Efficient Multiplication of Somewhat Small Integers using Number-Theoretic Transforms

Hanno Becker[1], Vincent Hwang[2,3], Matthias J. Kannwischer[3], Lorenz Panny[3], and Bo-Yin Yang[3]

[1] Arm Research, Cambridge, UK
`hanno.becker@arm.com`
[2] National Taiwan University, Taipei, Taiwan
`vincentvbh7@gmail.com`
[3] Academia Sinica, Taipei, Taiwan
`matthias@kannwischer.eu`, `lorenz@yx7.cc`, `by@crypto.tw`

Abstract. Conventional wisdom purports that FFT-based integer multiplication methods (such as the Schönhage–Strassen algorithm) begin to compete with Karatsuba and Toom–Cook only for integers of several tens of thousands of bits. In this work, we challenge this belief, leveraging recent advances in the implementation of number-theoretic transforms (NTT) stimulated by their use in post-quantum cryptography. We report on implementations of NTT-based integer arithmetic on two Arm Cortex-M CPUs on opposite ends of the performance spectrum: Cortex-M3 and Cortex-M55. Our results indicate that NTT-based multiplication is capable of outperforming the big-number arithmetic implementations of popular embedded cryptography libraries for integers as small as 2048 bits. To provide a realistic case study, we benchmark implementations of the RSA encryption and decryption operations. Our cycle counts on Cortex-M55 are about 10× lower than on Cortex-M3.

Keywords: FFT-based multiplication · NTT · Arm processors · RSA

1 Introduction

The development of fast algorithms for arithmetic on big numbers is a well-established field of research. As with any computational problem, its study can be dissected into two parts: First, the analysis of the *asymptotic* complexity. Second, the analysis of *concrete* complexity for a chosen size of input. The results are often different: An algorithm may have inferior asymptotic performance but superior practical performance for a certain input size. The analysis of the "crossover point", that is, the input size at which an asymptotically faster algorithm also becomes practically faster, is an important question when moving from theory to practice. The present paper is about the evaluation of such a crossover point in the case of big number arithmetic on microcontrollers.

[*] Date of this document: 2022-10-21.

The multiplication of big numbers can be performed in a variety of ways of decreasing asymptotic complexity and (unsurprisingly) increasing sophistication. At the base, so-called “schoolbook multiplication” approaches calculate the product of two n-limb numbers $(a_0, \ldots, a_{n-1})$ and $(b_0, \ldots, b_{n-1})$ by computing and accumulating all n^2 subproducts $a_i b_j$. While from a practical perspective, a lot of research has been conducted on the optimal *concrete* strategy, they all lead to an asymptotic complexity of $\mathcal{O}(n^2)$. Next, the Karatsuba method [KO63] and its generalization by Toom–Cook [Too63] lower the asymptotic complexity to $\mathcal{O}(n^{1+s})$ for varying $0 < s < 1$; for example, Karatsuba's method of computing

$$(a_0 + ta_1)(b_0 + tb_1) = a_0 b_0 + t^2 a_1 b_1 + t((a_0 + a_1)(b_0 + b_1) - a_0 a_0 - a_1 b_1)$$

leads to an asymptotic complexity of $\mathcal{O}(n^{\log_2 3}) \subseteq \mathcal{O}(n^{1.585})$. Moving further, starting with the famous Schönhage–Strassen algorithms [SS71], FFT-based integer multiplications achieve asymptotic complexity $\mathcal{O}(n \log n \log \log n)$ and better, and the long conjectured (and presumably final) complexity of $\mathcal{O}(n \log n)$ was only recently achieved in [HH21].

Despite its far superior asymptotic complexity, however, NTT-based integer multiplication is not used for number ranges found in contemporary public-key cryptography: In fact, quadratic multiplication strategies appear to be the most prominent choice in those contexts. At the same time, the past years have seen significant research and progress regarding fast implementation of the NTT, stimulated by their prominence in post-quantum cryptography. The primary objective of this paper is to evaluate how those optimizations affect the practical performance and viability of NTT-based big number arithmetic.

1.1 Results

We find that the crossover point for viability of NTT-based modular arithmetic is at around 2048 bits. More precisely, we compare to modular arithmetic implementations found in the popular TLS libraries BearSSL and Mbed TLS, and find that our NTT-based implementation outperforms both by 1.3×–2.2× on Cortex-M3 and by 1.8×–6.4× on Cortex-M55. We also notice that there is considerable optimization potential for the schoolbook multiplications in BearSSL and Mbed TLS — when this is implemented, 2048-bit NTT-based modular multiplication is only slightly better (1.1×) than schoolbook multiplication on Cortex-M3, and essentially equal on Cortex-M55. When moving to 4096-bit multiplication, however, our NTT-based implementation outperforms even those highly optimized schoolbook multiplications. We thus think that NTT-based modular arithmetic should be considered from 2048-bit onwards.

Software: All our Cortex-M3 code is available at `https://github.com/ntt-int-mul/ntt-int-mul-m3`. Our Cortex-M55 integer-multiplication code is available at `https://gitlab.com/arm-research/security/pqmx`.

Related work. Present-day general-purpose computer algebra systems switch to FFT-based multiplication only for very large numbers. For example, GMP [GMP] uses Schönhage–Strassen when multiplying numbers with more than 3000–10000 limbs (i.e., at least 96 000 bits) depending on the platform.[4] However, when tailoring an implementation to a specific integer size and platform, the crossover point appears to be lower. Previous work on implementing RSA using Schönhage–Strassen [GKZ07] in hardware concluded that it can only outperform Karatsuba and Toom–Cook for key sizes larger than 48 000 bits. [Gar07] reports similar findings: It estimates Schönhage–Strassen to be competitive only for RSA key sizes above $2^{17} \approx 131\,000$ bits, several orders of magnitude beyond typical RSA parameter choices. To the best of our knowledge, there is no competitive implementation of real-world RSA using FFT-based integer multiplication.

Other work. In addition to improvements to the efficiency of number-theoretic transforms, post-quantum cryptography has stimulated research into efficient schoolbook multiplication strategies for integers of a few hundred bits, as found in elliptic-curve or isogeny cryptography. It would be interesting to study and compare the performance of RSA based on the combination of Karatsuba and those new quadratic multiplication algorithms. Another avenue for further research is the evaluation of NTT-based arithmetic on high-end processors.

2 Preliminaries

2.1 RSA

The RSA (Rivest–Shamir–Adleman) cryptosystem [RSA78] was the most common public-key cryptosystem for decades and remains in widespread use, primarily with keys of 2048, 3072, or 4096 bits. We briefly recap how it works.

During key generation, a semiprime $N = pq$ with p and q of roughly equal size is generated. The public key is N and a small e to which power it is easy to raise, commonly $e = 2^{16} + 1$. We have $x^{k\phi(N)+1} \equiv x \pmod{N}$ for all x, k, where $\phi(N) = (p-1)(q-1)$ is the totient function. With $d \equiv e^{-1} \pmod{\phi(N)}$, the public map $x \mapsto x^e \bmod N$ is then inverted by the secret map $y \mapsto y^d \bmod N$, the secret key being d. Both encryption and signing primitives can be constructed based on this pair of public/private maps.

The private map can be evaluated using the Chinese Remainder Theorem (CRT) method, computing $x = y^d \bmod N$ by interpolating $x \equiv y^{d \bmod (p-1)} \pmod{p}$ and $x \equiv y^{d \bmod (q-1)} \pmod{q}$. Modular multiplications are commonly implemented using Montgomery multiplication, and modular exponentiation uses windowing methods (see Section 3).

2.2 FFT-based integer multiplication

Numerous versions of FFT-based integer multiplications are known, but their blueprint is typically the following: First, find an FFT-based quasi-linear time

[4] https://gmplib.org/manual/FFT-Multiplication

multiplication algorithm in a suitable polynomial ring. Second, find a means to reduce integer multiplication to the chosen kind of polynomial multiplications.

Starting with Schönhage–Strassen and Pollard [SS71; Pol71], numerous instantiations of this idea have been developed, using polynomials over $\mathbb{C}$, finite fields $\mathbb{F}_q$, integers modulo Fermat numbers $\mathbb{Z}/(2^{2^n}+1)\mathbb{Z}$, and also multivariate polynomial rings [HH21]. Here, we focus on NTT-based integer multiplication using polynomials in $\mathbb{Z}_q[X]/(X^n-1)$ with q a prime or bi-prime, which is close to [Pol71]. While variable-size integer multiplication requires recursive application of the above principle, it is not necessary for the integer sizes considered here.

Section 2.3 discusses how the NTT yields a quasi-linear multiplication in $\mathbb{Z}_q[X]/(X^n-1)$. We now explain the reduction from integer multiplication.

To turn a multiplication of $a, b \in \mathbb{Z}$ into a multiplication in $\mathbb{Z}_q[X]/(X^n-1)$, one first lifts a, b to integer *polynomials* $A, B \in \mathbb{Z}[X]$ along $f : \mathbb{Z}[X] \to \mathbb{Z}, X \mapsto 2^\ell$, the canonical choice being the radix-2^ℓ presentations of a, b. Since $f(AB) = ab$, it suffices to compute $AB \in \mathbb{Z}[X]$. To do so, one chooses q and n such that $AB \in \mathbb{Z}[X]$ is a canonical representative for the finite quotient $\mathbb{Z}_q[X]/(X^n-1)$, that is, it is of degree $< n$ with coefficients in $\{0, \ldots, q-1\}$. Under these circumstances, one can then uniquely recover AB from its image $g(AB) = g(A)g(B)$ under $g : \mathbb{Z}[X] \to \mathbb{Z}_q[X]/(X^n-1)$. We have thus reduced the computation of ab in $\mathbb{Z}$ to that of $g(A)g(B)$ in $\mathbb{Z}_q[X]/(X^n-1)$.

2.3 Number-theoretic transforms

The *number-theoretic transform* (NTT) is a generalization of the discrete Fourier transform, replacing the base ring $\mathbb{C}$ of the complex numbers by other commutative rings, commonly finite fields $\mathbb{F}_q$. In the present context, its value lies in the fact that it transforms convolutions into pointwise products in quasi-linear time, reducing the complexity of convolutions from quadratic to quasi-linear.

Definition. We're working over $\mathbb{Z}_q := \mathbb{Z}/q\mathbb{Z}$ for odd q and fix $\omega \in \mathbb{Z}_q$ an nth root of unity. We write $[n] := \{0, 1, \ldots, n-1\}$. The NTT [Für09; HH21] is the canonical projection $\mathbb{Z}_q[x]/\langle x^n - 1\rangle \to \prod_i \mathbb{Z}_q[x]/\langle x - \omega^i\rangle$, which under the isomorphism $\mathbb{Z}_q[x]/\langle x - \omega^i\rangle \cong \mathbb{Z}_q, \boldsymbol{a}(x) \mapsto \boldsymbol{a}(\omega^i)$ can also be described as

$$\texttt{NTT} : \mathbb{Z}_q[x]/\langle x^n - 1\rangle \to \mathbb{Z}_q^n, \quad \texttt{NTT}(a) := \left(\boldsymbol{a}(1), \boldsymbol{a}(\omega), \ldots, \boldsymbol{a}(\omega^{n-1})\right).$$

If ω is a principal nth root of unity and n is invertible in $\mathbb{Z}_q$, this constitutes a ring isomorphism $\texttt{NTT} : \mathbb{Z}_q[x]/\langle x^n - 1\rangle \cong \mathbb{Z}_q^n$; in particular, we have $ab = \texttt{NTT}^{-1}\left(\texttt{NTT}(a) \cdot_\Pi \texttt{NTT}(b)\right)$, where $\cdot_\Pi$ is the pointwise multiplication in $\mathbb{Z}_q^n$.

Fourier inversion. Domain and codomain of the NTT can be identified via the isomorphism of $\mathbb{Z}_q$-modules (not rings) $\varphi : \mathbb{Z}_q[x]/\langle x^n - 1\rangle \cong \mathbb{Z}_q^n$, $x^i \leftrightarrow e_i$ (where e_i is the ith unit vector). This renders the resulting $\texttt{NTT} : \mathbb{Z}_q^n \to \mathbb{Z}_q^n$ close to an involution: $\texttt{NTT}^2 = \mathrm{mul}_n \circ \texttt{neg}$, where $\mathrm{mul}_n : \mathbb{Z}_q^n \to \mathbb{Z}_q^n$ is pointwise multiplication with n and $\texttt{neg} : \mathbb{Z}_q^n \to \mathbb{Z}_q^n$ sends e_i to $e_{\texttt{neg}(i)}$ with $\texttt{neg}(0) = 0$ and $\texttt{neg}(i) = n - i$ for $i > 0$ (we don't distinguish between a permutation on

$[n]$ and the induced isomorphism on $\mathbb{Z}_q^n$). Another way of saying this is that $\mathtt{NTT}' : \mathbb{Z}_q[x]/\langle x^n - 1\rangle \cong \mathbb{Z}_q^n$ defined by $\mathtt{NTT}'(a) := \big(\boldsymbol{a}(1), \boldsymbol{a}(\omega^{-1}), \ldots, \boldsymbol{a}(\omega^{-(n-1)})\big)$ is, up to multiplication by n and application of φ, the inverse of $\mathtt{NTT} : \mathbb{Z}_q[x]/\langle x^n - 1\rangle \cong \mathbb{Z}_q^n$. This is the *Fourier Inversion Formula*, and the curious reader will find that it boils down to the orthogonality relations $\sum_j \omega^{ij} = n \cdot \delta_{i,0}$.

Fast Fourier transform. The NTT can be calculated using the Cooley–Tukey (CT) FFT algorithm: For $n = 2m$, CT splits $\mathbb{Z}_q[x]/\langle x^{2m} - \zeta^2\rangle$ into $\mathbb{Z}_q[x]/\langle x^m - \zeta\rangle \times \mathbb{Z}_q[x]/\langle x^m + \zeta\rangle$ via $\mathrm{CT}(a + x^m b, \zeta) = (a + \zeta b, a - \zeta b)$ for a, b of degree $< m$ — this is called a *CT butterfly.* The idea can be applied recursively, and for $n = 2^k$ we in particular obtain a map $\mathtt{NTT}_{\mathtt{CT}} : \mathbb{Z}_q[x]/\langle x^n - 1\rangle \cong \mathbb{Z}_q^n$ which is equal to $\mathtt{bitrev} \circ \mathtt{NTT}$, where $\mathtt{bitrev} : [2^k] \to [2^k]$ is the bitreversal permutation.

The CT strategy can also be applied for radices $r \neq 2$, performing one splitting $\mathbb{Z}_q[x]/\langle x^{rm} - \zeta^r\rangle \cong \prod_i \mathbb{Z}_q[x]/\langle x^m - \omega_r^i \zeta\rangle$ at a time. When applied recursively to a factorization $n = r_1 \cdots r_s$, the resulting map $\mathtt{NTT}_{\mathtt{CT}} : \mathbb{Z}_q[x]/\langle x^n - 1\rangle \cong \mathbb{Z}_q^n$ agrees with $\sigma(r_1, \ldots, r_s) \circ \mathtt{NTT}$, where $\sigma(r_1, \ldots, r_s)$ is given by

$$[n] \quad \cong \quad [r_1] \times \ldots \times [r_s] \xrightarrow{\text{reverse}} [r_s] \times \ldots \times [r_1] \quad \cong \quad [n]$$

where the first and last map are lexicographic orderings. Note that $\sigma(2, \ldots, 2) = \mathtt{bitrev}$, and $\sigma(r_1, \ldots, r_s)$ is an involution only if $(r_1, \ldots, r_s)$ is a palindrome.

Inverse NTT. For the computation of $\mathtt{NTT}_{\mathtt{CT}}^{-1}$, there are two approaches: First, one can invert CT butterflies via *Gentleman–Sande butterflies* $\mathrm{GS}(a, b, \zeta) = (a + b, (a - b)\zeta)$. Alternatively, one can leverage $\mathtt{NTT}_{\mathtt{CT}} = \sigma \circ \mathtt{NTT}$ and $\mathtt{NTT}^{-1} = \mathrm{mul}_{1/n} \circ \mathtt{NTT}'$ to compute $\mathtt{NTT}_{\mathtt{CT}}^{-1} = \mathrm{mul}_{1/n} \circ \mathtt{NTT}' \circ \sigma^{-1} = \mathrm{mul}_{1/n} \circ \sigma^{-1} \circ \mathtt{NTT}'_{\mathtt{CT}} \circ \sigma^{-1}$. If σ is an involution (e.g., if $n = 2^k$), this is $\mathrm{mul}_{1/n} \circ \sigma \circ \mathtt{NTT}'_{\mathtt{CT}} \circ \sigma^{-1}$ and can thus be implemented like $\mathtt{NTT}_{\mathtt{CT}}$ while implicitly applying the permutation σ; this leads to the implementation of $\mathtt{NTT}_{\mathtt{CT}}^{-1}$ as presented in [Abd+22, Figure 1], which does not require explicit permutations. For a general mixed-radix NTT, however, σ is not an involution, and an explicit permutation by σ^{-2} is needed; we avoid this via Good's trick, as explained in the next section.

GS butterflies lead to exponential growth for an exponentially shrinking number of coefficients, while CT butterflies yield linear growth for *all* coefficients. This impacts the amount and placement of reductions during $\mathtt{NTT}^{\pm 1}$.

Good's trick. For $n = rs$ with coprime r, s, another strategy to computing $\mathtt{NTT}_n$ is computing the bottom edge in the commutative diagram

$$\begin{tikzcd}
\mathbb{Z}_q[x]/\langle x^n - 1\rangle \arrow[rr, "\mathtt{NTT}_n^{\omega}"'] \arrow[d, "u^i \otimes v^j \leftrightarrow x^k", "\cong"', "\begin{smallmatrix} k\equiv i \pmod r \\ k\equiv j \pmod s \end{smallmatrix}"'] & & \mathbb{Z}_q^n \arrow[d, dash, "e_i \otimes e_j \leftrightarrow e_k"', "\cong"', "\begin{smallmatrix} k\equiv i \pmod r \\ k\equiv j \pmod s \end{smallmatrix}"] \\
\mathbb{Z}_q[u]/\langle u^r - 1\rangle \otimes_{\mathbb{Z}_q} \mathbb{Z}_q[v]/\langle v^s - 1\rangle \arrow[rr, "\mathtt{NTT}_r^{\omega^e} \otimes \mathtt{NTT}_s^{\omega^f}", "\begin{smallmatrix} e\equiv 0 \pmod s,\ e\equiv 1 \pmod r \\ f\equiv 1 \pmod s,\ f\equiv 0 \pmod r \end{smallmatrix}"'] & & \mathbb{Z}_q^r \otimes_{\mathbb{Z}_q} \mathbb{Z}_q^s
\end{tikzcd}$$

This has two benefits: First, if r, s are prime powers then $\mathtt{NTT}^{\pm 1}_{r/s}$ can be computed via CT as described above, avoiding non-involutive permutations. Second, fewer twiddle factors are needed for the computation of $\mathtt{NTT}_s \otimes \mathtt{NTT}_r$.

Incomplete NTTs. Denoting $R := \mathbb{Z}_q[x]/\langle x^n - 1\rangle$ and $R_i := \mathbb{Z}_q[x]/\langle x - \omega^i\rangle$, the NTT splitting $\mathtt{NTT} : R \xrightarrow{\cong} \prod_i R_i$ transfers to any R-algebra: If S is an R-algebra, we have $S \cong S \otimes_R R \cong S \otimes_R \prod_i R_i \cong \prod_i S \otimes_R R_i$. The most common example are *incomplete NTTs*: The ring $S := \mathbb{Z}_q[y]/\langle y^{nh} - 1\rangle$ is an algebra over its subring $R := \mathbb{Z}_q[y^h]/\langle y^{nh} - 1\rangle \cong \mathbb{Z}_q[x]/\langle x^n - 1\rangle$ to which the NTT applies, and so $S \cong \prod_i S \otimes_R \mathbb{Z}_q[y^h]/\langle y^h - \omega^i\rangle = \prod_i \mathbb{Z}_q[y]/\langle y^h - \omega^i\rangle$.

The benefits of using incomplete NTTs are: First, we only need an nth principal root of unity to partially split $\mathbb{Z}_q[y]/\langle y^{nh} - 1\rangle$. Second, polynomial multiplication using incomplete NTTs and "base multiplication" in $\mathbb{Z}_q[y]/\langle y^h - \omega^i\rangle$ may be faster than for full NTTs and base multiplication in $\mathbb{Z}_q$.

We use incomplete NTTs for all parameter sets — see below.

Fermat number transforms. The Fermat number transform (FNT) is a special case of NTT where the modulus is a Fermat number $F_t := 2^{2^t} + 1$ [AB74]. For the coefficient ring $\mathbb{Z}_{F_t}$, we can compute a size-n NTT if n divides 2^{t+2}. If we choose 2 to be the principal 2^{t+1}th root of unity, then the twiddle factors for a size-$(t+1)$ Cooley–Tukey FFT are all powers of 2.

Since there are square roots for ± 2, we can choose a principal 2^{t+2}th root of unity ω with $\omega = \sqrt{2}$ and compute a size-2^{t+2} NTT [AB74]. Furthermore, if F_t is a prime, then we can compute a size-2^{2^t} NTT. Note that the only known prime Fermat numbers are $F_0, \ldots, F_4$.

2.4 Modular reductions and multiplications

(Refined) Barrett reduction. Signed Barrett reduction approximates

$$a \bmod^{\pm} q = a - q\left\lfloor a/q \right\rceil = a - q\left\lfloor a\tfrac{\mathtt{R}}{q}/\mathtt{R} \right\rceil \approx a - \left\lfloor a \cdot [\![\mathtt{R}/q]\!]/\mathtt{R} \right\rceil =: \mathbf{bar}^{[\![\ ]\!]}_{q,\mathtt{R}}(a),$$

where $\mathtt{R} = 2^w$ is a power of 2 and $[\![\mathtt{R}/q]\!]$ is a precomputed integer approximation to $\frac{\mathtt{R}}{q}$. The quality of the resulting approximation $\mathbf{bar}^{[\![\ ]\!]}_{q,\mathtt{R}}(a) \approx a \bmod^{\pm} q$ — and in particular, the question of when it may in fact be an *equality* — depends on the value of w, and two choices for w are common, as we now recall.

First, $w = M$ where $M \in \{16, 32\}$ is the word or half-word size bitlength, allowing $\left\lfloor \frac{-}{\mathtt{R}} \right\rceil$ to be conveniently implemented using rounding high multiply instructions. We call this the "standard" Barrett reduction.

Second, $w = (M - 1) + \lfloor \log_2 q \rfloor$, which is maximal under the constraint that $[\![\mathtt{R}/q]\!]$ is a signed M-bit integer: This choice leads to higher accuracy of the approximation, but typically requires an additional instruction. We will henceforth call it the "refined" Barrett reduction. For standard Barrett reduction, both $[\![a]\!] := 2\left\lfloor \frac{a}{2} \right\rceil$ and $[\![a]\!] := \lfloor a \rceil$ can be useful, while for refined Barrett reduction, we always choose $[\![a]\!] = \lfloor a \rceil$ because of its tighter bound $|\lfloor a \rceil - a| \leq \frac{1}{2}$.

Note that both "standard" and "refined" Barrett reductions are already known in the literature as Barrett reduction. We make this distinction for introducing an extension of the signed Barrett multiplication introduced by [Bec+22a].

(Refined) Barrett multiplication. For two integers a, b and a modulus q, signed Barrett multiplication [Bec+22a] approximates

$$ab \bmod^{\pm} q = ab - q\left\lfloor \frac{ab}{q} \right\rceil = ab - q\left\lfloor a\frac{b\mathtt{R}}{q}/q \right\rceil \approx ab - \left\lfloor a \cdot \left[\!\left[\frac{b\mathtt{R}}{q} \right]\!\right] / \mathtt{R} \right\rceil q =: \mathbf{bar}^{[\![\ ]\!]}_{q,\mathtt{R}}(a,b),$$

where again $\mathtt{R} = 2^w$ is a power of 2 and $[\![b\mathtt{R}/q]\!]$ is a precomputed integer approximation to $\frac{b\mathtt{R}}{q}$. Previously, only the choice $w = M \in \{16, 32\}$ was considered. In analogy with refined Barrett reduction, we suggest to also consider $w = (M-1) + \lfloor \log_2 q \rfloor - \lceil \log_2 |b| \rceil$, which again is maximal under the constraint that $[\![b\mathtt{R}/q]\!]$ is a signed M-bit integer. We call the resulting approximation to $ab \bmod^{\pm} q$ the "refined" Barrett multiplication.

We summarize the quality and size of Barrett reduction and multiplication:

Fact 1. Let $q \in \mathbb{N}$ be odd and $a, b \in \mathbb{Z}$ with $|a|, |b| < 2^{M-1}$ for $M \in \{16, 32\}$. Moreover, let $[\![-]\!] : \mathbb{Q} \to \mathbb{Z}$ be any integer approximation, i.e. $|x - [\![x]\!]| \leq 1$ for all $x \in \mathbb{Q}$, and put $t \bmod^{[\![\ ]\!]} q := t - q\,[\![t/q]\!]$.

Then for $\mathtt{R} := 2^M$ we have $|\mathbf{bar}^{[\![\ ]\!]}_{q,\mathtt{R}}(a,b)| \leq \frac{a(b\mathtt{R} \bmod^{[\![\ ]\!]} q)}{\mathtt{R}} + \frac{\mathtt{R}}{2}$.

Proof. [Bec+22a, Corollary 2] □

Fact 2. Let $q \in \mathbb{N}$ be odd and $a, b \in \mathbb{Z}$ with $|a|, |b| < 2^{M-1}$ for $M \in \{16, 32\}$. Moreover, pick $k \geq 1$ maximal s.t. $\varepsilon := |\lfloor b\mathtt{R}/q \rceil - b\mathtt{R}/q| \leq 2^{-k}$. Finally, set $\mathtt{R} := 2^w$ for $w := (M-1) + \lfloor \log_2 q \rfloor - \lceil \log_2 |b| \rceil$. Then:

If $\log_2 |a| < (M-1) - (\lceil \log_2 |b| \rceil - (k-1))$, then $\mathbf{bar}^{[\![\ ]\!]}_{q,\mathtt{R}}(a,b) = ab \bmod^{\pm} q$.

Restating Fact 2 in simple terms: Refined Barrett *reduction* (the special case $b = 1$) yields canonical representatives for *all* inputs a with $|a| < 2^{M-1}$. For a refined Barrett multiplication, the range of inputs for which $\mathbf{bar}^{[\![\ ]\!]}_{q,\mathtt{R}}(a,b)$ is guaranteed to be canonical is narrowed by the bitwidth of b; *however*, this can be compensated for by an exceptionally close approximation $b\mathtt{R}/q \approx \lfloor b\mathtt{R}/q \rceil$.

Proof of Fact 2. Setting $\delta := a\lfloor b\mathtt{R}/q \rceil / \mathtt{R} - ab/q$, it follows from the definition of ε and k that $|\delta| \leq |a|/2^{k+w}$. Since $\lfloor - \rceil$ changes its value only when crossing values of the form $\{\frac{2n+1}{2}\}$ for $n \in \mathbb{Z}$, for $\left\lfloor \frac{ab}{q} \right\rceil$ and $\left\lfloor \frac{a\lfloor \frac{b\mathtt{R}}{q} \rceil}{\mathtt{R}} \right\rceil = \left\lfloor \frac{ab}{q} + \delta \right\rceil$ to agree it is sufficient to show that $|\delta| < \min\left\{ \left| \frac{2n+1}{2} - \frac{c}{q} \right| \mid c, n \in \mathbb{Z} \right\} = \frac{1}{2q}$—the last equality holds since q is odd. Refined Barrett multiplication is thus guaranteed to yield the canonical representative of ab if $\frac{|a|}{2^{k+w}} < \frac{1}{2q}$, i.e. $|a| < \frac{2^{k+w-1}}{q}$. Plugging in $w = M - 1 + \lfloor \log_2 q \rfloor - \lceil \log_2 |b| \rceil$ and estimating $q < 2^{\lfloor \log_2 |q| \rfloor + 1}$, this follows provided $\log_2 |a| < (M-1) - (\lceil \log_2 |b| \rceil - (k-1))$, as claimed. □

Example 1. Let $M = 32$, $q = 114826273$, and $b = 774$. Then $\lfloor \log_2 q \rfloor = 26$ and $\lceil \log_2 b \rceil = 10$, so $w = 47$. Moreover, $\varepsilon := |\lfloor b\mathtt{R}/q \rceil - b\mathtt{R}/q|$ satisfies $\varepsilon < 2^{-11}$. Thus, according to Fact 2, the refined Barrett multiplication $\mathbf{bar}^{\pm}_{q,\mathtt{R}}(-, b)$ for $\mathtt{R} := 2^{47}$ does therefore yield canonical representatives for all inputs a with $|a| < 2^{31}$: The exceptionally good approximation $\lfloor b\mathtt{R}/q \rceil \approx b\mathtt{R}/q$ makes up for the size of b.

Montgomery multiplication. The Montgomery multiplication [Mon85] of a, b with respect to a modulus q and a 2-power $\mathtt{R} > q$ is defined as $\mathbf{mont}^{+}_{q}(ab) = \mathtt{hi}\,(a \cdot b + q \cdot \mathtt{lo}\,(q' \cdot \mathtt{lo}\,(a \cdot b)))$, providing a representative of $ab\mathtt{R}^{-1}$ modulo q. Here, $q' = -q^{-1} \bmod \mathtt{R}$, and $\mathtt{lo}$ and $\mathtt{hi}$ are extractions of the lower and upper $\log_2 \mathtt{R}$ bits, respectively. Montgomery multiplication is defined and relevant for both small-width modular arithmetic such as modular arithmetic modulo a 16-bit or 32-bit prime, as well as large integer arithmetic as used, e.g., in RSA.

Multi-precision Montgomery multiplication. Montgomery multiplication for big integers is implemented iteratively: For $a, b = \sum_i b_i \mathtt{B}^i$, one computes a representative of $ab\mathtt{B}^{-n}$ by writing $ab\mathtt{B}^{-n} = \ldots (ab_2 + (ab_1 + (ab_0)\mathtt{B}^{-1})\mathtt{B}^{-1})\mathtt{B}^{-1} \ldots$ and computing each $x \mapsto (x + ab_i)\mathtt{B}^{-1}$ using a Montgomery multiplication w.r.t. $\mathtt{B}$. Each such step involves the computation and accumulation of $P = x + ab_i$ and of $Q = ((x+ab_i)_0 q' \bmod \mathtt{B})p$. If the products are computed separately, this is called *Coarsely Integrated Operand Scanning* (CIOS) [KAK96]. If $(x + ab_i)_0 q' \bmod \mathtt{B}$ is computed first and then $P + Q$ is computed in one loop, it is called *Finely Integrated Operand Scanning* (FIOS).

Divided-difference for Chinese remainder theorem (CRT). We compute polynomial products modulo $q_1 q_2$ by interpolating products modulo q_1 and q_2 using the divided-difference algorithm for CRT [Chu+21]: Let q_0, q_1 be two coprime integers and $m_1 := q_0^{-1} \bmod {}^{\pm}q_1$. For a system $u \equiv u_0 \pmod{q_0}, u \equiv u_1 \pmod{q_1}$ with $|u_0| < \frac{q_0}{2}, |u_1| < \frac{q_1}{2}$, we solve for u with $|u| < \frac{q_0 q_1}{2}$ by computing:

$$u = u_0 + \big((u_1 - u_0)m_1 \bmod {}^{\pm}q_1\big)\, q_0. \tag{1}$$

2.5 Implementation targets

We briefly explain our choice of implementation targets.

Cortex-M3 The Arm® Cortex®-M3 CPU is a low-cost processor found in a wide range of applications such as microcontrollers, automotive body systems, or wireless networking. It implements the Armv7-M architecture and features a 3-stage pipeline, an optional memory protection unit (MPU) and a single-cycle $32 \times 32 \to 32$-bit multiplier with optional 1-cycle accumulation or subtraction.

We select the Cortex-M3 primarily for two reasons: First, it is a popular choice of MCU for automotive hardware security modules (e.g. Infineon AURIX TC27X). Second, its $32 \times 32 \to 64$ long multiplication instructions `smull`, `smlal`, `umull`, `umlal` have data-dependent timing and lead to timing side channels when used to process sensitive data. To avoid those, implementations need to use single-width multiplication instructions `mul`, `mla`, and `mls` instead. We expect this reduction of basic multiplication width to have a more significant impact on the runtime of classical multiplication than on (quasi-linear) NTT-based multiplication. A goal of the paper is to evaluate this intuitive assessment.

Cortex-M55 The Cortex-M55 processor is the first implementation of the Armv8.1-M architecture, with optional support for the M-Profile Vector Extension (MVE), or Arm® Helium™ Technology. It features a 5-stage pipeline when Helium is enabled, and except for some pairs of Thumb instructions, it is single issue. In addition to the Helium vector extension, it supports the Low Overhead Branch Extension, as well as tightly coupled memory (TCM) for both code and data, with a total Data-TCM bandwidth of 128-bit/cycle, 64-bit/cycle for CPU processing and 64-bit/cycle for concurrent DMA transfers. For a more extensive introductions to both the Armv8.1-M architecture and the Cortex-M55 CPU, we refer to [Bec+22b, Section 3] and the references therein.

We select the Cortex-M55 for the following reasons: First, due to its support for SIMD vector processing, it is an exciting and powerful new implementation target — the cryptographic capabilities of which are still to be explored. Second, the authors are not aware of means to vectorize classical `umaal`-based multiplication strategies using MVE, while in contrast it has been demonstrated in [Bec+22b] that the NTT is amenable for significant speedup using MVE. We are thus curious to understand how a vectorized NTT-based integer multiplication fares compared to classical `umaal`-based integer multiplication.

3 Implementations

3.1 High-level strategy

We implement Montgomery multiplication on top of NTT-based large integer multiplication, the latter as described in Section 2.2. This is in contrast to CIOS/FIOS approaches for iterative Montgomery multiplication, which never need to compute the double-width product of two large integers.

We pick $\mathtt{R} = 2^{\ell \cdot n/2}$, which in contrast to $\mathtt{R} = 2^N$ aligns taking the low and high half w.r.t. $\mathtt{R}$ with taking the low resp. high halves of polynomials.

NTT-based large integer multiplication involves a considerable amount of precomputation, such as chunking and NTT. Since each Montgomery multiplication involves three integer multiplications — $a \cdot b$, $t := q' \cdot (a \cdot b)_{\mathrm{low}}$, and $p \cdot t$ — two of which involve static factors p and p', we buffer their precomputations. We also make use of asymmetric multiplication [Bec+22a] and refer to the resulting NTT and base multiplication as $\mathtt{NTT}_{\mathrm{heavy}}$ and $\mathtt{basemul}_{\mathrm{light}}$.

Algorithm 1, Algorithm 2 and Appendix E describe our modular multiplication strategy in more detail. Appendix B explains how to perform the non-trivial precomputation of $p^{-1} \bmod \mathtt{R}$ for our large choice of $\mathtt{R}$.

Algorithm 1:
Montgomery squaring using NTTs

```
Input: $p$, $aR \bmod p$,
       $\hat{p}^{-1} = \texttt{NTT}(\texttt{chk}(p^{-1} \bmod 2^k))$,
       $\hat{p} = \texttt{NTT}(\texttt{chk}(p))$
Output: $c = a^2 R \bmod p$
1: $\hat{a} = \texttt{NTT}(\texttt{chk}(a))$
2: $t = \texttt{dechk}(\texttt{NTT}^{-1}(\hat{a} \circ \hat{a}))$
3: $\hat{t} = \texttt{NTT}(\texttt{chk}(t \bmod 2^k))$
4: $l = \texttt{dechk}(\texttt{NTT}^{-1}(\hat{t} \circ \hat{p}^{-1}))$
5: $\hat{l} = \texttt{NTT}(\texttt{chk}(l \bmod 2^k))$
6: $r = \texttt{dechk}(\texttt{NTT}^{-1}(\hat{l} \circ \hat{p}))$
7: $c = \frac{t}{2^k} - \frac{r}{2^k}$
8: if $c < 0$ then $c = c + p$
9: return $c$
```

Algorithm 2:
Montgomery multiplication using NTTs

```
Input: $aR \bmod p$, $bR \bmod p$
       $\hat{p}^{-1} = \texttt{NTT}(\texttt{chk}(p^{-1} \bmod 2^k))$,
       $\hat{p} = \texttt{NTT}(\texttt{chk}(p))$
Output: $c = a \cdot b \cdot 2^{-k} \bmod p$
1: $\hat{a} = \texttt{NTT}(\texttt{chk}(a))$
2: $\hat{b} = \texttt{NTT}(\texttt{chk}(b))$
3: $t = \texttt{dechk}(\texttt{NTT}^{-1}(\hat{a} \circ \hat{b}))$
4: $\hat{t} = \texttt{NTT}(\texttt{chk}(t \bmod 2^k))$
5: $l = \texttt{dechk}(\texttt{NTT}^{-1}(\hat{t} \circ \hat{p}^{-1}))$
6: $\hat{l} = \texttt{NTT}(\texttt{chk}(l \bmod 2^k))$
7: $r = \texttt{dechk}(\texttt{NTT}^{-1}(\hat{l} \circ \hat{p}))$
8: $c = \frac{t}{2^k} - \frac{r}{2^k}$
9: if $c < 0$ then $c = c + p$
10: return $c$
```

3.2 Parameter choices

Recall from Section 2.2 that the Schönhage–Strassen algorithm involves lifting N-bit numbers to $\mathbb{Z}[X]$ along $X \mapsto 2^\ell$ and computing their product in $\mathbb{Z}_q[X]/(X^n - 1)$ using the NTT. We now describe our choices of N, ℓ, n, q; they were found by manually tailoring the algorithm to the given target architectures.

First, if we divide our inputs into ℓ-bit chunks, we need $n \geq 2\left\lceil \frac{N}{\ell} \right\rceil$; otherwise, we cannot lift from $\mathbb{Z}_q[X]/(X^n - 1)$ back to $\mathbb{Z}_q[X]$. For performance, we also want n so that NTT-based polynomial multiplication is fast, e.g., a 2-power. Hence, we may deliberately choose $n > 2\left\lceil \frac{N}{\ell} \right\rceil$ and pad with zeros when needed.

Secondly, the coefficients of the product of two dimension-$(n/2)$ polynomials with ℓ-bit coefficients are bounded by $\frac{n}{2} \cdot 2^{2\ell}$, so we need $q \geq \frac{n}{2} \cdot 2^{2\ell}$ to be able to lift from $\mathbb{Z}_q[X]$ back to $\mathbb{Z}[X]$. However, we also need to pick q so that $\mathbb{Z}_q$ has a principal nth root of unity, as otherwise the NTT is not defined. We pick $q = q_1 q_2$ a bi-prime and compute modulo q_1 and q_2 separately via CRT; using two half-size moduli maps to the available hardware multipliers better than a single larger q. Table 1 presents our choices, and we explain them in detail now.

On the Cortex-M3, we use chunks of $\ell = 11$ bits, so $\left\lceil \frac{N}{\ell} \right\rceil = 187$ for $N = 2048$ and $\left\lceil \frac{N}{\ell} \right\rceil = 373$ for $N = 4096$, but pick slightly larger $n = 384 > 2\left\lceil \frac{N}{\ell} \right\rceil$ for $N = 2048$ and $n = 768 > 2\left\lceil \frac{N}{\ell} \right\rceil$ for $N = 4096$ since both are dimensions for which a fast NTT can be implemented. Next, we need $q_1 q_2 \geq 192 \cdot 2^{22}$ for $N = 2048$ and $q_1 \cdot q_2 \geq 384 \cdot 2^{22}$ for $N = 4096$; we pick $(q_1, q_2) = (12289, 65537)$ for $N = 2048$, and $(q_1, q_2) = (25601, 65537)$ for $N = 4096$. The Fermat prime $q_2 = 65537$ allows particularly fast NTT computation using the FNT, while the other prime is chosen to be the smallest admissible prime for which a 128th (resp. 256th) primitive root of unity exists.

Table 1: Parameters

Cortex-M3				
bits (N)	chunking (ℓ)	poly length (n)	NTT	modulus $q = q_1 \cdot q_2$
2048	11 bits	384	$128 = 2^7$	$12289 \cdot 65537$
4096	11 bits	768	$256 = 2^8$	$25601 \cdot 65537$
Cortex-M55				
bits (N)	chunking (ℓ)	poly length (n)	NTT	modulus $q = q_1 \cdot q_2$
2048	22 bits	192	$64 \cdot 3 = 2^6 \cdot 3$	$114\,826\,273 \cdot 128\,919\,937$
4096	22 bits	384	$128 \cdot 3 = 2^7 \cdot 3$	$114\,826\,273 \cdot 128\,919\,937$

On the Cortex-M55, we use chunks of $\ell = 22$ bits, so $\lceil \frac{N}{\ell} \rceil = 94$ for $N = 2048$ and $\lceil \frac{N}{\ell} \rceil = 187$ for $N = 4096$, but again pick slightly larger $n = 192 > 2 \lceil \frac{N}{\ell} \rceil$ for $N = 2048$ and $n = 384 > 2 \lceil \frac{N}{\ell} \rceil$ for $N = 4096$ since those are NTT-friendly dimensions. For $q = q_1 q_2$, we pick $114\,826\,273 \cdot 128\,919\,937$ for both $N = 2048$ and $N = 4096$. Those choices are motivated as follows: First, we have $q \approx 2^{53.7} > 2^{51.58} \approx \frac{384}{2} \cdot 2^{44}$. In fact, since we even have $q > 4 \cdot (\frac{384}{2} \cdot 2^{44})$, we can recover the coefficients in the *sum* of two polynomial products as the *signed* canonical representatives of their image in $\mathbb{Z}_q$. The former allows saving one CRT during the Montgomery multiplication, while the latter means that we don't need a signed-to-unsigned conversion after the signed CRT. Second, q_1, q_2 are carefully chosen so that $(q_2 \bmod q_1)^{-1}$ in $\mathbb{Z}_{q_2}$ is amenable to refined Barrett multiplication — in fact, since $(q_2 \bmod q_1)^{-1} = 774$, this is what we observed in Example 1. Thirdly, both $q_1 - 1$ and $q_2 - 1$ are multiples of 96 and thus support incomplete dimension-96 NTTs. Finally, $q_1, q_2 < 2^{27}$ are small enough that during the dimension-96 NTTs, no explicit modular reduction is necessary.

3.3 Chunking and dechunking

We need to convert between multi-precision integers and polynomials, which we refer to as "chunking" `chk()` and "dechunking" `dechk()`. `chk()` takes an N-bit multi-precision integer and splits it into n chunks of ℓ bits each, viewed as the coefficients of a polynomial. In other words, we lift along $\mathbb{Z}[X] \to \mathbb{Z}, X \mapsto 2^\ell$. `dechk()` converts a polynomial to a multi-precision integer by evaluating the polynomial at $X = 2^\ell$. As the coefficients of polynomials may grow beyond 2^ℓ during computation, this requires carrying through the entire polynomial and packing into a multi-precision integer.

3.4 Modular exponentiation and table lookup

For the private-key operations, we use square-and-multiply with Algorithms 1 and 2 to implement constant-time exponentiation with a fixed window size of

w bits. This requires constant-time table lookups, and choosing the optimal w depends on the relative costs of a modular multiplication compared to such lookups: The cost of a lookup scales linearly in the table size 2^w, whereas the number of required multiplications only scales proportionally to $1/w$. We have determined that $w = 6$ is the fastest choice for both Cortex-M3 and Cortex-M55 and both 2048 and 4096 bits. We note that memory consumption will be an increasing concern as w grows, since the lookup table contains 2^w entries — exponentially large in w. In turn, reducing w will incur only a mild performance penalty while allowing for a significant reduction in the table size.

It may seem at first that storing the table entries in NTT domain should be preferable. However, the much larger size of elements in NTT domain results in drastically slower table lookups, which in our implementation clearly outweighs the cost of transforming to NTT domain on the fly after each load. Thus, our implementation stores the table entries as integer values.

For the public-key operation, we use a straightforward square-and-multiply for the fixed public exponent $2^{16}+1$ which is overwhelmingly common in practice.

3.5 Implementation details for Cortex-M3

Our Cortex-M3 NTT implementation relies on a code generator written in Python, featuring a bounds checker which determines when it should insert reductions, and which aborts if it cannot guarantee the correctness of the computation. The result is a set of fully unrolled assembly implementations of NTT, inverse NTT, base multiplication and squaring, for configurable moduli.

The code generator uses the same high-level structure for FNTs and “generic” NTTs, the main difference being in the reductions. The generator also recognizes multiplications by power-of-two constants and converts them to shifts when appropriate; this is one of the main optimizations employed by FNTs.

Number-theoretic transforms. We use incomplete NTTs of lengths $384 = 2^7 \cdot 3$ and $768 = 2^8 \cdot 3$. Both NTTs are implemented for the prime moduli $q_1 = 12289$ ($q_1 = 25601$) and $q_2 = 65537$, which by CRT correspond to a single NTT of the same length modulo $q_1 \cdot q_2$. We use CT butterflies for the forward NTT and GS butterflies for the inverse. Layers are merged as appropriate[5] to eliminate unnecessary store-load pairs. The base multiplication is a straightforward polynomial multiplication in a ring of the form $\mathbb{Z}_q[X]/(X^3 - \zeta)$. The CRT computation after the inverse NTTs is applied to each coefficient separately and follows Equation 1.

“General” number-theoretic transform. For most moduli including $q_1 = 12289$ and $q_1 = 25601$, we use a combination of (signed) Montgomery multiplication (Appendix A, Algorithm 4) and (signed) Barrett reductions (Appendix A, Algorithm 3). The Barrett reduction comes in two variants, the difference consisting

[5] The layers are merged as $4 + 3$ resp. $4 + 2 + 2$ in the forward NTTs, exploiting that the upper half of the input coefficients are zero, and $3 + 2 + 2$ resp. $3 + 3 + 2$ in the inverse NTTs. Register pressure prohibits more aggressive merging.

in the optional addition of `R`/2 before the right shift. Skipping the addition is faster, but results in worse reduction quality.

Fermat number transform. For the Fermat prime $q_2 = 2^{16} + 1 = 65537$, we use variants of the "FNT reduction" shown in Appendix A, Algorithm 5. Depending on the desired reduction quality, the algorithm is either applied (1) as written, or (2) with its input offset by 2^{15} and the output correspondingly offset by -2^{15}, or (3) followed by a conditional subtraction of $2^{16} + 1$ if the output is $> 2^{15}$. Method (2) produces a representative in $\{-2^{16} + 1, 2^{16} - 1\}$, while the output of method (3) is a canonical symmetric representative, i.e., lies in $\{-2^{15}, ..., 2^{15}\}$. Methods (2) and (3) are equally fast if the constant 2^{15} can be kept in a low register throughout. If register pressure renders this undesirable, method (2) provides a convenient "intermediate" solution between the very fast FNT reduction and the canonical symmetric reduction.

Constant-time lookup. We use predicated moves to extract the desired table entry in a "striding" fashion: For each slice of four 32-bit words, the respective part of each table entry is loaded and conditionally moved into a set of target registers using a `itttt eq; moveq; moveq; moveq; moveq` instruction sequence. The target registers are stored after processing all entries. Compared to the alternative of traversing the table entry by entry, this finalizes each output word immediately, and no partial outputs have to be stored and reloaded.

3.6 Implementation details for Cortex-M55

Pipeline efficiency. As explained in [Bec+22b], Cortex-M55 is a *dual-beat* implementation of MVE; that is, most MVE instructions execute over two cycles. To still achieve a Instructions per Cycle (IPC) rate of more than 0.5 without costly dual-issuing logic, Cortex-M55 supports *instruction overlapping* for neighboring vector instructions, provided they use different execution resources. The balance and ordering of instructions is therefore crucial for performance. We find that all our core subroutines have a good balance between load/store, addition and multiplication instructions and can be carefully arranged to maximize instruction overlapping, achieving an IPC > 0.9. Table 5 provides details.

Number-theoretic transform. We implement incomplete NTTs of degree $96 = 3 \cdot 32$ and $192 = 3 \cdot 64$ via Good's trick, using CT butterflies and Barrett multiplication throughout. Algorithm 6 is a translation of Barrett multiplication into MVE. No explicit modular reductions are necessary during $\texttt{NTT}$ or $\texttt{NTT}^{-1}$, as we confirm using a script tracking the bounds of modular representative throughout the NTT, applying Fact 1 repeatedly.

Base multiplication. The incomplete NTTs leave us with base multiplications in rings of the form $\mathbb{Z}_q[X]/(X^4 - \zeta)$ with a $<$ 32-bit prime q, which we implement essentially using the method of [Bec+22b]: A polynomial $\boldsymbol{a} = a_0 + a_1 X + a_2 X^2 +$

a_3X^3 is first expanded into a sequence $\tilde{\boldsymbol{a}} = (a_0, \ldots, a_3, \zeta a_0, \ldots, \zeta a_3)$, and 64-bit representatives of the coefficients of $\boldsymbol{a} \cdot \boldsymbol{b}$ are computed as dot products of (b_3, b_2, b_1, b_0) with length-4 subsequences of $\tilde{\boldsymbol{a}}$, using `vmalaldav`. Here, we instead compute $\tilde{\boldsymbol{a}} = \frac{1}{n}(a_0, \ldots, a_3, \zeta a_0, \ldots, \zeta a_3)$, where n is the incomplete NTT degree, taking care of the scaling by $\frac{1}{n}$ as part of the base multiplication.

CRT. We vectorize the divided-difference interpolation (1), producing chunked outputs. We allow non-reduced inputs and compute canonical reductions u_0' of u_0 and of $(u_1 - u_0')m_1$ as part of the CRT rather than at the end of $\texttt{NTT}^{-1}$. For the computation of $(u_1 - u_0')m_1$, we use refined Barrett multiplication, leveraging our choice of primes. The long multiplication $((u_1 - u_0')m_1 \bmod {}^{\pm}q_1)\, q_0$ is computed via `vmul[h]`, aligned to the 2^ℓ-boundary via $(a, b) \mapsto (a \bmod 2^\ell, b \cdot 2^{32-\ell} + \lfloor a/2^\ell \rfloor)$ (note $|b_i| < 2^{\lceil 52.7 \rceil - 32} = 2^{21}$, so $|2^{10} b_i| < 2^{31}$), and the low part added to u_0'. This results in a non-canonical chunked presentation of the CRT interpolation with 32-bit values, which are finally reduced to $< 2^\ell + 2^{32-\ell} = 2^{22} + 2^{10}$ via $a_i \mapsto (a_i \mod 2^\ell) + \lfloor a_{i-1}/2^\ell \rfloor$. We found that the slight non-canonicity of the coefficients does not impact functional correctness, while enabling vectorization of the above routine — a perfect reduction, in turn, is inherently sequential.

Constant time lookup. In contrast to Cortex-M3 we do not use predicated move operations: A block of loads followed by a block of predicated moves allows for only very little instruction overlapping. Instead, we use load-multiply-accumulate sequences with secret constant 0/1 for the conditional moves, achieving very good instruction overlapping. Overall, we obtain a constant-time lookup of 5184 cycles for a table of 8192-bytes — 26% over the theoretical minimum of 8192/2 cycles necessary to load each table entry once with a 64-bit data path. See Section C.

As our data resides in uncached Data-TCM, it is tempting to consider a plain load for a constant time lookup. We strongly advise against this: While access to D-TCM is *typically* single-cycle, it's not in general: On Cortex-M55 a D-TCM load with secret address could happen concurrently with a DMA transfer and trigger a memory bank conflict depending on the addresses being loaded. While this particularly issue could be circumvented in our present context, it might be problematic on future microarchitectures, and it appears prudent to simply stick to the principle that memory access patterns should not rely on secret data.

4 Results

4.1 Benchmark environment

Cortex-M3. We use the STM32 Nucleo-F207ZG with the STM32F207ZG Cortex-M3 core with 128 kB RAM and 1 MB flash.We clock the Cortex-M3 at 30 MHz (rather than the maximum frequency of 120 MHz) to void having any flash wait states when fetching code or constants from flash. We place the stack in SRAM1 (112 kB) only since it results in slightly better performance. We use libopencm3[6]

[6] https://github.com/libopencm3/libopencm3

Table 2: Performance of our NTTs and FNTs in cycles

Cortex-M3							
(N,n)	q	`NTT`	`NTT`$_{\text{heavy}}$	`basemul`	`basesqr`	`basemul`$_{\text{light}}$	`NTT`$^{-1}$
$(2048,384)$	12289	12 409	14 692	7 053	6 101	5 949	15 130
	65537	7 635	9 631	7 181	6 488	5 563	11 090
$(4096,768)$	25601	31 491	35 805	13 808	11 386	11 729	36 227
	65537	19 892	23 697	14 062	12 160	10 957	25 015

Cortex-M55							
(N,n)	q	`NTT`	`NTT`$_{\text{heavy}}$	`basemul`	`basesqr`	`basemul`$_{\text{light}}$	`NTT`$^{-1}$
$(2048,192)$	114 826 273 128 919 937	814	1 441	1 500	–	880	900
$(4096,384)$	114 826 273 128 919 937	2 027	3 230	2 894	–	1 696	2 195

and some hardware abstraction code is taken from pqm3[7]. We use the SysTick counter for benchmarking. We use `arm-none-eabi-gcc` version 11.2.0 with `-O3`.

Cortex-M55. We make use of the Arm MPS3 FPGA prototyping board with an FPGA model of the Cortex-M55r1 (AN552). Both the prototyping board and the FPGA model are publicly available[8]. Qemu supports a previous revision of the image (AN547) and can be used for running our code as well. However, for meaningful benchmarks, the FPGA board is required. We make use of the tightly coupled memory for code (ITCM) and data (DTCM). The core is clocked at the default frequency of 32 MHz. We use the PMU cycle counter for benchmarking. We use `arm-none-eabi-gcc` version 11.2.0 with `-O3`.

4.2 NTT and FNT performance

Table 2 contains the cycle counts for our core transformations. For the Cortex-M3, we implement four different transforms using specialized code for each combination of size and modulus. This allows us to minimize the number of explicit modular reductions taking into account the size of the modulus and its twiddles, and also to have a much faster FNT (modulo 65537) than the NTTs modulo 12289 and 25601. For the Cortex-M55 and a given size, the same code is used for both moduli with different precomputed constants; since no explicit modular reductions are required, we do not see prime-specific optimization potential. Base squaring and multiplication are the same, as we do not see optimization potential for squaring.

[7] https://github.com/mupq/pqm3

[8] https://developer.arm.com/tools-and-software/development-boards/fpga-prototyping-boards/download-fpga-images

Table 3: Performance of modular multiplication, squaring, exponentiation in cycles. $\texttt{expmod}_{\text{public}}$ is a modular exponentiation with the exponent 65537. $\texttt{expmod}_{\text{private}}$ is a modular exponentiation with a private n-bit exponent.

Cortex-M3					
	n	`mulmod`	`sqrmod`	$\texttt{expmod}_{\text{public}}$	$\texttt{expmod}_{\text{private}}$
This work	2048	220 047	196 830	4 227 473	494 923 435
This work (FIOS)		234 041	–	4 912 705	543 648 872
BearSSL [Bear]		283 038	–	18 350 210	718 347 177
This work	4096	510 708	454 128	9 752 690	2 250 748 647
This work (FIOS)		926 523	–	19 458 326	4 228 661 467
BearSSL [Bear]		1 102 151	–	70 443 207	5 505 856 187

Cortex-M55					
	n	`mulmod`	`sqrmod`	$\texttt{expmod}_{\text{public}}$	$\texttt{expmod}_{\text{private}}$
This work	2048	21 330	19 701	389 482	50 085 366
This work (FIOS)		20 260	–	426 707	50 683 718
MbedTLS [Mbed]		41 443	–	884 416	108 441 240
BearSSL [Bear]		83 517	–	5 400 650	217 123 645
This work	4096	47 660	43 620	861 450	218 110 707
This work (FIOS)		73 316	–	1 540 685	358 080 308
MbedTLS [Mbed]		152 371	–	3 223 797	755 391 521
BearSSL [Bear]		328 801	–	21 254 533	1 646834 048

4.3 Modular arithmetic: Multiplication, squaring, exponentiation

Table 3 presents timings for our modular arithmetic routines.

For Cortex-M3, we compare with BearSSL [Bear] (v0.6, `i15` implementation) which to our knowledge is the only library claiming to be constant-time on the Cortex-M3. We also consider a handwritten FIOS implementation (Section 2.4).

On Cortex-M55, we compare to BearSSL v0.6 (`i31` implementation), to Mbed TLS [Mbed] v3.1.0, and to our own handwritten FIOS implementation. The BearSSL implementation compiles down to `umlal`, while the Mbed TLS implementation uses CIOS (Section 2.4) with `umaal`-based inline assembly.

We find that our implementations outperform Mbed TLS and BearSSL significantly for both 2048-bit and 4096-bit parameters. Moreover, for Cortex-M3, our NTT-based implementation is also slightly faster than the handwritten FIOS implementation for 2048-bit, and considerably faster for 4096-bit.

Somewhat surprisingly, the `umaal`-based handwritten FIOS is much faster than the `umaal`-based CIOS in Mbed TLS, and on par with our NTT-based implementation for 2048-bit. For 4096-bit, however, the NTT-based implementation prevails. The optimization potential between `umaal`-based FIOS and CIOS lies within memory accesses: Mbed TLS' CIOS assembly does not leverage the

64-bit data path of Cortex-M55, and merging of loops in FIOS also saves accesses. We reported this optimization potential to the Mbed TLS team.[9]

Figure 1 shows the distribution of cycles spent in one modular multiplication.

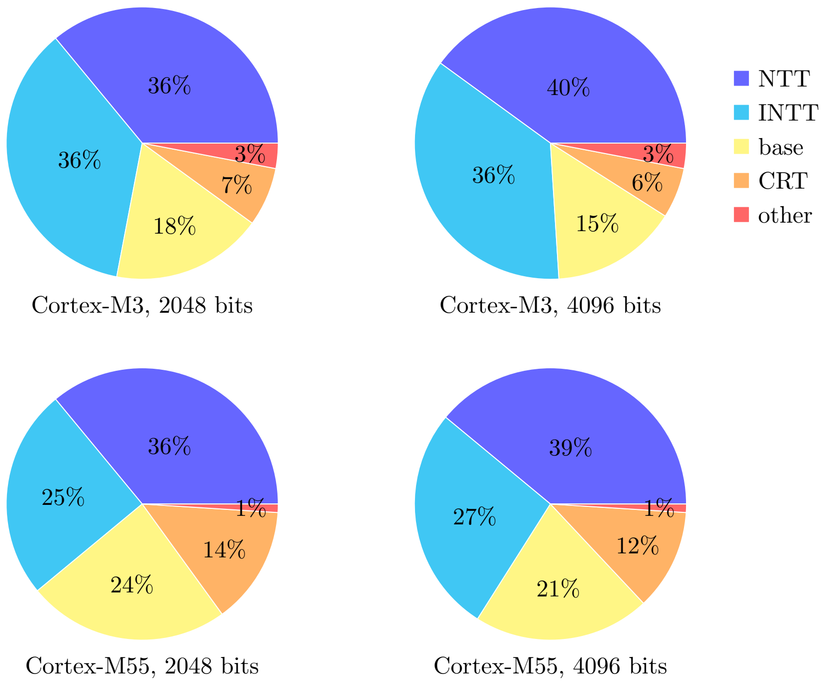

Fig. 1: Clock cycles spent on the subroutines of a single modular multiplication.

A Reduction algorithms for Cortex-M3 and Cortex-M55

Algorithm 3: ($\log_2 \mathtt{R}$)-bit Barrett reduction on Cortex-M3.

```
Input: a = $a$
Output: a = $a \bmod^{\pm} q$
1: mul t, a, $\lceil \mathtt{R}/q \rfloor$
2: (optional) add t, t, #(R/2)
3: asr t, t, #$\log_2 \mathtt{R}$
4: mls a, t, $q$, a
```

Algorithm 4: 16-bit Montgomery multiplication on Cortex-M3.

```
Input: (a, b) = $(a, b2^{16} \bmod^{\pm} q)$
Output: a = $ab \bmod^{\pm} q$
1: mul a, a, b
2: mul t, a, $-q^{-1} \bmod^{\pm} 2^{16}$
3: sxth t, t
4: mla a, t, $q$, a
5: asr a, a, #16
```

[9] See https://github.com/ARMmbed/mbedtls/issues/5666 and https://github.com/ARMmbed/mbedtls/issues/5360.

Algorithm 5: FNT reduction on Cortex-M3.

```
Input: a = $a$
Output: a = $a \bmod^{\pm} 65537 \in [-32767, 98303]$
1: ubfx t, a, #0, #16
2: sub a, t, a, asr#16
```

Algorithm 6: Barrett multiplication on Cortex-M55.

```
Input: (a, b, b') = $\left(a, b, \frac{\lfloor b 2^{32}/q \rceil_2}{2}\right)$
Output: a = $ab \bmod^{\pm} q$
1: vmul.s32 l, a, b
2: vqrdmulh.s32 h, a, b'
3: vmla.s32 l, h, $q$
```

B On precomputing the Montgomery constant

Montgomery multiplication (see Section 2.4) requires the precomputation of $q^{-1} \bmod \texttt{R}$. When implementing RSA via “large” Montgomery multiplication, rather than a FIOS approach, this means that we need to precompute $n^{-1} \bmod \texttt{R}$ for encryption and $p^{-1} \bmod \texttt{R}$ and $q^{-1} \bmod \texttt{R}$ for decryption. For decryption this can be computed as a part of key generation and stored as a part of the secret key. For encryption, however, it needs to be computed online.

Modular inversion $x^{-1} \bmod 2^r$ can be performed using “Hensel lifting”: If $xy - 1 = 2^k a$, so that y is an inverse to x modulo 2^k, then $y' = 2y - x^2 y$ satisfies $xy' - 1 = -(xy-1)^2 = 2^{2k}a^2$, and hence y' is an inverse of x modulo 2^{2k}. This yields $x^{-1} \bmod 2^k$ after $\mathcal{O}(\log k)$ iterations. One may observe that this is the sequence of approximate solutions to $xy = 1$ for x via the Newton–Raphson method in the 2-adic integers.

We prototyped Hensel-lifting to assess its relative cost compared to the modular exponentiation; we did not seek a fully optimized version. On the Cortex-M3 we implement both a variable-time variants using `umlal` for encryption and a constant-time variant using `mla` for key generation. For the Cortex-M55, we achieve the best performance using `umaal`. We list the performance in Table 4. We see that already a basic implementation has only a small performance overhead compared to an exponentiation (e.g., $< 5\%$ for RSA-4096).

Table 4: Performance of Hensel lifting; numbers for RSA-4096 in bold.

k	Cortex-M3		Cortex-M55
	`mla` (constant-time)	`umlal` (variable-time)	`umaal` (constant-time)
2112	**85 337**	45 326	**12 430**
4224	313 695	**163 107**	**38 575**

C Table lookup

Algorithm 7:
Conditional move on Cortex-M3

```
1: ldr.w a, [tbl, #4]
2: ldr.w b, [tbl, #8]
3: ldr.w c, [tbl, #12]
4: ldr.w d, [tbl], #16
5: cmp.n idx, #dst
6: itttt.n EQ
7: moveq.w A, a
8: moveq.w B, b
9: moveq.w C, c
10: moveq.w D, d
```

Algorithm 8:
Overlapping-friendly conditional accumulation on Cortex-M55

```
1: cmp idx, #dst
2: cset mask, EQ // idx == dst
3: vldrw.u32 t, [tbl], #16
4: vmla.s32 A, t, mask
5: vldrw.u32 t, [tbl], #16
6: vmla.s32 B, t, mask
7: vldrw.u32 t, [tbl], #16
8: vmla.s32 C, t, mask
9: vldrw.u32 t, [tbl], #16
10: vmla.s32 D, t, mask
```

D Pipeline efficiency of Cortex-M55 implementation

Table 5 shows Performance Monitoring Unit (PMU) statistics for the subroutines of our Cortex-M55 modular exponentiation ($N = 4096$). We use `ARM_PMU_CYCCNT`, `ARM_PMU_INST_RETIRED`, `ARM_PMU_MVE_INST_RETIRED`, and `ARM_PMU_MVE_STALL` for counting cycles, retired instructions, retired MVE instructions, and MVE instructions causing a stall, respectively. We derive the rate of Instructions per Cycle (IPC), as well as `ARM_PMU_MVE_INST_RETIRED`/`ARM_PMU_MVE_STALL` as a measure of the MVE overlapping efficiency. Despite most MVE instructions running for 2 cycles, instruction overlapping allows achieving an IPC > 0.9.

Table 5: Performance Monitoring Unit statistics for Cortex-M55 implementation.

Primitive	Cycles	Instructions	Instructions per Cycle (IPC)	MVE instructions	MVE stalls	MVE efficiency
`NTT`	2 027	1 936	0.95	1 876	27	98.6%
$\texttt{NTT}_{\text{heavy}}$	3 231	3 017	0.93	2 742	130	96.0%
$\texttt{NTT}^{-1}$	2 195	2 128	0.96	2 072	9	99.6%
`basemul`	2 894	2 737	0.94	2 500	109	95.6%
$\texttt{basemul}_{\texttt{light}}$	1 695	1 659	0.97	1 634	6	99.6%
CRT	4 287	4 216	0.98	3 563	13	99.6%
Table lookup	5 184	4 816	0.92	4 132	12	99.7%

References

[AB74] R. C. Agarwal and C. S. Burrus. "Fast Convolution Using Fermat Number Transforms with Applications to Digital Filtering". In: *IEEE Trans. Signal Process.* 22.2 (1974), pp. 87–97.

[Abd+22] A. Abdulrahman et al. “Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.1 (2022), pp. 127–151.

[Bear] T. Pornin. *BearSSL: a smaller TLS/SSL library.*

[Bec+22a] H. Becker et al. “Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.1 (2022), pp. 221–244.

[Bec+22b] H. Becker et al. “Polynomial multiplication on embedded vector architectures”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.1 (2022), pp. 482–505.

[Chu+21] C.-M. M. Chung et al. “NTT Multiplication for NTT-unfriendly Rings: New Speed Records for Saber and NTRU on Cortex-M4 and AVX2”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.2 (2021), pp. 159–188.

[Für09] M. Fürer. “Faster Integer Multiplication”. In: *SIAM Journal on Computing* 39.3 (2009), pp. 979–1005.

[Gar07] L. C. C. García. *Can Schönhage multiplication speed up the RSA decryption or encryption?* MoraviaCrypt 2007. 2007.

[GKZ07] P. Gaudry, A. Kruppa, and P. Zimmermann. “A GMP-based implementation of Schönhage–Strassen’s large integer multiplication algorithm”. In: *ISSAC 2007*. ACM, 2007, pp. 167–174.

[GMP] Free Software Foundation. *The GNU Multiple Precision Arithmetic Library.*

[HH21] D. Harvey and J. van der Hoeven. “Integer multiplication in time $O(n \log n)$”. In: *Annals of Mathematics* 193.2 (2021), pp. 563–617.

[KAK96] C. K. Koc, T. Acar, and B. S. Kaliski. “Analyzing and comparing Montgomery multiplication algorithms”. In: *IEEE Micro* 16.3 (1996), pp. 26–33.

[KO63] A. Karatsuba and Y. Ofman. “Multiplication of multidigit numbers on automata”. In: *Soviet Physics Doklady* 7 (1963). Translated from Doklady Akademii Nauk SSSR, Vol. 145, No. 2, pp. 293–294, July 1962, pp. 595–596.

[Mbed] Arm Ltd. *Mbed TLS.*

[Mon85] P. L. Montgomery. “Modular Multiplication without Trial division”. In: *Math. Comput.* 44.170 (1985), pp. 519–521.

[Pol71] J. M. Pollard. “The fast Fourier transform in a finite field”. In: *Mathematics of Computation* 25 (1971), pp. 365–374.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”. In: *Commun. ACM* 21.2 (1978), pp. 120–126.

[SS71] A. Schönhage and V. Strassen. “Schnelle Multiplikation großer Zahlen”. In: *Computing* 7.3-4 (1971), pp. 281–292.

[Too63] A. L. Toom. “The complexity of a scheme of functional elements realizing the multiplication of integers”. In: *Soviet Mathematics Doklady* 3 (1963), pp. 714–716.

E High-level multiplication structure

$$\begin{tikzcd}
 & & A \in \mathbb{Z} \arrow[d, "\boxed{\texttt{chk}}"] & & & \\
 & & \mathbb{Z}[x] \arrow[d, dotted] & & & \\
 & & (\mathbb{Z}/q_1q_2)[x] \arrow[dl, dotted] \arrow[dr, dotted] & & & \\
 & (\mathbb{Z}/q_1)[x] \arrow[dl, dotted] & & (\mathbb{Z}/q_2)[x] \arrow[dr, dotted] & & \\
\mathbb{Z}[x]/(q_1, x^N-1) \arrow[d, "\boxed{\texttt{NTT}_1}"] & & & & \mathbb{Z}[x]/(q_2, x^N-1) \arrow[d, "\boxed{\texttt{NTT}_2}"] & \\
\widehat{A}_1 \in \prod_i \mathbb{Z}[x]/(q_1, x^h-\zeta_i) \arrow[d] & & & & \widehat{A}_2 \in \prod_i \mathbb{Z}[x]/(q_2, x^h-\zeta_i) \arrow[d] & \\
\odot_1 \arrow[d] & \widehat{B}_1 \arrow[l] & & & \odot_2 \arrow[d] & \widehat{B}_2 \arrow[l] \\
\widehat{C}_1 \in \prod_i \mathbb{Z}[x]/(q_1, x^h-\zeta_i) \arrow[d, "\boxed{\texttt{NTT}_1^{-1}}"] & & & & \widehat{C}_2 \in \prod_i \mathbb{Z}[x]/(q_2, x^h-\zeta_i) \arrow[d, "\boxed{\texttt{NTT}_2^{-1}}"] & \\
\mathbb{Z}[x]/(q_1, x^N-1) \arrow[dr, dashed] & & & & \mathbb{Z}[x]/(q_2, x^N-1) \arrow[dl, dashed] & \\
 & (\mathbb{Z}/q_1)[x] \arrow[dr] & & (\mathbb{Z}/q_2)[x] \arrow[dl, "\boxed{\texttt{CRT}}"] & & \\
 & & (\mathbb{Z}/q_1q_2)[x] \arrow[d, dashed] & & & \\
 & & \mathbb{Z}[x] \arrow[d, "\boxed{\texttt{dechk}}"] & & & \\
 & & C \in \mathbb{Z} & & &
\end{tikzcd}$$

Fig. 2: High-level structure of our integer multiplication algorithm. Finely dotted arrows denote a conceptual reinterpretation with no change in representation. Dashed arrows denote a canonical choice of lift, e.g., a representative of minimal degree for polynomials or a smallest non-negative representative for integers.