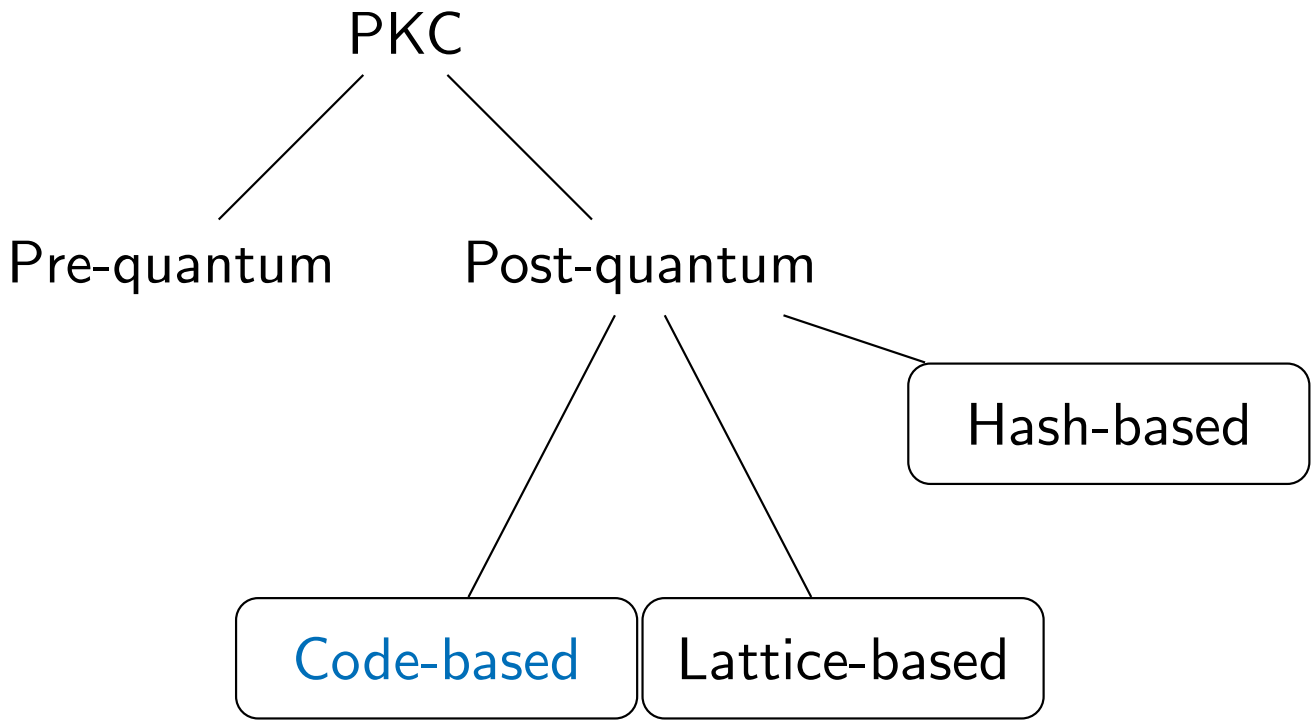
McBits:
fast constant-time code-based cryptography

Tung Chou

Joint work with Daniel J. Bernstein and Peter Schwabe

Code-based cryptography



Code-based versus Lattice-based encryption

	Code	Lattice
Confidence	McEliece ($\approx$ 40 years)	NTRU ($\approx$ 20 years)
Key size	$\approx$ 1 MB	$\approx$ 1 KB
Speed	"fast"	"fast"

Code-based versus Lattice-based encryption

	Code	Lattice
Confidence	McEliece ($\approx$ 40 years)	NTRU ($\approx$ 20 years)
Key size	$\approx$ 1 MB	$\approx$ 1 KB
Speed	"fast"	"fast"

- How fast can the decryption be?
- How about timing attacks?

What we achieved

m	n	t	sec	cycles
12	4096	41	129	60 493
13	6960	119	263	306 102

↑
constant-time

What we achieved

m	n	t	sec	cycles
12	4096	41	129	60 493
13	6960	119	263	306 102

↑
constant-time

- An order of magnitude faster than the previous work.
- non-conservative ECC: $\approx 60\,000$ cycles (2^{128} sec.)
- conservative ECC: $\approx 160\,000$ cycles (2^{128} sec.)
- NTRU Prime: $\approx 100\,000$ cycles ($\geq 2^{128}$ *quantum* sec.)

Binary linear codes

Code C

- a dimension-k linear subspace in $\mathbf{F}_2^n$ of **code words**.
- can be defined by a $k \times n$ generating matrix G:

$$C = \{mG \mid m \in \mathbf{F}_2^k\}$$

- can be defined by a $(n-k) \times n$ **parity-check matrix** H:

$$C = \{c \mid Hc = \vec{0}\}$$

Binary linear codes

Code C

- a dimension-k linear subspace in $\mathbf{F}_2^n$ of **code words**.
- can be defined by a $k \times n$ generating matrix G:

$$C = \{mG \mid m \in \mathbf{F}_2^k\}$$

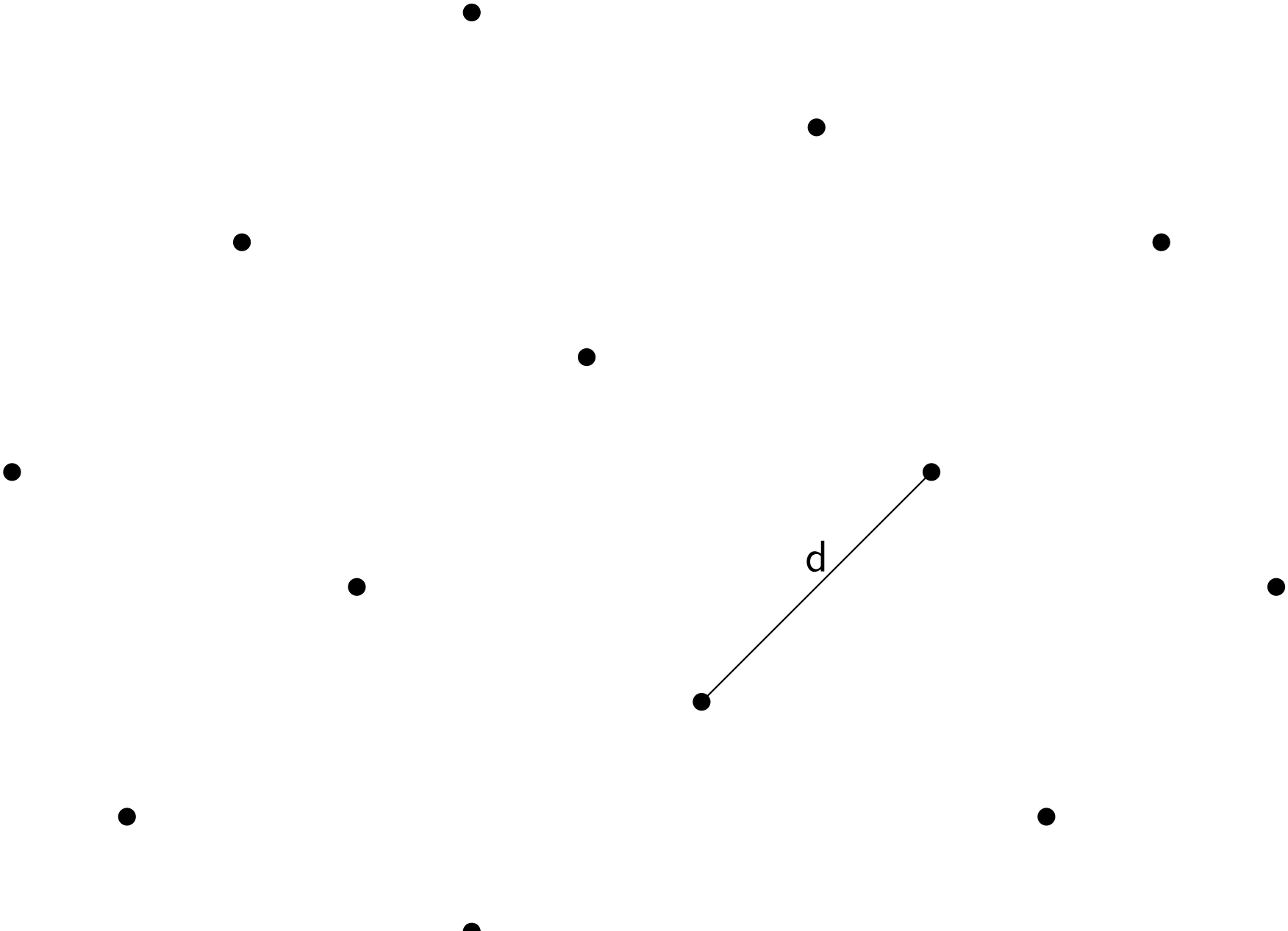
- can be defined by a $(n-k) \times n$ **parity-check matrix** H:

$$C = \{c \mid Hc = \vec{0}\}$$

Decoding

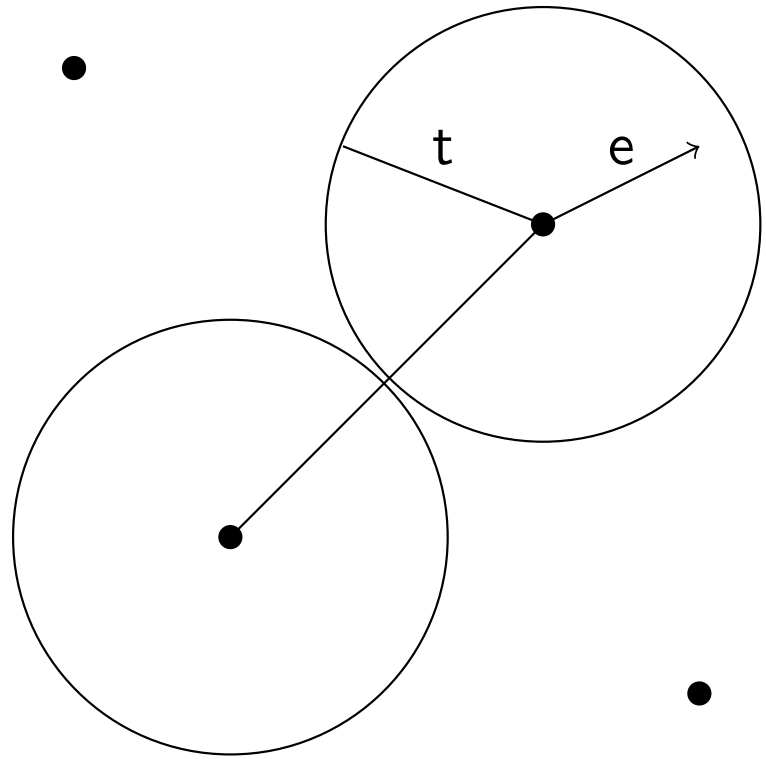
- compute **error vector** e (or c) given $c + e$, where $wt(e) \leq$ **t**
- compute e given the **syndrome** $He = H(c + e)$

Minimum distance



Minimum distance

$$t < d/2$$
$$\Downarrow$$
$$t \leq \lfloor (d-1)/2 \rfloor$$



Code-based encryption

- McEliece vs. Niederreiter
  - $(\mathcal{G}, \mathcal{H})$: "scrambled" version of G, H.

	encryption	decryption
McEliece	$m \to m\mathcal{G} + e$	$m\mathcal{G} + e \to mG + e \to e$
Niederreiter	$e \to \mathcal{H}e$	$\mathcal{H}e \to He \to e$

  - Decryption is essentially decoding.

Code-based encryption

- McEliece vs. Niederreiter
  - $(\mathcal{G}, \mathcal{H})$: "scrambled" version of G, H.

	encryption	decryption
McEliece	$m \to m\mathcal{G} + e$	$m\mathcal{G} + e \to mG + e \to e$
Niederreiter	$e \to \mathcal{H}e$	$\mathcal{H}e \to He \to e$

  - Decryption is essentially decoding.

- General shape

McEliece/Niederreiter + **code**

Binary Goppa codes

A binary Goppa code is defined by

- A **support** $L = (a_1, \ldots, a_n)$ of n distinct elements in $\mathbf{F}_{2^m}$.

  (Let's assume $n = 2^m$)

- A *square-free* polynomial $g(x) \in \mathbf{F}_{2^m}[x]$ of degree t s.t. $g(a_i) \neq 0, \forall i.$ $g(x)$ is called the **Goppa polynomial**.

Binary Goppa codes

A binary Goppa code is defined by

- A **support** $L = (a_1, \ldots, a_n)$ of n distinct elements in $\mathbf{F}_{2^m}$.

  (Let's assume $n = 2^m$)
- A *square-free* polynomial $g(x) \in \mathbf{F}_{2^m}[x]$ of degree t s.t. $g(a_i) \neq 0$, $\forall i$. $g(x)$ is called the **Goppa polynomial**.

The code $\Gamma(L, g)$ is the set of words $c = (c_1, \ldots, c_n) \in \mathbf{F}_2^n$, s.t.

$$\frac{c_1}{x - a_1} + \frac{c_2}{x - a_2} + \cdots + \frac{c_n}{x - a_n} \equiv 0 \pmod{g(x)}$$

- Can correct t errors
- In McEliece/Niederreiter L, g form the secret key.

Error locator

- The **error locator** for $\mathbf{e}$ is the polynomial

$$\sigma(x) = \prod_{\mathbf{e}_i \neq 0} (x - a_i) \in \mathbf{F}_q[x]$$

  With the roots, $\mathbf{e}$ can be reconstructed easily.

- For McEliece/Niederreiter, the error vector $\mathbf{e}$ is known to have Hamming weight t.

The Berlekamp's decoder

ciphertext $\vec{s} \Rightarrow \vec{c} + \vec{e}$

$\downarrow$ syndrome computation

$S(x)$

$\downarrow$ key-equation solving

$\sigma(x)$

$\downarrow$ root finding

$\vec{e}$

Timing attacks

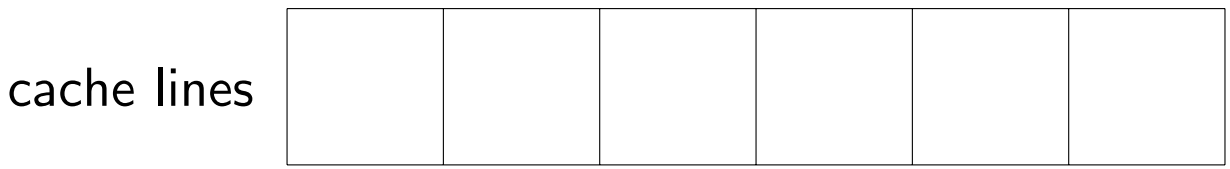
In general, there are two things you should avoid.

- Secret conditions (recall the case of scalar mult.).
- Secret memory indices (**cache timing attacks**).

Timing attacks

In general, there are two things you should avoid.

- Secret conditions (recall the case of scalar mult.).
- Secret memory indices (**cache timing attacks**).



Timing attacks

In general, there are two things you should avoid.

- Secret conditions (recall the case of scalar mult.).
- Secret memory indices (**cache timing attacks**).

Timing attacks

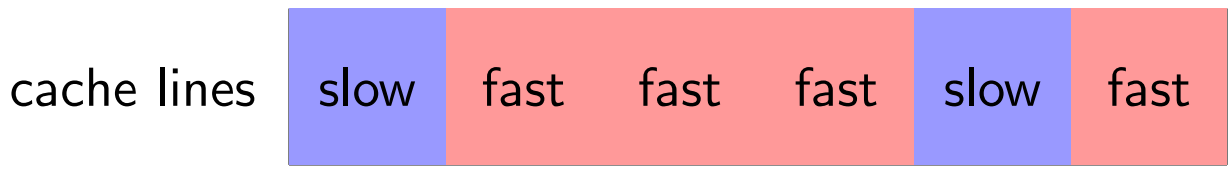
In general, there are two things you should avoid.

- Secret conditions (recall the case of scalar mult.).
- Secret memory indices (**cache timing attacks**).

Timing attacks

In general, there are two things you should avoid.

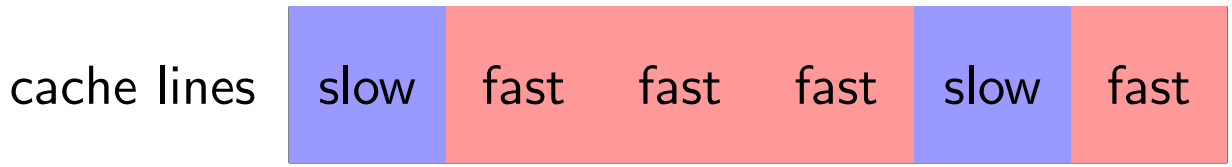
- Secret conditions (recall the case of scalar mult.).
- Secret memory indices (**cache timing attacks**).



Timing attacks

In general, there are two things you should avoid.

- Secret conditions (recall the case of scalar mult.).
- Secret memory indices (**cache timing attacks**).



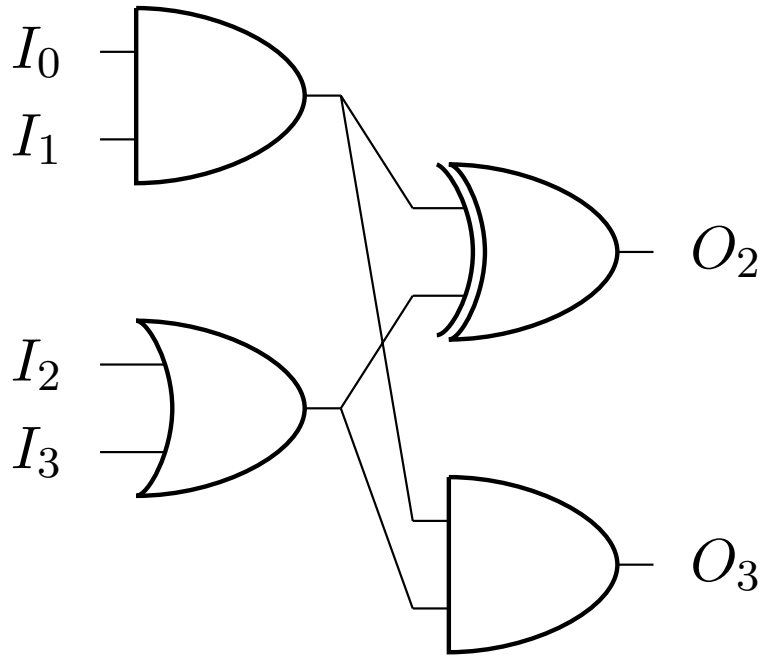
- Our strategy: representing the decoding process as a fix sequence of instructions.

Bitslicing

Simulating n copies of a combinatorial circuit using bitwise logical instructions.

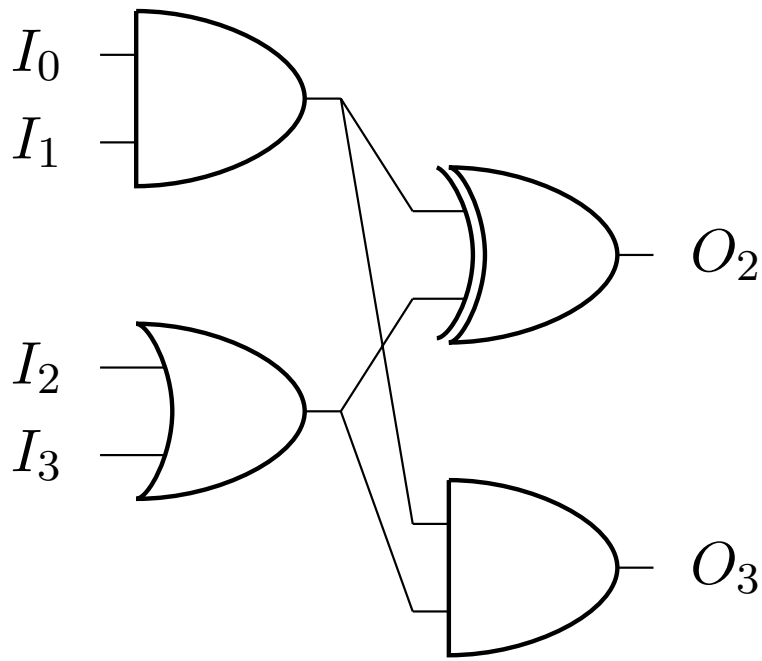
Bitslicing

Simulating n copies of a combinatorial circuit using bitwise logical instructions.



Bitslicing

Simulating n copies of a combinatorial circuit using bitwise logical instructions.



- We put the data for the ith decryption op. in the ith bit.
- Extra assumption required (or not?).

Root finding

Given $\sigma(x) \in \mathbf{F}_{2^m}[x]$ of degree t, find all the roots.

- Can do **multipoint evaluation**.
- Horner's rule: $O(nt)$ mults. Can we do better?

Root finding

Given $\sigma(x) \in \mathbf{F}_{2^m}[x]$ of degree t, find all the roots.

- Can do **multipoint evaluation**.
- Horner's rule: $O(nt)$ mults. Can we do better?

Gao–Mateer additive FFT (2010)

- Uses additive (instead of multiplicative) properties.
- Generalized complexity: $O(n \lg t)$ mults.
- Natural for characteristic-2 fields.

The Gao–Mateer Additive FFT

- Given a input polynomial $f(x) \in \mathbf{F}_{2^m}[x]$.
- An $\mathbf{F}_2$-linear basis $\beta_1, \beta_2, \ldots, \beta_k$ in $\mathbf{F}_{2^m}$.
- The FFT evaluates $\sigma(x)$ at all subset sums of the basis. That is, the output is the sequence:

$$f(0), f(\beta_1), f(\beta_2), f(\beta_1 + \beta_2), f(\beta_3), \ldots$$

- For decoding we have $k = m$.

The Gao–Mateer Additive FFT (cont.)

The Gao–Mateer Additive FFT (cont.)

- Assume that $\beta_k = 1$. The whole space is then

$$\alpha_1, \alpha_2, \ldots, \alpha_{2^{k-1}}, \quad \alpha_1 + 1, \alpha_2 + 1, \ldots, \alpha_{2^{k-1}} + 1.$$

The Gao–Mateer Additive FFT (cont.)

- Assume that $\beta_k = 1$. The whole space is then

$$\alpha_1, \alpha_2, \ldots, \alpha_{2^{k-1}}, \quad \alpha_1 + 1, \alpha_2 + 1, \ldots, \alpha_{2^{k-1}} + 1.$$

- Write f in the form $f^{(0)}(x^2 + x) + x \cdot f^{(1)}(x^2 + x)$.

  (multiplicative FFT: $f^{(0)}(x^2) + x \cdot f^{(1)}(x^2)$)

The Gao–Mateer Additive FFT (cont.)

- Assume that $\beta_k = 1$. The whole space is then

$$\alpha_1, \alpha_2, \ldots, \alpha_{2^{k-1}}, \quad \alpha_1 + 1, \alpha_2 + 1, \ldots, \alpha_{2^{k-1}} + 1.$$

- Write f in the form $f^{(0)}(x^2 + x) + x \cdot f^{(1)}(x^2 + x)$.

  (multiplicative FFT: $f^{(0)}(x^2) + x \cdot f^{(1)}(x^2)$)

- We have $(\alpha + 1)^2 + (\alpha + 1) = \alpha^2 + \alpha$. Therefore,

$$f(\alpha) = f^{(0)}(\alpha^2 + \alpha) + \alpha \cdot f^{(1)}(\alpha^2 + \alpha)$$

$$f(\alpha + 1) = f^{(0)}(\alpha^2 + \alpha) + (\alpha + 1) \cdot f^{(1)}(\alpha^2 + \alpha)$$

The Gao–Mateer Additive FFT (cont.)

- Assume that $\beta_k = 1$. The whole space is then

$$\alpha_1, \alpha_2, \ldots, \alpha_{2^{k-1}}, \quad \alpha_1 + 1, \alpha_2 + 1, \ldots, \alpha_{2^{k-1}} + 1.$$

- Write f in the form $f^{(0)}(x^2 + x) + x \cdot f^{(1)}(x^2 + x)$.

  (multiplicative FFT: $f^{(0)}(x^2) + x \cdot f^{(1)}(x^2)$)

- We have $(\alpha + 1)^2 + (\alpha + 1) = \alpha^2 + \alpha$. Therefore,

$$f(\alpha) = f^{(0)}(\alpha^2 + \alpha) + \alpha \cdot f^{(1)}(\alpha^2 + \alpha)$$

$$f(\alpha + 1) = f^{(0)}(\alpha^2 + \alpha) + (\alpha + 1) \cdot f^{(1)}(\alpha^2 + \alpha)$$

- $f^{(0)}(x)$ and $f^{(1)}(x)$ are evaluated using the new basis

$$\beta_1^2 + \beta_1, \ldots, \beta_{m-1}^2 + \beta_{m-1}$$

  recursively to obtain all $f^{(0)}(\alpha^2 + \alpha)$ and $f^{(1)}(\alpha^2 + \alpha)$.

The Gao–Mateer Additive FFT (cont.)

$m = 11$	t	27	32	35	40	53	63	69	79		
	adds	5.41	5.60	5.75	5.99	6.47	6.69	6.84	7.11		
	mults	1.85	2.12	2.13	2.16	2.40	2.73	2.77	2.82		
$m = 12$	t	21	41	45	56	67	81	89	111	133	
	adds	5.07	6.01	6.20	6.46	6.69	7.04	7.25	7.59	7.86	
	mults	1.55	2.09	2.10	2.40	2.64	2.68	2.70	2.99	3.28	
$m = 13$	t	18	29	35	57	95	115	119	189	229	237
	adds	4.78	5.45	5.70	6.44	7.33	7.52	7.56	8.45	8.71	8.77
	mults	1.52	1.91	2.04	2.38	2.62	2.94	3.01	3.24	3.57	3.64

Syndrome computation

Syndrome computation performs the linear map

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

Syndrome computation

Syndrome computation performs the linear map

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

For root finding, we perform the linear map

$$\begin{pmatrix} 1 & \alpha_1 & \cdots & \alpha_1^t \\ 1 & \alpha_2 & \cdots & \alpha_2^t \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_n & \cdots & \alpha_n^t \end{pmatrix}.$$

Syndrome computation

Syndrome computation performs the linear map

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

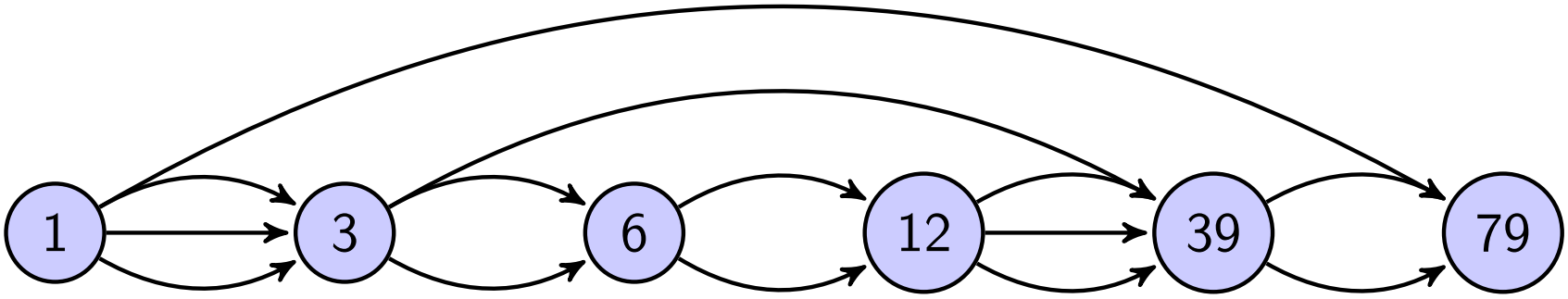
For root finding, we perform the linear map

$$\begin{pmatrix} 1 & \alpha_1 & \cdots & \alpha_1^t \\ 1 & \alpha_2 & \cdots & \alpha_2^t \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_n & \cdots & \alpha_n^t \end{pmatrix}.$$

Syndrome computation is a transposed multipoint evaluation.

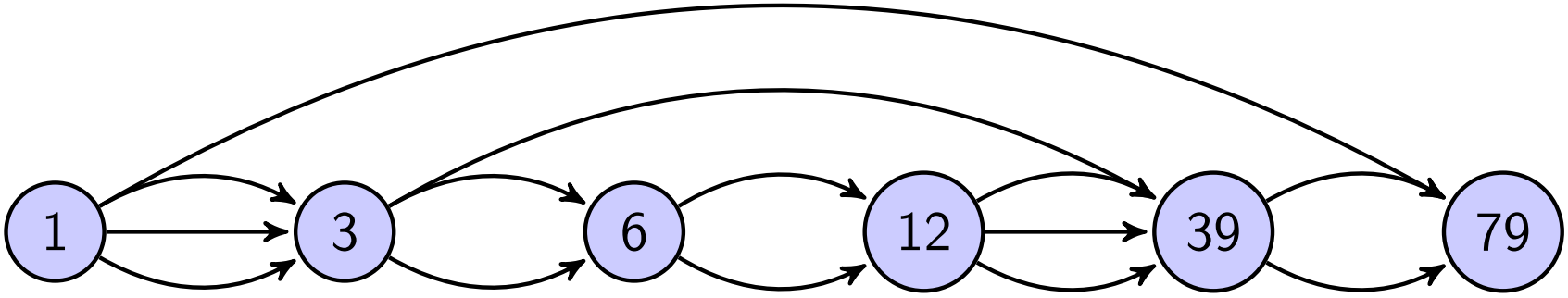
Transposing linear algorithms

Example: an addition chain for 79

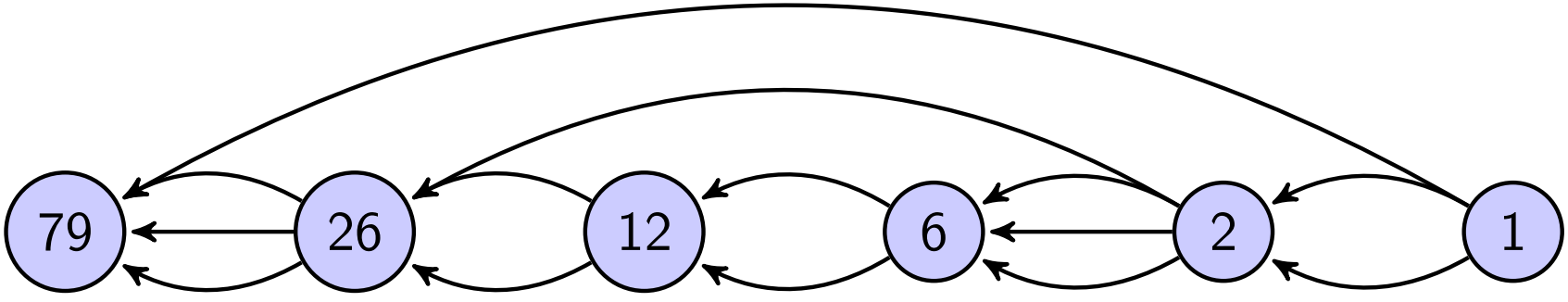


Transposing linear algorithms

Example: an addition chain for 79



By reversing the edges, we get another addition chain for 79:



Transposition principle

Transposition principle

"Given a linear algorithm represented as a directed graph, then the algorithm obtained by reversing the edges computes the transposed linear map."

Transposition principle

"Given a linear algorithm represented as a directed graph, then the algorithm obtained by reversing the edges computes the transposed linear map."

(We use a transposed additive FFT for syndrome computation.)

Secret permutations

FFT output:	$\sigma(\alpha_1),$	$\sigma(\alpha_2),$	$\sigma(\alpha_3),$	$\ldots,$	$\sigma(\alpha_n)$
Support:	$a_1,$	$a_2,$	$a_3,$	$\ldots,$	a_n

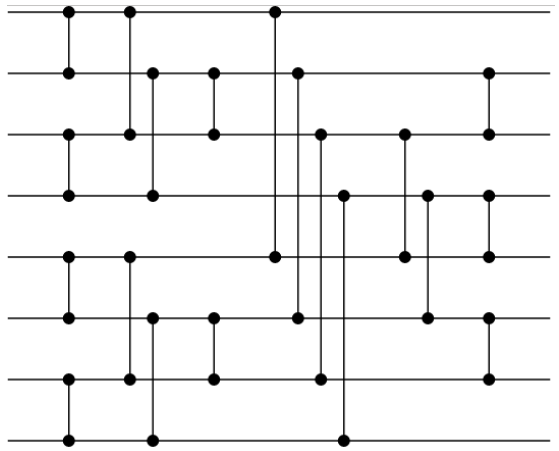
Secret permutations

FFT output:	$\sigma(\alpha_1),$	$\sigma(\alpha_2),$	$\sigma(\alpha_3),$	$\ldots,$	$\sigma(\alpha_n)$
Support:	$a_1,$	$a_2,$	$a_3,$	$\ldots,$	a_n

- Require a secret permutation for the output of the FFT.
  (The same issue for the input of the transposed FFT.)
- Can't just move data around by loads and stores!

Sorting network

- A sorting network for 8 elements



- The vertical line segments are **comparators**.
- Each comparator is a pair of indices (i, j).
- Swaps $S[i]$ and $S[j]$ if $S[i] > S[j]$.

Sorting network (Cont.)

- The permutation can be carried out by sorting.
- Each condition bit c depends only on the support:
  we precompute the condition bits.
- Each comparator can be implemented with 4 operations:

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

https://tungchou.github.io/mcbits/