McBits: Fast code-based cryptography

Peter Schwabe

Radboud University Nijmegen, The Netherlands

Joint work with Daniel Bernstein, Tung Chou

December 17, 2013

IMA International Conference on Cryptography and Coding

Introduction: the bigger context

Public-key encryption

- Alice generates a key pair (sk, pk), publishes pk, keeps sk secret
- Bob takes some message M and pk and computes an **ciphertext** C, sends C to Alice
- Alice uses sk to decrypt C and obtain M

Introduction: the bigger context

Public-key encryption

- Alice generates a key pair (sk, pk), publishes pk, keeps sk secret
- Bob takes some message M and pk and computes an **ciphertext** C, sends C to Alice
- Alice uses sk to decrypt C and obtain M

Implementation targets

- Secure
- Fast
- (Small, low energy, low-power,...)

Secure Implementations

- “Traditional” cryptographic security: all attacks take $\geq 2^{128}$ operations

Secure Implementations

- "Traditional" cryptographic security: all attacks take $\geq 2^{128}$ operations
- Implementation security: no leakage through side channels
- Most relevant for desktops and servers: timing attacks
- Idea:
  - Secret information influences time taken by software
  - Attacker measures time, computes influence^{-1} to obtain secret information

Secure Implementations

- "Traditional" cryptographic security: all attacks take $\geq 2^{128}$ operations
- Implementation security: no leakage through side channels
- Most relevant for desktops and servers: timing attacks
- Idea:
  - Secret information influences time taken by software
  - Attacker measures time, computes influence^{-1} to obtain secret information
- *Constant-time* software avoids such timing leaks:

Secure Implementations

- "Traditional" cryptographic security: all attacks take $\geq 2^{128}$ operations
- Implementation security: no leakage through side channels
- Most relevant for desktops and servers: timing attacks
- Idea:
  - Secret information influences time taken by software
  - Attacker measures time, computes influence^{-1} to obtain secret information
- *Constant-time* software avoids such timing leaks:
  - No secret branch conditions

Secure Implementations

- "Traditional" cryptographic security: all attacks take $\geq 2^{128}$ operations
- Implementation security: no leakage through side channels
- Most relevant for desktops and servers: timing attacks
- Idea:
  - Secret information influences time taken by software
  - Attacker measures time, computes influence^{-1} to obtain secret information
- *Constant-time* software avoids such timing leaks:
  - No secret branch conditions
  - No memory access with secret address (*cache timing*)

Fast Implementation

- This talk: focus on high throughput for servers
- Target micro-architecture: Intel Sandy Bridge/Ivy Bridge
- Techniques also interesting for other (micro-)architectures

Fast Implementation

- This talk: focus on high throughput for servers
- Target micro-architecture: Intel Sandy Bridge/Ivy Bridge
- Techniques also interesting for other (micro-)architectures

Vector arithmetic

- All "large" processors offer arithmetic on vectors of data
- Highest arithmetic throughput, example (Sandy Bridge):
  - Three 32-bit additions per cycle
  - Two 4×32-bit vector additions per cycle

Fast Implementation

- This talk: focus on high throughput for servers
- Target micro-architecture: Intel Sandy Bridge/Ivy Bridge
- Techniques also interesting for other (micro-)architectures

Vector arithmetic

- All "large" processors offer arithmetic on vectors of data
- Highest arithmetic throughput, example (Sandy Bridge):
  - Three 32-bit additions per cycle
  - Two 4×32-bit vector additions per cycle
- Also fast: full-vector loads
- Low performance for branches, independent vector-element loads

Fast Implementation

- This talk: focus on high throughput for servers
- Target micro-architecture: Intel Sandy Bridge/Ivy Bridge
- Techniques also interesting for other (micro-)architectures

Vector arithmetic

- All "large" processors offer arithmetic on vectors of data
- Highest arithmetic throughput, example (Sandy Bridge):
  - Three 32-bit additions per cycle
  - Two 4×32-bit vector additions per cycle
- Also fast: full-vector loads
- Low performance for branches, independent vector-element loads
- Synergie between efficient vectorization and timing-attack protection

Bitslicing

- Any n-bit register is a vector register with n 1-bit elements
- Operations on such bit vectors are `XOR`, `OR`, `AND`

Bitslicing

- Any n-bit register is a vector register with n 1-bit elements
- Operations on such bit vectors are `XOR`, `OR`, `AND`
- This is called *bitslicing*, introduced by Biham in 1997 for DES

Bitslicing

- Any n-bit register is a vector register with n 1-bit elements
- Operations on such bit vectors are `XOR`, `OR`, `AND`
- This is called *bitslicing*, introduced by Biham in 1997 for DES
- Other views on bitslicing:
  - Computations on a transposition of data
  - Simulation of hardware implementations in software

Bitslicing

- Any n-bit register is a vector register with n 1-bit elements
- Operations on such bit vectors are `XOR`, `OR`, `AND`
- This is called *bitslicing*, introduced by Biham in 1997 for DES
- Other views on bitslicing:
  - Computations on a transposition of data
  - Simulation of hardware implementations in software
- Needs large degree of data-level parallelism (e.g., $128\times$)
- Size of active data set increases massively (e.g., $128\times$)
- Typical consequence: more loads and stores (that easily become the performance bottleneck)

A code-based cryptosystem

System parameters

- Integers m, n, t, k, such that
  - $n \leq 2^m$
  - $k = n - mt$
  - $t \geq 2$

Example

- $m = 12$,
  $n = 4096$
  $k = 3604$
  $t = 41$

A code-based cryptosystem

System parameters

- Integers m, n, t, k, such that
  - $n \leq 2^m$
  - $k = n - mt$
  - $t \geq 2$
- An s-bit-key stream cipher S

Example

- $m = 12$,
  $n = 4096$
  $k = 3604$
  $t = 41$
- $S =$ Salsa20 $(s = 256)$

A code-based cryptosystem

System parameters

- Integers m, n, t, k, such that
  - $n \leq 2^m$
  - $k = n - mt$
  - $t \geq 2$
- An s-bit-key stream cipher S
- An a-bit-key authenticator (MAC) A

Example

- $m = 12$,
  $n = 4096$
  $k = 3604$
  $t = 41$
- $S =$ Salsa20 $(s = 256)$
- $A =$ Poly1305 $(a = 256)$

A code-based cryptosystem

System parameters

- Integers m, n, t, k, such that
  - $n \leq 2^m$
  - $k = n - mt$
  - $t \geq 2$
- An s-bit-key stream cipher S
- An a-bit-key authenticator (MAC) A
- An $(s + a)$-bit-output hash function H

Example

- $m = 12$,
  $n = 4096$
  $k = 3604$
  $t = 41$
- $S =$ Salsa20 $(s = 256)$
- $A =$ Poly1305 $(a = 256)$
- $H =$ SHA-512

Key generation

Secret key

- A random sequence $(\alpha_1, \ldots, \alpha_n)$ of distinct elements in $\mathbb{F}_{2^m}$
- An irreducible degree-t polynomial $g \in \mathbb{F}_{2^m}[x]$

Key generation

Secret key

- A random sequence $(\alpha_1, \ldots, \alpha_n)$ of distinct elements in $\mathbb{F}_{2^m}$
- An irreducible degree-t polynomial $g \in \mathbb{F}_{2^m}[x]$
- Compute the secret matrix

$$\begin{pmatrix} 1/g(\alpha_1) & 1/g(\alpha_2) & \cdots & 1/g(\alpha_n) \\ \alpha_1/g(\alpha_1) & \alpha_2/g(\alpha_2) & \cdots & \alpha_n/g(\alpha_n) \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{t-1}/g(\alpha_1) & \alpha_2^{t-1}/g(\alpha_2) & \cdots & \alpha_n^{t-1}/g(\alpha_n) \end{pmatrix} \in \mathbb{F}_{2^m}^{t \times n}$$

Key generation

Secret key

- A random sequence $(\alpha_1, \dots, \alpha_n)$ of distinct elements in $\mathbb{F}_{2^m}$
- An irreducible degree-t polynomial $g \in \mathbb{F}_{2^m}[x]$
- Compute the secret matrix

$$\begin{pmatrix} 1/g(\alpha_1) & 1/g(\alpha_2) & \cdots & 1/g(\alpha_n) \\ \alpha_1/g(\alpha_1) & \alpha_2/g(\alpha_2) & \cdots & \alpha_n/g(\alpha_n) \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{t-1}/g(\alpha_1) & \alpha_2^{t-1}/g(\alpha_2) & \cdots & \alpha_n^{t-1}/g(\alpha_n) \end{pmatrix} \in \mathbb{F}_{2^m}^{t \times n}$$

- Replace all entries by a column of m bits in a standard basis of $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$
- Obtain a matrix $H_{sec} \in \mathbb{F}_2^{mt \times n}$

Key generation

Secret key

- A random sequence $(\alpha_1, \ldots, \alpha_n)$ of distinct elements in $\mathbb{F}_{2^m}$
- An irreducible degree-t polynomial $g \in \mathbb{F}_{2^m}[x]$
- Compute the secret matrix

$$\begin{pmatrix} 1/g(\alpha_1) & 1/g(\alpha_2) & \cdots & 1/g(\alpha_n) \\ \alpha_1/g(\alpha_1) & \alpha_2/g(\alpha_2) & \cdots & \alpha_n/g(\alpha_n) \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{t-1}/g(\alpha_1) & \alpha_2^{t-1}/g(\alpha_2) & \cdots & \alpha_n^{t-1}/g(\alpha_n) \end{pmatrix} \in \mathbb{F}_{2^m}^{t \times n}$$

- Replace all entries by a column of m bits in a standard basis of $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$
- Obtain a matrix $H_{sec} \in \mathbb{F}_2^{mt \times n}$
- H_{sec} is a *secret* parity-check matrix of the Goppa code $\Gamma = \Gamma_2(\alpha_1, \ldots, \alpha_n, g)$

Key generation

Secret key

- A random sequence $(\alpha_1, \ldots, \alpha_n)$ of distinct elements in $\mathbb{F}_{2^m}$
- An irreducible degree-t polynomial $g \in \mathbb{F}_{2^m}[x]$
- Compute the secret matrix

$$\begin{pmatrix} 1/g(\alpha_1) & 1/g(\alpha_2) & \cdots & 1/g(\alpha_n) \\ \alpha_1/g(\alpha_1) & \alpha_2/g(\alpha_2) & \cdots & \alpha_n/g(\alpha_n) \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{t-1}/g(\alpha_1) & \alpha_2^{t-1}/g(\alpha_2) & \cdots & \alpha_n^{t-1}/g(\alpha_n) \end{pmatrix} \in \mathbb{F}_{2^m}^{t \times n}$$

- Replace all entries by a column of m bits in a standard basis of $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$
- Obtain a matrix $H_{sec} \in \mathbb{F}_2^{mt \times n}$
- H_{sec} is a *secret* parity-check matrix of the Goppa code $\Gamma = \Gamma_2(\alpha_1, \ldots, \alpha_n, g)$
- The secret key is $(\alpha_1, \ldots, \alpha_n, g)$

Key generation

Public key

- Perform Gaussian elimination on H_{sec} to obtain a matrix H_{pub} whose left $tm \times tm$ submatrix is the identity matrix
- H_{pub} is a *public* parity-check matrix for Γ
- The public key is H_{pub}

Encryption

- Generate a random weight-t vector $e \in \mathbb{F}_2^n$
- Compute $w = H_{pub}e$
- Compute $H(e)$ to obtain an $(s+a)$-bit string (k_{enc}, k_{auth})
- Encrypt the message M with the stream cipher S under key k_{enc} to obtain ciphertext C
- Compute authentication tag a on C using A with key k_{auth}
- Send (a, w, C)

Decryption

- Receive (a, w, C)
- Decode w to obtain weight-t string e
- Hash e with H to obtain (k_{enc}, k_{auth})
- Verify that a is a valid authentication tag on C using A with k_{auth}
- Use S with k_{enc} to decrypt and obtain M

Software implementation, first considerations

Key generation

- Key generation is not performance critical
- Some hassle to make constant-time, but possible

Software implementation, first considerations

Key generation

- Key generation is not performance critical
- Some hassle to make constant-time, but possible

Encryption

- Typical view: adding up t columns of mt bits each

Software implementation, first considerations

Key generation

- Key generation is not performance critical
- Some hassle to make constant-time, but possible

Encryption

- Typical view: adding up t columns of mt bits each
- Column positions are *secret*, need to load all columns
- Arithmetic (masking) to xor the desired columns

Software implementation, first considerations

Key generation

- Key generation is not performance critical
- Some hassle to make constant-time, but possible

Encryption

- Typical view: adding up t columns of mt bits each
- Column positions are *secret*, need to load all columns
- Arithmetic (masking) to xor the desired columns
- This talk: ignore implementation of H, S, and A

Software implementation, first considerations

Key generation

- Key generation is not performance critical
- Some hassle to make constant-time, but possible

Encryption

- Typical view: adding up t columns of mt bits each
- Column positions are *secret*, need to load all columns
- Arithmetic (masking) to xor the desired columns
- This talk: ignore implementation of H, S, and A

Decryption

- Decryption is mainly decoding, lots of operations in $\mathbb{F}_{2^m}$
- Decryption has to run in constant time!
- Obviously, decoding of w is the interesting part

A closer look at decoding

- Start with *some* $v \in \mathbb{F}_2^n$, such that $H_{pub}v = w$

A closer look at decoding

- Start with *some* $v \in \mathbb{F}_2^n$, such that $H_{pub}v = w$
- Compute a Goppa syndrome $s_0, \ldots, s_{2t-1}$
- Use Berlekamp's algorithm to obtain *error-locator polynomial* f of degree t

A closer look at decoding

- Start with *some* $v \in \mathbb{F}_2^n$, such that $H_{pub}v = w$
- Compute a Goppa syndrome $s_0, \ldots, s_{2t-1}$
- Use Berlekamp's algorithm to obtain *error-locator polynomial* f of degree t
- Compute t roots of this polynomial
- For each root $r_j = \alpha_i$, set error bit at position i in e

A closer look at decoding

- Start with *some* $v \in \mathbb{F}_2^n$, such that $H_{pub}v = w$
- Compute a Goppa syndrome $s_0, \ldots, s_{2t-1}$
- Use Berlekamp's algorithm to obtain *error-locator polynomial* f of degree t
- Compute t roots of this polynomial
- For each root $r_j = \alpha_i$, set error bit at position i in e
- All these computation work on medium-size polynomials over $\mathbb{F}_{2^m}$

A closer look at decoding

- Start with *some* $v \in \mathbb{F}_2^n$, such that $H_{pub}v = w$
- Compute a Goppa syndrome $s_0, \ldots, s_{2t-1}$
- Use Berlekamp's algorithm to obtain *error-locator polynomial* f of degree t
- Compute t roots of this polynomial
- For each root $r_j = \alpha_i$, set error bit at position i in e
- All these computation work on medium-size polynomials over $\mathbb{F}_{2^m}$
- Let's now fix the example parameters from above $(n = 2^m = 4096, t = 41)$

Representing elements of $\mathbb{F}_{2^m}$

Option I

- Use 16-bit integer values (`unsigned short`)
- Addition is simply XOR (we really XOR 64 bits, but ignore most of those)

Representing elements of $\mathbb{F}_{2^m}$

Option I

- Use 16-bit integer values (`unsigned short`)
- Addition is simply XOR (we really XOR 64 bits, but ignore most of those)
- Multiplication:
  - Use table lookups (not constant time!)

Representing elements of $\mathbb{F}_{2^m}$

Option I

- Use 16-bit integer values (`unsigned short`)
- Addition is simply XOR (we really XOR 64 bits, but ignore most of those)
- Multiplication:
  - Use table lookups (not constant time!)
  - Use carryless multiplier, e.g., `pclmulqdq` (not available on most architectures, again ignores most of the 64×64-bit multiplication)

Representing elements of $\mathbb{F}_{2^m}$

Option I

- Use 16-bit integer values (`unsigned short`)
- Addition is simply XOR (we really XOR 64 bits, but ignore most of those)
- Multiplication:
  - Use table lookups (not constant time!)
  - Use carryless multiplier, e.g., `pclmulqdq` (not available on most architectures, again ignores most of the 64×64-bit multiplication)
  - Squaring uses the same algorithm as multiplication

Representing elements of $\mathbb{F}_{2^m}$

Option II

- Use bitsliced representation in 256-bit `YMM` (or 128-bit `XMM` registers)
- Needs many parallel computations, obtain parallelism from independent decryption operations
- We only really care about speed when we have *many* decryptions

Representing elements of $\mathbb{F}_{2^m}$

Option II

- Use bitsliced representation in 256-bit `YMM` (or 128-bit `XMM` registers)
- Needs many parallel computations, obtain parallelism from independent decryption operations
- We only really care about speed when we have *many* decryptions
- Addition is 12 vector XORs for 256 parallel additions (much faster!)

Representing elements of $\mathbb{F}_{2^m}$

Option II

- Use bitsliced representation in 256-bit YMM (or 128-bit XMM registers)
- Needs many parallel computations, obtain parallelism from independent decryption operations
- We only really care about speed when we have *many* decryptions
- Addition is 12 vector XORs for 256 parallel additions (much faster!)
- Multiplication is easily constant time, but is it fast?
- How about squaring, can it be faster?

Bitsliced multiplication in $\mathbb{F}_{2^{12}}$

- Split into 12-coefficient polynomial multiplication and subsequent reduction
- Reduction trinomial $x^{12} + x^3 + 1$

Bitsliced multiplication in $\mathbb{F}_{2^{12}}$

- Split into 12-coefficient polynomial multiplication and subsequent reduction
- Reduction trinomial $x^{12} + x^3 + 1$
- Schoolbook multiplication needs 144 ANDs and 121 XORs

Bitsliced multiplication in $\mathbb{F}_{2^{12}}$

- Split into 12-coefficient polynomial multiplication and subsequent reduction
- Reduction trinomial $x^{12} + x^3 + 1$
- Schoolbook multiplication needs 144 ANDs and 121 XORs
- Much better: Karatsuba
  - Karatsuba:

$$\begin{aligned} & (a_0 + x^n a_1)(b_0 + x^n b_1) \\ = \quad & a_0 b_0 + x^n((a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1) + x^{2n} a_1 b_1 \end{aligned}$$

Bitsliced multiplication in $\mathbb{F}_{2^{12}}$

- Split into 12-coefficient polynomial multiplication and subsequent reduction
- Reduction trinomial $x^{12} + x^3 + 1$
- Schoolbook multiplication needs 144 ANDs and 121 XORs
- Much better: refined Karatsuba
  - Karatsuba:

$$
\begin{aligned}
& (a_0 + x^n a_1)(b_0 + x^n b_1) \\
= \quad & a_0 b_0 + x^n((a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1) + x^{2n} a_1 b_1
\end{aligned}
$$

  - Refined Karatsuba:

$$
\begin{aligned}
& (a_0 + x^n a_1)(b_0 + x^n b_1) \\
= \quad & (1 - x^n)(a_0 b_0 - x^n a_1 b_1) + x^n (a_0 + a_1)(b_0 + b_1)
\end{aligned}
$$

- Refined Karatsuba uses $M_{2n} = 3M_n + 7n - 3$ instead of $M_{2n} = 3M_n + 8n - 4$ bit operations
- For details see Bernstein, "Batch binary Edwards", Crypto 2009

Bitsliced performance

- One level of refined Karatsuba: 114 XORs, 108 ANDs

Bitsliced performance

- One level of refined Karatsuba: 114 XORs, 108 ANDs
- 222 bit operations are worse than 208 by Bernstein 2009, but better scheduling

Bitsliced performance

- One level of refined Karatsuba: 114 XORs, 108 ANDs
- 222 bit operations are worse than 208 by Bernstein 2009, but better scheduling
- Reduction takes 24 XORs, a total of 246 bit operations
- On Ivy Bridge: 247 cycles for 256 multiplications

Bitsliced performance

- One level of refined Karatsuba: 114 XORs, 108 ANDs
- 222 bit operations are worse than 208 by Bernstein 2009, but better scheduling
- Reduction takes 24 XORs, a total of 246 bit operations
- On Ivy Bridge: 247 cycles for 256 multiplications
- Bitsliced squaring is only reduction: 7 XORs

Bitsliced performance

- One level of refined Karatsuba: 114 XORs, 108 ANDs
- 222 bit operations are worse than 208 by Bernstein 2009, but better scheduling
- Reduction takes 24 XORs, a total of 246 bit operations
- On Ivy Bridge: 247 cycles for 256 multiplications
- Bitsliced squaring is only reduction: 7 XORs

Bitsliced performance

- One level of refined Karatsuba: 114 XORs, 108 ANDs
- 222 bit operations are worse than 208 by Bernstein 2009, but better scheduling
- Reduction takes 24 XORs, a total of 246 bit operations
- On Ivy Bridge: 247 cycles for 256 multiplications
- Bitsliced squaring is only reduction: 7 XORs

Summary:

- Bitsliced *addition* is much faster than non bitsliced
- Bitsliced *multiplication* is faster
- Bitsliced squaring is much faster (not very relevant)

Bitsliced performance

- One level of refined Karatsuba: 114 XORs, 108 ANDs
- 222 bit operations are worse than 208 by Bernstein 2009, but better scheduling
- Reduction takes 24 XORs, a total of 246 bit operations
- On Ivy Bridge: 247 cycles for 256 multiplications
- Bitsliced squaring is only reduction: 7 XORs

Summary:

- Bitsliced *addition* is much faster than non bitsliced
- Bitsliced *multiplication* is faster
- Bitsliced squaring is much faster (not very relevant)
- In the following: High-level algorithms that drastically reduce the number of multiplications

Root finding, the classical way

- Task: Find all t roots of a degree-t error-locator polynomial f
- Let $f = c_{41}x^{41} + c_{40} + x^{40} + \cdots + c_0$

Root finding, the classical way

- Task: Find all t roots of a degree-t error-locator polynomial f
- Let $f = c_{41}x^{41} + c_{40} + x^{40} + \cdots + c_0$
- Try all elements of F_{2^m}, Horner scheme takes 41 mul, 41 add per element

Root finding, the classical way

- Task: Find all t roots of a degree-t error-locator polynomial f
- Let $f = c_{41}x^{41} + c_{40} + x^{40} + \cdots + c_0$
- Try all elements of F_{2^m}, Horner scheme takes 41 mul, 41 add per element
- Chien search: Compute $c_i g^i, c_i g^{2i}, c_i g^{3i}$ etc.
- Same operation count but different structure

Root finding, the classical way

- Task: Find all t roots of a degree-t error-locator polynomial f
- Let $f = c_{41}x^{41} + c_{40} + x^{40} + \cdots + c_0$
- Try all elements of F_{2^m}, Horner scheme takes 41 mul, 41 add per element
- Chien search: Compute $c_i g^i, c_i g^{2i}, c_i g^{3i}$ etc.
- Same operation count but different structure
- Berlekamp's trace algorithm: not constant time

Multipoint evaluation via FFT

- Evaluate a polynomial $f = c_0 + c_1 x + \cdots + c_{n-1} x^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(x^2) + x f_1(x^2)$

Multipoint evaluation via FFT

- Evaluate a polynomial $f = c_0 + c_1 x + \cdots + c_{n-1} x^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(x^2) + x f_1(x^2)$
  - Huge overlap between evaluating

$$f(\alpha) = f_0(\alpha^2) + \alpha f_1(\alpha^2) \text{ and}$$
$$f(-\alpha) = f_0(\alpha^2) - \alpha f_1(\alpha^2)$$

Multipoint evaluation via FFT

- Evaluate a polynomial $f = c_0 + c_1 x + \cdots + c_{n-1} x^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(x^2) + x f_1(x^2)$
  - Huge overlap between evaluating

$$f(\alpha) = f_0(\alpha^2) + \alpha f_1(\alpha^2) \text{ and}$$
$$f(-\alpha) = f_0(\alpha^2) - \alpha f_1(\alpha^2)$$

- Problem: We have a binary field, and $\alpha = -\alpha$

Multipoint evaluation via FFT

- Evaluate a polynomial $f = c_0 + c_1 x + \cdots + c_{n-1} x^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(x^2) + x f_1(x^2)$
  - Huge overlap between evaluating

$$f(\alpha) = f_0(\alpha^2) + \alpha f_1(\alpha^2) \text{ and}$$
$$f(-\alpha) = f_0(\alpha^2) - \alpha f_1(\alpha^2)$$

- Problem: We have a binary field, and $\alpha = -\alpha$
- Wang, Zhu 1988, and independently Cantor 1989: additive FFT in characteristic 2 (quite slow)

Multipoint evaluation via FFT

- Evaluate a polynomial $f = c_0 + c_1 x + \cdots + c_{n-1}x^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(x^2) + x f_1(x^2)$
  - Huge overlap between evaluating

$$f(\alpha) = f_0(\alpha^2) + \alpha f_1(\alpha^2) \text{ and}$$
$$f(-\alpha) = f_0(\alpha^2) - \alpha f_1(\alpha^2)$$

- Problem: We have a binary field, and $\alpha = -\alpha$
- Wang, Zhu 1988, and independently Cantor 1989: additive FFT in characteristic 2 (quite slow)
- von zur Gathen 1996: some improvements (still slow)

Multipoint evaluation via FFT

- Evaluate a polynomial $f = c_0 + c_1 x + \cdots + c_{n-1}x^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(x^2) + x f_1(x^2)$
  - Huge overlap between evaluating

$$f(\alpha) = f_0(\alpha^2) + \alpha f_1(\alpha^2) \text{ and}$$
$$f(-\alpha) = f_0(\alpha^2) - \alpha f_1(\alpha^2)$$

- Problem: We have a binary field, and $\alpha = -\alpha$
- Wang, Zhu 1988, and independently Cantor 1989: additive FFT in characteristic 2 (quite slow)
- von zur Gathen 1996: some improvements (still slow)
- Gao, Mateer 2010: Much faster additive FFT

Gao-Mateer additive FFT

- Evaluate a polynomial $f = c_0 + c_1 x + \cdots + c_{n-1}x^{n-1}$ on a size-n $\mathbb{F}_2$-linear space S
- Idea: Write polynomial f as $f_0(x^2 + x) + xf_1(x^2 + x)$

Gao-Mateer additive FFT

- Evaluate a polynomial $f = c_0 + c_1 x + \cdots + c_{n-1}x^{n-1}$ on a size-n $\mathbb{F}_2$-linear space S
- Idea: Write polynomial f as $f_0(x^2 + x) + x f_1(x^2 + x)$
- Big overlap between evaluating

$$f(\alpha) = f_0(\alpha^2 + \alpha) + \alpha f_1(\alpha^2 + \alpha) \text{ and}$$
$$f(\alpha + 1) = f_0(\alpha^2 + \alpha) + (\alpha + 1) f_1(\alpha^2 + \alpha)$$

Gao-Mateer additive FFT

- Evaluate a polynomial $f = c_0 + c_1x + \cdots + c_{n-1}x^{n-1}$ on a size-n $\mathbb{F}_2$-linear space S
- Idea: Write polynomial f as $f_0(x^2 + x) + xf_1(x^2 + x)$
- Big overlap between evaluating

$$f(\alpha) = f_0(\alpha^2 + \alpha) + \alpha f_1(\alpha^2 + \alpha) \text{ and}$$
$$f(\alpha + 1) = f_0(\alpha^2 + \alpha) + (\alpha + 1)f_1(\alpha^2 + \alpha)$$

- Evaluate f_0 and f_1 at $\alpha^2 + \alpha$, obtain $f(\alpha)$ and $f(\alpha + 1)$ with only 1 multiplication and 2 additions

Gao-Mateer additive FFT

- Evaluate a polynomial $f = c_0 + c_1 x + \cdots + c_{n-1}x^{n-1}$ on a size-n $\mathbb{F}_2$-linear space S
- Idea: Write polynomial f as $f_0(x^2 + x) + x f_1(x^2 + x)$
- Big overlap between evaluating

$$f(\alpha) = f_0(\alpha^2 + \alpha) + \alpha f_1(\alpha^2 + \alpha) \text{ and}$$
$$f(\alpha + 1) = f_0(\alpha^2 + \alpha) + (\alpha + 1) f_1(\alpha^2 + \alpha)$$

- Evaluate f_0 and f_1 at $\alpha^2 + \alpha$, obtain $f(\alpha)$ and $f(\alpha + 1)$ with only 1 multiplication and 2 additions
- Again: apply the idea recursively

Gao-Mateer additive FFT

- Evaluate a polynomial $f = c_0 + c_1 x + \cdots + c_{n-1} x^{n-1}$ on a size-n $\mathbb{F}_2$-linear space S
- Idea: Write polynomial f as $f_0(x^2 + x) + x f_1(x^2 + x)$
- Big overlap between evaluating

$$f(\alpha) = f_0(\alpha^2 + \alpha) + \alpha f_1(\alpha^2 + \alpha) \text{ and}$$
$$f(\alpha + 1) = f_0(\alpha^2 + \alpha) + (\alpha + 1) f_1(\alpha^2 + \alpha)$$

- Evaluate f_0 and f_1 at $\alpha^2 + \alpha$, obtain $f(\alpha)$ and $f(\alpha + 1)$ with only 1 multiplication and 2 additions
- Again: apply the idea recursively
- Our paper: generalize the idea to small-degree f
  - Recursion can stop much earlier
  - Various speedups at the end of the recursion

Syndrome computation, the classical way

- Receive n-bit input word, scale bits by Goppa constants
- Apply linear map

$$M = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \alpha_1 & \alpha_2 & \cdots & \alpha_n \\ \alpha_1^2 & \alpha_2^2 & \cdots & \alpha_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{2t-1} & \alpha_2^{2t-1} & \cdots & \alpha_n^{2t-1} \end{pmatrix}$$

Syndrome computation, the classical way

- Receive n-bit input word, scale bits by Goppa constants
- Apply linear map

$$M = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \alpha_1 & \alpha_2 & \cdots & \alpha_n \\ \alpha_1^2 & \alpha_2^2 & \cdots & \alpha_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{2t-1} & \alpha_2^{2t-1} & \cdots & \alpha_n^{2t-1} \end{pmatrix}$$

- Can precompute matrix mapping bits to syndrome
- Yields pretty large secret key, larger than L1 cache

Another look at syndrome computation

Look at the syndrome-computation map again:

$$M = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \alpha_1 & \alpha_2 & \cdots & \alpha_n \\ \alpha_1^2 & \alpha_2^2 & \cdots & \alpha_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{2t-1} & \alpha_2^{2t-1} & \cdots & \alpha_n^{2t-1} \end{pmatrix}$$

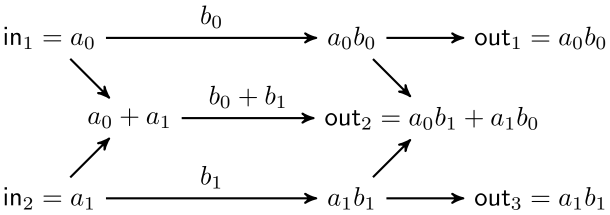
Consider the linear map $M^\intercal$:

$$\begin{pmatrix} 1 & \alpha_1 & \cdots & \alpha_1^{2t-1} \\ 1 & \alpha_2 & \cdots & \alpha_2^{2t-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_n & \cdots & \alpha_n^{2t-1} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_t \end{pmatrix} = \begin{pmatrix} v_1 + v_2\alpha_1 + \cdots + v_t\alpha_1^{2t-1} \\ v_1 + v_2\alpha_2 + \cdots + v_t\alpha_2^{2t-1} \\ \vdots \\ v_1 + v_2\alpha_n + \cdots + v_t\alpha_n^{2t-1} \end{pmatrix} = \begin{pmatrix} f(\alpha_1) \\ f(\alpha_2) \\ \vdots \\ f(\alpha_n) \end{pmatrix}$$

- This transposed linear map is actually doing multipoint evaluation
- Syndrome computation is a transposed multipoint evaluation

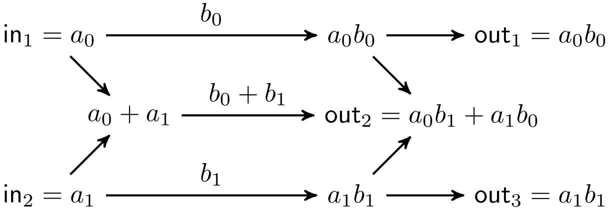
Transposing linear algorithms

- A linear map: $a_0, a_1 \to a_0 b_0, a_0 b_1 + a_1 b_0, a_1 b_1$

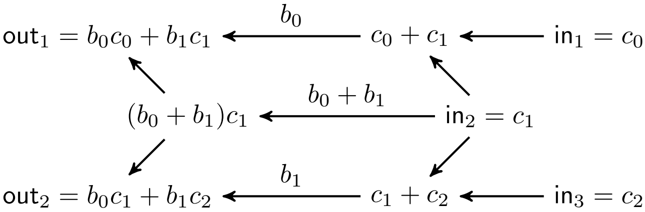


Transposing linear algorithms

- A linear map: $a_0, a_1 \to a_0b_0, a_0b_1 + a_1b_0, a_1b_1$



- Reversing the edges: $c_0, c_1, c_2 \to b_0c_0 + b_1c_1, b_0c_1 + b_1c_2$



What did we just do?

- The original linear map:

$$\begin{pmatrix} a_0b_0 \\ a_0b_1 + a_1b_0 \\ a_1b_1 \end{pmatrix} = \begin{pmatrix} b_0 & 0 \\ b_1 & b_0 \\ 0 & b_1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}$$

- The transposed map:

$$\begin{pmatrix} b_0c_0 + b_1c_1 \\ b_0c_1 + b_1c_2 \end{pmatrix} = \begin{pmatrix} b_0 & b_1 & 0 \\ 0 & b_0 & b_1 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix}$$

What did we just do?

- The original linear map:

$$\begin{pmatrix} a_0b_0 \\ a_0b_1 + a_1b_0 \\ a_1b_1 \end{pmatrix} = \begin{pmatrix} b_0 & 0 \\ b_1 & b_0 \\ 0 & b_1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}$$

- The transposed map:

$$\begin{pmatrix} b_0c_0 + b_1c_1 \\ b_0c_1 + b_1c_2 \end{pmatrix} = \begin{pmatrix} b_0 & b_1 & 0 \\ 0 & b_0 & b_1 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix}$$

- Reversing the edges automatically gives an algorithm for the transposed map
- This is called the *transposition principle*

What did we just do?

- The original linear map:

$$\begin{pmatrix} a_0b_0 \\ a_0b_1 + a_1b_0 \\ a_1b_1 \end{pmatrix} = \begin{pmatrix} b_0 & 0 \\ b_1 & b_0 \\ 0 & b_1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}$$

- The transposed map:

$$\begin{pmatrix} b_0c_0 + b_1c_1 \\ b_0c_1 + b_1c_2 \end{pmatrix} = \begin{pmatrix} b_0 & b_1 & 0 \\ 0 & b_0 & b_1 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix}$$

- Reversing the edges automatically gives an algorithm for the transposed map
- This is called the *transposition principle*
- Preserves number of multiplications
- References: Fiduccia 1972, Bordewijk 1956, Lupanov 1956

Transposing the additive FFT

The naive approach

- Idea: Compute syndrome by transposing the additive FFT
- Start with additive FFT program (sequence of additions and constant multiplications)
- Convert to directed acyclic graph (rename variables to remove cycles)
- Reverse edges, convert to C program
- Compile with gcc

Transposing the additive FFT

The naive approach

- Idea: Compute syndrome by transposing the additive FFT
- Start with additive FFT program (sequence of additions and constant multiplications)
- Convert to directed acyclic graph (rename variables to remove cycles)
- Reverse edges, convert to C program
- Compile with gcc
- Problems:
  - Huge program (all loops and function calls removed)

Transposing the additive FFT

The naive approach

- Idea: Compute syndrome by transposing the additive FFT
- Start with additive FFT program (sequence of additions and constant multiplications)
- Convert to directed acyclic graph (rename variables to remove cycles)
- Reverse edges, convert to C program
- Compile with gcc
- Problems:
  - Huge program (all loops and function calls removed)
  - At $m = 13$ or $m = 14$ gcc runs out of memory

Transposing the additive FFT

The naive approach

- Idea: Compute syndrome by transposing the additive FFT
- Start with additive FFT program (sequence of additions and constant multiplications)
- Convert to directed acyclic graph (rename variables to remove cycles)
- Reverse edges, convert to C program
- Compile with gcc
- Problems:
  - Huge program (all loops and function calls removed)
  - At $m = 13$ or $m = 14$ gcc runs out of memory
  - Can use better register allocators, but the program is still huge

Transposing the additive FFT

A better approach

- Analyze structure of additive FFT A: B, A_1, A_2, C
- A_1, A_2 are recursive calls

Transposing the additive FFT

A better approach

- Analyze structure of additive FFT A: B, A_1, A_2, C
- A_1, A_2 are recursive calls
- Transposition has structure C^T, A_2^T, A_1^T, B^T
- Use recursive calls to reduce code size

Secret permutations

- FFT evaluates f at elements in *standard order*
- We need output in a secret order
- Same problem for input of transposed FFT
- Similar problem during key generation (secret random permutation)

Secret permutations

- FFT evaluates f at elements in *standard order*
- We need output in a secret order
- Same problem for input of transposed FFT
- Similar problem during key generation (secret random permutation)
- Typical solution for permutation π: load from position i, store at position $\pi(i)$

Secret permutations

- FFT evaluates f at elements in *standard order*
- We need output in a secret order
- Same problem for input of transposed FFT
- Similar problem during key generation (secret random permutation)
- Typical solution for permutation π: load from position i, store at position $\pi(i)$
- This leaks through timing information
- We need to apply a secret permutation in constant time

Secret permutations

- FFT evaluates f at elements in *standard order*
- We need output in a secret order
- Same problem for input of transposed FFT
- Similar problem during key generation (secret random permutation)
- Typical solution for permutation π: load from position i, store at position $\pi(i)$
- This leaks through timing information
- We need to apply a secret permutation in constant time
- Solution: sorting networks

Sorting networks

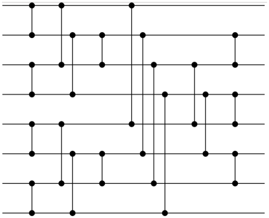
A *sorting network* sorts an array S of elements by using a sequence of *comparators*.

- A comparator can be expressed by a pair of indices (i, j).
- A comparator swaps $S[i]$ and $S[j]$ if $S[i] > S[j]$.

Sorting networks

A *sorting network* sorts an array S of elements by using a sequence of *comparators*.

- A comparator can be expressed by a pair of indices (i, j).
- A comparator swaps $S[i]$ and $S[j]$ if $S[i] > S[j]$.
- Efficient sorting network: Batcher sort (Batcher, 1968)



Batcher sorting network for sorting 8 elements
http://en.wikipedia.org/wiki/Batcher%27s_sort

Permuting by sorting

Example

Computing b_3, b_2, b_1 from b_1, b_2, b_3 can be done by sorting the key-value pairs $(3, b_1), (2, b_2), (1, b_3)$ the output is $(1, b_3), (2, b_2), (3, b_1)$

Permuting by sorting

Example

Computing b_3, b_2, b_1 from b_1, b_2, b_3 can be done by sorting the key-value pairs $(3, b_1), (2, b_2), (1, b_3)$ the output is $(1, b_3), (2, b_2), (3, b_1)$

- All the output bits of $>$ comparisons only depend on the secret permutation
- Those bits can be precomputed during key generation

Permuting by sorting

Example

Computing b_3, b_2, b_1 from b_1, b_2, b_3 can be done by sorting the key-value pairs $(3, b_1), (2, b_2), (1, b_3)$ the output is $(1, b_3), (2, b_2), (3, b_1)$

- All the output bits of $>$ comparisons only depend on the secret permutation
- Those bits can be precomputed during key generation
- Do conditional swap of $b[i]$ and $b[j]$ with condition bit c as

$$y \leftarrow b[i] \oplus b[j]; \quad y \leftarrow cy; \quad b[i] \leftarrow b[i] \oplus y; \quad b[j] \leftarrow b[j] \oplus y;$$

Permuting by sorting

Example

Computing b_3, b_2, b_1 from b_1, b_2, b_3 can be done by sorting the key-value pairs $(3, b_1), (2, b_2), (1, b_3)$ the output is $(1, b_3), (2, b_2), (3, b_1)$

- All the output bits of $>$ comparisons only depend on the secret permutation
- Those bits can be precomputed during key generation
- Do conditional swap of $b[i]$ and $b[j]$ with condition bit c as

$$y \leftarrow b[i] \oplus b[j]; \quad y \leftarrow cy; \quad b[i] \leftarrow b[i] \oplus y; \quad b[j] \leftarrow b[j] \oplus y;$$

- Possibly better than Batcher sort: Beneš permutation network (work in progress)

Results

Throughput cycles on Ivy Bridge

- Input secret permutation: 8622
- Syndrome computation: 20846
- Berlekamp-Massey: 7714
- Root finding: 14794
- Output secret permutation: 8520
- Total: **60493**

Results

Throughput cycles on Ivy Bridge

- Input secret permutation: 8622
- Syndrome computation: 20846
- Berlekamp-Massey: 7714
- Root finding: 14794
- Output secret permutation: 8520
- Total: **60493**
- These are amortized cycle counts across 256 parallel computations

Results

Throughput cycles on Ivy Bridge

- Input secret permutation: 8622
- Syndrome computation: 20846
- Berlekamp-Massey: 7714
- Root finding: 14794
- Output secret permutation: 8520
- Total: **60493**
- These are amortized cycle counts across 256 parallel computations
- All computations with full timing-attack protection!

Comparison

Public-key decryption speeds from eBATS

- `ntruees787ep1`: 700512 cycles
- `mceliece`: 1219344 cycles
- `ronald1024`: 1340040 cycles
- `ronald3072`: 16052564 cycles

Comparison

Public-key decryption speeds from eBATS

- `ntruees787ep1`: 700512 cycles
- `mceliece`: 1219344 cycles
- `ronald1024`: 1340040 cycles
- `ronald3072`: 16052564 cycles

Diffie-Hellman shared-secret speeds from eBATS

- `gls254`: 77468 cycles
- `kumfp127g` 116944 cycles
- `curve25519`: 182632 cycles

References

- Daniel J. Bernstein, Tung Chou, and Peter Schwabe. *McBits: fast constant-time code-based cryptography.*, CHES 2013. http://cryptojedi.org/papers/#mcbits

- Software will be online (public domain), for example, at http://cryptojedi.org/crypto/#mcbits