters for 128-bit security (based on a ring of degree $N = 443$), the given number of $+1$ and -1 coefficients of $f_1(x)$, $f_2(x)$, and $f_3(x)$ is 9, 8, and 5, respectively, which means that a convolution requires just a bit over 15,000 coefficient additions or subtractions. Despite the extremely low weight of these "sub-polynomials," it is possible to maintain security against all known attacks since the terms of $f_1(x)$ and $f_2(x)$ cross-multiply and the polynomial $f(x)$ has a weight of about $2N/3$. However, product-form parameters are rarely used in practice because the necessary index-based sparse polynomial multiplication is difficult to implement in a timing-attack-resistant fashion. Only recently it was shown that on AVR (and other microcontrollers without cache), product-form convolution can be fast *and* have constant execution time [7].

The designers of NTRU Prime decided not to support product-form parameters, claiming that product-form arithmetic "saves time for non-constant-time sparse-polynomial-multiplication algorithms, but loses time for constant-time algorithms" [4, Sect. T.3]. However, as recently demonstrated in [7], this claim is not necessarily true for microcontrollers without data cache. The advantages and disadvantages of the product form for NTRU Prime were also discussed on the official mailing list of NIST's PQC standardization project[4]. In light of the interest in product-form polynomials, we decided to assess how much they can accelerate NTRU Prime. Concretely, we evaluated the performance gain for the decapsulation when the ternary polynomial $f(x) \in \mathsf{Short}$, which is a part of the private key, is represented in product form. However, our work should not be seen as a recommendation to use the product form in practice.

A product-form parameter set for the classical NTRU cryptosystem includes the parameters d_1, d_2, d_3 specifying the number of $+1$ coefficients of the sub-polynomials $f_1(x)$, $f_2(x)$, $f_3(x)$, whereby the number of $+1$ coefficients equals the number of -1 coefficients (i.e. polynomial $f_i(x)$ has weight $w_i = 2d_i$). On the other hand, a set of parameters for NTRU Prime comes with just a single weight parameter w that specifies the number of non-0 coefficients of elements of Short. Hence, in order to use the product form for NTRU Prime, we have to

[4] `https://groups.google.com/a/list.nist.gov/d/msg/pqc-forum/fh2xGahC4LE/NycdEhTHAgAJ`

determine the weights w_1, w_2, w_3 of the sub-polynomials a ternary polynomial $f(x) \in \mathsf{Short}$ is composed of. The parameter generation approach we follow in this paper is derived from [23, Sect. 3.4.2] and assumes an equal split between $+1$ and -1 coefficients, though this requirement was dropped in NTRU Prime to allow for more choices of polynomials [4, Sect. 3.6]. Hoffstein and Silverman observed in one of the first papers about product-form polynomials that, when $f_1(x)$ and $f_2(x)$ are binary polynomials with d_1 and d_2 ones, respectively, the number of ones in the product $f_1(x)f_2(x)$ is essentially d_1d_2 [13]. Based on this observation, the weight of $f(x) = f_1(x) \star f_2(x) + f_3(x)$ can be estimated to be roughly $4d_1d_2 + 2d_3$ (see [23] for details). However, the weight of $f(x)$ depends not only on d_1, d_2, and d_3, but also on the irreducible polynomial used in the convolution. Since the irreducible polynomial P of NTRU Prime has the form $x^p - x - 1$, the reduction of the product $f_1(x)f_2(x)$ modulo P introduces more non-0 coefficients than a reduction modulo $x^p - 1$, the irreducible polynomial of NTRU. For example, any term of the form a_nx^n with $n \geq p$ gets reduced to $a_nx^{n-p+1} + a_nx^{n-p}$ in NTRU Prime, but to just a_nx^{n-p} in classical NTRU.

Our approach to calculate (d_1, d_2, d_3) for the NTRU Prime parameter sets (which require $f(x)$ to have a weight of $w = 288$, 286, and 322, respectively) is based on [23, Sect. 3.4.2], but takes the difference in the irreducible polynomial into account. For example, for the parameter set `sntrup653` (i.e. $w = 288$) we obtained $(d_1, d_2, d_3) = (9, 8, 4)$, i.e. the three sub-polynomials $f_1(x)$, $f_2(x)$, and $f_3(x)$ should have a weight of 18, 16, and 8, respectively. We conducted a large number of experiments for all three parameter sets of NTRU Prime to ensure that our approach to generate product-form polynomials is correct. In the case of `sntrup653`, the weight of $f(x)$ was always between 280 and 300.

While the security implications of using the product form have been studied in detail for classical NTRU [14], we are not aware of a similar security analysis for NTRU Prime. In the course of our work we discovered that the polynomial $f(x) = f_1(x) \star f_2(x) + f_3(x)$ has a linear distribution of non-0 terms (instead of a uniform distribution like in classical NTRU) if the non-0 coefficients of the sparse polynomials $f_1(x)$, $f_2(x)$, $f_3(x)$ are uniformly distributed. However, this effect can be compensated by choosing the distribution of the non-0 coefficients of $f_3(x)$ accordingly. We leave a full-fledged security analysis of product-form polynomials in NTRU Prime as part of our future work.

We implemented a product-form variant of NTRU Prime by re-using parts of the NTRU software for 8-bit AVR microcontrollers from [7], in particular the ring arithmetic. This software contains a ring multiplication function where one operand is an element of $\mathcal{R}/q$ (i.e. a polynomial with coefficients in the range $[0, q-1]$) and the second operand is a ternary polynomial in product form. We adapted this function to suit the requirements of NTRU Prime, which uses the field $\mathbb{Z}[x]/P$ with $P = x^p - x - 1$ as underlying algebraic structure. In concrete terms, this means we modified the reduction modulo the irreducible polynomial and the reduction of coefficient-sums modulo the prime q. The latter reduction can be performed in a similar way as described in Subsect. 3.1, except that the maximum length of a coefficient sum before modulo-q reduction is only 17 bits

Table 2. Execution time (in clock cycles) and code size (in bytes) of the main arithmetic operations and full encapsulation and decapsulation of NTRU Prime using the parameter set `sntrup653` on an ATmega1284 microcontroller. Operations annotated with "PF" use the product-form technique described in Subsect. 3.2.

Operation	Time	Size
$\mathcal{R}/q \times \mathsf{Short}$ multiplication	5,604,929	2,230
$\mathcal{R}/q \times \mathsf{Short}$ multiplication (PF)	740,980	2,812
$\mathcal{R}/3 \times \mathcal{R}/3$ multiplication	1,277,675	1,510
Full encapsulation	8,160,665	8,694
Full decapsulation	15,602,748	11,478
Full decapsulation (PF)	10,754,219	14,370

(for all three parameter sets of Streamlined NTRU Prime), i.e. Algorithm 1 can be slightly optimized. We refer to [7] for an in-depth description of the original product-form multiplication for 8-bit AVR. As explained in Sect. 2, the decapsulation of NTRU Prime includes as first step a multiplication of a polynomial that is an element of $\mathcal{R}/q$ by a ternary polynomial of fixed weight, namely the polynomial $f(x) \in \mathsf{Short}$. This multiplication can be accelerated by using the product-form technique described above when $f(x)$ is generated accordingly.

4 Results and Comparison

The 8-bit AVR device we used to test and benchmark our NTRU Prime implementation is an ATmega1284 microcontroller, which features 16 kB SRAM and 128 kB flash memory for storing program code. Our software consists of a mix of C and assembly language; we implement the main arithmetic operations in assembly to achieve fast and operand-independent execution time, whereas all functions that are neither performance-critical nor security-critical are written in C to maximize portability. We use the optimized Assembler implementation of the SHA-512 hash function introduced in [6] to minimize the execution time of certain auxiliary functions that are performance-critical. When executed on our target device, the compression function of SHA-512 takes slightly less than 60 k clock cycles, which corresponds to a compression rate of about 467 cycles per byte. Our implementation of (Streamlined) NTRU Prime can be compiled with Atmel Studio v7.0 under the `-O2` optimization option, which produces an executable that, according to our experiments, does not leak secret information through execution time and can, therefore, withstand timing attacks.

Table 2 summarizes the execution time and code size of the core arithmetic operations (i.e. polynomial multiplications) as well as a full encapsulation and decapsulation of our NTRU Prime software. The table shows the results of two implementations of the polynomial multiplication of the form $\mathcal{R}/q \times \mathsf{Short}$; the first uses a combination of Karatsuba's algorithm and product scanning at the lower level (see Subsect. 3.1), whereas the second is based on the product-form approach (see Subsect. 3.2). The results in Table 2 show that the product-form

Table 3. Comparison of our NTRU Prime implementation with other post-quantum key-establishment algorithms and ECC (all of which target 128 bits of security).

Implementation	Algorithm	Platform	Encaps.	Decaps.
This work	NTRU Prime	ATmega1284	8,160,665	15,602,748
Cheng et al [7]	NTRU (PF)	ATmega1281	847,973	1,051,871
Düll et al [8]	ECC-255	ATmega2560	13,900,397	13,900,397
Kannwischer et al [16]	NTRU Prime	Cortex M4	54,942,173	166,481,625
Kannwischer et al [16]	Frodo	Cortex M4	45,883,334	45,366,065
Kannwischer et al [16]	NewHope	Cortex M4	1,903,231	1,927,505
Kannwischer et al [16]	Kyber	Cortex M4	652,769	621,245
Kannwischer et al [16]	NTRU	Cortex M4	645,329	542,439

multiplication is significantly faster; it outperforms the Karatsuba-based multiplication by a factor of 7.56. On the other hand, these two implementations differ only marginally in terms of binary code size. The implementation of the $\mathcal{R}/3 \times \mathcal{R}/3$ polynomial multiplication combines Karatsuba's method with the hybrid technique and is much faster than the polynomial multiplication of the form $\mathcal{R}/q \times \mathsf{Short}$. This reduced running time is due to the smaller coefficients (enabling faster coefficient multiplication), smaller intermediate results (requiring fewer registers) and faster reduction (modulo 3 vs. modulo q). Also given in Table 2 are the execution times of encapsulation and decapsulation, which are primarily dominated by the polynomial arithmetic. The encapsulation includes just a single multiplication, namely a multiplication of an element of $\mathcal{R}/q$ by an element of Short (i.e. $\mathcal{R}/q \times \mathsf{Short}$) that accounts for roughly two thirds of the overall execution time. On the other hand, the decapsulation operation has to perform three polynomial multiplications (two of the form $\mathcal{R}/q \times \mathsf{Short}$ and one of the form $\mathcal{R}/3 \times \mathcal{R}/3$); together they contribute 80% to the overall execution time. The first $\mathcal{R}/q \times \mathsf{Short}$ multiplication, i.e. the multiplication of $c(x)$ by the ternary polynomial $f(x) \in \mathsf{Short}$, can be accelerated through the product-form technique, which reduces the execution time from 15.6 to 10.8 million cycles. In other words, product-form multiplication makes a decapsulation 31% faster.

Our software is, to the best of our knowledge, the first optimized implementation of Streamlined NTRU Prime for constrained devices. The only previous implementation of NTRU Prime for microcontrollers published in the literature is the implementation from `pqm4` [16], which is essentially the reference C code without any assembler optimizations. Compared with the `pqm4` timings on an ARM Cortex-M4, our implementation is 6.7 times faster for encapsulation and 10.7 times faster for decapsulation (see Table 3). However, it needs to be taken into account that a 32-bit ARM Cortex-M4 is significantly more powerful than an 8-bit AVR microcontroller. The AVR assembler implementation of classical NTRU (i.e. NTRUEncrypt with `ees443ep1` parameters) introduced in [7] uses a highly efficient product-form convolution and outperforms our NTRU Prime software by roughly an order of magnitude. On the other hand, when compared with ECC, our NTRU Prime encapsulation is much faster than a variable-base

scalar multiplication on Curve25519, while the decapsulation is a bit slower. Due to the limited number of state-of-the-art implementations of other NIST PQC candidates for 8-bit AVR, we give in Table 3 also a few recent results from the `pqm4` library for 32-bit ARM Cortex-M4 microcontrollers.

5 Conclusions

We presented the first highly-optimized implementation of NTRU Prime for an 8-bit microcontroller that is capable to resist timing attacks. When executed on an ATmega1284 device, the encapsulation takes about 8.2 million cycles, while the decapsulation has an execution time of 15.6 million cycles (both results are based on the parameter set `sntrup653`). For comparison, the reference C code from the designers requires 54.9 and 166.5 million cycles for encapsulation and decapsulation, respectively, on a much more powerful 32-bit Cortex-M4 microcontroller. To achieve these results, we implemented all expensive operations in AVR assembly language, most notably the polynomial arithmetic, whereby we strived for a balance between execution time and code size. We also discussed how the concept of product-form polynomials to speed up classical NTRU can be applied to NTRU Prime and demonstrated that product-form multiplication would make the decapsulation 30% faster. However, since a thorough analysis of the security implications of the product form in NTRU Prime is lacking, we do (currently) not recommend to use product-form polynomials in a real-world application. Furthermore, we showed that one cannot count on a C compiler to generate constant-time code for the modulo-3 reduction, which generally raises concerns about the security (i.e. resistance against timing attacks) of C implementations of NTRU Prime. In summary, our results show that NTRU Prime can be well optimized to run efficiently on small microcontrollers, which makes it an interesting candidate for securing the post-quantum IoT.

Acknowledgements. This work was supported by the European Union's Horizon 2020 research and innovation program under grant agreement No. 779391 (FutureTPM). The authors thank John Schanck for answering questions on the generation of product-form parameters for NTRU Prime. The research described in this paper was conducted before Daniel Dinu joined Intel and may not reflect the views of his current or previous employers.

References

1. E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Post-quantum key exchange – A new hope. In T. Holz and S. Savage, editors, *Proceedings of the 25th USENIX Security Symposium (USS 2016)*, pages 327–343. USENIX Association, 2016.
2. D. V. Bailey, D. Coffin, A. J. Elbirt, J. H. Silverman, and A. D. Woodbury. NTRU in constrained devices. In Ç. K. Koç, D. Naccache, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 262–272. Springer Verlag, 2001.

3. D. J. Bernstein, J. Buchmann, and E. Dahmen, editors. *Post-Quantum Cryptography*. Springer Verlag, 2009.
4. D. J. Bernstein, C. Chuengsatiansup, T. Lange, and C. van Vredendaal. NTRU Prime: Reducing attack surface at low cost. In C. Adams and J. Camenisch, editors, *Selected Areas in Cryptography — SAC 2017*, volume 10719 of *Lecture Notes in Computer Science*, pages 235–260. Springer Verlag, 2018.
5. D. J. Bernstein, C. Chuengsatiansup, T. Lange, and C. van Vredendaal. NTRU Prime: Round 2 specification, 2019. Available for download at `http://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions`.
6. H. Cheng, D. Dinu, and J. Großschädl. Efficient implementation of the SHA-512 hash function for 8-bit AVR microcontrollers. In J.-L. Lanet and C. Toma, editors, *Innovative Security Solutions for Information Technology and Communications — SecITC 2018*, volume 11359 of *Lecture Notes in Computer Science*, pages 273–287. Springer Verlag, 2019.
7. H. Cheng, J. Großschädl, P. B. Rønne, and P. Y. Ryan. A lightweight implementation of NTRUEncrypt for 8-bit AVR microcontrollers. In *Proceedings of the 2nd NIST PQC Standardization Conference*, 2019. Available online at `http://csrc.nist.gov/Events/2019/second-pqc-standardization-conference`.
8. M. Düll, B. Haase, G. Hinterwälder, M. Hutter, C. Paar, A. H. Sánchez, and P. Schwabe. High-speed Curve25519 on 8-bit, 16-bit and 32-bit microcontrollers. *Designs, Codes and Cryptography*, 77(2–3):493–514, Dec. 2015.
9. GCC Team. AVR-GCC Wiki. Available online at `http://gcc.gnu.org/wiki/avr-gcc#Exceptions_to_the_Calling_Convention`, 2017.
10. N. Gura, A. Patel, A. S. Wander, H. Eberle, and S. Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In M. Joye and J.-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer Verlag, 2004.
11. D. R. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Verlag, 2004.
12. J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A ring-based public key cryptosystem. In J. P. Buhler, editor, *Algorithmic Number Theory, Third International Symposium (ANTS-III)*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer Verlag, 1998.
13. J. Hoffstein and J. H. Silverman. Optimizations for NTRU. In K. Alster, J. Urbanowicz, and H. C. Williams, editors, *Public-Key Cryptography and Computational Number Theory*, De Gruyter Proceedings in Mathematics, pages 77–88. Walter de Gruyter, 2001.
14. J. Hoffstein and J. H. Silverman. Random small Hamming weight products with applications to cryptography. *Discrete Applied Mathematics*, 130(1):37–49, Aug. 2003.
15. M. J. Kannwischer, J. Rijneveld, and P. Schwabe. Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates. In R. H. Deng, V. Gauthier-Umaña, M. Ochoa, and M. Yung, editors, *Applied Cryptography and Network Security — ACNS 2019*, volume 11464 of *Lecture Notes in Computer Science*, pages 281–301. Springer Verlag, 2019.
16. M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen. `pqm4`: Testing and benchmarking NIST PQC on ARM Cortex-M4. Cryptology ePrint Archive, Report 2019/844, 2019. Available for download at `http://eprint.iacr.org`.
17. A. A. Karatsuba and Y. P. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics - Doklady*, 7(7):595–596, Jan. 1963.

18. A. Karmakar, J. M. Bermudo Mera, S. S. Roy, and I. Verbauwhede. Saber on ARM: CCA-secure module lattice-based key encapsulation on ARM. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):243–266, Aug. 2018.
19. P. R. Kaye, R. Laflamme, and M. Mosca. *An Introduction to Quantum Computing*. Oxford University Press, 2007.
20. V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. *Communications of the ACM*, 60(6):43:1–43:35, June 2013.
21. M. Mariantoni. Building a superconducting quantum computer. Invited presentation given at the 6th International Conference on Post-Quantum Cryptography (PQCrypto 2014), Waterloo, ON, Canada, Oct. 2014. Available online at `http://www.youtube.com/watch?v=wWHAs--HA1c`.
22. National Institute of Standards and Technology (NIST). NIST reveals 26 algorithms advancing to the post-quantum crypto 'semifinals'. Press release, available online at `http://www.nist.gov/news-events/news/2019/01/nist-reveals-26-algorithms-advancing-post-quantum-crypto-semifinals`, 2019.
23. J. M. Schanck. *Practical Lattice Cryptosystems: NTRUEncrypt and NTRUMLS*. M.Sc. Thesis, University of Waterloo, Waterloo, ON, Canada, 2015.
24. M. Scott. Missing a trick: Karatsuba variations. *Cryptography and Communications*, 10(1):5–15, Jan. 2018.
25. P. W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 124–134. IEEE Computer Society Press, 1994.
26. B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN 2005)*, pages 477–482. IEEE, 2005.
27. A. L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics - Doklady*, 4(3):714–716, May 1963.