McBits: fast constant-time code-based cryptography

(to appear at CHES 2013)

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*.

Examples of the competition

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

`mceliece` encrypt	61440
(2008 Biswas–Sendrier, 2^{80})	
`gls254` DH	77468
(binary elliptic curve; CHES 2013)	
`kumfp127g` DH	116944
(hyperelliptic; Eurocrypt 2013)	
`curve25519` DH	182632
(conservative elliptic curve)	
`mceliece` **decrypt**	1219344
`ronald1024` decrypt	1340040

New decoding speeds

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

New decoding speeds

$(n,t) = (4096, 41)$; 2^{128} security:
60493 Ivy Bridge cycles.
Talk will focus on this case.

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

New decoding speeds

$(n, t) = (4096, 41)$; 2^{128} security:
60493 Ivy Bridge cycles.
Talk will focus on this case.

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

$(n, t) = (2048, 32)$; 2^{80} security:
26544 Ivy Bridge cycles.

New decoding speeds

$(n,t) = (4096, 41)$; 2^{128} security:
60493 Ivy Bridge cycles.
Talk will focus on this case.

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

$(n,t) = (2048, 32)$; 2^{80} security:
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 (`^`), 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 (^), 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 $\mathbf{F}_{2^{12}}$.

Not immediately obvious that this “bitslicing” saves time for, e.g., multiplication in $\mathbf{F}_{2^{12}}$.

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

Not immediately obvious that this “bitslicing” saves time for, e.g., multiplication in $\mathbf{F}_{2^{12}}$.

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

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 $\mathbf{F}_{2^{12}}$
of $f = c_{41}x^{41} + \cdots + c_0x^0$.

For each $\alpha \in \mathbf{F}_{2^{12}}$,
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 $\mathbf{F}_{2^{12}}$
of $f = c_{41}x^{41} + \cdots + c_0x^0$.

For each $\alpha \in \mathbf{F}_{2^{12}}$,
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 $\mathbf{F}_{2^{12}}$
of $f = c_{41}x^{41} + \cdots + c_0x^0$.

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

Or use Chien search: compute
$c_i g^i$, $c_i g^{2i}$, $c_i g^{3i}$, 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 nth roots of 1.

Write f as $f_0(x^2) + x f_1(x^2)$.
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.

1988 Wang–Zhu,
independently 1989 Cantor:
"additive FFT" in char 2.
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 $\mathbf{F}_2$-linear space.

Main idea: Write f as $f_0(x^2 + x) + x f_1(x^2 + x)$.

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-$(n/2)$ $\mathbf{F}_2$-linear space. Apply same idea recursively.

We generalize to
$f = c_0 + c_1 x + \cdots + c_t x^t$
for any $t < n$.

$\Rightarrow$ several optimizations,
not all of which are automated
by simply tracking zeros.

For $t = 0$: copy c_0.

For $t \in \{1, 2\}$:
f_1 is a constant.
Instead of multiplying
this constant by each α,
multiply only by generators
and compute subset sums.

Syndrome computation

Initial decoding step: compute

$s_0 = r_1 + r_2 + \cdots + r_n$,

$s_1 = r_1\alpha_1 + r_2\alpha_2 + \cdots + r_n\alpha_n$,

$s_2 = r_1\alpha_1^2 + r_2\alpha_2^2 + \cdots + r_n\alpha_n^2$,

$\vdots$,

$s_t = r_1\alpha_1^t + r_2\alpha_2^t + \cdots + r_n\alpha_n^t$.

$r_1, r_2, \ldots, r_n$ are received bits scaled by Goppa constants.

Typically precompute matrix mapping bits to syndrome.

Not as slow as Chien search but still $n^{2+o(1)}$ and huge secret key.

Compare to multipoint evaluation:

$f(\alpha_1) = c_0 + c_1\alpha_1 + \cdots + c_t\alpha_1^t,$

$f(\alpha_2) = c_0 + c_1\alpha_2 + \cdots + c_t\alpha_2^t,$

$\vdots,$

$f(\alpha_n) = c_0 + c_1\alpha_n + \cdots + c_t\alpha_n^t.$

Compare to multipoint evaluation:

$f(\alpha_1) = c_0 + c_1\alpha_1 + \cdots + c_t\alpha_1^t$,

$f(\alpha_2) = c_0 + c_1\alpha_2 + \cdots + c_t\alpha_2^t$,

$\vdots$,

$f(\alpha_n) = c_0 + c_1\alpha_n + \cdots + c_t\alpha_n^t$.

Matrix for syndrome computation
is transpose of
matrix for multipoint evaluation.

Compare to multipoint evaluation:

$f(\alpha_1) = c_0 + c_1\alpha_1 + \cdots + c_t\alpha_1^t$,
$f(\alpha_2) = c_0 + c_1\alpha_2 + \cdots + c_t\alpha_2^t$,
$\vdots$,
$f(\alpha_n) = c_0 + c_1\alpha_n + \cdots + c_t\alpha_n^t$.

Matrix for syndrome computation is transpose of matrix for multipoint evaluation.

Amazing consequence: syndrome computation is as few ops as multipoint evaluation. Eliminate precomputed matrix.

Transposition principle:
If a linear algorithm
computes a matrix M
then reversing edges and
exchanging inputs/outputs
computes the transpose of M.

1956 Bordewijk;
independently 1957 Lupanov
for Boolean matrices.

1973 Fiduccia analysis:
preserves number of mults;
preserves number of adds plus
number of nontrivial outputs.

We built transposing compiler producing C code.

Too many variables for $m = 13$; gcc ran out of memory.

We built transposing compiler producing C code.
Too many variables for $m = 13$; `gcc` ran out of memory.

Used `qhasm` register allocator to optimize the variables.
Worked, but not very quickly.

We built transposing compiler producing C code.
Too many variables for $m = 13$; gcc ran out of memory.

Used qhasm register allocator to optimize the variables.
Worked, but not very quickly.

Wrote faster register allocator.
Still excessive code size.

We built transposing compiler
producing C code.
Too many variables for $m = 13$;
gcc ran out of memory.

Used `qhasm` register allocator
to optimize the variables.
Worked, but not very quickly.

Wrote faster register allocator.
Still excessive code size.

Built new interpreter,
allowing some code compression.
Still big; still some overhead.

Better solution:
stared at additive FFT,
wrote down transposition
with same loops etc.

Small code, no overhead.

Speedups of additive FFT
translate easily
to transposed algorithm.

Further savings:
merged first stage with
scaling by Goppa constants.

Secret permutation

Additive FFT $\Rightarrow f$ values at field elements *in a standard order*.

This is not the order needed in code-based crypto! Must apply a secret permutation, part of the secret key.

Same issue for syndrome.

Solution: Batcher sorting. Almost done with faster solution: Beneš network.

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.

More information:
cr.yp.to/papers.html#mcbits