

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.

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.

... at a high security level.

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.

... at a high security level.

... including protection  
against quantum computers.

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.

... at a high security level.

... including protection  
against quantum computers.

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

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.

... 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.

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.

... 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*.

constant-time  
based cryptography  
(presented at CHES 2013)

Barbara  
University of Illinois at Chicago &  
Technische Universiteit Eindhoven

collaborated with:

Barbara  
Technische Universiteit Eindhoven

Barbara  
University of Nijmegen

## 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*.

## Example

Some cy...  
(Intel Co...  
from [ber...](#)

mcelieo...  
(2008 B...  
g1s254...

(binary e...  
kumfp12...  
(hyperel...

curve25...  
(conserv...  
mcelieo...

ronald1...

graphy  
(AES 2013)

is at Chicago &  
iteit Eindhoven

iteit Eindhoven

ty Nijmegen

## 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 c

Some cycle counts  
(Intel Core i5-3210)  
from [bench.cr.yp.to](http://bench.cr.yp.to)

mceliece encrypt  
(2008 Biswas–Sen)  
g1s254 DH

(binary elliptic cur  
kumfp127g DH

(hyperelliptic; Euro  
curve25519 DH

(conservative ellipt  
mceliece **decrypt**

ronald1024 decry



## 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 Br) from [bench.cr.yp.to](http://bench.cr.yp.to):

```
mceliece encrypt  
(2008 Biswas–Sendrier,  $2^{80}$ )  
g1s254 DH  
(binary elliptic curve; CHES  
kumfp127g DH 1  
(hyperelliptic; Eurocrypt 201  
curve25519 DH 1  
(conservative elliptic curve)  
mceliece decrypt 12  
ronald1024 decrypt 13
```

## 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](http://bench.cr.yp.to):

mceliece encrypt	61440
(2008 Biswas–Sendrier, $2^{80}$ )	
g1s254 DH	77468
(binary elliptic curve; CHES 2013)	
kumfp127g DH	116944
(hyperelliptic; Eurocrypt 2013)	
curve25519 DH	182632
(conservative elliptic curve)	
mceliece <b>decrypt</b>	1219344
ronald1024 decrypt	1340040

es

speed records  
c-key cryptography.  
high security level.

uding protection  
quantum computers.

uding full protection  
cache-timing attacks,  
prediction attacks, etc.

g code-based crypto  
olid track record.

f the above *at once*.

## Examples of the competition

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

mceliece encrypt	61440
(2008 Biswas–Sendrier, $2^{80}$ )	
g1s254 DH	77468
(binary elliptic curve; CHES 2013)	
kumfp127g DH	116944
(hyperelliptic; Eurocrypt 2013)	
curve25519 DH	182632
(conservative elliptic curve)	
mceliece <b>decrypt</b>	1219344
ronald1024 decrypt	1340040

New dec

$(n, t) =$

## Examples of the competition

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

<code>mceliece encrypt</code>	61440
(2008 Biswas–Sendrier, $2^{80}$ )	
<code>g1s254 DH</code>	77468
(binary elliptic curve; CHES 2013)	
<code>kumfp127g DH</code>	116944
(hyperelliptic; Eurocrypt 2013)	
<code>curve25519 DH</code>	182632
(conservative elliptic curve)	
<code>mceliece <b>decrypt</b></code>	1219344
<code>ronald1024 decrypt</code>	1340040

## New decoding spe

$(n, t) = (4096, 41)$

## Examples of the competition

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

mceliece encrypt	61440
(2008 Biswas–Sendrier, $2^{80}$ )	
g1s254 DH	77468
(binary elliptic curve; CHES 2013)	
kumfp127g DH	116944
(hyperelliptic; Eurocrypt 2013)	
curve25519 DH	182632
(conservative elliptic curve)	
mceliece <b>decrypt</b>	1219344
ronald1024 decrypt	1340040

## New decoding speeds

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

## Examples of the competition

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

mceliece encrypt	61440
(2008 Biswas–Sendrier, $2^{80}$ )	
g1s254 DH	77468
(binary elliptic curve; CHES 2013)	
kumfp127g DH	116944
(hyperelliptic; Eurocrypt 2013)	
curve25519 DH	182632
(conservative elliptic curve)	
mceliece <b>decrypt</b>	1219344
ronald1024 decrypt	1340040

## New decoding speeds

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

## Examples of the competition

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

mceliece encrypt	61440
(2008 Biswas–Sendrier, $2^{80}$ )	
g1s254 DH	77468
(binary elliptic curve; CHES 2013)	
kumfp127g DH	116944
(hyperelliptic; Eurocrypt 2013)	
curve25519 DH	182632
(conservative elliptic curve)	
mceliece <b>decrypt</b>	1219344
ronald1024 decrypt	1340040

## 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.)

## Examples of the competition

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

mceliece encrypt	61440
(2008 Biswas–Sendrier, $2^{80}$ )	
g1s254 DH	77468
(binary elliptic curve; CHES 2013)	
kumfp127g DH	116944
(hyperelliptic; Eurocrypt 2013)	
curve25519 DH	182632
(conservative elliptic curve)	
mceliece <b>decrypt</b>	1219344
ronald1024 decrypt	1340040

## 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.



## Examples of the competition

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

mceliece encrypt	61440
(2008 Biswas–Sendrier, $2^{80}$ )	
g1s254 DH	77468
(binary elliptic curve; CHES 2013)	
kumfp127g DH	116944
(hyperelliptic; Eurocrypt 2013)	
curve25519 DH	182632
(conservative elliptic curve)	
mceliece <b>decrypt</b>	1219344
ronald1024 decrypt	1340040

## 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.

## Results of the competition

cycle counts on h9ivy

(Core i5-3210M, Ivy Bridge)

[bench.cr.yp.to](http://bench.cr.yp.to):

128-bit AES encrypt 61440

(Serpent–Sendrier,  $2^{80}$ )

256-bit DH 77468

(Curve25519 elliptic curve; CHES 2013)

256-bit DH 116944

(Curve25519 elliptic; Eurocrypt 2013)

256-bit DH 182632

(Curve25519 elliptic curve)

128-bit AES **decrypt** 1219344

128-bit AES 1024 decrypt 1340040

## 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

The extra

to elimin

Handle a

using on

XOR ( $\sim$ )

## Competition

s on h9ivy

DM, Ivy Bridge)

p.to:

61440

drier,  $2^{80}$ )

77468

ve; CHES 2013)

116944

ocrypt 2013)

182632

tic curve)

t 1219344

ypt 1340040

## 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 fan

The extremist's ap

to eliminate timing

Handle all secret c

using only bit oper

XOR ( $\sim$ ), AND ( $\&$ )

## 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 ( $\wedge$ ), AND ( $\&$ ), etc.

## 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 ( $\sim$ ), AND ( $\&$ ), etc.

## 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 ( $\sim$ ), AND ( $\&$ ), etc.

We take this approach.

## 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 ( $\sim$ ), 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?”

coding speeds

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

vy Bridge cycles.

l focus on this case.

tion is slightly slower:

hash, cipher, MAC.)

(2048, 32);  $2^{80}$  security:

vy Bridge cycles.

/store addresses

branch conditions

ic. Eliminates

ming attacks etc.

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.

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

Not as s

On a typ

the XOR

is actual

operatin

on vecto



eds

;  $2^{128}$  security:

cycles.

this case.

htly slower:

ner, MAC.)

;  $2^{80}$  security:

cycles.

resses

ditions

ates

cks etc.

ents for CFS.

## Constant-time fanaticism

The extremist's approach  
to eliminate timing attacks:

Handle all secret data  
using only bit operations—  
XOR ( $\sim$ ), 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 s  
On a typical 32-bit  
the XOR instruction  
is actually 32-bit X  
operating in parall  
on vectors of 32 b

## Constant-time fanaticism

The extremist's approach  
to eliminate timing attacks:

Handle all secret data  
using only bit operations—  
XOR ( $\wedge$ ), 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.

## Constant-time fanaticism

The extremist's approach  
to eliminate timing attacks:

Handle all secret data  
using only bit operations—  
XOR ( $\sim$ ), 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.

## Constant-time fanaticism

The extremist's approach to eliminate timing attacks:

Handle all secret data using only bit operations—XOR ( $\sim$ ), 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.

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

Ivy Bridge: 256-bit XOR every cycle, or three 128-bit XORs.

Real-time fanaticism

remist's approach  
to timing attacks:  
all secret data  
is only bit operations—  
XOR (^), AND (&), etc.

this approach.  
Can this be  
competitive in speed?  
Is it really simulating  
multiplication with  
a series of bit operations  
or 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.

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

Ivy Bridge:  
256-bit XOR every cycle,  
or three 128-bit XORs.

Not imm  
that this  
saves tim  
multiplic

aticism

approach

g attacks:

data

rations—

), etc.

oach.

ed?

ulating

n with

operations

og 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.

Low-end smartphone CPU:

128-bit XOR every cycle.

Ivy Bridge:

256-bit XOR every cycle,

or three 128-bit XORs.

Not immediately o

that this “bitslicin

saves time for, e.g.

multiplication in F

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  $\mathbf{F}_{2^{12}}$ .

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  $\mathbf{F}_{2^{12}}$ .



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  $\mathbf{F}_{2^{12}}$ .

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

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  $\mathbf{F}_{2^{12}}$ .

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.

are.

slow as it sounds!

typical 32-bit CPU,

R instruction

ly 32-bit XOR,

g in parallel

ors of 32 bits.

smartphone CPU:

XOR every cycle.

ge:

XOR every cycle,

128-bit XORs.

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 add

Fix  $n =$

Big final

is to find

of  $f = a$

For each

compute

41 adds,

sounds!  
t CPU,  
on  
KOR,  
el  
its.  
one CPU:  
/ cycle.  
/ cycle,  
ORs.

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  $n = 4096 = 2^{12}$

Big final decoding  
is to find all roots  
of  $f = c_{41}x^{41} + \dots$

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

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  $f = c_{41}x^{41} + \dots + c_0x^0$ .

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

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.

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.

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.

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.

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.



Immediately obvious

is “bitslicing”

same for, e.g.,

addition in  $\mathbf{F}_{2^{12}}$ .

It is obvious that it

is the same for addition in  $\mathbf{F}_{2^{12}}$ .

Other decoding algorithms

are not, mult roughly balanced.

Next: how to save

adds and *most* mults.

Save energy 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  $f = c_{41}x^{41} + \dots + c_0x^0$ .

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.

Asymptotically

normally

so Horner's

$\Theta(nt) =$

obvious

$g''$

.

$\mathbb{F}_{2^{12}}$ .

that it

is in  $\mathbb{F}_{2^{12}}$ .

algorithms

are roughly balanced.

to save

most mults.

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  $f = c_{41}x^{41} + \dots + c_0x^0$ .

For each  $\alpha \in \mathbb{F}_{2^{12}}$ ,

compute  $f(\alpha)$  by Horner's rule:

41 adds, 41 mults.

Or use Chien search: compute

$c_i g^i, c_i g^{2i}, c_i g^{3i}$ , 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

$\Theta(nt) = \Theta(n^2 / \lg n)$

## 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  $f = c_{41}x^{41} + \dots + c_0x^0$ .

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

$\Theta(nt) = \Theta(n^2 / \lg n)$ .

## 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

$\Theta(nt) = \Theta(n^2 / \lg n)$ .

## 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  $f = c_{41}x^{41} + \dots + c_0x^0$ .

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

$\Theta(nt) = \Theta(n^2 / \lg n)$ .

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  $n^2 / \lg n$ ?

## Iterative FFT

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

decoding step

find all roots in  $\mathbf{F}_{2^{12}}$

$$c_{41}x^{41} + \dots + c_0x^0.$$

for  $\alpha \in \mathbf{F}_{2^{12}}$ ,

evaluate  $f(\alpha)$  by Horner's rule:

41 mults.

Chien search: compute

$\alpha^{2^i}, c_i\alpha^{3^i}$ , etc. Cost per

gain 41 adds, 41 mults.

total: **6.01** adds, **2.09** mults.

Asymptotics:

normally  $t \in \Theta(n / \lg n)$ ,

so Horner's rule costs

$$\Theta(nt) = \Theta(n^2 / \lg n).$$

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

Want to

$$f = c_0 +$$

at all the

Write  $f$

Observe

$$f(\alpha) =$$

$$f(-\alpha) =$$

$f_0$  has  $n$

evaluate

by same

Similarly

$12$ ,  $t = 41$ .

step

in  $\mathbf{F}_{2^{12}}$

$\dots + c_0 x^0$ .

Horner's rule:

ch: compute

etc. Cost per

adds, 41 mults.

adds, **2.09** mults.

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

$f = c_0 + c_1 x + \dots$

at all the  $n$ th roots

Write  $f$  as  $f_0(x^2)$

Observe big overlap

$f(\alpha) = f_0(\alpha^2) + \dots$

$f(-\alpha) = f_0(\alpha^2) - \dots$

$f_0$  has  $n/2$  coeffs;

evaluate at  $(n/2)$  roots

by same idea recursively

Similarly  $f_1$ .

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  $f_0(x^2) + x f_1(x^2)$ .

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  $f_1$ .



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

$$f = c_0 + c_1x + \cdots + c_{n-1}x^{n-1}$$

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  $f_1$ .

otics:

$t \in \Theta(n / \lg n)$ ,

er's rule costs

$= \Theta(n^2 / \lg n)$ .

minute.

ve learn in school

T evaluates

eff polynomial

nts

$+o(1)$  operations?

s better than  $n^2 / \lg n$ ?

Standard radix-2 FFT:

Want to evaluate

$$f = c_0 + c_1x + \dots + c_{n-1}x^{n-1}$$

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 i

Standard

FFT com

1988 Wa

independ

“additive

Still quit

1996 vor

some im

2010 Ga

much be

We use

plus som

$(\lg n)$ ,  
costs  
 $n$ ).  
school  
s  
mial  
rations?  
an  $n^2 / \lg n$ ?

Standard radix-2 FFT:

Want to evaluate

$$f = c_0 + c_1x + \dots + c_{n-1}x^{n-1}$$

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:

Standard workarou

FFT considered im

1988 Wang–Zhu,

independently 198

“additive FFT” in

Still quite expensiv

1996 von zur Gath

some improvement

2010 Gao–Mateer:

much better addit

We use Gao–Mate

plus some new imp

Standard radix-2 FFT:

Want to evaluate

$$f = c_0 + c_1x + \cdots + c_{n-1}x^{n-1}$$

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 p

FFT considered impractical.

1988 Wang–Zhu,

independently 1989 Cantor:

“additive FFT” in char 2.

Still quite expensive.

1996 von zur Gathen–Gerha

some improvements.

2010 Gao–Mateer:

much better additive FFT.

We use Gao–Mateer,

plus some new improvement

$n?$

Standard radix-2 FFT:

Want to evaluate

$$f = c_0 + c_1x + \cdots + c_{n-1}x^{n-1}$$

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.

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.

radix-2 FFT:

evaluate

$$c_0 + c_1x + \dots + c_{n-1}x^{n-1}$$

at the  $n$ th roots of 1.

$$\text{as } f_0(x^2) + xf_1(x^2).$$

big overlap between

$$f_0(\alpha^2) + \alpha f_1(\alpha^2),$$

$$= f_0(\alpha^2) - \alpha f_1(\alpha^2).$$

$n/2$  coeffs;

at  $(n/2)$ nd roots of 1

idea recursively.

$f_1$ .

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

$$f = c_0 +$$

on a size

Main ide

$$f_0(x^2 +$$

Big over

$$f_0(\alpha^2 +$$

and  $f(\alpha$

$$f_0(\alpha^2 +$$

“Twist”

Then  $\{c$

size- $(n/$

Apply sa

FFT:

$$\dots + c_{n-1}x^{n-1}$$

roots of 1.

$$+ x f_1(x^2).$$

map between

$$\alpha f_1(\alpha^2),$$

$$- \alpha f_1(\alpha^2).$$

and roots of 1

recursively.

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 e

$$f = c_0 + c_1x + \dots$$

on a size- $n$   $\mathbf{F}_2$ -line

Main idea: Write

$$f_0(x^2 + x) + x f_1(x)$$

Big overlap between

$$f_0(\alpha^2 + \alpha) + \alpha f_1$$

$$\text{and } f(\alpha + 1) =$$

$$f_0(\alpha^2 + \alpha) + (\alpha -$$

“Twist” to ensure

Then  $\{\alpha^2 + \alpha\}$  is

size- $(n/2)$   $\mathbf{F}_2$ -line

Apply same idea r

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  
 $f = c_0 + c_1x + \cdots + c_{n-1}x$   
on a size- $n$   $\mathbf{F}_2$ -linear space.

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



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  
 $f = c_0 + c_1x + \cdots + c_{n-1}x^{n-1}$   
on a size- $n$   $\mathbf{F}_2$ -linear space.

Main idea: Write  $f$  as  
 $f_0(x^2 + x) + xf_1(x^2 + x)$ .

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

in char 2:  $\alpha = -\alpha$ .

and workarounds are painful.

considered impractical.

ang-Zhu,

idently 1989 Cantor:

the FFT" in char 2.

is expensive.

n zur Gathen-Gerhard:

improvements.

o-Mateer:

better additive FFT.

Gao-Mateer,

the new improvements.

Gao and Mateer evaluate

$$f = c_0 + c_1x + \cdots + c_{n-1}x^{n-1}$$

on a size- $n$   $\mathbf{F}_2$ -linear space.

Main idea: Write  $f$  as

$$f_0(x^2 + x) + xf_1(x^2 + x).$$

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

Apply same idea recursively.

We gener

$$f = c_0 +$$

for any  $t$

$\Rightarrow$  sever

not all o

by simpl

For  $t =$

For  $t \in$

$f_1$  is a c

Instead o

this cons

multiply

and com

$$\alpha = -\alpha.$$

unds are painful.

npractical.

9 Cantor:

char 2.

ve.

men–Gerhard:

ts.

ive FFT.

er,

provements.

Gao and Mateer evaluate

$$f = c_0 + c_1x + \cdots + c_{n-1}x^{n-1}$$

on a size- $n$   $\mathbf{F}_2$ -linear space.

Main idea: Write  $f$  as

$$f_0(x^2 + x) + xf_1(x^2 + x).$$

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

Apply same idea recursively.

We generalize to

$$f = c_0 + c_1x + \cdots$$

for any  $t < n$ .

$\Rightarrow$  several optimiz

not all of which ar

by simply tracking

For  $t = 0$ : copy  $c_0$

For  $t \in \{1, 2\}$ :

$f_1$  is a constant.

Instead of multiply

this constant by ea

multiply only by g

and compute subs

ainful.

Gao and Mateer evaluate

$$f = c_0 + c_1x + \cdots + c_{n-1}x^{n-1}$$

on a size- $n$   $\mathbf{F}_2$ -linear space.

Main idea: Write  $f$  as

$$f_0(x^2 + x) + xf_1(x^2 + x).$$

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

Apply same idea recursively.

rd:

s.

We generalize to

$$f = c_0 + c_1x + \cdots + c_t x^t$$

for any  $t < n$ .

$\Rightarrow$  several optimizations,

not all of which are automa

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  $\alpha$ ,

multiply only by generators

and compute subset sums.

Gao and Mateer evaluate

$$f = c_0 + c_1x + \cdots + c_{n-1}x^{n-1}$$

on a size- $n$   $\mathbf{F}_2$ -linear space.

Main idea: Write  $f$  as

$$f_0(x^2 + x) + xf_1(x^2 + x).$$

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

Apply same idea recursively.

We generalize to

$$f = c_0 + c_1x + \cdots + c_t x^t$$

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  $\alpha$ ,  
multiply only by generators  
and compute subset sums.

Mateer evaluate  
 $c_1 x + \dots + c_{n-1} x^{n-1}$   
 e- $n$   $\mathbf{F}_2$ -linear space.

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

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

to ensure  $1 \in$  space.  
 $\{x^2 + x\}$  is a

2)  $\mathbf{F}_2$ -linear space.

same idea recursively.

We generalize to

$f = c_0 + c_1 x + \dots + c_t x^t$   
 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  $\alpha$ ,  
 multiply only by generators  
 and compute subset sums.

Syndrom

Initial de

$$s_0 = r_1$$

$$s_1 = r_1 c$$

$$s_2 = r_1 c$$

$\vdots$

$$s_t = r_1 c$$

$$r_1, r_2, \dots$$

scaled by

Typically

mapping

Not as s

still  $n^2+$

evaluate

$$\dots + c_{n-1}x^{n-1}$$

near space.

$f$  as

$$(x^2 + x).$$

then  $f(\alpha) =$

$$(\alpha^2 + \alpha)$$

$$+ 1) f_1(\alpha^2 + \alpha).$$

$1 \in$  space.

a

near space.

recursively.

We generalize to

$$f = c_0 + c_1x + \dots + c_t x^t$$

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  $\alpha$ ,

multiply only by generators

and compute subset sums.

Syndrome computation

Initial decoding step

$$s_0 = r_1 + r_2 + \dots$$

$$s_1 = r_1\alpha_1 + r_2\alpha_2$$

$$s_2 = r_1\alpha_1^2 + r_2\alpha_2^2$$

$\vdots$ ,

$$s_t = r_1\alpha_1^t + r_2\alpha_2^t$$

$r_1, r_2, \dots, r_n$  are

scaled by Goppa c

Typically precomp

mapping bits to sy

Not as slow as Ch

still  $n^{2+o(1)}$  and h

We generalize to

$$f = c_0 + c_1x + \dots + c_t x^t$$

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  $\alpha$ ,  
multiply only by generators  
and compute subset sums.

## Syndrome computation

Initial decoding step: compute

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

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

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

$\vdots$

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

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

Typically precompute matrix  
mapping bits to syndrome.

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



We generalize to

$$f = c_0 + c_1x + \cdots + c_t x^t$$

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  $\alpha$ ,  
multiply only by generators  
and compute subset sums.

## Syndrome computation

Initial decoding step: compute

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

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

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

$\vdots$ ,

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

$r_1, r_2, \dots, r_n$  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.

Generalize to

$$c_0 + c_1 x + \dots + c_t x^t$$

$t < n$ .

Various optimizations,  
many of which are automated  
by tracking zeros.

0: copy  $c_0$ .

{1, 2}:

constant.

of multiplying

constant by each  $\alpha$ ,

only by generators

compute subset sums.

## Syndrome computation

Initial decoding step: compute

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

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

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

$\vdots$ ,

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

$r_1, r_2, \dots, r_n$  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

$$f(\alpha_1) =$$

$$f(\alpha_2) =$$

$\vdots$ ,

$$f(\alpha_n) =$$

## Syndrome computation

Initial decoding step: compute

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

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

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

$\vdots$ ,

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

$r_1, r_2, \dots, r_n$  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 multip

$$f(\alpha_1) = c_0 + c_1\alpha_1$$

$$f(\alpha_2) = c_0 + c_1\alpha_2$$

$\vdots$ ,

$$f(\alpha_n) = c_0 + c_1\alpha_n$$

## Syndrome computation

Initial decoding step: compute

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

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

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

$\vdots$ ,

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

$r_1, r_2, \dots, r_n$  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

$$f(\alpha_1) = c_0 + c_1\alpha_1 + \cdots + c_n\alpha_1^n$$

$$f(\alpha_2) = c_0 + c_1\alpha_2 + \cdots + c_n\alpha_2^n$$

$\vdots$ ,

$$f(\alpha_n) = c_0 + c_1\alpha_n + \cdots + c_n\alpha_n^n$$

## Syndrome computation

Initial decoding step: compute

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

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

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

$\vdots$ ,

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

$r_1, r_2, \dots, r_n$  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:

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

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

$\vdots$ ,

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

## Syndrome computation

Initial decoding step: compute

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

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

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

$\vdots$ ,

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

$r_1, r_2, \dots, r_n$  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:

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

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

$\vdots$ ,

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

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

## Syndrome computation

Initial decoding step: compute

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

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

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

$\vdots$ ,

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

$r_1, r_2, \dots, r_n$  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:

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

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

$\vdots$ ,

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

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.

## the computation

Decoding step: compute

$$r_1 + r_2 + \dots + r_n,$$

$$r_1 \alpha_1 + r_2 \alpha_2 + \dots + r_n \alpha_n,$$

$$r_1 \alpha_1^2 + r_2 \alpha_2^2 + \dots + r_n \alpha_n^2,$$

$$r_1 \alpha_1^t + r_2 \alpha_2^t + \dots + r_n \alpha_n^t.$$

$r_1, \dots, r_n$  are received bits

by Goppa constants.

Precompute matrix

of bits to syndrome.

As slow as Chien search but

$O(1)$  and huge secret key.

Compare to multipoint evaluation:

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

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

$$\vdots,$$

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

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.

Transpose

If a linear

computation

then reverse

exchange

computation

1956 Bo

independence

for Boolean

1973 Fic

preserves

preserves

number



ation

ep: compute

$$+ r_n,$$

$$+ \dots + r_n \alpha_n,$$

$$+ \dots + r_n \alpha_n^2,$$

$$+ \dots + r_n \alpha_n^t.$$

received bits

constants.

ute matrix

ndrome.

ien search but

uge secret key.

Compare to multipoint evaluation:

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

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

$\vdots$ ,

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

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

then reversing edge

exchanging inputs,

computes the trans

1956 Bordewijk;

independently 195

for Boolean matrices

1973 Fiduccia ana

preserves number

preserves number

number of nontriv

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

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.

to multipoint evaluation:

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

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

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

for syndrome computation

use of

for multipoint evaluation.

ing consequence:

the computation is as few

multipoint evaluation.

the 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

producing

Too many

gcc ran

point evaluation:

$$c_1 + \dots + c_t \alpha_1^t,$$
$$c_2 + \dots + c_t \alpha_2^t,$$

$$c_n + \dots + c_t \alpha_n^t.$$

me computation

point evaluation.

ence:

ation is as few

evaluation.

puted 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 transposi

producing C code.

Too many variable

gcc ran out of me

uation:

$c_t \alpha_1^t,$

$c_t \alpha_2^t,$

$c_t \alpha_n^t.$

tation

tion.

few

rix.

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  $m =$   
gcc ran out of memory.

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.

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.

Used qhasm register allocator  
to optimize the variables.

Worked, but not very quickly.



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.

Used qhasm register allocator  
to optimize the variables.

Worked, but not very quickly.

Wrote faster register allocator.  
Still excessive code size.

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.

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.

position principle:  
ear algorithm  
es a matrix  $M$   
ersing edges and  
ing inputs/outputs  
es the transpose of  $M$ .  
rdewijk;  
dently 1957 Lupanov  
ean matrices.  
duccia analysis:  
s number of mults;  
s number of adds plus  
of nontrivial outputs.

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 s  
stared at  
wrote do  
with sam  
Small co  
Speedup  
translate  
to transp  
Further  
merged  
scaling b

principle:  
m  
k  $M$   
es and  
/outputs  
sponse of  $M$ .  
7 Lupanov  
ces.  
lysis:  
of mults;  
of adds plus  
ial outputs.

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  
wrote down transp  
with same loops e  
Small code, no ov  
Speedups of addit  
translate easily  
to transposed algo  
Further savings:  
merged first stage  
scaling by Goppa c

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:

started 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.

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:

started 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.

transposing compiler  
C code.  
many variables for  $m = 13$ ;  
out of memory.  
asm register allocator  
optimize the variables.  
but not very quickly.  
faster register allocator.  
excessive code size.  
new interpreter,  
some code compression.  
still some overhead.

Better solution:  
started 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 p  
Additive  
field elem  
This is m  
needed i  
Must ap  
part of t  
Same iss  
Solution  
Almost c  
Beneš ne

ng compiler

es for  $m = 13$ ;  
emory.

er allocator  
riables.

very quickly.

ter allocator.  
e size.

ter,  
e compression.  
e 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  $\Rightarrow j$   
field elements *in a*

This is not the ord  
needed in code-ba  
Must apply a secre  
part of the secret

Same issue for sym

Solution: Batcher  
Almost done with  
Beneš network.



Better solution:

started 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*

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: Batchier sorting.  
Almost done with faster solution  
Beneš network.

Better solution:

started 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.

solution:  
t additive FFT,  
own transposition  
ne loops etc.  
ode, no overhead.  
s of additive FFT  
e easily  
posed algorithm.  
savings:  
first stage with  
y 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: Batchier sorting.

Almost done with faster solution:  
Beneš network.

## Results

60493 Iv

8622 fo

20846 fo

7714 fo

14794 fo

8520 fo

Code wi

We're st

More inf

[cr.yp.to](http://cr.yp.to)

FFT,  
position  
tc.  
erhead.  
ive FFT  
rithm.  
with  
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 c

8622 for permuta

20846 for syndrom

7714 for BM.

14794 for roots.

8520 for permuta

Code will be publi

We're still speedin

More information:

[cr.yp.to/papers](http://cr.yp.to/papers)

## 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: Batchier 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.yep.to/papers.html#m](http://cr.yep.to/papers.html#m)

## 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: Batchier 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](http://cr.yp.to/papers.html#mcbits)