Efficient implementation of code-based cryptography

D. J. Bernstein
University of Illinois at Chicago & Technische Universiteit Eindhoven

Joint work with:

Tung Chou
Technische Universiteit Eindhoven

Peter Schwabe
Radboud University Nijmegen
Objectives

Set new speed records for public-key cryptography.
Objectives

Set new speed records for public-key cryptography.

... at a high security level.
Objectives

Set new speed records for public-key cryptography. . . at a high security level. . . including protection against quantum computers.
Objectives

Set new speed records for public-key cryptography.

... at a high security level.

... including protection against quantum computers.

... including full protection against cache-timing attacks, branch-prediction attacks, etc.
Objectives

Set new speed records for public-key cryptography. . . at a high security level.

. . . including protection against quantum computers.

. . . including full protection against cache-timing attacks, branch-prediction attacks, etc.

. . . using code-based crypto with a solid track record.
Objectives

Set new speed records for public-key cryptography.

... at a high security level.

... including protection against quantum computers.

... including full protection against cache-timing attacks, branch-prediction attacks, etc.

... using code-based crypto with a solid track record.

... all of the above at once.
The track record

1978 McEliece proposed public-key code-based crypto.

1994 van Tilburg.
1994 Canteaut–Chabanne.
1998 Canteaut–Chabaud.
2009 Bernstein (post-quantum).
2009 Finiasz–Sendrier.
2010 Bernstein–Lange–Peters.
2011 May–Meurer–Thomae.
Examples of the competition

Some cycle counts on h9ivy (Intel Core i5-3210M, Ivy Bridge) from bench.cr.yp.to:

mceliece encrypt 73092
(2008 Biswas–Sendrier, \(\approx 2^{80}\))
gls254 DH 76212
(binary elliptic curve; CHES 2013)
kummer DH 88448
(hyperelliptic; Asiacrypt 2014)
curve25519 DH 182708
(conservative elliptic curve)
mceliece decrypt 1130908
ronald1024 decrypt 1313324
New decoding speeds

\[ \approx 2^{128} \text{ security } (n, t) = (4096, 41) : \]
New decoding speeds

$\approx 2^{128}$ security \((n, t) = (4096, 41)\):

**60493** Ivy Bridge cycles.

Talk will focus on this case.

(Decryption is slightly slower: includes hash, cipher, MAC.)
New decoding speeds

\[ \approx 2^{128} \text{ security } (n, t) = (4096, 41): 60493 \text{ Ivy Bridge cycles.} \]

Talk will focus on this case.

(Decryption is slightly slower: includes hash, cipher, MAC.)

\[ \approx 2^{80} \text{ security } (n, t) = (2048, 32): 26544 \text{ Ivy Bridge cycles.} \]
New decoding speeds

\[ \approx 2^{128} \text{ security } (n, t) = (4096, 41) : \text{60493 Ivy Bridge cycles.} \]

Talk will focus on this case.

(Decryption is slightly slower: includes hash, cipher, MAC.)

\[ \approx 2^{80} \text{ security } (n, t) = (2048, 32) : \text{26544 Ivy Bridge cycles.} \]

All load/store addresses and all branch conditions are public. Eliminates cache-timing attacks etc.

Similar improvements for CFS.
Constant-time fanaticism

The extremist’s approach to eliminate timing attacks:
Handle all secret data using only bit operations—
XOR (\(^\wedge\)), AND (\(\&\)), etc.
Constant-time fanaticism

The extremist’s approach to eliminate timing attacks: Handle all secret data using only bit operations—XOR (^), AND (\&), etc.

We take this approach.
Constant-time fanaticism

The extremist’s approach to eliminate timing attacks: Handle all secret data using only bit operations—XOR (\(^\cdot\)), AND (\(\&\)), etc.

We take this approach.

“How can this be competitive in speed? Are you really simulating field multiplication with hundreds of bit operations instead of simple log tables?”
Yes, we are.

Not as slow as it sounds! On a typical 32-bit CPU, the XOR instruction is actually 32-bit XOR, operating in parallel on vectors of 32 bits.
Yes, we are.

Not as slow as it sounds! On a typical 32-bit CPU, the XOR instruction is actually 32-bit XOR, operating in parallel on vectors of 32 bits.

Low-end smartphone CPU: 128-bit XOR every cycle.

Ivy Bridge: 256-bit XOR every cycle, or three 128-bit XORs.
Not immediately obvious that this “bitslicing” saves time for, e.g., multiplication in $\mathbb{F}_{2^{12}}$. 
Not immediately obvious that this “bitslicing” saves time for, e.g., multiplication in \( \mathbb{F}_{2^{12}} \).

But quite obvious that it saves time for addition in \( \mathbb{F}_{2^{12}} \).
Not immediately obvious that this “bitslicing” saves time for, e.g., multiplication in $\mathbb{F}_{2^{12}}$.

But quite obvious that it saves time for addition in $\mathbb{F}_{2^{12}}$.

Typical decoding algorithms have add, mult roughly balanced.

Coming next: how to save many adds and most mults. Nice synergy with bitslicing.
The additive FFT

Fix \( n = 4096 = 2^{12} \), \( t = 41 \).

Big final decoding step is to find all roots in \( \mathbb{F}_{2^{12}} \)
of \( f = c_{41}x^{41} + \cdots + c_0x^0 \).

For each \( \alpha \in \mathbb{F}_{2^{12}} \),
compute \( f(\alpha) \) by Horner’s rule:
41 adds, 41 mults.
The additive FFT

Fix $n = 4096 = 2^{12}$, $t = 41$.

Big final decoding step is to find all roots in $\mathbb{F}_{2^{12}}$ of $f = c_{41}x^{41} + \cdots + c_0x^0$.

For each $\alpha \in \mathbb{F}_{2^{12}}$, compute $f(\alpha)$ by Horner’s rule: 41 adds, 41 mults.

Or use Chien search: compute $c_ig^i$, $c_ig^{2i}$, $c_ig^{3i}$, etc. Cost per point: again 41 adds, 41 mults.
The additive FFT

Fix \( n = 4096 = 2^{12} \), \( t = 41 \).

Big final decoding step
is to find all roots in \( \mathbb{F}_{2^{12}} \)
of \( f = c_{41}x^{41} + \cdots + c_0x^0 \).

For each \( \alpha \in \mathbb{F}_{2^{12}} \),
compute \( f(\alpha) \) by Horner’s rule:
41 adds, 41 mults.

Or use Chien search: compute \( c_ig^i, c_ig^{2i}, c_ig^{3i}, \) etc. Cost per point: again 41 adds, 41 mults.

Our cost: **6.01** adds, **2.09** mults.
Asymptotics:
normally $t \in \Theta(n/\lg n)$,
so Horner’s rule costs
$\Theta(nt) = \Theta(n^2/\lg n)$. 
Asymptotics:

normally \( t \in \Theta(n/\lg n) \), so Horner’s rule costs
\[ \Theta(nt) = \Theta(n^2/\lg n). \]

Wait a minute.

Didn’t we learn in school that FFT evaluates
an \( n \)-coeff polynomial
at \( n \) points
using \( n^{1+o(1)} \) operations?
Isn’t this better than \( n^2/\lg n \)?
Standard radix-2 FFT:
Want to evaluate 
\[ f = c_0 + c_1 x + \cdots + c_{n-1} x^{n-1} \]
at all the \( n \)th roots of 1.

Write \( f \) as \( f_0(x^2) + xf_1(x^2) \).
Observe big overlap between 
\[ f(\alpha) = f_0(\alpha^2) + \alpha f_1(\alpha^2), \]
\[ f(-\alpha) = f_0(\alpha^2) - \alpha f_1(\alpha^2). \]

\( f_0 \) has \( n/2 \) coeffs;
evaluate at \((n/2)\)nd roots of 1
by same idea recursively.
Similarly \( f_1 \).
Useless in char 2: $\alpha = -\alpha$.
Standard workarounds are painful. FFT considered impractical.

Still quite expensive.

1996 von zur Gathen–Gerhard: some improvements.

2010 Gao–Mateer: much better additive FFT.
We use Gao–Mateer, plus some new improvements.
Gao and Mateer evaluate
\[ f = c_0 + c_1 x + \cdots + c_{n-1} x^{n-1} \]
on a size-\( n \) \( \mathbb{F}_2 \)-linear space.

Their main idea: Write \( f \) as
\[ f_0(x^2 + x) + xf_1(x^2 + x). \]

Big overlap between \( f(\alpha) = f_0(\alpha^2 + \alpha) + \alpha f_1(\alpha^2 + \alpha) \)
and \( f(\alpha + 1) = f_0(\alpha^2 + \alpha) + (\alpha + 1)f_1(\alpha^2 + \alpha). \)

“Twist” to ensure \( 1 \in \) space.
Then \( \{\alpha^2 + \alpha\} \) is a
size-\( (n/2) \) \( \mathbb{F}_2 \)-linear space.
Apply same idea recursively.
Results

60493 Ivy Bridge cycles:
  8622 for permutation.
20846 for syndrome.
  7714 for BM.
14794 for roots.
  8520 for permutation.

Code will be public domain.
We’re still speeding it up.

Also $10 \times$ speedup for CFS.

More information:

cr.yp.to/papers.html#mcbits
What you find in paper:

Cryptosystem specification.

Our speedups to additive FFT. (We now have more speedups: cr yp.to/papers.html#auth256.)

Fast syndrome computation without big precomputed matrix. Important for lightweight!