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*.

stant-time
sed cryptography

ear at CHES 2013)

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 com

bench.c

CPU cyc
(Intel Co
to encry

46940
61440
94464
398912

mceliec
$(n, t) =$
from Bis
See pape

graphy

ES 2013)

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 competition

bench.cr.yp.to:

CPU cycles on h9
(Intel Core i5-3210
to encrypt 59 byte

46940 ronald10
61440 mceliece
94464 ronald20
398912 ntruees7

mceliece:
$(n, t) = (2048, 32$
from Biswas and S
See paper at PQC

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 competition

`bench.cr.yp.to`:

CPU cycles on `h9ivy`
(Intel Core i5-3210M, Ivy Br
to encrypt 59 bytes:

46940 `ronald1024` (RSA-
61440 `mceliece`
94464 `ronald2048`
398912 `ntruees787ep1`

`mceliece`:
$(n, t) = (2048, 32)$ software
from Biswas and Sendrier.
See paper at PQCrypto 200

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 competition

`bench.cr.yp.to`:

CPU cycles on `h9ivy`
(Intel Core i5-3210M, Ivy Bridge)
to encrypt 59 bytes:

46940	`ronald1024` (RSA-1024)
61440	`mceliece`
94464	`ronald2048`
398912	`ntruees787ep1`

`mceliece`:
$(n, t) = (2048, 32)$ software from Biswas and Sendrier.
See paper at PQCrypto 2008.

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*.

The competition

bench.cr.yp.to:

CPU cycles on `h9ivy`
(Intel Core i5-3210M, Ivy Bridge)
to encrypt 59 bytes:

46940 `ronald1024` (RSA-1024)
61440 `mceliece`
94464 `ronald2048`
398912 `ntruees787ep1`

`mceliece`:
$(n, t) = (2048, 32)$ software
from Biswas and Sendrier.
See paper at PQCrypto 2008.

Sounds
What's

ords
tography.

rity level.

ection
computers.

protection
ng attacks,
attacks, etc.

sed crypto
record.

e *at once*.

The competition

bench.cr.yp.to:

CPU cycles on `h9ivy`
(Intel Core i5-3210M, Ivy Bridge)
to encrypt 59 bytes:

46940	`ronald1024` (RSA-1024)
61440	`mceliece`
94464	`ronald2048`
398912	`ntruees787ep1`

`mceliece`:
$(n, t) = (2048, 32)$ software
from Biswas and Sendrier.
See paper at PQCrypto 2008.

Sounds reasonably
What's the proble

The competition

bench.cr.yp.to:

CPU cycles on h9ivy
(Intel Core i5-3210M, Ivy Bridge)
to encrypt 59 bytes:

```
 46940 ronald1024 (RSA-1024)
 61440 mceliece
 94464 ronald2048
398912 ntruees787ep1
```

mceliece:
$(n, t) = (2048, 32)$ software
from Biswas and Sendrier.
See paper at PQCrypto 2008.

Sounds reasonably fast.
What's the problem?

The competition

bench.cr.yp.to:

CPU cycles on `h9ivy`
(Intel Core i5-3210M, Ivy Bridge)
to encrypt 59 bytes:

46940	`ronald1024` (RSA-1024)
61440	`mceliece`
94464	`ronald2048`
398912	`ntruees787ep1`

`mceliece`:
$(n, t) = (2048, 32)$ software
from Biswas and Sendrier.
See paper at PQCrypto 2008.

Sounds reasonably fast.
What's the problem?

The competition

bench.cr.yp.to:

CPU cycles on `h9ivy` (Intel Core i5-3210M, Ivy Bridge) to encrypt 59 bytes:

46940	`ronald1024` (RSA-1024)
61440	`mceliece`
94464	`ronald2048`
398912	`ntruees787ep1`

`mceliece`: $(n, t) = (2048, 32)$ software from Biswas and Sendrier. See paper at PQCrypto 2008.

Sounds reasonably fast. What's the problem?

Decryption is much slower:

700512	`ntruees787ep1`
1219344	`mceliece`
1340040	`ronald1024`
5766752	`ronald2048`

The competition

bench.cr.yp.to:

CPU cycles on `h9ivy`
(Intel Core i5-3210M, Ivy Bridge)
to encrypt 59 bytes:

46940 `ronald1024` (RSA-1024)
61440 `mceliece`
94464 `ronald2048`
398912 `ntruees787ep1`

`mceliece`:
$(n, t) = (2048, 32)$ software
from Biswas and Sendrier.
See paper at PQCrypto 2008.

Sounds reasonably fast.
What's the problem?

Decryption is much slower:

700512 `ntruees787ep1`
1219344 `mceliece`
1340040 `ronald1024`
5766752 `ronald2048`

But Biswas and Sendrier
say they're faster now,
even beating NTRU.
What's the problem?

npetition

cr.yp.to:

cles on h9ivy
ore i5-3210M, Ivy Bridge)
pt 59 bytes:

ronald1024 (RSA-1024)
mceliece
ronald2048
ntruees787ep1

ce:
(2048, 32) software
swas and Sendrier.
er at PQCrypto 2008.

Sounds reasonably fast.
What's the problem?

Decryption is much slower:

```
 700512 ntruees787ep1
1219344 mceliece
1340040 ronald1024
5766752 ronald2048
```

But Biswas and Sendrier
say they're faster now,
even beating NTRU.
What's the problem?

The seri

Some D

bench.c

77468
(binary e
116944
(hyperel
182632
(conserv

Use DH
Decrypti
Encrypti
+ key-ge

ivy
0M, Ivy Bridge)
es:

024 (RSA-1024)
e
048
787ep1

) software
Sendrier.
rypto 2008.

Sounds reasonably fast.
What's the problem?

Decryption is much slower:

700512	ntruees787ep1
1219344	mceliece
1340040	ronald1024
5766752	ronald2048

But Biswas and Sendrier
say they're faster now,
even beating NTRU.
What's the problem?

The serious compe

Some Diffie–Hellm

bench.cr.yp.to:

77468 gls254
(binary elliptic cur
116944 kumfp127
(hyperelliptic; Euro
182632 curve255
(conservative ellipt

Use DH for public
Decryption time ≈
Encryption time ≈
+ key-generation

ridge)

1024)

8.

Sounds reasonably fast.
What's the problem?

Decryption is much slower:

```
 700512 ntruees787ep1
1219344 mceliece
1340040 ronald1024
5766752 ronald2048
```

But Biswas and Sendrier say they're faster now, even beating NTRU. What's the problem?

The serious competition

Some Diffie–Hellman speeds

bench.cr.yp.to:

```
 77468 gls254
```
(binary elliptic curve; CHES
```
116944 kumfp127g
```
(hyperelliptic; Eurocrypt 201
```
182632 curve25519
```
(conservative elliptic curve)

Use DH for public-key encry
Decryption time $\approx$ DH time
Encryption time $\approx$ DH time
$+$ key-generation time.

Sounds reasonably fast.
What's the problem?

Decryption is much slower:

```
 700512 ntruees787ep1
1219344 mceliece
1340040 ronald1024
5766752 ronald2048
```

But Biswas and Sendrier say they're faster now, even beating NTRU. What's the problem?

The serious competition

Some Diffie–Hellman speeds from `bench.cr.yp.to`:

`77468 gls254`
(binary elliptic curve; CHES 2013)
`116944 kumfp127g`
(hyperelliptic; Eurocrypt 2013)
`182632 curve25519`
(conservative elliptic curve)

Use DH for public-key encryption.
Decryption time $\approx$ DH time.
Encryption time $\approx$ DH time + key-generation time.

reasonably fast.
the problem?

ion is much slower:

2 `ntruees787ep1`
4 `mceliece`
0 `ronald1024`
2 `ronald2048`

was and Sendrier
're faster now,
ating NTRU.
the problem?

The serious competition

Some Diffie–Hellman speeds from `bench.cr.yp.to`:

`77468 gls254`
(binary elliptic curve; CHES 2013)
`116944 kumfp127g`
(hyperelliptic; Eurocrypt 2013)
`182632 curve25519`
(conservative elliptic curve)

Use DH for public-key encryption.
Decryption time ≈ DH time.
Encryption time ≈ DH time
+ key-generation time.

Elliptic/
fast encr

(Also sig
key exch
let's foc
Also sho
let's foc

`kumfp12`
protect a
branch-p

Broken
but high
for the s

y fast.
m?

h slower:

3787ep1
ce
1024
2048

endrier
now,
RU.
m?

The serious competition

Some Diffie–Hellman speeds from `bench.cr.yp.to`:

`77468 gls254`
(binary elliptic curve; CHES 2013)
`116944 kumfp127g`
(hyperelliptic; Eurocrypt 2013)
`182632 curve25519`
(conservative elliptic curve)

Use DH for public-key encryption.
Decryption time ≈ DH time.
Encryption time ≈ DH time
+ key-generation time.

Elliptic/hyperellipt
fast encryption *an*

(Also signatures, r
key exchange, mor
let's focus on encr
Also short keys etc
let's focus on spee

`kumfp127g` and cu
protect against tin
branch-prediction

Broken by quantu
but high security l
for the short term.

The serious competition

Some Diffie–Hellman speeds from bench.cr.yp.to:

`77468 gls254`
(binary elliptic curve; CHES 2013)
`116944 kumfp127g`
(hyperelliptic; Eurocrypt 2013)
`182632 curve25519`
(conservative elliptic curve)

Use DH for public-key encryption.
Decryption time ≈ DH time.
Encryption time ≈ DH time
+ key-generation time.

Elliptic/hyperelliptic curves c
fast encryption *and* decrypti

(Also signatures, non-interac
key exchange, more; but
let's focus on encrypt/decry
Also short keys etc.; but
let's focus on speed.)

`kumfp127g` and `curve2551`
protect against timing attac
branch-prediction attacks, et

Broken by quantum comput
but high security level
for the short term.

The serious competition

Some Diffie–Hellman speeds from
bench.cr.yp.to:

`77468 gls254`
(binary elliptic curve; CHES 2013)
`116944 kumfp127g`
(hyperelliptic; Eurocrypt 2013)
`182632 curve25519`
(conservative elliptic curve)

Use DH for public-key encryption.
Decryption time ≈ DH time.
Encryption time ≈ DH time
+ key-generation time.

Elliptic/hyperelliptic curves offer fast encryption *and* decryption.

(Also signatures, non-interactive key exchange, more; but let's focus on encrypt/decrypt. Also short keys etc.; but let's focus on speed.)

`kumfp127g` and `curve25519` protect against timing attacks, branch-prediction attacks, etc.

Broken by quantum computers, but high security level for the short term.

ous competition

iffie–Hellman speeds from
cr.yp.to:

gls254
elliptic curve; CHES 2013)
kumfp127g
liptic; Eurocrypt 2013)
curve25519
ative elliptic curve)

for public-key encryption.
ion time ≈ DH time.
on time ≈ DH time
eneration time.

Elliptic/hyperelliptic curves offer fast encryption *and* decryption.

(Also signatures, non-interactive key exchange, more; but let's focus on encrypt/decrypt. Also short keys etc.; but let's focus on speed.)

kumfp127g and curve25519 protect against timing attacks, branch-prediction attacks, etc.

Broken by quantum computers, but high security level for the short term.

New dec

$(n, t) =$

etition

han speeds from

ve; CHES 2013)
/g
ocrypt 2013)
519
tic curve)

-key encryption.
& DH time.
& DH time
time.

Elliptic/hyperelliptic curves offer fast encryption *and* decryption.

(Also signatures, non-interactive key exchange, more; but let's focus on encrypt/decrypt. Also short keys etc.; but let's focus on speed.)

`kumfp127g` and `curve25519` protect against timing attacks, branch-prediction attacks, etc.

Broken by quantum computers, but high security level for the short term.

New decoding spe

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

s from

2013)

13)

ption.

.

Elliptic/hyperelliptic curves offer fast encryption *and* decryption.

(Also signatures, non-interactive key exchange, more; but let's focus on encrypt/decrypt. Also short keys etc.; but let's focus on speed.)

`kumfp127g` and `curve25519` protect against timing attacks, branch-prediction attacks, etc.

Broken by quantum computers, but high security level for the short term.

New decoding speeds

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

Elliptic/hyperelliptic curves offer fast encryption *and* decryption.

(Also signatures, non-interactive key exchange, more; but let's focus on encrypt/decrypt. Also short keys etc.; but let's focus on speed.)

`kumfp127g` and `curve25519` protect against timing attacks, branch-prediction attacks, etc.

Broken by quantum computers, but high security level for the short term.

New decoding speeds

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

Elliptic/hyperelliptic curves offer fast encryption *and* decryption.

(Also signatures, non-interactive key exchange, more; but let's focus on encrypt/decrypt. Also short keys etc.; but let's focus on speed.)

`kumfp127g` and `curve25519` protect against timing attacks, branch-prediction attacks, etc.

Broken by quantum computers, but high security level for the short term.

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.)

Elliptic/hyperelliptic curves offer fast encryption *and* decryption.

(Also signatures, non-interactive key exchange, more; but let's focus on encrypt/decrypt. Also short keys etc.; but let's focus on speed.)

`kumfp127g` and `curve25519` protect against timing attacks, branch-prediction attacks, etc.

Broken by quantum computers, but high security level for the short term.

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.

Elliptic/hyperelliptic curves offer fast encryption *and* decryption.

(Also signatures, non-interactive key exchange, more; but let's focus on encrypt/decrypt. Also short keys etc.; but let's focus on speed.)

`kumfp127g` and `curve25519` protect against timing attacks, branch-prediction attacks, etc.

Broken by quantum computers, but high security level for the short term.

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.

hyperelliptic curves offer
ryption *and* decryption.

gnatures, non-interactive
ange, more; but
us on encrypt/decrypt.
rt keys etc.; but
us on speed.)

27g and `curve25519`
against timing attacks,
rediction attacks, etc.

by quantum computers,
security level
hort term.

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.

Constan

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

tic curves offer
d decryption.

non-interactive
re; but
rypt/decrypt.
c.; but
ed.)

`urve25519`
ning attacks,
attacks, etc.

m computers,
evel
.

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 d
using only bit oper
XOR (^), AND (&

offer
on.
ctive
pt.
9
ks,
tc.
ers,

New decoding speeds

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

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

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

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

Similar improvements for CFS.

Constant-time fanaticism

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

New decoding speeds

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

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

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

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

Similar improvements for CFS.

Constant-time fanaticism

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

New decoding speeds

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

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

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

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

Similar improvements for CFS.

Constant-time fanaticism

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

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 (^), 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.

mprovements 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

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
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 (^), 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 instructi
is actually 32-bit X
operating in parall
on vectors of 32 b

urity:

r:

)

rity:

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 tir
multiplic

aticism

pproach
g attacks:
data
rations—
), etc.

oach.

ed?
ulating
n with
perations
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 $n = 4096 = 2^{1}$

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 $f = c_{41}x^{41} + \cdots + c_0 x^0$.

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.

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

te 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.
nergy 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"
.,
$\mathbf{F}_{2^{12}}$.

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 $f = c_{41}x^{41} + \cdots + c_0 x^0$.

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

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}, $c_i g^{3i}$, 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, **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 $(n/2)$r
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
$_{1+o(1)}$ 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
"additive
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

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

e nth roots of 1.

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

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 $\{\alpha$

size-$(n/$

Apply sa

FFT:

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

ts of 1.

$+ x f_1(x^2)$.

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

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

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

Main idea: Write

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

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.

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.

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.

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 o
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.

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
$f = c_0 + c_1 x + \cdots$
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.

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.

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.

ea: 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

$s_0 = r_1$

$s_1 = r_1 \alpha$

$s_2 = r_1 \alpha$

$\vdots$,

$s_t = r_1 \alpha$

$r_1, r_2, \ldots$

scaled by

Typically

mapping

Not as s

still n^{2+}

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

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
$f = c_0 + c_1x + \cdots + c_tx^t$
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

$n-1$

$=$

$+ \alpha)$.

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.

<u>Syndrome computation</u>

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,
of 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

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

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

$\vdots,$

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

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

$+ r_2 + \cdots + r_n,$

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

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

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

$\ldots, r_n$ 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

preserves

preserves

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 M

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 san

Small c

Speedup
translat
to trans

Further
merged
scaling b

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 ov

Speedups of addit
translate easily
to transposed algo

Further savings:
merged first stage
scaling by Goppa

M.

/

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 n
needed i
Must ap
part of t

Same iss

Solution
Almost c
Beneš n

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 ⇒ 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 $\Rightarrow$ 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

More inf
paper or

FFT,
position
tc.

erhead.

ive FFT

orithm.

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:
paper online very s

Secret permutation

Additive FFT ⇒ f values at field elements *in a standard order*.

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

Same issue for syndrome.

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

Results

60493 Ivy Bridge cycles:

8622 for permutation.
20846 for syndrome.
7714 for BM.
14794 for roots.
8520 for permutation.

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

More information: paper online very soon.

Secret permutation

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

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

Same issue for syndrome.

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

Results

60493 Ivy Bridge cycles:

8622 for permutation.
20846 for syndrome.
7714 for BM.
14794 for roots.
8520 for permutation.

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

More information: paper online very soon.