Efficient Multiplication of Somewhat Small Integers using Number-Theoretic Transforms

Hanno Becker Vincent Hwang Matthias J. Kannwischer
Lorenz Panny Bo-Yin Yang

31 August 2022, IWSEC 2022, Tokyo, Japan [online]

From integers to polynomials and back

Task: Given $a, b \in \mathbb{Z}$, compute $c = a \cdot b$.

From integers to polynomials and back

Task: Given $a, b \in \mathbb{Z}$, compute $c = a \cdot b$.

Observation: This is equivalent to multiplying polynomials.

From integers to polynomials and back

Task: Given $a, b \in \mathbb{Z}$, compute $c = a \cdot b$.

Observation: This is equivalent to multiplying polynomials.

- *ℓ-bit chunking:* Write $a = \sum_{i=0}^{n} (2^\ell)^i a_i$ and replace 2^ℓ by x, giving $f_a = \sum a_i x^i \in \mathbb{Z}[x]$.

From integers to polynomials and back

Task: Given $a, b \in \mathbb{Z}$, compute $c = a \cdot b$.

Observation: This is equivalent to multiplying polynomials.

- *ℓ-bit chunking:* Write $a = \sum_{i=0}^{n} (2^{\ell})^i a_i$ and replace 2^{ℓ} by x, giving $f_a = \sum a_i x^i \in \mathbb{Z}[x]$.
- *polynomial multiplication:* Compute $f_c = f_a \cdot f_b \in \mathbb{Z}[x]$ using any method.

From integers to polynomials and back

Task: Given $a, b \in \mathbb{Z}$, compute $c = a \cdot b$.

Observation: This is equivalent to multiplying polynomials.

- *ℓ-bit chunking:* Write $a = \sum_{i=0}^{n} (2^\ell)^i a_i$ and replace 2^ℓ by x, giving $f_a = \sum a_i x^i \in \mathbb{Z}[x]$.
- *polynomial multiplication:* Compute $f_c = f_a \cdot f_b \in \mathbb{Z}[x]$ using any method.
- *"dechunking":* Replace x in f_c by 2^ℓ. (In other words, *evaluate* f_c at 2^ℓ.)

From integers to polynomials and back

Task: Given $a, b \in \mathbb{Z}$, compute $c = a \cdot b$.

Observation: This is equivalent to multiplying polynomials.

- *ℓ-bit chunking:* Write $a = \sum_{i=0}^{n} (2^\ell)^i a_i$ and replace 2^ℓ by x, giving $f_a = \sum a_i x^i \in \mathbb{Z}[x]$.
- *polynomial multiplication:* Compute $f_c = f_a \cdot f_b \in \mathbb{Z}[x]$ using any method.
- *"dechunking":* Replace x in f_c by 2^ℓ. (In other words, *evaluate* f_c at 2^ℓ.)

Correctness: The map $x \mapsto 2^\ell$ is a *ring homomorphism*.

From integers to polynomials and back

Task: Given $a, b \in \mathbb{Z}$, compute $c = a \cdot b$.

Observation: This is equivalent to multiplying polynomials.

- *ℓ-bit chunking:* Write $a = \sum_{i=0}^{n} (2^\ell)^i a_i$ and replace 2^ℓ by x, giving $f_a = \sum a_i x^i \in \mathbb{Z}[x]$.
- *polynomial multiplication:* Compute $f_c = f_a \cdot f_b \in \mathbb{Z}[x]$ using any method.
- *"dechunking":* Replace x in f_c by 2^ℓ. (In other words, *evaluate* f_c at 2^ℓ.)

Correctness: The map $x \mapsto 2^\ell$ is a *ring homomorphism*.

(The reverse reduction works as well, using *Kronecker substitution*: Given $f, g \in \mathbb{Z}[x]$, choose large 2^ℓ, compute $c = f(2^\ell) \cdot g(2^\ell)$, and recover $f \cdot g$ from c via ℓ-bit chunking.)

FFT-based polynomial multiplication

Observation: Multiplication in product rings $R_1 \times \cdots \times R_n$ is *component-wise* $\Longrightarrow O(n)$!

FFT-based polynomial multiplication

Observation: Multiplication in product rings $R_1 \times \cdots \times R_n$ is *component-wise* $\implies O(n)$!

Chinese Remainder Theorem (CRT): If $\gcd(f, g) = 1$, then $R[x]/(f \cdot g) \cong R[x]/f \times R[x]/g$.

FFT-based polynomial multiplication

Observation: Multiplication in product rings $R_1 \times \cdots \times R_n$ is *component-wise* $\Longrightarrow O(n)$!

Chinese Remainder Theorem (CRT): If $\gcd(f, g) = 1$, then $R[x]/(f \cdot g) \cong R[x]/f \times R[x]/g$.

Fourier transform computes $R[x]/(x^n - 1) \xrightarrow{\sim} \prod_{i=0}^{n-1} R[x]/(x - \omega_n^i)$.

FFT-based polynomial multiplication

Observation: Multiplication in product rings $R_1 \times \cdots \times R_n$ is *component-wise* $\implies O(n)$!

Chinese Remainder Theorem (CRT): If $\gcd(f, g) = 1$, then $R[x]/(f \cdot g) \cong R[x]/f \times R[x]/g$.

Fourier transform computes $R[x]/(x^n - 1) \xrightarrow{\sim} \prod_{i=0}^{n-1} R[x]/(x - \omega_n^i)$.

(Here ω_n is a *principal n^{th} root of unity*. Over $\mathbb{C}$, can use $\omega_n = \exp(2\pi\mathrm{i}/n)$.)

FFT-based polynomial multiplication

Observation: Multiplication in product rings $R_1 \times \cdots \times R_n$ is *component-wise* $\implies O(n)$!

Chinese Remainder Theorem (CRT): If $\gcd(f, g) = 1$, then $R[x]/(f \cdot g) \cong R[x]/f \times R[x]/g$.

Fourier transform computes $R[x]/(x^n - 1) \xrightarrow{\sim} \prod_{i=0}^{n-1} R[x]/(x - \omega_n^i)$.
(Here ω_n is a *principal n^{th} root of unity*. Over $\mathbb{C}$, can use $\omega_n = \exp(2\pi\mathrm{i}/n)$.)

Fast Fourier transform takes time only $O(n \log n)$!

FFT-based polynomial multiplication

Observation: Multiplication in product rings $R_1 \times \cdots \times R_n$ is *component-wise* $\implies O(n)$!

Chinese Remainder Theorem (CRT): If $\gcd(f, g) = 1$, then $R[x]/(f \cdot g) \;\cong\; R[x]/f \times R[x]/g$.

Fourier transform computes $R[x]/(x^n - 1) \xrightarrow{\sim} \prod_{i=0}^{n-1} R[x]/(x - \omega_n^i)$.

(Here ω_n is a *principal n^{th} root of unity*. Over $\mathbb{C}$, can use $\omega_n = \exp(2\pi\mathrm{i}/n)$.)

Fast Fourier transform takes time only $O(n \log n)$! Essential trick: *remainder tree*.

FFT-based polynomial multiplication

Observation: Multiplication in product rings $R_1 \times \cdots \times R_n$ is *component-wise* $\implies O(n)$!

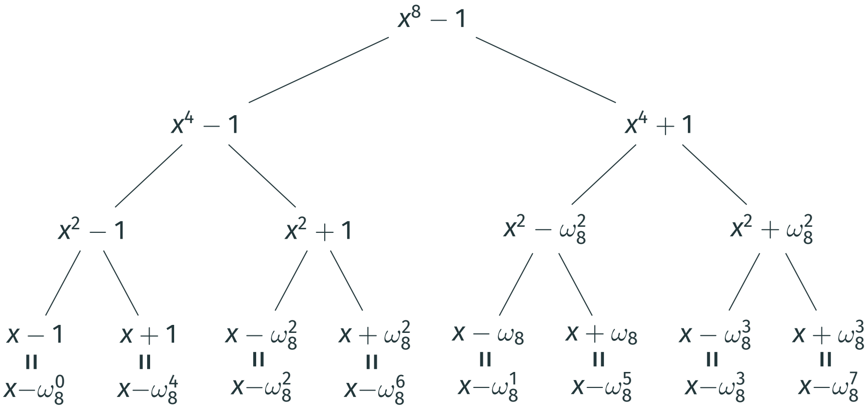
Chinese Remainder Theorem (CRT): If $\gcd(f, g) = 1$, then $R[x]/(f \cdot g) \;\cong\; R[x]/f \times R[x]/g$.

Fourier transform computes $R[x]/(x^n - 1) \xrightarrow{\sim} \prod_{i=0}^{n-1} R[x]/(x - \omega_n^i)$.
(Here ω_n is a *principal n^{th} root of unity*. Over $\mathbb{C}$, can use $\omega_n = \exp(2\pi\mathrm{i}/n)$.)

Fast Fourier transform takes time only $O(n \log n)$! Essential trick: *remainder tree*.

$\implies$ [FFT + pointwise multiplication + inverse FFT] is only $O(n \log n)$ operations in R.

FFT tree for $R[x]/(x^{2^m}-1)$



Computing the FFT (and inverse FFT)

Going <u>down</u> one layer: Compute $R[x]/(x^{2k} - \tau^2) \xrightarrow{\sim} R[x]/(x^k - \tau) \times R[x]/(x^k + \tau)$.

Computing the FFT (and inverse FFT)

Going <u>down</u> one layer: Compute $R[x]/(x^{2k} - \tau^2) \xrightarrow{\simeq} R[x]/(x^k - \tau) \times R[x]/(x^k + \tau)$.

- Input: $c_0 + c_1 x + \ldots + c_{2k-1} x^{2k-1}$.

Computing the FFT (and inverse FFT)

Going down one layer: Compute $R[x]/(x^{2k} - \tau^2) \xrightarrow{\sim} R[x]/(x^k - \tau) \times R[x]/(x^k + \tau)$.

- Input: $c_0 + c_1 x + \ldots + c_{2k-1} x^{2k-1}$.
- x^i-coefficient of "left" output is $(c_i + \tau c_{k+i})$.

Computing the FFT (and inverse FFT)

Going down one layer: Compute $R[x]/(x^{2k} - \tau^2) \xrightarrow{\sim} R[x]/(x^k - \tau) \times R[x]/(x^k + \tau)$.

- Input: $c_0 + c_1 x + \ldots + c_{2k-1} x^{2k-1}$.
- x^i-coefficient of "left" output is $(c_i + \tau c_{k+i})$.
- x^i-coefficient of "right" output is $(c_i - \tau c_{k+i})$.

Computing the FFT (and inverse FFT)

Going down one layer: Compute $R[x]/(x^{2k} - \tau^2) \xrightarrow{\sim} R[x]/(x^k - \tau) \times R[x]/(x^k + \tau)$.

- Input: $c_0 + c_1 x + \ldots + c_{2k-1} x^{2k-1}$.
- x^i-coefficient of "left" output is $(c_i + \tau c_{k+i})$.
- x^i-coefficient of "right" output is $(c_i - \tau c_{k+i})$.

Going up one layer: Compute the inverse map.

Computing the FFT (and inverse FFT)

Going <u>down</u> one layer: Compute $R[x]/(x^{2k} - \tau^2) \xrightarrow{\sim} R[x]/(x^k - \tau) \times R[x]/(x^k + \tau)$.

- Input: $c_0 + c_1 x + \ldots + c_{2k-1} x^{2k-1}$.
- x^i-coefficient of "left" output is $(c_i + \tau c_{k+i})$.
- x^i-coefficient of "right" output is $(c_i - \tau c_{k+i})$.

Going <u>up</u> one layer: Compute the inverse map.

- For each i, *have* $d_i = c_i + \tau c_{k+i}$ and $e_i = c_i - \tau c_{k+i}$; *want* (c_i, c_{k+i}).

Computing the FFT (and inverse FFT)

Going <u>down</u> one layer: Compute $R[x]/(x^{2k} - \tau^2) \xrightarrow{\sim} R[x]/(x^k - \tau) \times R[x]/(x^k + \tau)$.

- Input: $c_0 + c_1 x + \ldots + c_{2k-1}x^{2k-1}$.
- x^i-coefficient of "left" output is $(c_i + \tau c_{k+i})$.
- x^i-coefficient of "right" output is $(c_i - \tau c_{k+i})$.

Going <u>up</u> one layer: Compute the inverse map.

- For each i, *have* $d_i = c_i + \tau c_{k+i}$ and $e_i = c_i - \tau c_{k+i}$; *want* (c_i, c_{k+i}).
- Linear algebra $\rightsquigarrow$ $c_i = (d_i + e_i)/2$ and $c_{k+i} = \tau^{-1}(d_i - e_i)/2$.

Computing the FFT (and inverse FFT)

Going <u>down</u> one layer: Compute $R[x]/(x^{2k}-\tau^2) \xrightarrow{\sim} R[x]/(x^k-\tau) \times R[x]/(x^k+\tau)$.

- Input: $c_0 + c_1x + \ldots + c_{2k-1}x^{2k-1}$.
- x^i-coefficient of "left" output is $(c_i + \tau c_{k+i})$.
- x^i-coefficient of "right" output is $(c_i - \tau c_{k+i})$.

Going <u>up</u> one layer: Compute the inverse map.

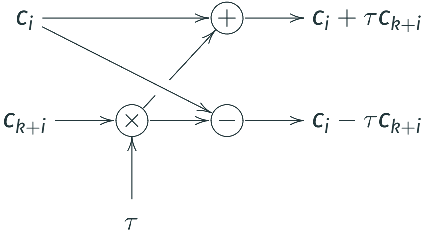
- For each i, *have* $d_i = c_i + \tau c_{k+i}$ and $e_i = c_i - \tau c_{k+i}$; *want* (c_i, c_{k+i}).
- Linear algebra $\rightsquigarrow$ $c_i = (d_i + e_i)/2$ and $c_{k+i} = \tau^{-1}(d_i - e_i)/2$.

$\implies$ Work per layer is $O(n)$, and there are $O(\log n)$ layers. $\implies O(n \log n)$.

Butterflies

Butterflies

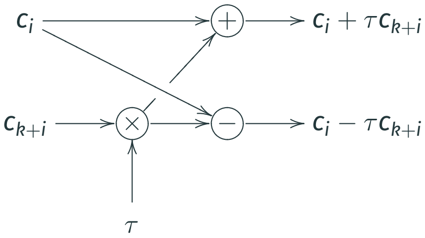
Cooley–Tukey butterfly



Reduce $f = c_0 + c_1 x + \ldots + c_{2k-1} c^{2k-1}$ modulo $(x^k - \tau)$ and $(x^k + \tau)$.

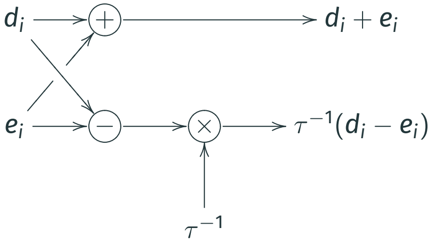
Butterflies

Cooley–Tukey butterfly



Reduce $f = c_0 + c_1 x + \ldots + c_{2k-1} c^{2k-1}$
modulo $(x^k - \tau)$ and $(x^k + \tau)$.

Gentleman–Sande butterfly



Recover $2 \cdot f \in R[x]/(x^{2k} - \tau^2)$
from $\left(f \bmod (x^k - \tau), f \bmod (x^k + \tau)\right)$.

NTT-based polynomial multiplication

Number-Theoretic Transform is a Fourier transform over a *finite* ring (typically $\mathbb{F}_q$).
Much more convenient for computers than, say, working over $\mathbb{C}$.

NTT-based polynomial multiplication

Number-Theoretic Transform is a Fourier transform over a *finite* ring (typically $\mathbb{F}_q$).
Much more convenient for computers than, say, working over $\mathbb{C}$.

- Multiplications in $\mathbb{Z}[x]$ can be emulated by choosing q large enough.

NTT-based polynomial multiplication

Number-Theoretic Transform is a Fourier transform over a *finite* ring (typically $\mathbb{F}_q$).
Much more convenient for computers than, say, working over $\mathbb{C}$.

- Multiplications in $\mathbb{Z}[x]$ can be emulated by choosing q large enough.

We require an n^{th} principal root of unity. For $\mathbb{F}_q$, this means $n \mid (q-1)$.

NTT-based polynomial multiplication

Number-Theoretic Transform is a Fourier transform over a *finite* ring (typically $\mathbb{F}_q$).
Much more convenient for computers than, say, working over $\mathbb{C}$.

- Multiplications in $\mathbb{Z}[x]$ can be emulated by choosing q large enough.

We require an n^{th} principal root of unity. For $\mathbb{F}_q$, this means $n \mid (q-1)$.

- If ω_n doesn't exist but $\omega_{n/d}$ does, can do *incomplete NTT*: $x^n - 1 = \prod_{i=0}^{n/d-1}(x^d - \omega_{n/d}^i)$.
  Base multiplication will be on degree-d polynomials instead of base-ring elements.

NTT-based polynomial multiplication

Number-Theoretic Transform is a Fourier transform over a *finite* ring (typically $\mathbb{F}_q$).
Much more convenient for computers than, say, working over $\mathbb{C}$.

- Multiplications in $\mathbb{Z}[x]$ can be emulated by choosing q large enough.

We require an n^{th} principal root of unity. For $\mathbb{F}_q$, this means $n \mid (q-1)$.

- If ω_n doesn't exist but $\omega_{n/d}$ does, can do *incomplete NTT*: $x^n - 1 = \prod_{i=0}^{n/d-1}(x^d - \omega_{n/d}^i)$.
  Base multiplication will be on degree-d polynomials instead of base-ring elements.

Note that R doesn't have to be a field: Another useful choice is $\mathbb{Z}/q$ with $q = q_1 q_2$.

NTT-based polynomial multiplication

Number-Theoretic Transform is a Fourier transform over a *finite* ring (typically $\mathbb{F}_q$).
Much more convenient for computers than, say, working over $\mathbb{C}$.

- Multiplications in $\mathbb{Z}[x]$ can be emulated by choosing q large enough.

We require an n^{th} principal root of unity. For $\mathbb{F}_q$, this means $n \mid (q-1)$.

- If ω_n doesn't exist but $\omega_{n/d}$ does, can do *incomplete NTT*: $x^n - 1 = \prod_{i=0}^{n/d-1}(x^d - \omega_{n/d}^i)$.
  Base multiplication will be on degree-d polynomials instead of base-ring elements.

Note that R doesn't have to be a field: Another useful choice is $\mathbb{Z}/q$ with $q = q_1 q_2$.

- Compute NTT modulo q_1 and q_2 separately, recombine via CRT $\mathbb{F}_{q_1} \times \mathbb{F}_{q_2} \xrightarrow{\sim} \mathbb{Z}/q$.

Asymptotics aside: Concrete performance

Recent focus on lattice-based cryptography taught us a lot about fast NTTs.
We can leverage these insights to speed up integer multiplication too.

Asymptotics aside: Concrete performance

Recent focus on lattice-based cryptography taught us a lot about fast NTTs. We can leverage these insights to speed up integer multiplication too.

Schönhage-Strassen applies FFT multiplication *recursively* to the coefficient multiplications occurring within the FFT. $\Longrightarrow$ Good asymptotic complexity.

Asymptotics aside: Concrete performance

Recent focus on lattice-based cryptography taught us a lot about fast NTTs.
We can leverage these insights to speed up integer multiplication too.

Schönhage-Strassen applies FFT multiplication *recursively* to the coefficient multiplications occurring within the FFT. $\implies$ Good asymptotic complexity.

In practice, want to move away from big integers as soon as possible.

Asymptotics aside: Concrete performance

Recent focus on lattice-based cryptography taught us a lot about fast NTTs. We can leverage these insights to speed up integer multiplication too.

Schönhage-Strassen applies FFT multiplication *recursively* to the coefficient multiplications occurring within the FFT. $\implies$ Good asymptotic complexity.

In practice, want to move away from big integers as soon as possible.
$\implies$ Chop into $\leq$word-sized coefficients; use longer NTT if needed.

Asymptotics aside: Concrete performance

Recent focus on lattice-based cryptography taught us a lot about fast NTTs.
We can leverage these insights to speed up integer multiplication too.

Schönhage-Strassen applies FFT multiplication *recursively* to the coefficient multiplications occurring within the FFT. $\implies$ Good asymptotic complexity.

In practice, want to move away from big integers as soon as possible.
$\implies$ Chop into $\leq$word-sized coefficients; use longer NTT if needed.
$\implies$ Use $\leq$word-sized moduli q_i suitable for fast reductions.

Asymptotics aside: Concrete performance

Recent focus on lattice-based cryptography taught us a lot about fast NTTs.
We can leverage these insights to speed up integer multiplication too.

Schönhage-Strassen applies FFT multiplication *recursively* to the coefficient multiplications occurring within the FFT. $\implies$ Good asymptotic complexity.

In practice, want to move away from big integers as soon as possible.
$\implies$ Chop into $\leq$word-sized coefficients; use longer NTT if needed.
$\implies$ Use $\leq$word-sized moduli q_i suitable for fast reductions.

Our algorithm isn't even properly specified for arbitrary lengths. If it were, it would scale worse than Schönhage–Strassen. Still, it appears to be *faster for medium-sized integers!*

Our work

- Optimized implementation of NTT-based multiplication for two popular Arm microcontrollers (one low-end, one high-end).

Our work

- Optimized implementation of NTT-based multiplication for two popular Arm microcontrollers (one low-end, one high-end).
- Parameters were carefully adjusted to our target architectures, for integer sizes of cryptographic relevance.

Our work

- Optimized implementation of NTT-based multiplication for two popular Arm microcontrollers (one low-end, one high-end).
- Parameters were carefully adjusted to our target architectures, for integer sizes of cryptographic relevance.
- Comparison to existing cryptographic software and other, less sophisticated integer-multiplication algorithms.

Our work

- Optimized implementation of NTT-based multiplication for two popular Arm microcontrollers (one low-end, one high-end).
- Parameters were carefully adjusted to our target architectures, for integer sizes of cryptographic relevance.
- Comparison to existing cryptographic software and other, less sophisticated integer-multiplication algorithms.

Conclusion: NTTs can compete — even win! — for integers around **a few thousand bits**.

Our work

- Optimized implementation of NTT-based multiplication for two popular Arm microcontrollers (one low-end, one high-end).
- Parameters were carefully adjusted to our target architectures, for integer sizes of cryptographic relevance.
- Comparison to existing cryptographic software and other, less sophisticated integer-multiplication algorithms.

Conclusion: NTTs can compete — even win! — for integers around **a few thousand bits**.

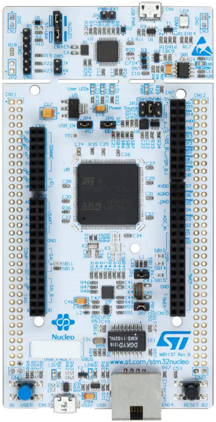
Compare conventional wisdom:

“[Schönhage–Strassen] starts to outperform [...] for numbers beyond $2^{2^{15}}$ *to* $2^{2^{17}}$*.”*

(Wikipedia)

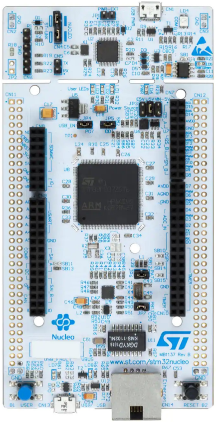
Target Architectures

- Focus on 32-bit Arm microcontrollers
- First target: Arm Cortex-M3
  - Announced in 2004
  - Implements Armv7-M
  - Interesting/dangerous feature: Timing of long multiplications (e.g., `UMULL`) is input-dependent
    $\rightarrow$ Avoid for constant-time code
  - We make use of STM32 Nucleo-F207ZG with the STM32F207ZG
  - \$ 20



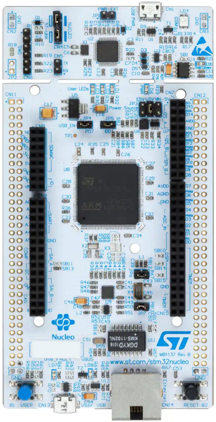
Target Architectures

- Focus on 32-bit Arm microcontrollers
- First target: Arm Cortex-M3
  - Announced in 2004
  - Implements Armv7-M
  - Interesting/dangerous feature:
    Timing of long multiplications (e.g., `UMULL`) is input-dependent
    $\rightarrow$ Avoid for constant-time code
  - We make use of STM32 Nucleo-F207ZG with the STM32F207ZG
  - \$ 20



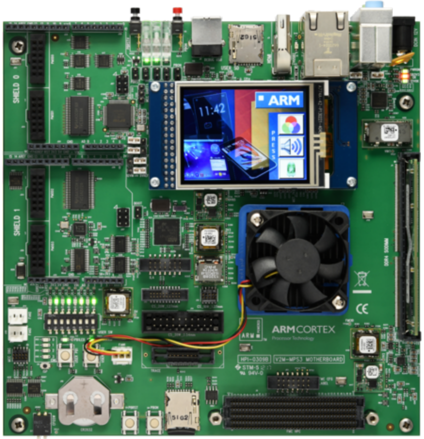
Target Architectures

- Focus on 32-bit Arm microcontrollers
- First target: Arm Cortex-M3
    - Announced in 2004
    - Implements Armv7-M
    - Interesting/dangerous feature:
      Timing of long multiplications (e.g., `UMULL`) is input-dependent
      $\rightarrow$ Avoid for constant-time code
    - We make use of STM32 Nucleo-F207ZG with the STM32F207ZG
    - \$ 20



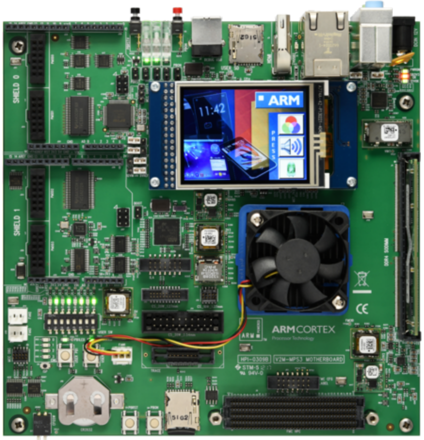
Target Architectures (2)

- Second target: Arm Cortex-M55
  - Announced in 2020
  - Implements Armv8-M
  - First core to implement the M-profile vector extension (MVE) a.k.a. Helium
  - No development boards available as of now
  - We use an FPGA prototyping board (Arm MPS3) with the AN552 model
  - $ 1500



Target Architectures (2)

- Second target: Arm Cortex-M55
  - Announced in 2020
  - Implements Armv8-M
  - First core to implement the M-profile vector extension (MVE) a.k.a. Helium
  - No development boards available as of now
  - We use an FPGA prototyping board (Arm MPS3) with the AN552 model
  - $ 1500



Fermat Number Transforms (FNT)

- Recall: For NTTs we require $2^k \mid q - 1$ with prime q
- Fermat numbers: $2^{2^k} + 1$
- Fermat primes: 3, 5, 17, 257, 65537
- Example: 65537
  - $\omega_2 = -1 = 2^{16}$
  - $\omega_4 = 2^8$
  - $\omega_8 = 2^4$
  - $\omega_{16} = 2^2$
  - $\omega_{32} = 2^1$
- First 5 layers of the FFT have multiplications by powers of two
  $\rightarrow$ Can use shifts instead of multiplications
  $\rightarrow$ Particularly useful on the Cortex-M3!

Fermat Number Transforms (FNT)

- Recall: For NTTs we require $2^k \mid q - 1$ with prime q
- Fermat numbers: $2^{2^k} + 1$
- Fermat primes: 3, 5, 17, 257, 65537
- Example: 65537
  - $\omega_2 = -1 = 2^{16}$
  - $\omega_4 = 2^8$
  - $\omega_8 = 2^4$
  - $\omega_{16} = 2^2$
  - $\omega_{32} = 2^1$
- First 5 layers of the FFT have multiplications by powers of two
  $\rightarrow$ Can use shifts instead of multiplications
  $\rightarrow$ Particularly useful on the Cortex-M3!

Fermat Number Transforms (FNT)

- Recall: For NTTs we require $2^k \mid q - 1$ with prime q
- Fermat numbers: $2^{2^k} + 1$
- Fermat primes: 3, 5, 17, 257, 65537
- Example: 65537
  - $\omega_2 = -1 = 2^{16}$
  - $\omega_4 = 2^8$
  - $\omega_8 = 2^4$
  - $\omega_{16} = 2^2$
  - $\omega_{32} = 2^1$
- First 5 layers of the FFT have multiplications by powers of two
  $\rightarrow$ Can use shifts instead of multiplications
  $\rightarrow$ Particularly useful on the Cortex-M3!

Parameter Choices

- High-level goal: Efficient N-bit (2048, 4096) multiplication
  - Chunk up number in ℓ-bit coefficients
  - Pad with zeros to have an n-coefficient polynomial
  - Each coefficient is modulo small $q = q_1 q_2$
  - Perform a k-layer NTT-based multiplication
- Constraint 1: We want to make efficient use of the available multipliers
  - M3:
    `mul` $32 \times 32 \rightarrow 32$ bit (low multiplication)
    $\rightarrow$ want to limit moduli q_i to 16 bit
    $\rightarrow$ special case: FNT with $q_2 = 65537$ for NTT
  - M55:
    `vmul` $32 \times 32 \rightarrow 32$ bit (low multiplication)
    `vqrdmulh`: $32 \times 32 \rightarrow 32$ bit (rounding doubling high multiplication)
    $\rightarrow$ want to limit moduli q_i to 32 bit

Parameter Choices

- High-level goal: Efficient N-bit (2048, 4096) multiplication
  - Chunk up number in ℓ-bit coefficients
  - Pad with zeros to have an n-coefficient polynomial
  - Each coefficient is modulo small $q = q_1 q_2$
  - Perform a k-layer NTT-based multiplication
- Constraint 1: We want to make efficient use of the available multipliers
  - M3:
    `mul` $32 \times 32 \rightarrow 32$ bit (low multiplication)
    $\rightarrow$ want to limit moduli q_i to 16 bit
    $\rightarrow$ special case: FNT with $q_2 = 65537$ for NTT
  - M55:
    `vmul` $32 \times 32 \rightarrow 32$ bit (low multiplication)
    `vqrdmulh`: $32 \times 32 \rightarrow 32$ bit (rounding doubling high multiplication)
    $\rightarrow$ want to limit moduli q_i to 32 bit

Parameter Choices (2)

- Constraint 2: Need to be able to represent the $2N$-bit result
  $\rightarrow n \geq \lceil 2N/\ell \rceil$
- Constraint 3: n should be NTT-friendly
  $\rightarrow$ power of two or small multiple of power of two
- Constraint 4: Require NTT-friendly modulus
  $\rightarrow$ restrict to prime q_1, q_2
  $\rightarrow 2^k \mid q_1 - 1$ and $2^k \mid q_2 - 1$ to have the required principal roots of unity
- Constraint 5: Coefficients of polynomial product must not overflow q
  $\rightarrow q \geq \lceil N/\ell \rceil \cdot 2^{2\ell}$

Parameter Choices (2)

- Constraint 2: Need to be able to represent the $2N$-bit result
  $\rightarrow n \geq \lceil 2N/\ell \rceil$
- Constraint 3: n should be NTT-friendly
  $\rightarrow$ power of two or small multiple of power of two
- Constraint 4: Require NTT-friendly modulus
  $\rightarrow$ restrict to prime q_1, q_2
  $\rightarrow 2^k \mid q_1 - 1$ and $2^k \mid q_2 - 1$ to have the required principal roots of unity
- Constraint 5: Coefficients of polynomial product must not overflow q
  $\rightarrow q \geq \lceil N/\ell \rceil \cdot 2^{2\ell}$

Parameter Choices (2)

- Constraint 2: Need to be able to represent the $2N$-bit result
  $\rightarrow n \geq \lceil 2N/\ell \rceil$
- Constraint 3: n should be NTT-friendly
  $\rightarrow$ power of two or small multiple of power of two
- Constraint 4: Require NTT-friendly modulus
  $\rightarrow$ restrict to prime q_1, q_2
  $\rightarrow 2^k \mid q_1 - 1$ and $2^k \mid q_2 - 1$ to have the required principal roots of unity
- Constraint 5: Coefficients of polynomial product must not overflow q
  $\rightarrow q \geq \lceil N/\ell \rceil \cdot 2^{2\ell}$

Parameter Choices (2)

- Constraint 2: Need to be able to represent the $2N$-bit result
  $\rightarrow n \geq \lceil 2N/\ell \rceil$
- Constraint 3: n should be NTT-friendly
  $\rightarrow$ power of two or small multiple of power of two
- Constraint 4: Require NTT-friendly modulus
  $\rightarrow$ restrict to prime q_1, q_2
  $\rightarrow 2^k \mid q_1 - 1$ and $2^k \mid q_2 - 1$ to have the required principal roots of unity
- Constraint 5: Coefficients of polynomial product must not overflow q
  $\rightarrow q \geq \lceil N/\ell \rceil \cdot 2^{2\ell}$

Parameter Choices (3)

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	$128 \cdot 3 = 2^7 \cdot 3$	$114\,826\,273 \cdot 128\,919\,937$

Low-level: Modular Coefficient Multiplication on Cortex-M3

NTT: Montgomery mult

`mul a, a, b`

`mul t, a,` $-q^{-1} \bmod {}^{\pm}2^{16}$

`sxth t, t`

`mla a, t, q, a`

`asr a, a, #16`

NTT: Barrett reductions

`mul t, a,` $\lceil R/q \rfloor$

`add t, t, #(R/2)`

`asr t, t,` #$\log_2$ `R`

`mls a, t, q, a`

FNT: Reduction mod 65537

`ubfx t, a, #0, #16`

`sub a, t, a, asr#16`

Low-level: Modular Coefficient Multiplication on Cortex-M55

- We make use of “Barrett multiplication” from Becker–Hwang–Kannwischer–Yang–Yang (CHES 2022) `https://tches.iacr.org/index.php/TCHES/article/view/9295`
- Pre-compute: $b' = \frac{\lfloor b2^{32}/q \rceil}{2}$
- Implement 4 parallel Barrett multiplications

```
vmul l, a, b
vqrdmulh h, a, b'
vmla l, h, q
```

Application: RSA

- Integer multiplication is dominating operation within RSA
- Need to compute `expmod` modulo $n = pq$ (4096-bit n, 2048-bit p, q)
- Encryption:
  $c = m^e \bmod n$ (usually, $e = 65537$)
  $\rightarrow$ requires 4096-bit multiplication; e may leak via timing
- Decryption:
  $c^d \bmod n = CRT(c^d \bmod p, c^d \bmod q)$
  $\rightarrow$ requires 2048-bit multiplication; d must not leak via timing
- Fixed-window exponentiation for decryption
  $\rightarrow$ Use constant-time table look-up!

Application: RSA

- Integer multiplication is dominating operation within RSA
- Need to compute `expmod` modulo $n = pq$ (4096-bit n, 2048-bit p, q)
- Encryption:
  $c = m^e \bmod n$ (usually, $e = 65537$)
  $\rightarrow$ requires 4096-bit multiplication; e may leak via timing
- Decryption:
  $c^d \bmod n = CRT(c^d \bmod p, c^d \bmod q)$
  $\rightarrow$ requires 2048-bit multiplication; d must not leak via timing
- Fixed-window exponentiation for decryption
  $\rightarrow$ Use constant-time table look-up!

Application: RSA

- Integer multiplication is dominating operation within RSA
- Need to compute `expmod` modulo $n = pq$ (4096-bit n, 2048-bit p, q)
- Encryption:
  $c = m^e \bmod n$ (usually, $e = 65537$)
  $\rightarrow$ requires 4096-bit multiplication; e may leak via timing
- Decryption:
  $c^d \bmod n = CRT(c^d \bmod p, c^d \bmod q)$
  $\rightarrow$ requires 2048-bit multiplication; d must not leak via timing
- Fixed-window exponentiation for decryption
  $\rightarrow$ Use constant-time table look-up!

Application: RSA

- Integer multiplication is dominating operation within RSA
- Need to compute `expmod` modulo $n = pq$ (4096-bit n, 2048-bit p, q)
- Encryption:
  $c = m^e \bmod n$ (usually, $e = 65537$)
  $\rightarrow$ requires 4096-bit multiplication; e may leak via timing
- Decryption:
  $c^d \bmod n = CRT(c^d \bmod p, c^d \bmod q)$
  $\rightarrow$ requires 2048-bit multiplication; d must not leak via timing
- Fixed-window exponentiation for decryption
  $\rightarrow$ Use constant-time table look-up!

RSA: `modmul`

- Within `expmod`, we need a `modmul`
- Common way to implement `modmul`: Montgomery multiplication

  $c = a \cdot b$

  $t = c \cdot p^{-1} \bmod R$

  $r = (c - t \cdot p)/R$
- We can actually implement this using NTTs:

  $c = iNTT(NTT(a) \circ NTT(b))$

  $t = iNTT(NTT(c \bmod R) \circ NTT(p^{-1} \bmod R))$

  $r = (c - iNTT(NTT(t \bmod R) \circ NTT(p)))/R$
- We can pre-compute $NTT(p)$ and $NTT(p^{-1} \bmod R)$
- Need $4\times$ NTT and $3\times$ iNTT
- Squaring: $a = b \rightarrow$ only $3\times$ NTT

RSA: `modmul`

- Within `expmod`, we need a `modmul`
- Common way to implement `modmul`: Montgomery multiplication
  $c = a \cdot b$
  $t = c \cdot p^{-1} \bmod R$
  $r = (c - t \cdot p)/R$
- We can actually implement this using NTTs:
  $c = iNTT(NTT(a) \circ NTT(b))$
  $t = iNTT(NTT(c \bmod R) \circ NTT(p^{-1} \bmod R))$
  $r = (c - iNTT(NTT(t \bmod R) \circ NTT(p)))/R$
- We can pre-compute $NTT(p)$ and $NTT(p^{-1} \bmod R)$
- Need $4\times$ NTT and $3\times$ iNTT
- Squaring: $a = b \rightarrow$ only $3\times$ NTT

RSA: `modmul`

- Within `expmod`, we need a `modmul`
- Common way to implement `modmul`: Montgomery multiplication
  $c = a \cdot b$
  $t = c \cdot p^{-1} \bmod R$
  $r = (c - t \cdot p)/R$
- We can actually implement this using NTTs:
  $c = iNTT(NTT(a) \circ NTT(b))$
  $t = iNTT(NTT(c \bmod R) \circ NTT(p^{-1} \bmod R))$
  $r = (c - iNTT(NTT(t \bmod R) \circ NTT(p)))/R$
- We can pre-compute $NTT(p)$ and $NTT(p^{-1} \bmod R)$
- Need $4\times$ NTT and $3\times$ iNTT
- Squaring: $a = b \rightarrow$ only $3\times$ NTT

RSA: `modmul`

- Within `expmod`, we need a `modmul`
- Common way to implement `modmul`: Montgomery multiplication
  $c = a \cdot b$
  $t = c \cdot p^{-1} \bmod R$
  $r = (c - t \cdot p)/R$
- We can actually implement this using NTTs:
  $c = iNTT(NTT(a) \circ NTT(b))$
  $t = iNTT(NTT(c \bmod R) \circ NTT(p^{-1} \bmod R))$
  $r = (c - iNTT(NTT(t \bmod R) \circ NTT(p)))/R$
- We can pre-compute $NTT(p)$ and $NTT(p^{-1} \bmod R)$
- Need $4\times$ NTT and $3\times$ iNTT
- Squaring: $a = b \rightarrow$ only $3\times$ NTT

RSA: `modmul`

- Within `expmod`, we need a `modmul`
- Common way to implement `modmul`: Montgomery multiplication

  $c = a \cdot b$

  $t = c \cdot p^{-1} \bmod R$

  $r = (c - t \cdot p)/R$
- We can actually implement this using NTTs:

  $c = iNTT(NTT(a) \circ NTT(b))$

  $t = iNTT(NTT(c \bmod R) \circ NTT(p^{-1} \bmod R))$

  $r = (c - iNTT(NTT(t \bmod R) \circ NTT(p)))/R$
- We can pre-compute $NTT(p)$ and $NTT(p^{-1} \bmod R)$
- Need $4\times$ NTT and $3\times$ iNTT
- Squaring: $a = b \rightarrow$ only $3\times$ NTT

Results: 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[1]		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[1]		1 102 151	–	70 443 207	5 505 856 187

RSA-2048 using CRT for decryption

[1] `https://bearssl.org/`

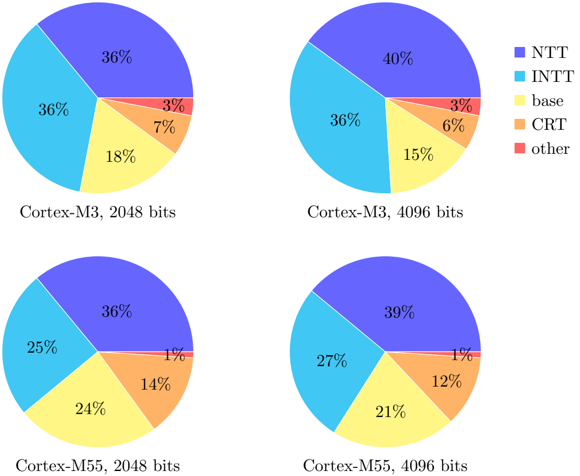
Results: Cortex-M55

	n	mulmod	sqrmod	expmod$_{\text{public}}$	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[1]		41 443	–	884 416	108 441 240
BearSSL[2]		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[1]		152 371	–	3 223 797	755 391 521
BearSSL[2]		328 801	–	21 254 533	1 646 834 048

RSA-2048 using CRT for decryption

[1] https://github.com/Mbed-TLS/mbedtls
[2] https://bearssl.org/

Profiling of `mulmod`



Conclusions

- NTT-based integer multiplication can be superior for relatively small sizes
  - We implemented 2048-bit and 4096-bit multiplications
  - We target two common Arm platforms: Cortex-M3 and Cortex-M55
- Progress in post-quantum cryptography (lattice-based crypto) helps speeding up pre-quantum crypto
- NTT are much easier to vectorize than other integer-multiplication algorithms
  - Gives advantage on platforms supporting vector instructions, e.g., Cortex-M55

Conclusions

- NTT-based integer multiplication can be superior for relatively small sizes
  - We implemented 2048-bit and 4096-bit multiplications
  - We target two common Arm platforms: Cortex-M3 and Cortex-M55
- Progress in post-quantum cryptography (lattice-based crypto) helps speeding up pre-quantum crypto
- NTT are much easier to vectorize than other integer-multiplication algorithms
  - Gives advantage on platforms supporting vector instructions, e.g., Cortex-M55

Conclusions

- NTT-based integer multiplication can be superior for relatively small sizes
  - We implemented 2048-bit and 4096-bit multiplications
  - We target two common Arm platforms: Cortex-M3 and Cortex-M55
- Progress in post-quantum cryptography (lattice-based crypto) helps speeding up pre-quantum crypto
- NTT are much easier to vectorize than other integer-multiplication algorithms
  - Gives advantage on platforms supporting vector instructions, e.g., Cortex-M55

Conclusions: Limitations

- Limited to certain integer sizes
- Limited to chosen platforms (Cortex-M3, Cortex-M55)
- Our code is heavily unrolled
  - May be problematic on the Cortex-M3 due to ROM/flash constraints
  - Performance overhead of re-rolling the code is hopefully small
- `expmod` allows some pre-computation (modulus and its inverse in NTT domain) favouring NTT-based multiplication
  - General-purpose modular multiplication will be slower

Conclusions: Limitations

- Limited to certain integer sizes
- Limited to chosen platforms (Cortex-M3, Cortex-M55)
- Our code is heavily unrolled
  - May be problematic on the Cortex-M3 due to ROM/flash constraints
  - Performance overhead of re-rolling the code is hopefully small
- `expmod` allows some pre-computation (modulus and its inverse in NTT domain) favouring NTT-based multiplication
  - General-purpose modular multiplication will be slower

Conclusions: Limitations

- Limited to certain integer sizes
- Limited to chosen platforms (Cortex-M3, Cortex-M55)
- Our code is heavily unrolled
    - May be problematic on the Cortex-M3 due to ROM/flash constraints
    - Performance overhead of re-rolling the code is hopefully small
- `expmod` allows some pre-computation (modulus and its inverse in NTT domain) favouring NTT-based multiplication
    - General-purpose modular multiplication will be slower

Conclusions: Limitations

- How does this translate to other platforms?
  - Unclear
  - For example, Arm Cortex-M4 has powerful multiplication instructions (single cycle `umaal`) that help schoolbook much more than NTTs
  - Armv8-A/Armv9-A processors would be interesting to look at in the future
- Can we scale this to any size of integers (≥ 2048)?
  - Unclear
  - Parameters have been carefully picked for the two sizes (2048, 4096)
  - General-purpose integer multiplication code is trickier!

Conclusions: Limitations

- How does this translate to other platforms?
  - Unclear
  - For example, Arm Cortex-M4 has powerful multiplication instructions (single cycle `umaal`) that help schoolbook much more than NTTs
  - Armv8-A/Armv9-A processors would be interesting to look at in the future
- Can we scale this to any size of integers (≥ 2048)?
  - Unclear
  - Parameters have been carefully picked for the two sizes (2048, 4096)
  - General-purpose integer multiplication code is trickier!

Thanks!

https://eprint.iacr.org/2022/439

https://github.com/ntt-int-mul/ntt-int-mul-m3

https://gitlab.com/arm-research/security/pqmx