McBits: fast constant-time 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.

McBits:
fast constant-time
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.

. . . at a high security level.

McBits:
fast constant-time 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.

. . . at a high security level.

. . . including protection against quantum computers.

McBits:
fast constant-time
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.

. . . 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

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

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*.

stant-time
sed cryptography

ernstein
ty of Illinois at Chicago &
che Universiteit Eindhoven

ork with:

hou
che Universiteit Eindhoven

chwabe
d 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*.

The trac

1978 Mc
public-ke

Has held
optimiza
1962 Pra
1988 Lee
1989 Kr
1989 Du
1990 Co
1990 va
1991 Co
1993 Ch
1993 Ch

graphy

is at Chicago &
siteit Eindhoven

siteit 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*.

The track record

1978 McEliece pro
public-key code-ba

Has held up well a
optimization of at
1962 Prange. 198
1988 Lee–Brickell.
1989 Krouk. 1989
1989 Dumer.
1990 Coffey–Good
1990 van Tilburg.
1991 Coffey–Good
1993 Chabanne–C
1993 Chabaud.

ago &
hoven

hoven

en

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

Has held up well after exten
optimization of attack algori
1962 Prange. 1981 Omura.
1988 Lee–Brickell. 1988 Leo
1989 Krouk. 1989 Stern.
1989 Dumer.
1990 Coffey–Goodman.
1990 van Tilburg. 1991 Dur
1991 Coffey–Goodman–Farr
1993 Chabanne–Courteau.
1993 Chabaud.

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.

Has held up well after extensive
optimization of attack algorithms:
1962 Prange. 1981 Omura.
1988 Lee–Brickell. 1988 Leon.
1989 Krouk. 1989 Stern.
1989 Dumer.
1990 Coffey–Goodman.
1990 van Tilburg. 1991 Dumer.
1991 Coffey–Goodman–Farrell.
1993 Chabanne–Courteau.
1993 Chabaud.

es

speed records
c-key cryptography.

high security level.

uding protection
quantum computers.

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

g code-based crypto
olid track record.

f the above *at once*.

The track record

1978 McEliece proposed
public-key code-based crypto.

Has held up well after extensive
optimization of attack algorithms:
1962 Prange. 1981 Omura.
1988 Lee–Brickell. 1988 Leon.
1989 Krouk. 1989 Stern.
1989 Dumer.
1990 Coffey–Goodman.
1990 van Tilburg. 1991 Dumer.
1991 Coffey–Goodman–Farrell.
1993 Chabanne–Courteau.
1993 Chabaud.

1994 var
1994 Ca
1998 Ca
1998 Ca
2008 Be
2009 Be
Peters–v
2009 Be
2009 Fir
2010 Be
2011 Ma
2011 Be
2012 Be
2013 Be
Meurer (

ords
otography.

rity level.

ection
computers.

protection
ng attacks,
attacks, etc.

sed crypto
record.

ve *at once*.

The track record

1978 McEliece proposed public-key code-based crypto.

Has held up well after extensive optimization of attack algorithms:
1962 Prange. 1981 Omura.
1988 Lee–Brickell. 1988 Leon.
1989 Krouk. 1989 Stern.
1989 Dumer.
1990 Coffey–Goodman.
1990 van Tilburg. 1991 Dumer.
1991 Coffey–Goodman–Farrell.
1993 Chabanne–Courteau.
1993 Chabaud.

1994 van Tilburg.
1994 Canteaut–Ch
1998 Canteaut–Ch
1998 Canteaut–Se
2008 Bernstein–La
2009 Bernstein–La
Peters–van Tilborg
2009 Bernstein (p
2009 Finiasz–Send
2010 Bernstein–La
2011 May–Meurer
2011 Becker–Coro
2012 Becker–Joux
2013 Bernstein–Je
Meurer (post-quar

The track record

1978 McEliece proposed public-key code-based crypto.

Has held up well after extensive optimization of attack algorithms:

1962 Prange. 1981 Omura.
1988 Lee–Brickell. 1988 Leon.
1989 Krouk. 1989 Stern.
1989 Dumer.
1990 Coffey–Goodman.
1990 van Tilburg. 1991 Dumer.
1991 Coffey–Goodman–Farrell.
1993 Chabanne–Courteau.
1993 Chabaud.
1994 van Tilburg.
1994 Canteaut–Chabanne.
1998 Canteaut–Chabaud.
1998 Canteaut–Sendrier.
2008 Bernstein–Lange–Peter
2009 Bernstein–Lange–Peters–van Tilborg.
2009 Bernstein (post-quantu
2009 Finiasz–Sendrier.
2010 Bernstein–Lange–Peter
2011 May–Meurer–Thomae.
2011 Becker–Coron–Joux.
2012 Becker–Joux–May–Me
2013 Bernstein–Jeffery–Lang
Meurer (post-quantum).

The track record

1978 McEliece proposed public-key code-based crypto.

Has held up well after extensive optimization of attack algorithms:

1962 Prange. 1981 Omura.
1988 Lee–Brickell. 1988 Leon.
1989 Krouk. 1989 Stern.
1989 Dumer.
1990 Coffey–Goodman.
1990 van Tilburg. 1991 Dumer.
1991 Coffey–Goodman–Farrell.
1993 Chabanne–Courteau.
1993 Chabaud.
1994 van Tilburg.
1994 Canteaut–Chabanne.
1998 Canteaut–Chabaud.
1998 Canteaut–Sendrier.
2008 Bernstein–Lange–Peters.
2009 Bernstein–Lange–Peters–van Tilborg.
2009 Bernstein (post-quantum).
2009 Finiasz–Sendrier.
2010 Bernstein–Lange–Peters.
2011 May–Meurer–Thomae.
2011 Becker–Coron–Joux.
2012 Becker–Joux–May–Meurer.
2013 Bernstein–Jeffery–Lange–Meurer (post-quantum).

ck record

cEliece proposed
ey code-based crypto.

d up well after extensive
ntion of attack algorithms:
ange. 1981 Omura.
e–Brickell. 1988 Leon.
ouk. 1989 Stern.
umer.
offey–Goodman.
n Tilburg. 1991 Dumer.
offey–Goodman–Farrell.
abanne–Courteau.
abaud.

1994 van Tilburg.
1994 Canteaut–Chabanne.
1998 Canteaut–Chabaud.
1998 Canteaut–Sendrier.
2008 Bernstein–Lange–Peters.
2009 Bernstein–Lange–Peters–van Tilborg.
2009 Bernstein (post-quantum).
2009 Finiasz–Sendrier.
2010 Bernstein–Lange–Peters.
2011 May–Meurer–Thomae.
2011 Becker–Coron–Joux.
2012 Becker–Joux–May–Meurer.
2013 Bernstein–Jeffery–Lange–Meurer (post-quantum).

Example

Some cy
(Intel Co
from ber

`mcelie`
(2008 B
`gls254`
(binary e
`kumfp12`
(hyperel
`curve25`
(conserv
`mcelie`
`ronald1`

posed
ased crypto.

fter extensive
tack algorithms:
1 Omura.
1988 Leon.
Stern.

lman.
1991 Dumer.
lman–Farrell.
ourteau.

1994 van Tilburg.
1994 Canteaut–Chabanne.
1998 Canteaut–Chabaud.
1998 Canteaut–Sendrier.
2008 Bernstein–Lange–Peters.
2009 Bernstein–Lange–Peters–van Tilborg.
2009 Bernstein (post-quantum).
2009 Finiasz–Sendrier.
2010 Bernstein–Lange–Peters.
2011 May–Meurer–Thomae.
2011 Becker–Coron–Joux.
2012 Becker–Joux–May–Meurer.
2013 Bernstein–Jeffery–Lange–Meurer (post-quantum).

Examples of the c

Some cycle counts
(Intel Core i5-321
from bench.cr.y

`mceliece` encrypt
(2008 Biswas–Sen
`gls254` DH
(binary elliptic cu
`kumfp127g` DH
(hyperelliptic; Eur
`curve25519` DH
(conservative ellip
`mceliece` **decryp**
`ronald1024` decry

o.

sive

ithms:

on.

mer.

ell.

1994 van Tilburg.

1994 Canteaut–Chabanne.

1998 Canteaut–Chabaud.

1998 Canteaut–Sendrier.

2008 Bernstein–Lange–Peters.

2009 Bernstein–Lange–Peters–van Tilborg.

2009 Bernstein (post-quantum).

2009 Finiasz–Sendrier.

2010 Bernstein–Lange–Peters.

2011 May–Meurer–Thomae.

2011 Becker–Coron–Joux.

2012 Becker–Joux–May–Meurer.

2013 Bernstein–Jeffery–Lange–Meurer (post-quantum).

Examples of the competitio

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

`mceliece` encrypt
(2008 Biswas–Sendrier, ≈2^{8}
`gls254` DH
(binary elliptic curve; CHES
`kumfp127g` DH 1
(hyperelliptic; Eurocrypt 201
`curve25519` DH 1
(conservative elliptic curve)
`mceliece` **decrypt** 12
`ronald1024` decrypt 13

1994 van Tilburg.
1994 Canteaut–Chabanne.
1998 Canteaut–Chabaud.
1998 Canteaut–Sendrier.
2008 Bernstein–Lange–Peters.
2009 Bernstein–Lange–Peters–van Tilborg.
2009 Bernstein (post-quantum).
2009 Finiasz–Sendrier.
2010 Bernstein–Lange–Peters.
2011 May–Meurer–Thomae.
2011 Becker–Coron–Joux.
2012 Becker–Joux–May–Meurer.
2013 Bernstein–Jeffery–Lange–Meurer (post-quantum).

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, $\approx 2^{80}$)	
`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

n Tilburg.
nteaut–Chabanne.
nteaut–Chabaud.
nteaut–Sendrier.
rnstein–Lange–Peters.
rnstein–Lange–
van Tilborg.
rnstein (post-quantum).
niasz–Sendrier.
rnstein–Lange–Peters.
ay–Meurer–Thomae.
cker–Coron–Joux.
cker–Joux–May–Meurer.
rnstein–Jeffery–Lange–
(post-quantum).

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, $\approx 2^{80}$)	
`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 dec

$\approx 2^{128}$ se

nabanne.
nabaud.
endrier.
ange–Peters.
ange–
g.
ost-quantum).
drier.
ange–Peters.
–Thomae.
n–Joux.
–May–Meurer.
effery–Lange–
ntum).

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, $\approx 2^{80}$)	
`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 spe

$\approx 2^{128}$ security (n,

rs.

um).

rs.

urer.

ge–

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, $\approx 2^{80}$)	
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

$\approx 2^{128}$ security $(n, t) = (409$

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, $\approx 2^{80}$)	
`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

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

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, $\approx 2^{80}$)	
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

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

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

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, $\approx 2^{80}$)	
`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

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

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

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

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, $\approx 2^{80}$)	
`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

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

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

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

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

Similar improvements for CFS.

es of the competition

ycle counts on h9ivy
ore i5-3210M, Ivy Bridge)
nch.cr.yp.to:

ce encrypt	61440
iswas–Sendrier, $\approx 2^{80}$)	
DH	77468
elliptic curve; CHES 2013)	
27g DH	116944
liptic; Eurocrypt 2013)	
5519 DH	182632
ative elliptic curve)	
ce **decrypt**	1219344
1024 decrypt	1340040

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.)

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

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

Similar improvements for CFS.

Constan

The extr
to elimin
Handle a
using on
XOR (^)

ompetition

s on `h9ivy`
0M, Ivy Bridge)
`p.to`:

61440
drier, $\approx 2^{80}$)
77468
ve; CHES 2013)
116944
ocrypt 2013)
182632
tic curve)
t 1219344
ypt 1340040

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.)

$\approx 2^{80}$ security $(n, t) = (2048, 32)$:
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 d
using only bit oper
XOR (^), AND (&

n

y
ridge)

61440
0)
77468
2013)
16944
13)
82632

219344
340040

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.)

$\approx 2^{80}$ security $(n, t) = (2048, 32)$:
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.

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.)

$\approx 2^{80}$ security $(n,t)=(2048,32)$:
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.

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.)

$\approx 2^{80}$ security $(n,t) = (2048, 32)$:
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.

We take this approach.

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.)

$\approx 2^{80}$ security $(n, t) = (2048, 32)$:
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.

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

ecurity $(n, t) = (4096, 41)$:
vy Bridge cycles.
l focus on this case.

tion is slightly slower:
hash, cipher, MAC.)

curity $(n, t) = (2048, 32)$:
vy Bridge cycles.

/store addresses
branch conditions
ic. Eliminates
ming attacks etc.

mprovements for CFS.

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

Not as s
On a typ
the XOR
is actual
operatin
on vecto

eds

$t) = (4096, 41)$:
cycles.
this case.

htly slower:
her, MAC.)

$t) = (2048, 32)$:
cycles.

resses
nditions
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 (ˆ), 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-bi
the XOR instructio
is actually 32-bit X
operating in parall
on vectors of 32 b

6, 41):

r:

)

, 32):

FS.

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.

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.

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.

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

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

t-time fanaticism

remist's approach
nate timing attacks:
all secret data
ly bit operations—
), AND (&), etc.

this approach.

an this be
tive in speed?
really simulating
ltiplication with
s of bit operations
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.

Not imm
that this
saves ti
multiplic

aticism

pproach
g attacks:
data
rations—
), etc.

oach.

ed?
ulating
n with
erations
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!
pical 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 $f = c$

For each
compute
41 adds,

sounds!
t CPU,
on
XOR,
el
its.

one CPU:
y cycle.

y 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 $n = 4096 = 2$

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

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

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_ig^i, c_ig^{2i}, c_ig^{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_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.

nediately obvious
s "bitslicing"
me for, e.g.,
cation in $\mathbf{F}_{2^{12}}$.

e obvious that it
me for addition in $\mathbf{F}_{2^{12}}$.

decoding algorithms
d, mult roughly balanced.

next: how to save
lds and *most* mults.
ergy 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_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.

Asympto
normally
so Horne
$\Theta(nt) =$

bvious
g"
.,
$_{2^{12}}$.
that it
ition in $\mathbf{F}_{2^{12}}$.
lgorithms
ughly balanced.
to save
ost mults.
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_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 $t \in \Theta(n/$
so Horner's rule co
$\Theta(nt) = \Theta(n^2/\lg$

2^{12}.

nced.

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.

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

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.

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 $n^2/\lg n$?

itive FFT

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

decoding step
d all roots in $\mathbf{F}_{2^{12}}$
$c_{41}x^{41} + \cdots + c_0x^0$.

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

Chien search: compute
g^{2i}, c_ig^{3i}, etc. Cost per
gain 41 adds, 41 mults.

t: **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 $\mathbf{F}_{2^{12}}$

$\cdots + c_0 x^0$.

,

Horner's rule:

.

ch: compute

etc. Cost per

dds, 41 mults.

lds, $\mathbf{2.09}$ 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 F

Want to evaluate

$f = c_0 + c_1 x + \cdots$

at all the nth root

Write f as $f_0(x^2)$

Observe big overla

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

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

f_0 has $n/2$ coeffs;

evaluate at $(n/2)$n

by same idea recur

Similarly f_1.

.

ule:

ute

per

ults.

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
$f = c_0 + c_1 x + \cdots + c_{n-1} x$
at all the nth roots of 1.

Write f as $f_0(x^2) + x f_1(x^2$
Observe big overlap betweer
$f(\alpha) = f_0(\alpha^2) + \alpha f_1(\alpha^2)$,
$f(-\alpha) = f_0(\alpha^2) - \alpha f_1(\alpha^2)$

f_0 has $n/2$ coeffs;
evaluate at $(n/2)$nd roots o
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 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.

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
$f = c_0 + c_1 x + \cdots + c_{n-1} x^{n-1}$
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 i
Standard
FFT con

1988 Wa
independ
"additiv
Still quit

1996 vo
some im

2010 Ga
much be

We use
plus som

/ lg n),
osts
n).

school
es
mial

rations?
han $n^2/\lg n$?

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:
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

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 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

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.

d radix-2 FFT:

evaluate

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

e nth roots of 1.

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

Their ma

$f_0(x^2 +$

Big over

$f_0(\alpha^2 +$

and $f(\alpha$

$f_0(\alpha^2 +$

"Twist"

Then $\{\alpha$

size-$(n/$

Apply sa

FFT:

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

ts of 1.

$+ xf_1(x^2)$.

ap between

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

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

nd roots of 1

rsively.

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

$f = c_0 + c_1x + \cdot\cdot$

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

Their main idea: V

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

Big overlap betwe

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

and $f(\alpha + 1) =$

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

"Twist" to ensure

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

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

Apply same idea r

$n-1$

).

n

.

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
$f = c_0 + c_1 x + \cdots + c_{n-1} x$
on a size-n $\mathbf{F}_2$-linear space.

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

“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
$f = c_0 + c_1 x + \cdots + c_{n-1} x^{n-1}$
on a size-n $\mathbf{F}_2$-linear space.

Their 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.

in char 2: $\alpha = -\alpha$.
d workarounds are painful.
nsidered impractical.

ang–Zhu,
dently 1989 Cantor:
e FFT" in char 2.
te expensive.

n zur Gathen–Gerhard:
provements.

o–Mateer:
etter additive FFT.

Gao–Mateer,
he 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.

Their 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 gene
$f = c_0 +$
for any t

$\Rightarrow$ sever
not all o
by simpl

For $t =$

For $t \in$
f_1 is a c
Instead
this cons
multiply
and com

$\alpha = -\alpha$.
unds are painful.
npractical.

9 Cantor:
char 2.
ve.

hen–Gerhard:
ts.

:
ive FFT.

eer,
provements.

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.

Their 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
$f = c_0 + c_1 x + \cdot\cdot$
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.

rd:

ts.

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.

Their 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 automat
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.

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.

Their 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.

Mateer evaluate

$+ c_1 x + \cdots + c_{n-1} x^{n-1}$

e-n $\mathbf{F}_2$-linear space.

ain idea: Write f as

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

rlap between $f(\alpha) =$

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

$+ 1) =$

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

to ensure $1 \in$ space.

$\alpha^2 + \alpha\}$ is a

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

ame 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.

Syndrom

Initial de

$s_0 = r_1$

$s_1 = r_1 c$

$s_2 = r_1 c$

$\vdots$,

$s_t = r_1 c$

$r_1, r_2, \ldots$

scaled by

Typically

mapping

Not as s

still n^{2+}

valuate
$\cdot + c_{n-1}x^{n-1}$
ear space.

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

en $f(\alpha) =$
$(\alpha^2 + \alpha)$

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

$1 \in$ space.
a
ar space.
ecursively.

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 comput

Initial decoding ste
$s_0 = r_1 + r_2 + \cdots$
$s_1 = r_1\alpha_1 + r_2\alpha_2$
$s_2 = r_1\alpha_1^2 + r_2\alpha_2^2$
$\vdots$,
$s_t = r_1\alpha_1^t + r_2\alpha_2^t$

$r_1, r_2, \ldots, r_n$ are
scaled by Goppa c
Typically precomp
mapping bits to sy
Not as slow as Ch
still $n^{2+o(1)}$ and h

$n-1$

s

=

$+ \alpha)$.

e.

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: compu
$s_0 = r_1 + r_2 + \cdots + r_n$,
$s_1 = r_1\alpha_1 + r_2\alpha_2 + \cdots + r$
$s_2 = r_1\alpha_1^2 + r_2\alpha_2^2 + \cdots + r$
$\vdots$,
$s_t = r_1\alpha_1^t + r_2\alpha_2^t + \cdots + r$

$r_1, r_2, \ldots, r_n$ are received b
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
$f = c_0 + c_1 x + \cdots + c_t x^t$
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.

eralize to

$+ c_1 x + \cdots + c_t x^t$

$t < n$.

al optimizations,
f which are automated
y tracking zeros.

0: copy c_0.

$\{1, 2\}$:
constant.
of multiplying
stant by each α,
only by generators
npute 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

$f(\alpha_1) =$

$f(\alpha_2) =$

$\vdots$,

$f(\alpha_n) =$

$\cdot + c_t x^t$

ations,

re automated

zeros.

$_0$.

ying

ach α,

enerators

et 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 multi

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

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

$\vdots,$

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

ted

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 evalu

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

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

$\vdots,$

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

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:

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

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:

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

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

$\vdots,$

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

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

Syndrome computation

Initial decoding step: compute

$s_0 = r_1 + r_2 + \cdots + r_n$,
$s_1 = r_1\alpha_1 + r_2\alpha_2 + \cdots + r_n\alpha_n$,
$s_2 = r_1\alpha_1^2 + r_2\alpha_2^2 + \cdots + r_n\alpha_n^2$,
$\vdots$,
$s_t = r_1\alpha_1^t + r_2\alpha_2^t + \cdots + r_n\alpha_n^t$.

$r_1, r_2, \ldots, r_n$ 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:

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

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.

ne computation

ecoding step: compute

$+ r_2 + \cdots + r_n,$

$\alpha_1 + r_2\alpha_2 + \cdots + r_n\alpha_n,$

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

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

. ., r_n are received bits

y Goppa constants.

y precompute matrix

g bits to syndrome.

slow as Chien search but

$o(1)$ 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.

Transpo

If a linea

compute

then rev

exchang

compute

1956 Bo

independ

for Bool

1973 Fid

preserve

preserve

number

ation

ep: compute

$\cdot + r_n,$

$+ \cdots + r_n\alpha_n,$

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

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

received bits
onstants.
ute matrix
yndrome.
ien search but
uge 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.

Transposition prin
If a linear algorith
computes a matrix
then reversing edg
exchanging inputs
computes the tran

1956 Bordewijk;
independently 195
for Boolean matric

1973 Fiduccia ana
preserves number
preserves number
number of nontriv

ute

$_n\alpha_n$,

$_n\alpha_n^2$,

$_n\alpha_n^t$.

its

k

but

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.

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

1956 Bordewijk; independently 1957 Lupanov for Boolean matrices.

1973 Fiduccia analysis: preserves number of mults; preserves number of adds pl number of nontrivial output

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.

e to multipoint evaluation:

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

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

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

or syndrome computation
ose of
or multipoint evaluation.

g consequence:
e computation is as few
nultipoint evaluation.
e 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
producin
Too mar
gcc ran

point evaluation:

$_1 + \cdots + c_t \alpha_1^t$,

$_2 + \cdots + c_t \alpha_2^t$,

$_n + \cdots + c_t \alpha_n^t$.

me computation

oint 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 compil
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.

sition principle:
ar 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 a
wrote d
with sa

Small c

Speedu
translat
to trans

Further
merged
scaling

ciple:
m
k M
ges and
/outputs
spose 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 ove

Speedups of addit
translate easily
to transposed algo

Further savings:
merged first stage
scaling by Goppa

M.

y

us

s.

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.

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.

t transposing compiler
ng C code.
ny variables for $m = 13$;
out of memory.

asm register allocator
nize the variables.
but not very quickly.

aster register allocator.
essive code size.

w interpreter,
some code compression.
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 p

Additive
field ele

This is r
needed i
Must ap
part of t

Same iss

Solution
Almost c
Beneš ne

ng compiler

es for $m = 13$;
emory.

er allocator
riables.
ery 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 permutatio

Additive FFT $\Rightarrow$ f
field elements *in a*

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

Same issue for syn

Solution: Batcher
Almost done with
Beneš network.

er

= 13;

or

y.

or.

ssion.

.

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 ⇒ f values at
field elements *in a standard*

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

Same issue for syndrome.

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

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.

olution:
t additive FFT,
own transposition
ne loops etc.

ode, no overhead.

os of additive FFT
e easily
posed algorithm.

savings:
first stage with
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 Iv

8622 fo
20846 fo
7714 fo
14794 fo
8520 fo

Code wi
We're st

Also 10×

More inf
cr.yp.t

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

Also 10× speedup

More information:
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: 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.

Also 10× speedup for CFS.

More information:

`cr.yp.to/papers.html#mc`

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.

Also $10\times$ speedup for CFS.

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