McBits: fast constant-time code-based cryptography

Tung Chou

Technische Universiteit Eindhoven, The Netherlands

October 13, 2015

Joint work with Daniel J. Bernstein and Peter Schwabe

Outline

- Summary of Our Work
- Background
- Main Components of Our Software

Summary of Our Work

Motivation

Code-based public-key **encryption** system:

- Confidence: The original McEliece system using Goppa code proposed in 1978 remains hard to break.
- Post-quantum security
- Known to provide fast encryption and decryption.

The state-of-the-art implementation before our work

- Biswas and Sendrier. *McEliece Cryptosystem Implementation: Theory and Practice.* 2008.

Issues:

- Decryption time: Lots of interesting things to do...
- Usability: haven't seen implementations that claim to be secure against timing attacks.

What we achieved

- For 80-bit security, we achieved decryption time of 26 544 cycles, while the previous work requires 288 681 cycles.
- For 128-bit security, we achieved decryption time of 60 493 cycles, while the previous work requires 540 960 cycles.
- We set new speed records for decryption of code-based system. Actually these are also speed records for public-key cryptography in general.
  - followed by 77 468 cycles for an binary-elliptic-curve Diffie–Hellman implementation (128-bit security). CHES 2013.
- Our software is fully protected against timing attacks.

Novelty

Novelty in our work:

- Using an additive FFT for fast root computation.
  - Conventional approach: using Horner-like algorithms.
- Using an transposed additive FFT for fast syndrome computation.
  - Conventional approach: matrix-vector multiplication.
- Using a sorting network to avoid cache-timing attacks.
  - Existing softwares did not deal with this issue.

Background

Binary Linear Codes

A binary linear code C of length n and dimension k is a k-dimensional subspace of $\mathbf{F}_2^n$.

C is usually specified as

- the row space of a generating matrix $G \in \mathbf{F}_2^{k\times n}$

$$C = \{\mathbf{m}G | \mathbf{m} \in \mathbf{F}_2^k\}$$

- the kernel space of a parity-check matrix $H \in \mathbf{F}_2^{(n-k)\times n}$

$$C = \{\mathbf{c} | H\mathbf{c}^\intercal = 0,\ \ \mathbf{c} \in \mathbf{F}_2^n\}$$

Example:

$$G = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

$\mathbf{c} = (111)G = (10011)$ is a codeword.

Decoding problem

Decoding problem: find the closest codeword $\mathbf{c} \in C$ to a given $\mathbf{r} \in \mathbf{F}_2^n$, assuming that there is a unique closest codeword. Let $\mathbf{r} = \mathbf{c} + \mathbf{e}$. Note that finding $\mathbf{e}$ is an equivalent problem.

- $\mathbf{r}$ is called the received word. $\mathbf{e}$ is called the error vector.
- There are lots of code families with fast decoding algorithms, e.g., Reed–Solomon codes, Goppa codes/alternant codes, etc.
- However, the general decoding problem is hard: best known algorithm takes exponential time.

Binary Goppa code

A binary Goppa code is often defined by

- a list $L = (a_1, \ldots, a_n)$ of n distinct elements in $\mathbf{F}_q$, called the support. For convenience we assume $n = q$ in this talk.
- a square-free polynomial $g(x) \in \mathbf{F}_q[x]$ of degree t such that $g(a) \neq 0$ for all $a \in L$. $g(x)$ is called the Goppa polynomial.
- In code-base encryption system these form the secret key.

Then the corresponding binary Goppa code, denoted as $\Gamma_2(L, g)$, is the set of words $c = (c_1, \ldots, c_n) \in \mathbf{F}_2^n$ that satisfy

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

- can correct t errors
- suitable for building secure code-based encryption system.

The Niederreiter cryptosystem

Developed in 1986 by Harald Niederreiter as a variant of the McEliece cryptosystem.

- Public Key: a parity-check matrix $K \in \mathbf{F}_q^{(n-k)\times n}$ for the binary Goppa code
- Encryption: The plaintext $\mathbf{e}$ is an n-bit vector of weight t. The ciphertext $\mathbf{s}$ is an $(n-k)$-bit vector:

$$\mathbf{s}^\mathsf{T} = K\mathbf{e}^\mathsf{T}.$$

- Decryption: Find a n-bit vector $\mathbf{r}$ such that

$$\mathbf{s}^\mathsf{T} = K\mathbf{r}^\mathsf{T}.$$

  $\mathbf{r}$ would be of the form $\mathbf{c}+\mathbf{e}$, where $\mathbf{c}$ is a codeword. Then we use any available decoder to decode $\mathbf{r}$.

- A passive attacker is facing a t-error correcting problem for the public key, which seems to be random.

Decoder

- A syndrome is $H\mathbf{r}$, where H is a parity-check matrix.
- 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 cryptographic use the error vector $\mathbf{e}$ is known to have Hamming weight t.

Typical decoders decode by performing

- Syndrome computation
- Solving key equation
- Root finding (for the error locator)

The decoder we used is the Berlekamp decoder.

Timing attacks

Secret memory indices

- Cryptographic software C and attacker software A runs on a machine.
- A overwrites several caches lines $L = \{L_1, L_2, \ldots, L_k\}$.
- C then overwrites a subset of L. The indices of the data are secret.
- A reads from L_i and gains information from the timing.

Secret branch conditions

- Whether the branch is taken or not causes difference in timing.

Bitslicing

- Simulating logic gates by performing bitwise logic operations on m-bit words ($m = 8$, 16, 32, 64, 128, 256, etc.). In our implementation $m = 128$ or 256.
- Naturally process m instances in parallel. Our software handles m decryptions for m secret keys at the same time.
- It's constant-time.
- Can be much faster than a non-bitsliced implementation, depending on the application.
  - e.g., Eli Biham, *A fast new DES implementation in software*: implementing S-boxes with bitslicing instead of table lookups, gaining $2\times$ speedup.

Main Components of the Implementation

- Root finding
- Syndrome computation
- Secret permutation

Root finding

- Input:

$$f(x) = v_0 + v_1 x + \cdots + v_t x^t \in \mathbf{F}_q[x]$$

(assume $t < q$ without loss of generality)

- Output: a sequence of q bits $\mathbf{w}_{\alpha_i}$ indexed by $\alpha_i \in \mathbf{F}_q$ where $\mathbf{w}_{\alpha_i} = 0$ iff $f(\alpha_i) = 0$. Example:

$$(\mathbf{w}_{\alpha_1}, \mathbf{w}_{\alpha_2}, \ldots, \mathbf{w}_{\alpha_q}) = (1, 0, 1, 1, 1, 0, 1, \ldots)$$

- Can be done by doing multipoint evaluation:
  - Compute all the images $f(\alpha_1), f(\alpha_2), \ldots, f(\alpha_q)$.
  - And then for each α_i, OR together the bits of $f(\alpha_i)$.

- The multipoint evaluation we used: Gao–Mateer additive FFT

The Gao–Mateer Additive FFT

- Shuhong Gao and Todd Mateer. *Additive Fast Fourier Transforms over Finite Fields*. 2010.

- Deal with the problem of evaluating a 2^m-coefficient polynomial $f \in \mathbf{F}_q[x]$ over $\hat{S}$, the sequence of all subset sums of $\{\beta_1, \beta_2, \ldots, \beta_m\} \in \mathbf{F}_q$. That is, the output is 2^m elements in $\mathbf{F}_q$:

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

- A recursive algorithm. Recursion stops when m is small.

- In decoding applications f would be the error locator, and $\{\beta_1, \beta_2, \ldots, \beta_m\}$ can be any basis of $\mathbf{F}_q$ over $\mathbf{F}_2$.

The Gao–Mateer Additive FFT: main idea

- Assume that the sequence $\hat{S}$ can be divided into two partitions S and $S+1$.
- Write f in the form $f_0(x^2 - x) + x \cdot f_1(x^2 - x)$. For comparison, a multiplicative FFT would use $f = f_0(x^2) + x \cdot f_1(x^2)$.
- For all $\alpha \in \mathbf{F}_q$, $(\alpha+1)^2 - (\alpha+1) = \alpha^2 - \alpha$. Therefore,

$$f(\alpha) = f_0(\alpha^2 - \alpha) + \alpha \cdot f_1(\alpha^2 - \alpha)$$

$$f(\alpha+1) = f_0(\alpha^2 - \alpha) + (\alpha+1) \cdot f_1(\alpha^2 - \alpha)$$

  Once we have $f_i(\alpha^2 - \alpha)$, $f(\alpha)$ and $f(\alpha+1)$ can be computed in a few field operations.

- Computing the f_0 and f_1 value for all $\alpha \in S$ recursively gives $f(\beta)$ for all $\beta \in \hat{S}$.

The Gao–Mateer Additive FFT: Improvements

In code-based cryptography $t \ll q$, which can be exploited to make the additive FFT much faster. Some typical choices of (q, t):

q	t				
2^{11}	27	32	35	40	
2^{12}	21	41	45	56	67
2^{13}	18	29	95	115	119

We keep track of the actual degree of polynomials being evaluated. In this way, the depth of recursion can be made smaller.

Take $q = 2^{12}$, $t = 41$ for example. Let L be the length of f. Then $(L, 2^m)$ would go like:

- Original: $(2^{12}, 2^{12}) \to (2^{11}, 2^{11}) \to (2^{10}, 2^{10}) \to \cdots \to (1, 1)$
- Improved: $(42, 2^{12}) \to (21, 2^{11}) \to (11, 2^{10}) \to \cdots \to (1, 2^6)$

The Gao–Mateer Additive FFT: Improvements

Recall that for all $\alpha \in S$

$$f(\alpha) = f_0(\alpha^2 - \alpha) + \alpha \cdot f_1(\alpha^2 - \alpha)$$

In order to compute $f(\alpha)$, we need to compute $\alpha \cdot f_1(\alpha^2 - \alpha)$ for all $\alpha \in S$, which requires $2^{m-1} - 1$ multiplications.

However, when $t + 1 = 2, 3$, f_1 is a 1-coefficient polynomial, so $f_1(\alpha) = f_1(0) = c$.

$$c \cdot \langle \delta_1, \ldots, \delta_{m-1} \rangle = \langle c \cdot \delta_1, \ldots, c \cdot \delta_{m-1} \rangle$$

Once we have all the $c \cdot \delta_i$ the subset sums can be computed in $2^{m-1} - m$ additions. Computing all the $c \cdot \delta_i$ requires $m - 1$ multiplications. Therefore $2^{m-1} - m$ of $2^{m-1} - 1$ multiplications are replaced by the same number of additions.

Syndrome computation

Syndrome computation is defined as the following linear map:

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

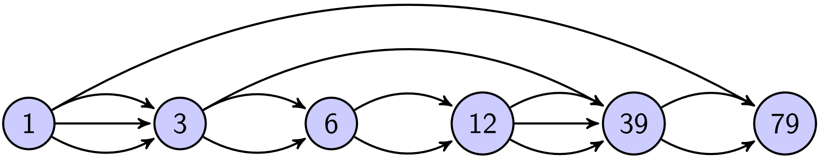
Consider the linear map $M^\intercal$:

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

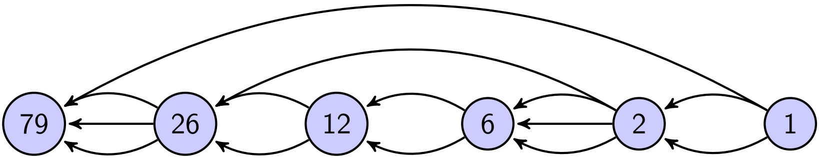
This transposed linear map is actually doing multipoint evaluation: syndrome computation is a transposed multipoint evaluation.

Transposing linear algorithms

Example: an addition chain for 79



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



Transposing linear algorithms

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

$$\begin{tikzcd}
\mathsf{in}_1 = a_0 \arrow[rr, "b_0"] \arrow[dr] & & a_0 b_0 \arrow[rr] \arrow[dr] & & \mathsf{out}_1 = a_0 b_0 \\
& a_0 + a_1 \arrow[rr, "b_0 + b_1"] & & \mathsf{out}_2 = a_0 b_1 + a_1 b_0 & \\
\mathsf{in}_2 = a_1 \arrow[rr, "b_1"] \arrow[ur] & & a_1 b_1 \arrow[rr] \arrow[ur] & & \mathsf{out}_3 = a_1 b_1
\end{tikzcd}$$

- Reversing the edges: $c_0, c_1, c_2 \to b_0 c_0 + b_1 c_1, b_0 c_1 + b_1 c_2$

$$\begin{tikzcd}
\mathsf{out}_1 = b_0 c_0 + b_1 c_1 & & c_0 + c_1 \arrow[ll, "b_0"'] & & \mathsf{in}_1 = c_0 \arrow[ll] \\
& (b_0 + b_1) c_1 \arrow[ul] \arrow[dl] & & \mathsf{in}_2 = c_1 \arrow[ll, "b_0 + b_1"'] \arrow[ul] \arrow[dl] & \\
\mathsf{out}_2 = b_0 c_1 + b_1 c_2 & & c_1 + c_2 \arrow[ll, "b_1"'] & & \mathsf{in}_3 = c_2 \arrow[ll]
\end{tikzcd}$$

Transposing linear algorithms

- 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.

Transposition principle

References:

- J. L. Bordewijk. *Inter-reciprocity applied to electrical networks*. 1956.
- O. B. Lupanov. *On rectifier and contact-rectifier circuits*. 1956.
- Charles M. Fiduccia. *On the algebraic complexity of matrix multiplication*. 1972.

Properties of the transposition principle:

- The reversal preserves the number of multiplications.
- The reversal preserves the number of additions plus the number of (nontrivial) outputs.

We compute the syndrome using a transposed additive FFT, including all the improvements.

Transposing the additive FFT

Naive approach

- The resulting algorithm is straight-line: no recursion/loops.
- This leads to efficiency problems: big code size, big memory demand.

Our current implementation: figure out the underlying code structure

- The order of components will be reversed in the transposed algorithm.

$$(M_1 M_2 \cdots M_n)^\intercal = (M_n^\intercal M_{n-1}^\intercal \cdots M_1^\intercal).$$

- The additive FFT can be combined with the divisions by $g(\alpha)^2$'s to save bit operations.

Secret permutation

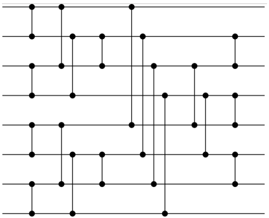
FFT output	1	0	0	1	0	...
$\mathbf{F}_q$ elements	α_1	α_2	α_3	α_4	α_5	...
support	$\alpha_{\pi(1)}$	$\alpha_{\pi(2)}$	$\alpha_{\pi(3)}$	$\alpha_{\pi(4)}$	$\alpha_{\pi(5)}$	...

- Need to apply some secret permutation to the output of the additive FFT. The same issue arises for the input of the transposed additive FFT.
- The secret permutation should not leak information about the permutation being performed: Can't just move data around by loads and stores.
- The approach we took: sorting network

Sorting network

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 $S[i] > S[j]$.



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

Sorting network

Permuting by sorting:

- Example: compute 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 $(1, b_3), (2, b_2), (3, b_1)$

Turning comparators into conditional swaps: Since the keys are independent of the input data b_i's, the conditions can be precomputed.

- Each comparator can be implemented with 4 operations:

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

A possibly better alternative: Beneš permutation network.

Timings

n	t	sec	perm	synd	key eq	root	perm	total
2048	32	87	3326	9081	4267	6699	3172	26544
4096	41	129	8622	20846	7714	14794	8520	60493

Table : Number of cycles for decoding

Future works

- Optimizing key equation solving using asymptotically faster algorithms
- Explore other decoding algorithms
- Optimizing constant multiplications
- Tower fields
- ...

Thanks for your attention.