Timing attacks

1970s: TENEX operating system compares user-supplied string against secret password one character at a time, stopping at first difference:

- AAAAAA vs. SECRET: stop at 1.
- SAAAAA vs. SECRET: stop at 2.
- SEAAAA vs. SECRET: stop at 3.

Attacker sees comparison time, deduces position of difference. A few hundred tries reveal secret password.

How typical software checks 16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow from secrets to timings:

```
uint32 diff = 0;
for (i = 0;i < 16;++i)
  diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language makes the wrong thing simple and the right thing complex.

attacks

TENEX operating system
es user-supplied string
secret password
racter at a time,
g at first difference:

A vs. SECRET: stop at 1.
A vs. SECRET: stop at 2.
A vs. SECRET: stop at 3.

r sees comparison time,
position of difference.
undred tries
ecret password.

How typical software checks
16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow
from secrets to timings:

```
uint32 diff = 0;
for (i = 0;i < 16;++i)
  diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language
makes the wrong thing simple
and the right thing complex.

Languag
“right” i

So mista

erating system
plied string
sword
time,
ifference:

RET: stop at 1.
RET: stop at 2.
RET: stop at 3.

parison time,
of difference.
es
word.

How typical software checks
16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow
from secrets to timings:

```
uint32 diff = 0;
for (i = 0;i < 16;++i)
  diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language
makes the wrong thing simple
and the right thing complex.

Language designer
“right” is too wea

So mistakes contin

ystem
g

at 1.
at 2.
at 3.

me,
ce.

How typical software checks 16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow from secrets to timings:

```
uint32 diff = 0;
for (i = 0;i < 16;++i)
  diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language makes the wrong thing simple and the right thing complex.

Language designer's notion
"right" is too weak for secu

So mistakes continue to hap

How typical software checks
16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow
from secrets to timings:

```
uint32 diff = 0;
for (i = 0;i < 16;++i)
  diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language
makes the wrong thing simple
and the right thing complex.

Language designer's notion of
"right" is too weak for security.

So mistakes continue to happen.

How typical software checks
16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow
from secrets to timings:

```
uint32 diff = 0;
for (i = 0;i < 16;++i)
  diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language
makes the wrong thing simple
and the right thing complex.

Language designer's notion of
"right" is too weak for security.

So mistakes continue to happen.

One of many current examples,
part of the reference software for
CAESAR candidate CLOC:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```

ical software checks
authenticator:

```
(i = 0;i < 16;++i)
 (x[i] != y[i]) return 0;
rn 1;
```

inating information flow
rets to timings:

```
32 diff = 0;
(i = 0;i < 16;++i)
ff |= x[i] ^ y[i];
rn 1 & ((diff-1) >> 8);
```

hat the language
ne wrong thing simple
right thing complex.

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many current examples, part of the reference software for CAESAR candidate CLOC:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```


Do timin

Objectio

are checks
ator:

```
< 16;++i)
 y[i]) return 0;
```

formation flow
nings:

```
 0;
< 16;++i)
] ^ y[i];
diff-1) >> 8);
```

nguage
thing simple
g complex.

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many current examples, part of the reference software for CAESAR candidate CLOC:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```

Do timing attacks

Objection: "Timir

turn 0;

flow

> 8);

le

.

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many current examples, part of the reference software for CAESAR candidate CLOC:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```


Do timing attacks really wor

Objection: "Timings are noi

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many current examples, part of the reference software for CAESAR candidate CLOC:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```


Do timing attacks really work?

Objection: "Timings are noisy!"

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many current examples, part of the reference software for CAESAR candidate CLOC:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```


Do timing attacks really work?

Objection: "Timings are noisy!"

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender must block *all* information flow.

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many current examples, part of the reference software for CAESAR candidate CLOC:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```


Do timing attacks really work?

Objection: "Timings are noisy!"

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender must block *all* information flow.

Answer #2: Attacker uses statistics to eliminate noise.

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many current examples, part of the reference software for CAESAR candidate CLOC:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```


Do timing attacks really work?

Objection: "Timings are noisy!"

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender must block *all* information flow.

Answer #2: Attacker uses statistics to eliminate noise.

Answer #3, what the 1970s attackers actually did:
Cross page boundary,
inducing page faults,
to amplify timing signal.

ge designer's notion of
is too weak for security.

akes continue to happen.

many current examples,
the reference software for
R candidate CLOC:

```
are the tag */

 0;i < CRYPTO_ABYTES;i++)
g[i] != c[(*mlen) + i]){
urn RETURN_TAG_NO_MATCH;

RETURN_SUCCESS;
```


Do timing attacks really work?

Objection: "Timings are noisy!"

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender
must block *all* information flow.

Answer #2: Attacker uses
statistics to eliminate noise.

Answer #3, what the
1970s attackers actually did:
Cross page boundary,
inducing page faults,
to amplify timing signal.

Example

2005 Tr
65ms to
used for

2013 Al
Thirteen
DTLS re
plaintext

2014 va
steals Bi
of 25 Op

2016 Ya
"CacheB
key via t

r's notion of
k for security.

nue to happen.

ent examples,
ce software for
te CLOC:

```
ag */

YPTO_ABYTES;i++)
[(*mlen) + i]){
N_TAG_NO_MATCH;

CCESS;
```


Do timing attacks really work?

Objection: “Timings are noisy!”

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender
must block *all* information flow.

Answer #2: Attacker uses
statistics to eliminate noise.

Answer #3, what the
1970s attackers actually did:
Cross page boundary,
inducing page faults,
to amplify timing signal.

Examples of succe

2005 Tromer–Osvi
65ms to steal Linu
used for hard-disk

2013 AlFardan–Pa
Thirteen: breaking
DTLS record proto
plaintext using dec

2014 van de Pol–S
steals Bitcoin key
of 25 OpenSSL sig

2016 Yarom–Genk
“CacheBleed” stea
key via timings of

of

rity.

pen.

les,

re for

```
ES;i++)
+ i]){
MATCH;
```

Do timing attacks really work?

Objection: "Timings are noisy!"

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender must block *all* information flow.

Answer #2: Attacker uses statistics to eliminate noise.

Answer #3, what the 1970s attackers actually did:
Cross page boundary,
inducing page faults,
to amplify timing signal.

Examples of successful attac

2005 Tromer–Osvik–Shamir:
65ms to steal Linux AES key
used for hard-disk encryption

2013 AlFardan–Paterson "Lu
Thirteen: breaking the TLS
DTLS record protocols" stea
plaintext using decryption ti

2014 van de Pol–Smart–Yar
steals Bitcoin key from timin
of 25 OpenSSL signatures.

2016 Yarom–Genkin–Hening
"CacheBleed" steals RSA se
key via timings of OpenSSL.

Do timing attacks really work?

Objection: "Timings are noisy!"

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender must block *all* information flow.

Answer #2: Attacker uses statistics to eliminate noise.

Answer #3, what the 1970s attackers actually did:
Cross page boundary,
inducing page faults,
to amplify timing signal.

Examples of successful attacks:

2005 Tromer–Osvik–Shamir: 65ms to steal Linux AES key used for hard-disk encryption.

2013 AlFardan–Paterson "Lucky Thirteen: breaking the TLS and DTLS record protocols" steals plaintext using decryption timings.

2014 van de Pol–Smart–Yarom steals Bitcoin key from timings of 25 OpenSSL signatures.

2016 Yarom–Genkin–Heninger "CacheBleed" steals RSA secret key via timings of OpenSSL.

ng attacks really work?

on: "Timings are noisy!"

#1:
ise stop *all* attacks?
antee security, defender
ock *all* information flow.

#2: Attacker uses
s to eliminate noise.

#3, what the
ttackers actually did:
ge boundary,
page faults,
fy timing signal.

Examples of successful attacks:

2005 Tromer–Osvik–Shamir: 65ms to steal Linux AES key used for hard-disk encryption.

2013 AlFardan–Paterson "Lucky Thirteen: breaking the TLS and DTLS record protocols" steals plaintext using decryption timings.

2014 van de Pol–Smart–Yarom steals Bitcoin key from timings of 25 OpenSSL signatures.

2016 Yarom–Genkin–Heninger "CacheBleed" steals RSA secret key via timings of OpenSSL.

Constan

ECDH c
where a

Key gen

Signing:

All of th
Does tin

Are ther
ECC ops
Do the u
take vari

really work?

ngs are noisy!”

ll attacks?
rity, defender
ormation flow.

cker uses
ate noise.

the
ctually did:
ary,
ts,
signal.

Examples of successful attacks:

2005 Tromer–Osvik–Shamir: 65ms to steal Linux AES key used for hard-disk encryption.

2013 AlFardan–Paterson “Lucky Thirteen: breaking the TLS and DTLS record protocols” steals plaintext using decryption timings.

2014 van de Pol–Smart–Yarom steals Bitcoin key from timings of 25 OpenSSL signatures.

2016 Yarom–Genkin–Heninger “CacheBleed” steals RSA secret key via timings of OpenSSL.

Constant-time EC

ECDH computatio
where a is your se

Key generation: a

Signing: $r \mapsto rB$.

All of these use se
Does timing leak t

Are there any bran
ECC ops? Point o
Do the underlying
take variable time

rk?

sy!"

der

low.

:

Examples of successful attacks:

2005 Tromer–Osvik–Shamir: 65ms to steal Linux AES key used for hard-disk encryption.

2013 AlFardan–Paterson "Lucky Thirteen: breaking the TLS and DTLS record protocols" steals plaintext using decryption timings.

2014 van de Pol–Smart–Yarom steals Bitcoin key from timings of 25 OpenSSL signatures.

2016 Yarom–Genkin–Heninger "CacheBleed" steals RSA secret key via timings of OpenSSL.

Constant-time ECC

ECDH computation: $a, P \mapsto$

where a is your secret key.

Key generation: $a \mapsto aB$.

Signing: $r \mapsto rB$.

All of these use secret data.

Does timing leak this data?

Are there any branches in ECC ops? Point ops? Field

Do the underlying machine i

take variable time?

Examples of successful attacks:

2005 Tromer–Osvik–Shamir: 65ms to steal Linux AES key used for hard-disk encryption.

2013 AlFardan–Paterson "Lucky Thirteen: breaking the TLS and DTLS record protocols" steals plaintext using decryption timings.

2014 van de Pol–Smart–Yarom steals Bitcoin key from timings of 25 OpenSSL signatures.

2016 Yarom–Genkin–Heninger "CacheBleed" steals RSA secret key via timings of OpenSSL.

Constant-time ECC

ECDH computation: $a, P \mapsto aP$ *where a is your secret key.*

Key generation: $a \mapsto aB$.

Signing: $r \mapsto rB$.

All of these use secret data. Does timing leak this data?

Are there any branches in ECC ops? Point ops? Field ops? Do the underlying machine insns take variable time?

es of successful attacks:

omer–Osvik–Shamir:
steal Linux AES key
hard-disk encryption.

Fardan–Paterson "Lucky
n: breaking the TLS and
ecord protocols" steals
t using decryption timings.

n de Pol–Smart–Yarom
itcoin key from timings
penSSL signatures.

rom–Genkin–Heninger
Bleed" steals RSA secret
timings of OpenSSL.

Constant-time ECC

ECDH computation: $a, P \mapsto aP$ *where a is your secret key*.

Key generation: $a \mapsto aB$.

Signing: $r \mapsto rB$.

All of these use secret data.
Does timing leak this data?

Are there any branches in ECC ops? Point ops? Field ops? Do the underlying machine insns take variable time?

Recall le
to comp
using po

```
def sca
  if n
  if n
  R = s
  R = R
  if n %
  retur
```

Many br
NAF etc

ssful attacks:

ik–Shamir:
ux AES key
encryption.

terson "Lucky
g the TLS and
ocols" steals
cryption timings.

Smart–Yarom
from timings
gnatures.

in–Heninger
als RSA secret
OpenSSL.

Constant-time ECC

ECDH computation: $a, P \mapsto aP$ *where a is your secret key.*

Key generation: $a \mapsto aB$.

Signing: $r \mapsto rB$.

All of these use secret data.
Does timing leak this data?

Are there any branches in ECC ops? Point ops? Field ops? Do the underlying machine insns take variable time?

Recall left-to-right
to compute $n, P \mapsto$
using point additio

```
def scalarmult(n
  if n == 0: ret
  if n == 1: ret
  R = scalarmult
  R = R + R
  if n % 2: R =
  return R
```

Many branches he
NAF etc. also use

cks:

:

y

n.

ucky

and

als

mings.

om

ngs

er

ecret

.

Constant-time ECC

ECDH computation: $a, P \mapsto aP$
where a is your secret key.

Key generation: $a \mapsto aB$.

Signing: $r \mapsto rB$.

All of these use secret data.
Does timing leak this data?

Are there any branches in
ECC ops? Point ops? Field ops?
Do the underlying machine insns
take variable time?

Recall left-to-right binary m
to compute $n, P \mapsto nP$
using point addition:

```
def scalarmult(n,P):
  if n == 0: return 0
  if n == 1: return P
  R = scalarmult(n//2,P)
  R = R + R
  if n % 2: R = R + P
  return R
```

Many branches here.
NAF etc. also use many bra

Constant-time ECC

ECDH computation: $a, P \mapsto aP$
where a is your secret key.

Key generation: $a \mapsto aB$.

Signing: $r \mapsto rB$.

All of these use secret data.
Does timing leak this data?

Are there any branches in
ECC ops? Point ops? Field ops?
Do the underlying machine insns
take variable time?

Recall left-to-right binary method
to compute $n, P \mapsto nP$
using point addition:

```
def scalarmult(n,P):
  if n == 0: return 0
  if n == 1: return P
  R = scalarmult(n//2,P)
  R = R + R
  if n % 2: R = R + P
  return R
```

Many branches here.
NAF etc. also use many branches.

t-time ECC

omputation: $a, P \mapsto aP$
is your secret key.

eration: $a \mapsto aB$.

$r \mapsto rB$.

ese use secret data.
ning leak this data?

e any branches in
s? Point ops? Field ops?
underlying machine insns
iable time?

Recall left-to-right binary method
to compute $n, P \mapsto nP$
using point addition:

```
def scalarmult(n,P):
  if n == 0: return 0
  if n == 1: return P
  R = scalarmult(n//2,P)
  R = R + R
  if n % 2: R = R + P
  return R
```

Many branches here.
NAF etc. also use many branches.

Even if e
takes th
(certainl
total tim

If $2^{e-1} \leq$
n has ex
number

Particula
usually i
"Lattice
compute
positions

C

on: $a, P \mapsto aP$
cret key.

$\mapsto aB$.

cret data.
this data?

nches in
ps? Field ops?
machine insns
?

Recall left-to-right binary method
to compute $n, P \mapsto nP$
using point addition:

```
def scalarmult(n,P):
  if n == 0: return 0
  if n == 1: return P
  R = scalarmult(n//2,P)
  R = R + R
  if n % 2: R = R + P
  return R
```

Many branches here.
NAF etc. also use many branches.

Even if each point
takes the same am
(certainly not true
total time depends

If $2^{e-1} \leq n < 2^e$ a
n has exactly w bi
number of additio

Particularly fast to
usually indicates v
"Lattice attacks"
compute the secre
positions of very s

$\mapsto aP$

ops?

insns

Recall left-to-right binary method to compute $n, P \mapsto nP$ using point addition:

```
def scalarmult(n,P):
  if n == 0: return 0
  if n == 1: return P
  R = scalarmult(n//2,P)
  R = R + R
  if n % 2: R = R + P
  return R
```

Many branches here.
NAF etc. also use many branches.

Even if each point addition takes the same amount of ti (certainly not true in Pytho total time depends on n.

If $2^{e-1} \le n < 2^e$ and n has exactly w bits set: number of additions is $e +$

Particularly fast total time usually indicates very small
“Lattice attacks” on signatu compute the secret key give positions of very small nonc

Recall left-to-right binary method
to compute $n, P \mapsto nP$
using point addition:

```
def scalarmult(n,P):
  if n == 0: return 0
  if n == 1: return P
  R = scalarmult(n//2,P)
  R = R + R
  if n % 2: R = R + P
  return R
```

Many branches here.
NAF etc. also use many branches.

Even if each point addition
takes the same amount of time
(certainly not true in Python),
total time depends on n.

If $2^{e-1} \leq n < 2^e$ and
n has exactly w bits set:
number of additions is $e + w - 2$.

Particularly fast total time
usually indicates very small n.
"Lattice attacks" on signatures
compute the secret key given
positions of very small nonces r.

eft-to-right binary method
ute $n, P \mapsto nP$
oint addition:

```
larmult(n,P):
== 0: return 0
== 1: return P
calarmult(n//2,P)
 + R
% 2: R = R + P
n R
```

anches here.
:. also use many branches.

Even if each point addition takes the same amount of time (certainly not true in Python), total time depends on n.

If $2^{e-1} \leq n < 2^e$ and n has exactly w bits set: number of additions is $e + w - 2$.

Particularly fast total time usually indicates very small n. "Lattice attacks" on signatures compute the secret key given positions of very small nonces r.

Even wo
CPUs do
metadat

Actual t
affects, a
detailed
branch p

Attacker
often see
Exploite

t binary method

→ nP

on:

```
,P):
urn 0
urn P
(n//2,P)

R + P
```

re.

many branches.

Even if each point addition takes the same amount of time (certainly not true in Python), total time depends on n.

If $2^{e-1} \le n < 2^e$ and n has exactly w bits set: number of additions is $e + w - 2$.

Particularly fast total time usually indicates very small n. "Lattice attacks" on signatures compute the secret key given positions of very small nonces r.

Even worse:

CPUs do not try t

metadata regardin

Actual time for a

affects, and is affe

detailed state of c

branch predictor, e

Attacker interacts

often sees pattern

Exploited in, e.g.,

ethod

nches.

Even if each point addition takes the same amount of time (certainly not true in Python), total time depends on n.

If $2^{e-1} \leq n < 2^e$ and n has exactly w bits set: number of additions is $e + w - 2$.

Particularly fast total time usually indicates very small n. "Lattice attacks" on signatures compute the secret key given positions of very small nonces r.

Even worse:
CPUs do not try to protect
metadata regarding branche

Actual time for a branch
affects, and is affected by,
detailed state of code cache
branch predictor, etc.

Attacker interacts with this
often sees pattern of branch
Exploited in, e.g., Bitcoin at

Even if each point addition takes the same amount of time (certainly not true in Python), total time depends on n.

If $2^{e-1} \leq n < 2^e$ and n has exactly w bits set: number of additions is $e + w - 2$.

Particularly fast total time usually indicates very small n. "Lattice attacks" on signatures compute the secret key given positions of very small nonces r.

Even worse:
CPUs do not try to protect metadata regarding branches.

Actual time for a branch affects, and is affected by, detailed state of code cache, branch predictor, etc.

Attacker interacts with this state, often sees pattern of branches. Exploited in, e.g., Bitcoin attack.

Even if each point addition
takes the same amount of time
(certainly not true in Python),
total time depends on n.

If $2^{e-1} \leq n < 2^e$ and
n has exactly w bits set:
number of additions is $e + w - 2$.

Particularly fast total time
usually indicates very small n.
"Lattice attacks" on signatures
compute the secret key given
positions of very small nonces r.

Even worse:
CPUs do not try to protect
metadata regarding branches.

Actual time for a branch
affects, and is affected by,
detailed state of code cache,
branch predictor, etc.

Attacker interacts with this state,
often sees pattern of branches.
Exploited in, e.g., Bitcoin attack.

Confidence-inspiring solution:
Avoid all data flow from secrets to branch conditions.

each point addition
e same amount of time
y not true in Python),
ne depends on n.

$\leq n < 2^e$ and
xactly w bits set:
of additions is $e + w - 2$.

arly fast total time
ndicates very small n.
attacks" on signatures
e the secret key given
s of very small nonces r.

Even worse:
CPUs do not try to protect metadata regarding branches.

Actual time for a branch affects, and is affected by, detailed state of code cache, branch predictor, etc.

Attacker interacts with this state, often sees pattern of branches. Exploited in, e.g., Bitcoin attack.

Confidence-inspiring solution:
Avoid all data flow from secrets to branch conditions.

Double-a

Eliminat
always c

```
def scal
  if b =
  R = sc
  R2 = R
  S = [R
  return
```

Works fo
Always t
(includin
Use pub

addition
ount of time
in Python),
s on n.

and
its set:
ns is $e + w - 2$.

tal time
ery small n.
on signatures
t key given
mall nonces r.

Even worse:
CPUs do not try to protect metadata regarding branches.

Actual time for a branch affects, and is affected by, detailed state of code cache, branch predictor, etc.

Attacker interacts with this state, often sees pattern of branches. Exploited in, e.g., Bitcoin attack.

Confidence-inspiring solution:
Avoid all data flow from secrets to branch conditions.

Double-and-add-al

Eliminate branche
always computing

```
def scalarmult(n
  if b == 0: ret
  R = scalarmult
  R2 = R + R
  S = [R2,R2 + P
  return S[n % 2
```

Works for $0 \leq n <$
Always takes $2b$ a
(including b doubl
Use public b: bits

me
n),

$w - 2$.

n.
ures
n
es r.

Even worse:
CPUs do not try to protect metadata regarding branches.

Actual time for a branch affects, and is affected by, detailed state of code cache, branch predictor, etc.

Attacker interacts with this state, often sees pattern of branches. Exploited in, e.g., Bitcoin attack.

Confidence-inspiring solution:
Avoid all data flow from secrets to branch conditions.

Double-and-add-always

Eliminate branches by always computing both resu

```
def scalarmult(n,b,P):
  if b == 0: return 0
  R = scalarmult(n//2,b-1
  R2 = R + R
  S = [R2,R2 + P]
  return S[n % 2]
```

Works for $0 \le n < 2^b$.
Always takes $2b$ additions (including b doublings).
Use public b: bits *allowed* i

Even worse:
CPUs do not try to protect metadata regarding branches.

Actual time for a branch affects, and is affected by, detailed state of code cache, branch predictor, etc.

Attacker interacts with this state, often sees pattern of branches.
Exploited in, e.g., Bitcoin attack.

Confidence-inspiring solution:
Avoid all data flow from secrets to branch conditions.

Double-and-add-always

Eliminate branches by always computing both results:

```
def scalarmult(n,b,P):
  if b == 0: return 0
  R = scalarmult(n//2,b-1,P)
  R2 = R + R
  S = [R2,R2 + P]
  return S[n % 2]
```

Works for $0 \le n < 2^b$.
Always takes $2b$ additions (including b doublings).
Use public b: bits *allowed* in n.

rse:
o not try to protect
a regarding branches.

ime for a branch
and is affected by,
state of code cache,
predictor, etc.

interacts with this state,
es pattern of branches.
d in, e.g., Bitcoin attack.

nce-inspiring solution:
ll data flow from
to branch conditions.

<u>Double-and-add-always</u>

Eliminate branches by
always computing both results:

```
def scalarmult(n,b,P):
  if b == 0: return 0
  R = scalarmult(n//2,b-1,P)
  R2 = R + R
  S = [R2,R2 + P]
  return S[n % 2]
```

Works for $0 \le n < 2^b$.
Always takes $2b$ additions
(including b doublings).
Use public b: bits *allowed* in n.

Another
CPUs do
metadat

Actual t
affects, a
detailed
store-to-

Exploite
despite I
claiming

o protect
g branches.

branch
cted by,
ode cache,
etc.

with this state,
of branches.
Bitcoin attack.

ng solution:
ow from
n conditions.

Double-and-add-always

Eliminate branches by
always computing both results:

```
def scalarmult(n,b,P):
  if b == 0: return 0
  R = scalarmult(n//2,b-1,P)
  R2 = R + R
  S = [R2,R2 + P]
  return S[n % 2]
```

Works for $0 \le n < 2^b$.
Always takes $2b$ additions
(including b doublings).
Use public b: bits *allowed* in n.

Another big proble
CPUs do not try t
metadata regardin

Actual time for x[
affects, and is affe
detailed state of d
store-to-load forwa

Exploited in, e.g.,
despite Intel and O
claiming their code

s.

,

state,

es.

tack.

n:

ns.

Double-and-add-always

Eliminate branches by always computing both results:

```
def scalarmult(n,b,P):
  if b == 0: return 0
  R = scalarmult(n//2,b-1,P)
  R2 = R + R
  S = [R2,R2 + P]
  return S[n % 2]
```

Works for $0 \le n < 2^b$.
Always takes $2b$ additions (including b doublings).
Use public b: bits *allowed* in n.

Another big problem:
CPUs do not try to protect metadata regarding *array in*

Actual time for `x[i]` affects, and is affected by, detailed state of data cache, store-to-load forwarder, etc.

Exploited in, e.g., CacheBlee despite Intel and OpenSSL claiming their code was safe

Double-and-add-always

Eliminate branches by always computing both results:

```
def scalarmult(n,b,P):
  if b == 0: return 0
  R = scalarmult(n//2,b-1,P)
  R2 = R + R
  S = [R2,R2 + P]
  return S[n % 2]
```

Works for $0 \leq n < 2^b$.
Always takes $2b$ additions (including b doublings).
Use public b: bits *allowed* in n.

Another big problem:
CPUs do not try to protect metadata regarding *array indices*.

Actual time for `x[i]` affects, and is affected by, detailed state of data cache, store-to-load forwarder, etc.

Exploited in, e.g., CacheBleed, despite Intel and OpenSSL claiming their code was safe.

Double-and-add-always

Eliminate branches by
always computing both results:

```
def scalarmult(n,b,P):
  if b == 0: return 0
  R = scalarmult(n//2,b-1,P)
  R2 = R + R
  S = [R2,R2 + P]
  return S[n % 2]
```

Works for $0 \le n < 2^b$.
Always takes $2b$ additions
(including b doublings).
Use public b: bits *allowed* in n.

Another big problem:
CPUs do not try to protect
metadata regarding *array indices*.

Actual time for `x[i]`
affects, and is affected by,
detailed state of data cache,
store-to-load forwarder, etc.

Exploited in, e.g., CacheBleed,
despite Intel and OpenSSL
claiming their code was safe.

Confidence-inspiring solution:
Avoid all data flow from secrets to memory addresses.

and-add-always

e branches by
omputing both results:

```
larmult(n,b,P):
== 0: return 0
calarmult(n//2,b-1,P)
R + R
R2,R2 + P]
n S[n % 2]
```

or $0 \le n < 2^b$.
takes $2b$ additions
ng b doublings).
lic b: bits *allowed* in n.

Another big problem:
CPUs do not try to protect
metadata regarding *array indices*.

Actual time for `x[i]`
affects, and is affected by,
detailed state of data cache,
store-to-load forwarder, etc.

Exploited in, e.g., CacheBleed,
despite Intel and OpenSSL
claiming their code was safe.

Confidence-inspiring solution:
Avoid all data flow from secrets to memory addresses.

Table lo

Always r
Use bit o
the desi

```
def sca
  if b =
  R = s
  R2 = 
  S = [
  mask =
  retur
```

lways

s by
both results:

```
,b,P):
urn 0
(n//2,b-1,P)

]
]
```

< 2^b.
dditions
ings).
allowed in n.

Another big problem:
CPUs do not try to protect
metadata regarding *array indices*.

Actual time for `x[i]`
affects, and is affected by,
detailed state of data cache,
store-to-load forwarder, etc.

Exploited in, e.g., CacheBleed,
despite Intel and OpenSSL
claiming their code was safe.

Confidence-inspiring solution:
Avoid all data flow from secrets to memory addresses.

Table lookups via

Always read all tal
Use bit operations
the desired table e

```
def scalarmult(n
  if b == 0: ret
  R = scalarmult
  R2 = R + R
  S = [R2,R2 + P
  mask = -(n % 2
  return S[0]^(m
```

lts:

,P)

n *n*.

Another big problem:
CPUs do not try to protect
metadata regarding *array indices*.

Actual time for `x[i]`
affects, and is affected by,
detailed state of data cache,
store-to-load forwarder, etc.

Exploited in, e.g., CacheBleed,
despite Intel and OpenSSL
claiming their code was safe.

Confidence-inspiring solution:
Avoid all data flow from secrets to memory addresses.

Table lookups via arithmetic

Always read all table entries
Use bit operations to select
the desired table entry:

```
def scalarmult(n,b,P):
  if b == 0: return 0
  R = scalarmult(n//2,b-1
  R2 = R + R
  S = [R2,R2 + P]
  mask = -(n % 2)
  return S[0]^(mask&(S[1]
```

Another big problem:
CPUs do not try to protect
metadata regarding *array indices*.

Actual time for `x[i]`
affects, and is affected by,
detailed state of data cache,
store-to-load forwarder, etc.

Exploited in, e.g., CacheBleed,
despite Intel and OpenSSL
claiming their code was safe.

Confidence-inspiring solution:
Avoid all data flow from secrets to memory addresses.

Table lookups via arithmetic

Always read all table entries.
Use bit operations to select
the desired table entry:

```
def scalarmult(n,b,P):
  if b == 0: return 0
  R = scalarmult(n//2,b-1,P)
  R2 = R + R
  S = [R2,R2 + P]
  mask = -(n % 2)
  return S[0]^(mask&(S[1]^S[0]))
```

big problem:
o not try to protect
a regarding *array indices*.

ime for x[i]
and is affected by,
state of data cache,
-load forwarder, etc.

d in, e.g., CacheBleed,
ntel and OpenSSL
their code was safe.

nce-inspiring solution:
ll data flow from
to memory addresses.

Table lookups via arithmetic

Always read all table entries.
Use bit operations to select
the desired table entry:

```
def scalarmult(n,b,P):
  if b == 0: return 0
  R = scalarmult(n//2,b-1,P)
  R2 = R + R
  S = [R2,R2 + P]
  mask = -(n % 2)
  return S[0]^(mask&(S[1]^S[0]))
```

Width-2

```
def fixw
  if b <
  T = ta
  mask =
  T ^= t
  mask =
  T ^= t
  mask =
  T ^= t
  R = fi
  R = R
  R = R
  return
```

em:
o protect
g *array indices*.

[i]
cted by,
ata cache,
arder, etc.

CacheBleed,
OpenSSL
e was safe.

ng solution:
ow from
ry addresses.

Table lookups via arithmetic

Always read all table entries.
Use bit operations to select
the desired table entry:

```
def scalarmult(n,b,P):
  if b == 0: return 0
  R = scalarmult(n//2,b-1,P)
  R2 = R + R
  S = [R2,R2 + P]
  mask = -(n % 2)
  return S[0]^(mask&(S[1]^S[0]))
```

Width-2 unsigned

```
def fixwin2(n,b,
  if b <= 0: ret
  T = table[0]
  mask = (-(1 ^
  T ^= ~mask & (
  mask = (-(2 ^
  T ^= ~mask & (
  mask = (-(3 ^
  T ^= ~mask & (
  R = fixwin2(n/
  R = R + R
  R = R + R
  return R + T
```

dices.

,

ed,

.

n:

ses.

Table lookups via arithmetic

Always read all table entries.

Use bit operations to select the desired table entry:

```
def scalarmult(n,b,P):
  if b == 0: return 0
  R = scalarmult(n//2,b-1,P)
  R2 = R + R
  S = [R2,R2 + P]
  mask = -(n % 2)
  return S[0]^(mask&(S[1]^S[0]))
```


Width-2 unsigned fixed winc

```
def fixwin2(n,b,table):
  if b <= 0: return 0
  T = table[0]
  mask = (-(1 ^ (n % 4)))
  T ^= ~mask & (T^table[1
  mask = (-(2 ^ (n % 4)))
  T ^= ~mask & (T^table[2
  mask = (-(3 ^ (n % 4)))
  T ^= ~mask & (T^table[3
  R = fixwin2(n//4,b-2,ta
  R = R + R
  R = R + R
  return R + T
```

Table lookups via arithmetic

Always read all table entries.

Use bit operations to select the desired table entry:

```
def scalarmult(n,b,P):
  if b == 0: return 0
  R = scalarmult(n//2,b-1,P)
  R2 = R + R
  S = [R2,R2 + P]
  mask = -(n % 2)
  return S[0]^(mask&(S[1]^S[0]))
```


Width-2 unsigned fixed windows

```
def fixwin2(n,b,table):
  if b <= 0: return 0
  T = table[0]
  mask = (-(1 ^ (n % 4))) >> 2
  T ^= ~mask & (T^table[1])
  mask = (-(2 ^ (n % 4))) >> 2
  T ^= ~mask & (T^table[2])
  mask = (-(3 ^ (n % 4))) >> 2
  T ^= ~mask & (T^table[3])
  R = fixwin2(n//4,b-2,table)
  R = R + R
  R = R + R
  return R + T
```

okups via arithmetic

read all table entries.
operations to select
red table entry:

```
larmult(n,b,P):
== 0: return 0
calarmult(n//2,b-1,P)
R + R
R2,R2 + P]
= -(n % 2)
n S[0]^(mask&(S[1]^S[0]))
```


Width-2 unsigned fixed windows

```
def fixwin2(n,b,table):
  if b <= 0: return 0
  T = table[0]
  mask = (-(1 ^ (n % 4))) >> 2
  T ^= ~mask & (T^table[1])
  mask = (-(2 ^ (n % 4))) >> 2
  T ^= ~mask & (T^table[2])
  mask = (-(3 ^ (n % 4))) >> 2
  T ^= ~mask & (T^table[3])
  R = fixwin2(n//4,b-2,table)
  R = R + R
  R = R + R
  return R + T
```

```
def sca
  P2 =
  table
  retur
```

Public b

For $b \in$
Always
Always
Always 2

Can sim
larger-wi
Unsigned
Signed i

arithmetic

ble entries.

to select

entry:

```
,b,P):
urn 0
(n//2,b-1,P)

]
)
ask&(S[1]^S[0]))
```


Width-2 unsigned fixed windows

```
def fixwin2(n,b,table):
  if b <= 0: return 0
  T = table[0]
  mask = (-(1 ^ (n % 4))) >> 2
  T ^= ~mask & (T^table[1])
  mask = (-(2 ^ (n % 4))) >> 2
  T ^= ~mask & (T^table[2])
  mask = (-(3 ^ (n % 4))) >> 2
  T ^= ~mask & (T^table[3])
  R = fixwin2(n//4,b-2,table)
  R = R + R
  R = R + R
  return R + T
```

```
def scalarmult(n
  P2 = P+P
  table = [0,P,P
  return fixwin2
```

Public branches, p

For $b \in 2\mathbf{Z}$:

Always b doubling

Always $b/2$ additio

Always 2 additions

Can similarly prote

larger-width fixed

Unsigned is slightl

Signed is slightly f

t

.

,P)

^S[0]))

Width-2 unsigned fixed windows

```
def fixwin2(n,b,table):
  if b <= 0: return 0
  T = table[0]
  mask = (-(1 ^ (n % 4))) >> 2
  T ^= ~mask & (T^table[1])
  mask = (-(2 ^ (n % 4))) >> 2
  T ^= ~mask & (T^table[2])
  mask = (-(3 ^ (n % 4))) >> 2
  T ^= ~mask & (T^table[3])
  R = fixwin2(n//4,b-2,table)
  R = R + R
  R = R + R
  return R + T
```



```
def scalarmult(n,b,P):
  P2 = P+P
  table = [0,P,P2,P2+P]
  return fixwin2(n,b,tabl
```

Public branches, public indic

For $b \in 2\mathbf{Z}$:
Always b doublings.
Always $b/2$ additions of T.
Always 2 additions for table.

Can similarly protect
larger-width fixed windows.
Unsigned is slightly easier.
Signed is slightly faster.

Width-2 unsigned fixed windows

```
def fixwin2(n,b,table):
  if b <= 0: return 0
  T = table[0]
  mask = (-(1 ^ (n % 4))) >> 2
  T ^= ~mask & (T^table[1])
  mask = (-(2 ^ (n % 4))) >> 2
  T ^= ~mask & (T^table[2])
  mask = (-(3 ^ (n % 4))) >> 2
  T ^= ~mask & (T^table[3])
  R = fixwin2(n//4,b-2,table)
  R = R + R
  R = R + R
  return R + T
```



```
def scalarmult(n,b,P):
  P2 = P+P
  table = [0,P,P2,P2+P]
  return fixwin2(n,b,table)
```

Public branches, public indices.

For $b \in 2\mathbf{Z}$:
Always b doublings.
Always $b/2$ additions of T.
Always 2 additions for table.

Can similarly protect larger-width fixed windows.
Unsigned is slightly easier.
Signed is slightly faster.

unsigned fixed windows

```
win2(n,b,table):
<= 0: return 0
able[0]
= (-(1 ^ (n % 4))) >> 2
~mask & (T^table[1])
= (-(2 ^ (n % 4))) >> 2
~mask & (T^table[2])
= (-(3 ^ (n % 4))) >> 2
~mask & (T^table[3])
ixwin2(n//4,b-2,table)
 + R
 + R
n R + T
```



```
def scalarmult(n,b,P):
  P2 = P+P
  table = [0,P,P2,P2+P]
  return fixwin2(n,b,table)
```

Public branches, public indices.

For $b \in 2\mathbf{Z}$:
Always b doublings.
Always $b/2$ additions of T.
Always 2 additions for table.

Can similarly protect
larger-width fixed windows.
Unsigned is slightly easier.
Signed is slightly faster.

Fixed-ba

Obvious
$a \mapsto aB$
reuse n,

fixed windows

```
table):
urn 0

(n % 4))) >> 2
T^table[1])
(n % 4))) >> 2
T^table[2])
(n % 4))) >> 2
T^table[3])
/4,b-2,table)
```



```
def scalarmult(n,b,P):
  P2 = P+P
  table = [0,P,P2,P2+P]
  return fixwin2(n,b,table)
```

Public branches, public indices.

For $b \in 2\mathbf{Z}$:
Always b doublings.
Always $b/2$ additions of T.
Always 2 additions for table.

Can similarly protect
larger-width fixed windows.
Unsigned is slightly easier.
Signed is slightly faster.

Fixed-base scalar r

Obvious way to ha
$a \mapsto aB$ and signir
reuse $n, P \mapsto nP$ f

dows

```
 >> 2
]])
 >> 2
2])
 >> 2
3])
able)
```



```
def scalarmult(n,b,P):
  P2 = P+P
  table = [0,P,P2,P2+P]
  return fixwin2(n,b,table)
```

Public branches, public indices.

For $b \in 2\mathbf{Z}$:
Always b doublings.
Always $b/2$ additions of T.
Always 2 additions for table.

Can similarly protect
larger-width fixed windows.
Unsigned is slightly easier.
Signed is slightly faster.

Fixed-base scalar multiplicat

Obvious way to handle keyg
$a \mapsto aB$ and signing $r \mapsto rB$
reuse $n, P \mapsto nP$ from ECDI

```
def scalarmult(n,b,P):
  P2 = P+P
  table = [0,P,P2,P2+P]
  return fixwin2(n,b,table)
```

Public branches, public indices.

For $b \in 2\mathbf{Z}$:
Always b doublings.
Always $b/2$ additions of T.
Always 2 additions for table.

Can similarly protect
larger-width fixed windows.
Unsigned is slightly easier.
Signed is slightly faster.

Fixed-base scalar multiplication

Obvious way to handle keygen
$a \mapsto aB$ and signing $r \mapsto rB$:
reuse $n, P \mapsto nP$ from ECDH.

```
def scalarmult(n,b,P):
  P2 = P+P
  table = [0,P,P2,P2+P]
  return fixwin2(n,b,table)
```

Public branches, public indices.

For $b \in 2\mathbf{Z}$:
Always b doublings.
Always $b/2$ additions of T.
Always 2 additions for table.

Can similarly protect larger-width fixed windows.
Unsigned is slightly easier.
Signed is slightly faster.

Fixed-base scalar multiplication

Obvious way to handle keygen $a \mapsto aB$ and signing $r \mapsto rB$: reuse $n, P \mapsto nP$ from ECDH.

Can do much better since B is a constant: standard base point.

e.g. For $b = 256$: Compute $(2^{128}n_1 + n_0)B$ as $n_1B_1 + n_0B$ using double-scalar fixed windows, after precomputing $B_1 = 2^{128}B$.

Fun exercise: For each k, try to minimize number of additions using k precomputed points.

```
larmult(n,b,P):
P+P
 = [0,P,P2,P2+P]
n fixwin2(n,b,table)
```

ranches, public indices.

$2\mathbf{Z}$:

b doublings.

$b/2$ additions of T.

2 additions for table.

ilarly protect

idth fixed windows.

d is slightly easier.

s slightly faster.

Fixed-base scalar multiplication

Obvious way to handle keygen $a \mapsto aB$ and signing $r \mapsto rB$: reuse $n, P \mapsto nP$ from ECDH.

Can do much better since B is a constant: standard base point.

e.g. For $b = 256$: Compute $(2^{128} n_1 + n_0)B$ as $n_1 B_1 + n_0 B$ using double-scalar fixed windows, after precomputing $B_1 = 2^{128} B$.

Fun exercise: For each k, try to minimize number of additions using k precomputed points.

Recall C

57164 cy

63526 cy

205741 c

159128 c

ECDH is

Verificat

somewha

(But bat

Keygen

much fa

Signing

dependir

```
,b,P):

2,P2+P]
(n,b,table)
```

ublic indices.

s.
ons of T.
s for table.

ect
windows.
y easier.
aster.

Fixed-base scalar multiplication

Obvious way to handle keygen $a \mapsto aB$ and signing $r \mapsto rB$: reuse $n, P \mapsto nP$ from ECDH.

Can do much better since B is a constant: standard base point.

e.g. For $b = 256$: Compute $(2^{128}n_1 + n_0)B$ as $n_1 B_1 + n_0 B$ using double-scalar fixed windows, after precomputing $B_1 = 2^{128}B$.

Fun exercise: For each k, try to minimize number of additions using k precomputed points.

Recall Chou timing
57164 cycles for k
63526 cycles for si
205741 cycles for
159128 cycles for

ECDH is single-sca

Verification is dou
somewhat slower t
(But batch verifica

Keygen is fixed-ba
much faster than

Signing is keygen
depending on mes

e)

ces.

.

Fixed-base scalar multiplication

Obvious way to handle keygen $a \mapsto aB$ and signing $r \mapsto rB$: reuse $n, P \mapsto nP$ from ECDH.

Can do much better since B is a constant: standard base point.

e.g. For $b = 256$: Compute $(2^{128} n_1 + n_0)B$ as $n_1 B_1 + n_0 B$ using double-scalar fixed windows, after precomputing $B_1 = 2^{128} B$.

Fun exercise: For each k, try to minimize number of additions using k precomputed points.

Recall Chou timings:
57164 cycles for keygen,
63526 cycles for signature,
205741 cycles for verificatio
159128 cycles for ECDH.

ECDH is single-scalar mult.

Verification is double-scalar
somewhat slower than ECD
(But batch verification is fa

Keygen is fixed-base scalar
much faster than ECDH.

Signing is keygen plus overh
depending on message lengt

Fixed-base scalar multiplication

Obvious way to handle keygen $a \mapsto aB$ and signing $r \mapsto rB$: reuse $n, P \mapsto nP$ from ECDH.

Can do much better since B is a constant: standard base point.

e.g. For $b = 256$: Compute $(2^{128}n_1 + n_0)B$ as $n_1 B_1 + n_0 B$ using double-scalar fixed windows, after precomputing $B_1 = 2^{128}B$.

Fun exercise: For each k, try to minimize number of additions using k precomputed points.

Recall Chou timings:
57164 cycles for keygen,
63526 cycles for signature,
205741 cycles for verification,
159128 cycles for ECDH.

ECDH is single-scalar mult.

Verification is double-scalar mult, somewhat slower than ECDH. (But batch verification is faster.)

Keygen is fixed-base scalar mult, much faster than ECDH.

Signing is keygen plus overhead depending on message length.

<u>se scalar multiplication</u>

way to handle keygen
and signing $r \mapsto rB$:
$P \mapsto nP$ from ECDH.

much better since B is
nt: standard base point.

$b = 256$: Compute
$+ n_0)B$ as $n_1 B_1 + n_0 B$
uble-scalar fixed windows,
ecomputing $B_1 = 2^{128} B$.

rcise: For each k, try to
e number of additions
precomputed points.

Recall Chou timings:
57164 cycles for keygen,
63526 cycles for signature,
205741 cycles for verification,
159128 cycles for ECDH.

ECDH is single-scalar mult.

Verification is double-scalar mult,
somewhat slower than ECDH.
(But batch verification is faster.)

Keygen is fixed-base scalar mult,
much faster than ECDH.

Signing is keygen plus overhead
depending on message length.

Let's mo

ECC
verify S

Point
P, Q

Field
$x_1, x_2 \mapsto$

Machin
32-bit

Gat
AND,

multiplication

andle keygen
ng $r \mapsto rB$:
from ECDH.

ter since B is
ard base point.

Compute
$n_1 B_1 + n_0 B$
r fixed windows,
g $B_1 = 2^{128} B$.

each k, try to
of additions
ted points.

Recall Chou timings:
57164 cycles for keygen,
63526 cycles for signature,
205741 cycles for verification,
159128 cycles for ECDH.

ECDH is single-scalar mult.

Verification is double-scalar mult,
somewhat slower than ECDH.
(But batch verification is faster.)

Keygen is fixed-base scalar mult,
much faster than ECDH.

Signing is keygen plus overhead
depending on message length.

Let's move down a

ECC ops: e.g.,
verify $SB = R +$

window

Point ops: e.g.
$P, Q \mapsto P + Q$

faster

Field ops: e.g.,
$x_1, x_2 \mapsto x_1 x_2$ in

delayed

Machine insns: e.
32-bit multiplicat

pipelin

Gates: e.g.,
AND, OR, XOR

tion

en

3:

H.

3 is

oint.

$_0B$

ndows,

^{28}B.

y to

ns

.

Recall Chou timings:

57164 cycles for keygen,
63526 cycles for signature,
205741 cycles for verification,
159128 cycles for ECDH.

ECDH is single-scalar mult.

Verification is double-scalar mult, somewhat slower than ECDH.
(But batch verification is faster.)

Keygen is fixed-base scalar mult, much faster than ECDH.

Signing is keygen plus overhead depending on message length.

Let's move down a level:

ECC ops: e.g., verify $SB = R + hA$

↓ windowing etc.

Point ops: e.g., $P, Q \mapsto P + Q$

↓ faster doubling e

Field ops: e.g., $x_1, x_2 \mapsto x_1x_2$ in $\mathbf{F}_p$

↓ delayed carries et

Machine insns: e.g., 32-bit multiplication

↓ pipelining etc.

Gates: e.g., AND, OR, XOR

Recall Chou timings:
57164 cycles for keygen,
63526 cycles for signature,
205741 cycles for verification,
159128 cycles for ECDH.

ECDH is single-scalar mult.

Verification is double-scalar mult, somewhat slower than ECDH.
(But batch verification is faster.)

Keygen is fixed-base scalar mult, much faster than ECDH.

Signing is keygen plus overhead depending on message length.

Let's move down a level:

ECC ops: e.g., verify $SB = R + hA$

↓ windowing etc.

Point ops: e.g., $P, Q \mapsto P + Q$

↓ faster doubling etc.

Field ops: e.g., $x_1, x_2 \mapsto x_1 x_2$ in $\mathbf{F}_p$

↓ delayed carries etc.

Machine insns: e.g., 32-bit multiplication

↓ pipelining etc.

Gates: e.g., AND, OR, XOR

hou timings:
ycles for keygen,
ycles for signature,
cycles for verification,
cycles for ECDH.

s single-scalar mult.

tion is double-scalar mult,
at slower than ECDH.
tch verification is faster.)

is fixed-base scalar mult,
ster than ECDH.

is keygen plus overhead
ng on message length.

Let's move down a level:

ECC ops: e.g., verify $SB = R + hA$

↓ windowing etc.

Point ops: e.g., $P, Q \mapsto P + Q$

↓ faster doubling etc.

Field ops: e.g., $x_1, x_2 \mapsto x_1 x_2$ in $\mathbf{F}_p$

↓ delayed carries etc.

Machine insns: e.g., 32-bit multiplication

↓ pipelining etc.

Gates: e.g., AND, OR, XOR

Eliminat

Have to
of curve
How to
addition

Addition
$((x_1 y_2 +$
$(y_1 y_2 -$
uses exp

gs:
eygen,
gnature,
verification,
ECDH.

alar mult.

ble-scalar mult,
than ECDH.
ation is faster.)

se scalar mult,
ECDH.

plus overhead
sage length.

Let's move down a level:

ECC ops: e.g., verify $SB = R + hA$

↓ windowing etc.

Point ops: e.g., $P, Q \mapsto P + Q$

↓ faster doubling etc.

Field ops: e.g., $x_1, x_2 \mapsto x_1 x_2$ in $\mathbf{F}_p$

↓ delayed carries etc.

Machine insns: e.g., 32-bit multiplication

↓ pipelining etc.

Gates: e.g., AND, OR, XOR

Eliminating divisio

Have to do many
of curve points: P
How to efficiently
additions into field

Addition $(x_1, y_1) +$
$((x_1 y_2 + y_1 x_2)/(1$
$(y_1 y_2 - x_1 x_2)/(1$
uses expensive div

n,

mult,

H.

ster.)

mult,

ead

h.

Let's move down a level:

ECC ops: e.g., verify $SB = R + hA$

↓ windowing etc.

Point ops: e.g., $P, Q \mapsto P + Q$

↓ faster doubling etc.

Field ops: e.g., $x_1, x_2 \mapsto x_1x_2$ in $\mathbf{F}_p$

↓ delayed carries etc.

Machine insns: e.g., 32-bit multiplication

↓ pipelining etc.

Gates: e.g., AND, OR, XOR

Eliminating divisions

Have to do many additions of curve points: $P, Q \mapsto P +$ How to efficiently decompos additions into field ops?

Addition $(x_1, y_1) + (x_2, y_2) =$
$((x_1y_2 + y_1x_2)/(1 + dx_1x_2y_1$
$(y_1y_2 - x_1x_2)/(1 - dx_1x_2y_1$
uses expensive divisions.

Let's move down a level:

ECC ops: e.g.,
verify $SB = R + hA$

$\downarrow$ windowing etc.

Point ops: e.g.,
$P, Q \mapsto P + Q$

$\downarrow$ faster doubling etc.

Field ops: e.g.,
$x_1, x_2 \mapsto x_1 x_2$ in $\mathbf{F}_p$

$\downarrow$ delayed carries etc.

Machine insns: e.g.,
32-bit multiplication

$\downarrow$ pipelining etc.

Gates: e.g.,
AND, OR, XOR

Eliminating divisions

Have to do many additions of curve points: $P, Q \mapsto P + Q$. How to efficiently decompose additions into field ops?

Addition $(x_1, y_1) + (x_2, y_2) =$
$((x_1 y_2 + y_1 x_2)/(1 + d x_1 x_2 y_1 y_2),$
$(y_1 y_2 - x_1 x_2)/(1 - d x_1 x_2 y_1 y_2))$
uses expensive divisions.

Let's move down a level:

$$\begin{array}{c}
\boxed{\text{ECC ops: e.g., verify } SB = R + hA} \\
\Big\downarrow \text{ windowing etc.} \\
\boxed{\text{Point ops: e.g., } P, Q \mapsto P + Q} \\
\Big\downarrow \text{ faster doubling etc.} \\
\boxed{\text{Field ops: e.g., } x_1, x_2 \mapsto x_1 x_2 \text{ in } \mathbf{F}_p} \\
\Big\downarrow \text{ delayed carries etc.} \\
\boxed{\text{Machine insns: e.g., 32-bit multiplication}} \\
\Big\downarrow \text{ pipelining etc.} \\
\boxed{\text{Gates: e.g., AND, OR, XOR}}
\end{array}$$

Eliminating divisions

Have to do many additions of curve points: $P, Q \mapsto P + Q$. How to efficiently decompose additions into field ops?

Addition $(x_1, y_1) + (x_2, y_2) =$
$((x_1 y_2 + y_1 x_2)/(1 + d x_1 x_2 y_1 y_2),$
$(y_1 y_2 - x_1 x_2)/(1 - d x_1 x_2 y_1 y_2))$
uses expensive divisions.

Better: postpone divisions and work with fractions. Represent (x, y) as $(X : Y : Z)$ with $x = X/Z$, $y = Y/Z$, $Z \neq 0$.

ove down a level:

ops: e.g.,
$B = R + hA$

windowing etc.

ops: e.g.,
$\mapsto P + Q$

faster doubling etc.

ops: e.g.,
$\to x_1 x_2$ in $\mathbf{F}_p$

delayed carries etc.

e insns: e.g.,
multiplication

pipelining etc.

tes: e.g.,
, OR, XOR

Eliminating divisions

Have to do many additions of curve points: $P, Q \mapsto P + Q$. How to efficiently decompose additions into field ops?

Addition $(x_1, y_1) + (x_2, y_2) =$
$((x_1 y_2 + y_1 x_2)/(1 + d x_1 x_2 y_1 y_2),$
$(y_1 y_2 - x_1 x_2)/(1 - d x_1 x_2 y_1 y_2))$
uses expensive divisions.

Better: postpone divisions and work with fractions. Represent (x, y) as $(X : Y : Z)$ with $x = X/Z$, $y = Y/Z$, $Z \neq 0$.

Addition
handle f

$\left(\frac{X_1}{Z_1}, \frac{Y}{Z}\right.$

$\left(\frac{\frac{X_1}{Z_1}\frac{Y}{Z}}{1 + d}\right.$

$\frac{\frac{Y_1}{Z_1}\frac{Y}{Z}}{1 - d}$

a level:

hA

ving etc.

,

doubling etc.

$\mathbf{F}_p$

d carries etc.

g.,
ion

ing etc.

R

Eliminating divisions

Have to do many additions of curve points: $P, Q \mapsto P + Q$.

How to efficiently decompose additions into field ops?

Addition $(x_1, y_1) + (x_2, y_2) =$
$((x_1 y_2 + y_1 x_2)/(1 + d x_1 x_2 y_1 y_2),$
$(y_1 y_2 - x_1 x_2)/(1 - d x_1 x_2 y_1 y_2))$
uses expensive divisions.

Better: postpone divisions and work with fractions.

Represent (x, y) as $(X : Y : Z)$ with $x = X/Z$, $y = Y/Z$, $Z \neq 0$.

Addition now has

handle fractions as

$$\left(\frac{X_1}{Z_1}, \frac{Y_1}{Z_1}\right) + \left(\frac{X_2}{Z_2}, \ldots\right.$$

$$\left(\frac{\frac{X_1}{Z_1}\frac{Y_2}{Z_2} + \frac{Y_1}{Z_1}\frac{X_2}{Z_2}}{1 + d\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}}\right.$$

$$\frac{\frac{Y_1}{Z_1}\frac{Y_2}{Z_2} - \frac{X_1}{Z_1}\frac{X_2}{Z_2}}{1 - d\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}}$$

etc.

tc.

Eliminating divisions

Have to do many additions of curve points: $P, Q \mapsto P + Q$.
How to efficiently decompose additions into field ops?

Addition $(x_1, y_1) + (x_2, y_2) =$
$((x_1y_2 + y_1x_2)/(1 + dx_1x_2y_1y_2),$
$(y_1y_2 - x_1x_2)/(1 - dx_1x_2y_1y_2))$
uses expensive divisions.

Better: postpone divisions and work with fractions.
Represent (x, y) as $(X : Y : Z)$ with $x = X/Z$, $y = Y/Z$, $Z \neq 0$.

Addition now has to handle fractions as input:

$$\left(\frac{X_1}{Z_1}, \frac{Y_1}{Z_1}\right) + \left(\frac{X_2}{Z_2}, \frac{Y_2}{Z_2}\right) =$$

$$\left(\frac{\frac{X_1}{Z_1}\frac{Y_2}{Z_2} + \frac{Y_1}{Z_1}\frac{X_2}{Z_2}}{1 + d\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}}, \frac{\frac{Y_1}{Z_1}\frac{Y_2}{Z_2} - \frac{X_1}{Z_1}\frac{X_2}{Z_2}}{1 - d\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}}\right) =$$

Eliminating divisions

Have to do many additions of curve points: $P, Q \mapsto P + Q$.
How to efficiently decompose additions into field ops?

Addition $(x_1, y_1) + (x_2, y_2) =$
$((x_1y_2 + y_1x_2)/(1 + dx_1x_2y_1y_2),$
$(y_1y_2 - x_1x_2)/(1 - dx_1x_2y_1y_2))$
uses expensive divisions.

Better: postpone divisions and work with fractions.
Represent (x, y) as $(X : Y : Z)$ with $x = X/Z$, $y = Y/Z$, $Z \neq 0$.

Addition now has to handle fractions as input:

$$\left(\frac{X_1}{Z_1}, \frac{Y_1}{Z_1}\right) + \left(\frac{X_2}{Z_2}, \frac{Y_2}{Z_2}\right) =$$

$$\left(\frac{\frac{X_1}{Z_1}\frac{Y_2}{Z_2} + \frac{Y_1}{Z_1}\frac{X_2}{Z_2}}{1 + d\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}},\right.$$

$$\left.\frac{\frac{Y_1}{Z_1}\frac{Y_2}{Z_2} - \frac{X_1}{Z_1}\frac{X_2}{Z_2}}{1 - d\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}}\right) =$$

Eliminating divisions

Have to do many additions of curve points: $P, Q \mapsto P + Q$.
How to efficiently decompose additions into field ops?

Addition $(x_1, y_1) + (x_2, y_2) =$
$((x_1 y_2 + y_1 x_2)/(1 + d x_1 x_2 y_1 y_2),$
$(y_1 y_2 - x_1 x_2)/(1 - d x_1 x_2 y_1 y_2))$
uses expensive divisions.

Better: postpone divisions and work with fractions.
Represent (x, y) as $(X : Y : Z)$ with $x = X/Z$, $y = Y/Z$, $Z \neq 0$.

Addition now has to handle fractions as input:

$$\left(\frac{X_1}{Z_1}, \frac{Y_1}{Z_1}\right) + \left(\frac{X_2}{Z_2}, \frac{Y_2}{Z_2}\right) =$$

$$\left(\frac{\frac{X_1}{Z_1}\frac{Y_2}{Z_2} + \frac{Y_1}{Z_1}\frac{X_2}{Z_2}}{1 + d\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}}, \frac{\frac{Y_1}{Z_1}\frac{Y_2}{Z_2} - \frac{X_1}{Z_1}\frac{X_2}{Z_2}}{1 - d\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}}\right) =$$

$$\left(\frac{Z_1 Z_2(X_1 Y_2 + Y_1 X_2)}{Z_1^2 Z_2^2 + d X_1 X_2 Y_1 Y_2}, \frac{Z_1 Z_2(Y_1 Y_2 - X_1 X_2)}{Z_1^2 Z_2^2 - d X_1 X_2 Y_1 Y_2}\right)$$

ing divisions

do many additions

points: $P, Q \mapsto P + Q$.

efficiently decompose

s into field ops?

n $(x_1, y_1) + (x_2, y_2) =$

$y_1 x_2)/(1 + d x_1 x_2 y_1 y_2)$,

$x_1 x_2)/(1 - d x_1 x_2 y_1 y_2))$

pensive divisions.

postpone divisions

k with fractions.

nt (x, y) as $(X : Y : Z)$

$= X/Z$, $y = Y/Z$, $Z \neq 0$.

Addition now has to handle fractions as input:

$$\left(\frac{X_1}{Z_1}, \frac{Y_1}{Z_1}\right) + \left(\frac{X_2}{Z_2}, \frac{Y_2}{Z_2}\right) =$$

$$\left(\frac{\frac{X_1}{Z_1}\frac{Y_2}{Z_2} + \frac{Y_1}{Z_1}\frac{X_2}{Z_2}}{1 + d\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}}, \frac{\frac{Y_1}{Z_1}\frac{Y_2}{Z_2} - \frac{X_1}{Z_1}\frac{X_2}{Z_2}}{1 - d\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}}\right) =$$

$$\left(\frac{Z_1 Z_2 (X_1 Y_2 + Y_1 X_2)}{Z_1^2 Z_2^2 + d X_1 X_2 Y_1 Y_2}, \frac{Z_1 Z_2 (Y_1 Y_2 - X_1 X_2)}{Z_1^2 Z_2^2 - d X_1 X_2 Y_1 Y_2}\right)$$

i.e. $\left(\frac{X_1}{Z_1}\right.$

$= \left(\frac{X_3}{Z_3},\right.$

where

$F = Z_1^2$

$G = Z_1^2$

$X_3 = Z_1$

$Y_3 = Z_1$

$Z_3 = F$

Input to

X_1, Y_1, Z

Output

X_3, Y_3, Z

ons

additions

$P, Q \mapsto P + Q$.

decompose

d ops?

$-(x_2, y_2) =$

$+ d x_1 x_2 y_1 y_2),$

$- d x_1 x_2 y_1 y_2))$

isions.

divisions

ctions.

s $(X : Y : Z)$

$= Y/Z,\ Z \neq 0.$

Addition now has to handle fractions as input:

$$\left(\frac{X_1}{Z_1}, \frac{Y_1}{Z_1}\right) + \left(\frac{X_2}{Z_2}, \frac{Y_2}{Z_2}\right) =$$

$$\left(\frac{\frac{X_1}{Z_1}\frac{Y_2}{Z_2} + \frac{Y_1}{Z_1}\frac{X_2}{Z_2}}{1 + d\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}},\right.$$

$$\left.\frac{\frac{Y_1}{Z_1}\frac{Y_2}{Z_2} - \frac{X_1}{Z_1}\frac{X_2}{Z_2}}{1 - d\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}}\right) =$$

$$\left(\frac{Z_1 Z_2 (X_1 Y_2 + Y_1 X_2)}{Z_1^2 Z_2^2 + d X_1 X_2 Y_1 Y_2},\right.$$

$$\left.\frac{Z_1 Z_2 (Y_1 Y_2 - X_1 X_2)}{Z_1^2 Z_2^2 - d X_1 X_2 Y_1 Y_2}\right)$$

i.e. $\left(\frac{X_1}{Z_1}, \frac{Y_1}{Z_1}\right) +$

$= \left(\frac{X_3}{Z_3}, \frac{Y_3}{Z_3}\right)$

where

$F = Z_1^2 Z_2^2 - d X_1$

$G = Z_1^2 Z_2^2 + d X_1$

$X_3 = Z_1 Z_2 (X_1 Y_2$

$Y_3 = Z_1 Z_2 (Y_1 Y_2 -$

$Z_3 = FG.$

Input to addition a

$X_1, Y_1, Z_1, X_2, Y_2,$

Output from addit

X_3, Y_3, Z_3. No div

$-\ Q.$

se

$=$

$_1 y_2),$

$_1 y_2))$

$:\ Z)$

$\neq 0.$

Addition now has to handle fractions as input:

$$\left(\frac{X_1}{Z_1}, \frac{Y_1}{Z_1}\right) + \left(\frac{X_2}{Z_2}, \frac{Y_2}{Z_2}\right) =$$

$$\left(\frac{\frac{X_1}{Z_1}\frac{Y_2}{Z_2} + \frac{Y_1}{Z_1}\frac{X_2}{Z_2}}{1 + d\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}}, \frac{\frac{Y_1}{Z_1}\frac{Y_2}{Z_2} - \frac{X_1}{Z_1}\frac{X_2}{Z_2}}{1 - d\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}}\right) =$$

$$\left(\frac{Z_1 Z_2(X_1 Y_2 + Y_1 X_2)}{Z_1^2 Z_2^2 + d X_1 X_2 Y_1 Y_2}, \frac{Z_1 Z_2(Y_1 Y_2 - X_1 X_2)}{Z_1^2 Z_2^2 - d X_1 X_2 Y_1 Y_2}\right)$$

i.e. $\left(\frac{X_1}{Z_1}, \frac{Y_1}{Z_1}\right) + \left(\frac{X_2}{Z_2}, \frac{Y_2}{Z_2}\right.$

$$= \left(\frac{X_3}{Z_3}, \frac{Y_3}{Z_3}\right)$$

where

$F = Z_1^2 Z_2^2 - d X_1 X_2 Y_1 Y_2,$

$G = Z_1^2 Z_2^2 + d X_1 X_2 Y_1 Y_2,$

$X_3 = Z_1 Z_2(X_1 Y_2 + Y_1 X_2)F$

$Y_3 = Z_1 Z_2(Y_1 Y_2 - X_1 X_2)G,$

$Z_3 = FG.$

Input to addition algorithm:

$X_1, Y_1, Z_1, X_2, Y_2, Z_2.$

Output from addition algorit

$X_3, Y_3, Z_3.$ No divisions nee

Addition now has to handle fractions as input:

$$\left(\frac{X_1}{Z_1},\frac{Y_1}{Z_1}\right)+\left(\frac{X_2}{Z_2},\frac{Y_2}{Z_2}\right)=$$

$$\left(\frac{\frac{X_1}{Z_1}\frac{Y_2}{Z_2}+\frac{Y_1}{Z_1}\frac{X_2}{Z_2}}{1+d\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}},\right.$$

$$\left.\frac{\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}-\frac{X_1}{Z_1}\frac{X_2}{Z_2}}{1-d\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}}\right)=$$

$$\left(\frac{Z_1Z_2(X_1Y_2+Y_1X_2)}{Z_1^2Z_2^2+dX_1X_2Y_1Y_2},\right.$$

$$\left.\frac{Z_1Z_2(Y_1Y_2-X_1X_2)}{Z_1^2Z_2^2-dX_1X_2Y_1Y_2}\right)$$

i.e. $\left(\frac{X_1}{Z_1},\frac{Y_1}{Z_1}\right)+\left(\frac{X_2}{Z_2},\frac{Y_2}{Z_2}\right)$

$=\left(\frac{X_3}{Z_3},\frac{Y_3}{Z_3}\right)$

where

$F=Z_1^2Z_2^2-dX_1X_2Y_1Y_2$,
$G=Z_1^2Z_2^2+dX_1X_2Y_1Y_2$,
$X_3=Z_1Z_2(X_1Y_2+Y_1X_2)F$,
$Y_3=Z_1Z_2(Y_1Y_2-X_1X_2)G$,
$Z_3=FG$.

Input to addition algorithm:
X_1,Y_1,Z_1,X_2,Y_2,Z_2.
Output from addition algorithm:
X_3,Y_3,Z_3. No divisions needed!

n now has to

ractions as input:

$$1\Big) + \left(\frac{X_2}{Z_2}, \frac{Y_2}{Z_2}\right) =$$

$$\frac{2}{2} + \frac{Y_1}{Z_1}\frac{X_2}{Z_2}$$

$$\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2},$$

$$\frac{2}{2} - \frac{X_1}{Z_1}\frac{X_2}{Z_2}$$

$$\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}\Big) =$$

$$(X_1Y_2 + Y_1X_2)$$

$$+ dX_1X_2Y_1Y_2,$$

$$(Y_1Y_2 - X_1X_2)$$

$$- dX_1X_2Y_1Y_2\Big)$$

i.e. $\left(\frac{X_1}{Z_1}, \frac{Y_1}{Z_1}\right) + \left(\frac{X_2}{Z_2}, \frac{Y_2}{Z_2}\right)$

$= \left(\frac{X_3}{Z_3}, \frac{Y_3}{Z_3}\right)$

where

$F = Z_1^2 Z_2^2 - dX_1X_2Y_1Y_2,$

$G = Z_1^2 Z_2^2 + dX_1X_2Y_1Y_2,$

$X_3 = Z_1Z_2(X_1Y_2 + Y_1X_2)F,$

$Y_3 = Z_1Z_2(Y_1Y_2 - X_1X_2)G,$

$Z_3 = FG.$

Input to addition algorithm:

$X_1, Y_1, Z_1, X_2, Y_2, Z_2.$

Output from addition algorithm:

$X_3, Y_3, Z_3.$ No divisions needed!

Eliminat

to save

$A = Z_1$

$C = X_1$

$D = Y_1$

$E = d \cdot$

$F = B -$

$X_3 = A$

$Y_3 = A \cdot$

$Z_3 = F$

Cost: 11

M, **S** are

Choose

to

s input:

$$\frac{Y_2}{Z_2}\Big) =$$

$$\Big) =$$

$$\frac{{}_1X_2)}{Y_1Y_2},$$

$$\frac{{}_1X_2)}{Y_1Y_2}\Big)$$

i.e. $\left(\frac{X_1}{Z_1}, \frac{Y_1}{Z_1}\right) + \left(\frac{X_2}{Z_2}, \frac{Y_2}{Z_2}\right)$

$= \left(\frac{X_3}{Z_3}, \frac{Y_3}{Z_3}\right)$

where

$F = Z_1^2 Z_2^2 - dX_1X_2Y_1Y_2,$
$G = Z_1^2 Z_2^2 + dX_1X_2Y_1Y_2,$
$X_3 = Z_1Z_2(X_1Y_2 + Y_1X_2)F,$
$Y_3 = Z_1Z_2(Y_1Y_2 - X_1X_2)G,$
$Z_3 = FG.$

Input to addition algorithm:
$X_1, Y_1, Z_1, X_2, Y_2, Z_2.$
Output from addition algorithm:
$X_3, Y_3, Z_3.$ No divisions needed!

Eliminate commor
to save multiplicat

$A = Z_1 \cdot Z_2;\ B =$
$C = X_1 \cdot X_2;$
$D = Y_1 \cdot Y_2;$
$E = d \cdot C \cdot D;$
$F = B - E;\ G =$
$X_3 = A \cdot F \cdot (X_1 \cdot$
$Y_3 = A \cdot G \cdot (D -$
$Z_3 = F \cdot G.$

Cost: $11\mathbf{M} + 1\mathbf{S} +$
$\mathbf{M}, \mathbf{S}$ are costs of
Choose small d fo

i.e. $\left(\frac{X_1}{Z_1}, \frac{Y_1}{Z_1}\right) + \left(\frac{X_2}{Z_2}, \frac{Y_2}{Z_2}\right)$

$= \left(\frac{X_3}{Z_3}, \frac{Y_3}{Z_3}\right)$

where

$F = Z_1^2 Z_2^2 - dX_1X_2Y_1Y_2,$
$G = Z_1^2 Z_2^2 + dX_1X_2Y_1Y_2,$
$X_3 = Z_1Z_2(X_1Y_2 + Y_1X_2)F,$
$Y_3 = Z_1Z_2(Y_1Y_2 - X_1X_2)G,$
$Z_3 = FG.$

Input to addition algorithm:
$X_1, Y_1, Z_1, X_2, Y_2, Z_2.$
Output from addition algorithm:
$X_3, Y_3, Z_3.$ No divisions needed!

Eliminate common subexpre
to save multiplications:

$A = Z_1 \cdot Z_2$; $B = A^2$;
$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$E = d \cdot C \cdot D$;
$F = B - E$; $G = B + E$;
$X_3 = A \cdot F \cdot (X_1 \cdot Y_2 + Y_1 \cdot$
$Y_3 = A \cdot G \cdot (D - C)$;
$Z_3 = F \cdot G.$

Cost: $11\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$ wh
$\mathbf{M}, \mathbf{S}$ are costs of mult, squa
Choose small d for cheap $\mathbf{M}$

i.e. $\left(\frac{X_1}{Z_1}, \frac{Y_1}{Z_1}\right) + \left(\frac{X_2}{Z_2}, \frac{Y_2}{Z_2}\right)$

$= \left(\frac{X_3}{Z_3}, \frac{Y_3}{Z_3}\right)$

where

$F = Z_1^2 Z_2^2 - dX_1X_2Y_1Y_2$,

$G = Z_1^2 Z_2^2 + dX_1X_2Y_1Y_2$,

$X_3 = Z_1Z_2(X_1Y_2 + Y_1X_2)F$,

$Y_3 = Z_1Z_2(Y_1Y_2 - X_1X_2)G$,

$Z_3 = FG$.

Input to addition algorithm:
$X_1, Y_1, Z_1, X_2, Y_2, Z_2$.

Output from addition algorithm:
X_3, Y_3, Z_3. No divisions needed!

Eliminate common subexpressions to save multiplications:

$A = Z_1 \cdot Z_2$; $B = A^2$;

$C = X_1 \cdot X_2$;

$D = Y_1 \cdot Y_2$;

$E = d \cdot C \cdot D$;

$F = B - E$; $G = B + E$;

$X_3 = A \cdot F \cdot (X_1 \cdot Y_2 + Y_1 \cdot X_2)$;

$Y_3 = A \cdot G \cdot (D - C)$;

$Z_3 = F \cdot G$.

Cost: $11\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$ where $\mathbf{M}, \mathbf{S}$ are costs of mult, square. Choose small d for cheap $\mathbf{M}_d$.

$, \frac{Y_1}{Z_1}\Big) + \Big(\frac{X_2}{Z_2}, \frac{Y_2}{Z_2}\Big)$

$\frac{Y_3}{Z_3}\Big)$

$Z_2^2 - dX_1X_2Y_1Y_2,$

$Z_2^2 + dX_1X_2Y_1Y_2,$

$Z_2(X_1Y_2 + Y_1X_2)F,$

$Z_2(Y_1Y_2 - X_1X_2)G,$

$G.$

addition algorithm:

$Z_1, X_2, Y_2, Z_2.$

from addition algorithm:

Z_3. No divisions needed!

Eliminate common subexpressions to save multiplications:

$A = Z_1 \cdot Z_2;\ B = A^2;$

$C = X_1 \cdot X_2;$

$D = Y_1 \cdot Y_2;$

$E = d \cdot C \cdot D;$

$F = B - E;\ G = B + E;$

$X_3 = A \cdot F \cdot (X_1 \cdot Y_2 + Y_1 \cdot X_2);$

$Y_3 = A \cdot G \cdot (D - C);$

$Z_3 = F \cdot G.$

Cost: $11\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$ where $\mathbf{M}, \mathbf{S}$ are costs of mult, square. Choose small d for cheap $\mathbf{M}_d$.

Can do

Obvious

compute

of polys

$C = X_1$

$D = Y_1$

$M = X_1$

$\left(\frac{X_2}{Z_2}, \frac{Y_2}{Z_2}\right)$

$X_2 Y_1 Y_2,$

$X_2 Y_1 Y_2,$

$+ Y_1 X_2)F,$

$- X_1 X_2)G,$

algorithm:

$Z_2.$

tion algorithm:

isions needed!

Eliminate common subexpressions to save multiplications:

$A = Z_1 \cdot Z_2$; $B = A^2$;
$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$E = d \cdot C \cdot D$;
$F = B - E$; $G = B + E$;
$X_3 = A \cdot F \cdot (X_1 \cdot Y_2 + Y_1 \cdot X_2)$;
$Y_3 = A \cdot G \cdot (D - C)$;
$Z_3 = F \cdot G$.

Cost: $11\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$ where $\mathbf{M}, \mathbf{S}$ are costs of mult, square. Choose small d for cheap $\mathbf{M}_d$.

Can do better: 10

Obvious $4\mathbf{M}$ meth

compute product (

of polys $X_1 + Y_1 t,$

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = X_1 \cdot Y_2 + Y_1$

)

,

thm:

ded!

Eliminate common subexpressions to save multiplications:

$A = Z_1 \cdot Z_2$; $B = A^2$;
$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$E = d \cdot C \cdot D$;
$F = B - E$; $G = B + E$;
$X_3 = A \cdot F \cdot (X_1 \cdot Y_2 + Y_1 \cdot X_2)$;
$Y_3 = A \cdot G \cdot (D - C)$;
$Z_3 = F \cdot G$.

Cost: $11\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$ where $\mathbf{M}, \mathbf{S}$ are costs of mult, square. Choose small d for cheap $\mathbf{M}_d$.

Can do better: $10\mathbf{M} + 1\mathbf{S} +$

Obvious $4\mathbf{M}$ method to compute product $C + Mt +$ of polys $X_1 + Y_1 t$, $X_2 + Y_2 t$

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = X_1 \cdot Y_2 + Y_1 \cdot X_2$.

Eliminate common subexpressions to save multiplications:

$A = Z_1 \cdot Z_2$; $B = A^2$;
$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$E = d \cdot C \cdot D$;
$F = B - E$; $G = B + E$;
$X_3 = A \cdot F \cdot (X_1 \cdot Y_2 + Y_1 \cdot X_2)$;
$Y_3 = A \cdot G \cdot (D - C)$;
$Z_3 = F \cdot G$.

Cost: $11\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$ where $\mathbf{M}, \mathbf{S}$ are costs of mult, square. Choose small d for cheap $\mathbf{M}_d$.

Can do better: $10\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$.

Obvious $4\mathbf{M}$ method to compute product $C + Mt + Dt^2$ of polys $X_1 + Y_1 t$, $X_2 + Y_2 t$:

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = X_1 \cdot Y_2 + Y_1 \cdot X_2$.

Eliminate common subexpressions to save multiplications:

$A = Z_1 \cdot Z_2$; $B = A^2$;
$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$E = d \cdot C \cdot D$;
$F = B - E$; $G = B + E$;
$X_3 = A \cdot F \cdot (X_1 \cdot Y_2 + Y_1 \cdot X_2)$;
$Y_3 = A \cdot G \cdot (D - C)$;
$Z_3 = F \cdot G$.

Cost: $11\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$ where $\mathbf{M}, \mathbf{S}$ are costs of mult, square. Choose small d for cheap $\mathbf{M}_d$.

Can do better: $10\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$.

Obvious $4\mathbf{M}$ method to compute product $C + Mt + Dt^2$ of polys $X_1 + Y_1 t$, $X_2 + Y_2 t$:

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = X_1 \cdot Y_2 + Y_1 \cdot X_2$.

Karatsuba's $3\mathbf{M}$ method:

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = (X_1 + Y_1) \cdot (X_2 + Y_2) - C - D$.

e common subexpressions
multiplications:

$\cdot Z_2$; $B = A^2$;

$\cdot X_2$;

$\cdot Y_2$;

$C \cdot D$;

$- E$; $G = B + E$;

$\cdot F \cdot (X_1 \cdot Y_2 + Y_1 \cdot X_2)$;

$G \cdot (D - C)$;

$\cdot G$.

$1\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$ where
e costs of mult, square.
small d for cheap $\mathbf{M}_d$.

Can do better: $10\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$.

Obvious $4\mathbf{M}$ method to compute product $C + Mt + Dt^2$ of polys $X_1 + Y_1 t$, $X_2 + Y_2 t$:

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = X_1 \cdot Y_2 + Y_1 \cdot X_2$.

Karatsuba's $3\mathbf{M}$ method:

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = (X_1 + Y_1) \cdot (X_2 + Y_2) - C - D$.

Faster d

(x_1, y_1)

$((x_1 y_1 +$

$(y_1 y_1 -$

$((2x_1 y_1)$

$(y_1^2 - x_1^2$

n subexpressions

tions:

A^2;

$B + E$;

$Y_2 + Y_1 \cdot X_2)$;

$C)$;

$+ 1\mathbf{M}_d$ where

mult, square.

r cheap $\mathbf{M}_d$.

Can do better: $10\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$.

Obvious $4\mathbf{M}$ method to compute product $C + Mt + Dt^2$ of polys $X_1 + Y_1 t$, $X_2 + Y_2 t$:

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = X_1 \cdot Y_2 + Y_1 \cdot X_2$.

Karatsuba's $3\mathbf{M}$ method:

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = (X_1 + Y_1) \cdot (X_2 + Y_2) - C - D$.

Faster doubling

$(x_1, y_1) + (x_1, y_1)$
$((x_1 y_1 + y_1 x_1)/(1+$
$(y_1 y_1 - x_1 x_1)/(1-$
$((2x_1 y_1)/(1 + dx_1^2$
$(y_1^2 - x_1^2)/(1 - dx$

ssions

$X_2)$;

ere

are.

$\mathbf{M}_d$.

Can do better: $10\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$.

Obvious $4\mathbf{M}$ method to compute product $C + Mt + Dt^2$ of polys $X_1 + Y_1 t$, $X_2 + Y_2 t$:

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = X_1 \cdot Y_2 + Y_1 \cdot X_2$.

Karatsuba's $3\mathbf{M}$ method:

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = (X_1 + Y_1) \cdot (X_2 + Y_2) - C - D$.

Faster doubling

$(x_1, y_1) + (x_1, y_1) =$
$((x_1 y_1 + y_1 x_1)/(1 + d x_1 x_1 y_1 y_1$
$(y_1 y_1 - x_1 x_1)/(1 - d x_1 x_1 y_1 y_1$
$((2 x_1 y_1)/(1 + d x_1^2 y_1^2),$
$(y_1^2 - x_1^2)/(1 - d x_1^2 y_1^2))$.

Can do better: $10\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$.

Obvious $4\mathbf{M}$ method to compute product $C + Mt + Dt^2$ of polys $X_1 + Y_1 t$, $X_2 + Y_2 t$:

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = X_1 \cdot Y_2 + Y_1 \cdot X_2$.

Karatsuba's $3\mathbf{M}$ method:

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = (X_1 + Y_1) \cdot (X_2 + Y_2) - C - D$.

Faster doubling

$(x_1, y_1) + (x_1, y_1) =$
$((x_1 y_1 + y_1 x_1)/(1 + d x_1 x_1 y_1 y_1),$
$(y_1 y_1 - x_1 x_1)/(1 - d x_1 x_1 y_1 y_1)) =$
$((2 x_1 y_1)/(1 + d x_1^2 y_1^2),$
$(y_1^2 - x_1^2)/(1 - d x_1^2 y_1^2))$.

Can do better: $10\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$.

Obvious $4\mathbf{M}$ method to compute product $C + Mt + Dt^2$ of polys $X_1 + Y_1 t$, $X_2 + Y_2 t$:

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = X_1 \cdot Y_2 + Y_1 \cdot X_2$.

Karatsuba's $3\mathbf{M}$ method:

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = (X_1 + Y_1) \cdot (X_2 + Y_2) - C - D$.

Faster doubling

$$(x_1, y_1) + (x_1, y_1) =$$
$$((x_1 y_1 + y_1 x_1)/(1 + d x_1 x_1 y_1 y_1),$$
$$(y_1 y_1 - x_1 x_1)/(1 - d x_1 x_1 y_1 y_1)) =$$
$$((2 x_1 y_1)/(1 + d x_1^2 y_1^2),$$
$$(y_1^2 - x_1^2)/(1 - d x_1^2 y_1^2)).$$

$x_1^2 + y_1^2 = 1 + d x_1^2 y_1^2$ so

$$(x_1, y_1) + (x_1, y_1) =$$
$$((2 x_1 y_1)/(x_1^2 + y_1^2),$$
$$(y_1^2 - x_1^2)/(2 - x_1^2 - y_1^2)).$$

Can do better: $10\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$.

Obvious $4\mathbf{M}$ method to compute product $C + Mt + Dt^2$ of polys $X_1 + Y_1 t$, $X_2 + Y_2 t$:

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = X_1 \cdot Y_2 + Y_1 \cdot X_2$.

Karatsuba's $3\mathbf{M}$ method:

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = (X_1 + Y_1) \cdot (X_2 + Y_2) - C - D$.

Faster doubling

$(x_1, y_1) + (x_1, y_1) =$
$((x_1 y_1 + y_1 x_1)/(1 + d x_1 x_1 y_1 y_1),$
$(y_1 y_1 - x_1 x_1)/(1 - d x_1 x_1 y_1 y_1)) =$
$((2 x_1 y_1)/(1 + d x_1^2 y_1^2),$
$(y_1^2 - x_1^2)/(1 - d x_1^2 y_1^2))$.

$x_1^2 + y_1^2 = 1 + d x_1^2 y_1^2$ so
$(x_1, y_1) + (x_1, y_1) =$
$((2 x_1 y_1)/(x_1^2 + y_1^2),$
$(y_1^2 - x_1^2)/(2 - x_1^2 - y_1^2))$.

Again eliminate divisions using $(X : Y : Z)$: only $3\mathbf{M} + 4\mathbf{S}$. Much faster than addition.

better: $10\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$.

$4\mathbf{M}$ method to

e product $C + Mt + Dt^2$

$X_1 + Y_1 t$, $X_2 + Y_2 t$:

$\cdot X_2$;

$\cdot Y_2$;

$\cdot Y_2 + Y_1 \cdot X_2$.

pa's $3\mathbf{M}$ method:

$\cdot X_2$;

$\cdot Y_2$;

$_1 + Y_1) \cdot (X_2 + Y_2) - C - D$.

Faster doubling

$(x_1, y_1) + (x_1, y_1) =$
$((x_1 y_1 + y_1 x_1)/(1 + d x_1 x_1 y_1 y_1),$
$(y_1 y_1 - x_1 x_1)/(1 - d x_1 x_1 y_1 y_1)) =$
$((2 x_1 y_1)/(1 + d x_1^2 y_1^2),$
$(y_1^2 - x_1^2)/(1 - d x_1^2 y_1^2))$.

$x_1^2 + y_1^2 = 1 + d x_1^2 y_1^2$ so
$(x_1, y_1) + (x_1, y_1) =$
$((2 x_1 y_1)/(x_1^2 + y_1^2),$
$(y_1^2 - x_1^2)/(2 - x_1^2 - y_1^2))$.

Again eliminate divisions
using $(X : Y : Z)$: only $3\mathbf{M} + 4\mathbf{S}$.
Much faster than addition.

More ad

Dual ad

(x_1, y_1) +

$((x_1 y_1 +$

$(x_1 y_1 -$

Low deg

$\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$.

od to

$C + Mt + Dt^2$

$X_2 + Y_2 t$:

$\cdot X_2$.

ethod:

$X_2+Y_2) - C - D$.

Faster doubling

$(x_1, y_1) + (x_1, y_1) =$
$((x_1y_1+y_1x_1)/(1+dx_1x_1y_1y_1),$
$(y_1y_1-x_1x_1)/(1-dx_1x_1y_1y_1)) =$
$((2x_1y_1)/(1+dx_1^2y_1^2),$
$(y_1^2-x_1^2)/(1-dx_1^2y_1^2))$.

$x_1^2 + y_1^2 = 1 + dx_1^2y_1^2$ so
$(x_1, y_1) + (x_1, y_1) =$
$((2x_1y_1)/(x_1^2+y_1^2),$
$(y_1^2-x_1^2)/(2-x_1^2-y_1^2))$.

Again eliminate divisions
using $(X : Y : Z)$: only $3\mathbf{M} + 4\mathbf{S}$.
Much faster than addition.

More addition stra

Dual addition form

$(x_1, y_1) + (x_2, y_2)$
$((x_1y_1 + x_2y_2)/(x_1$
$(x_1y_1 - x_2y_2)/(x_1$

Low degree, no ne

$+ 1\mathbf{M}_d.$

Dt^2

:

$C - D.$

Faster doubling

$(x_1, y_1) + (x_1, y_1) =$
$((x_1y_1+y_1x_1)/(1+dx_1x_1y_1y_1),$
$(y_1y_1-x_1x_1)/(1-dx_1x_1y_1y_1)) =$
$((2x_1y_1)/(1+dx_1^2y_1^2),$
$(y_1^2-x_1^2)/(1-dx_1^2y_1^2)).$

$x_1^2 + y_1^2 = 1 + dx_1^2y_1^2$ so
$(x_1, y_1) + (x_1, y_1) =$
$((2x_1y_1)/(x_1^2 + y_1^2),$
$(y_1^2-x_1^2)/(2 - x_1^2 - y_1^2)).$

Again eliminate divisions
using $(X : Y : Z)$: only $3\mathbf{M} + 4\mathbf{S}$.
Much faster than addition.

More addition strategies

Dual addition formula:
$(x_1, y_1) + (x_2, y_2) =$
$((x_1y_1 + x_2y_2)/(x_1x_2 + y_1y_2$
$(x_1y_1 - x_2y_2)/(x_1y_2 - x_2y_1$
Low degree, no need for d.

Faster doubling

$(x_1, y_1) + (x_1, y_1) =$
$((x_1 y_1 + y_1 x_1)/(1 + d x_1 x_1 y_1 y_1),$
$(y_1 y_1 - x_1 x_1)/(1 - d x_1 x_1 y_1 y_1)) =$
$((2 x_1 y_1)/(1 + d x_1^2 y_1^2),$
$(y_1^2 - x_1^2)/(1 - d x_1^2 y_1^2)).$

$x_1^2 + y_1^2 = 1 + d x_1^2 y_1^2$ so
$(x_1, y_1) + (x_1, y_1) =$
$((2 x_1 y_1)/(x_1^2 + y_1^2),$
$(y_1^2 - x_1^2)/(2 - x_1^2 - y_1^2)).$

Again eliminate divisions
using $(X : Y : Z)$: only $3\mathbf{M} + 4\mathbf{S}$.
Much faster than addition.

More addition strategies

Dual addition formula:
$(x_1, y_1) + (x_2, y_2) =$
$((x_1 y_1 + x_2 y_2)/(x_1 x_2 + y_1 y_2),$
$(x_1 y_1 - x_2 y_2)/(x_1 y_2 - x_2 y_1)).$
Low degree, no need for d.

Faster doubling

$(x_1, y_1) + (x_1, y_1) =$
$((x_1 y_1 + y_1 x_1)/(1 + d x_1 x_1 y_1 y_1),$
$(y_1 y_1 - x_1 x_1)/(1 - d x_1 x_1 y_1 y_1)) =$
$((2 x_1 y_1)/(1 + d x_1^2 y_1^2),$
$(y_1^2 - x_1^2)/(1 - d x_1^2 y_1^2)).$

$x_1^2 + y_1^2 = 1 + d x_1^2 y_1^2$ so
$(x_1, y_1) + (x_1, y_1) =$
$((2 x_1 y_1)/(x_1^2 + y_1^2),$
$(y_1^2 - x_1^2)/(2 - x_1^2 - y_1^2)).$

Again eliminate divisions using $(X : Y : Z)$: only $3\mathbf{M} + 4\mathbf{S}$. Much faster than addition.

More addition strategies

Dual addition formula:
$(x_1, y_1) + (x_2, y_2) =$
$((x_1 y_1 + x_2 y_2)/(x_1 x_2 + y_1 y_2),$
$(x_1 y_1 - x_2 y_2)/(x_1 y_2 - x_2 y_1)).$
Low degree, no need for d.

Warning: fails for doubling! Is this really "addition"? Most EC formulas have failures.

Faster doubling

$(x_1, y_1) + (x_1, y_1) =$
$((x_1 y_1 + y_1 x_1)/(1 + d x_1 x_1 y_1 y_1),$
$(y_1 y_1 - x_1 x_1)/(1 - d x_1 x_1 y_1 y_1)) =$
$((2x_1 y_1)/(1 + d x_1^2 y_1^2),$
$(y_1^2 - x_1^2)/(1 - d x_1^2 y_1^2)).$

$x_1^2 + y_1^2 = 1 + d x_1^2 y_1^2$ so
$(x_1, y_1) + (x_1, y_1) =$
$((2x_1 y_1)/(x_1^2 + y_1^2),$
$(y_1^2 - x_1^2)/(2 - x_1^2 - y_1^2)).$

Again eliminate divisions
using $(X : Y : Z)$: only $3\mathbf{M} + 4\mathbf{S}$.
Much faster than addition.

More addition strategies

Dual addition formula:
$(x_1, y_1) + (x_2, y_2) =$
$((x_1 y_1 + x_2 y_2)/(x_1 x_2 + y_1 y_2),$
$(x_1 y_1 - x_2 y_2)/(x_1 y_2 - x_2 y_1)).$
Low degree, no need for d.

Warning: fails for doubling!
Is this really “addition”?
Most EC formulas have failures.

Can test for failure cases.
Can produce constant-time code
by eliminating branches.
For some ECC ops, can prove
that failure cases never happen.

oubling

$+ (x_1, y_1) =$
$y_1x_1)/(1{+}dx_1x_1y_1y_1),$
$x_1x_1)/(1{-}dx_1x_1y_1y_1)) =$
$/(1 + dx_1^2y_1^2),$
$)/(1 - dx_1^2y_1^2)).$

$= 1 + dx_1^2y_1^2$ so
$+ (x_1, y_1) =$
$/(x_1^2 + y_1^2),$
$)/(2 - x_1^2 - y_1^2)).$

iminate divisions
$: Y : Z)$: only $3\mathbf{M} + 4\mathbf{S}$.
ster than addition.

More addition strategies

Dual addition formula:

$$(x_1, y_1) + (x_2, y_2) =$$
$$((x_1y_1 + x_2y_2)/(x_1x_2 + y_1y_2),$$
$$(x_1y_1 - x_2y_2)/(x_1y_2 - x_2y_1)).$$

Low degree, no need for d.

Warning: fails for doubling!
Is this really “addition”?
Most EC formulas have failures.

Can test for failure cases.
Can produce constant-time code by eliminating branches.
For some ECC ops, can prove that failure cases never happen.

More co

- inverte
- extend
- compl

“$-$1-twi
$-x^2 + y$
further s
$-x^2 + y$

Inside m
$8\mathbf{M}$ for a
$3\mathbf{M} + 4\mathbf{S}$

$=$

$dx_1x_1y_1y_1),$

$dx_1x_1y_1y_1)) =$

$y_1^2),$

$x_1^2y_1^2)).$

y_1^2 so

$=$

$),$

$- y_1^2)).$

visions

only $3\mathbf{M} + 4\mathbf{S}$.

addition.

More addition strategies

Dual addition formula:

$(x_1, y_1) + (x_2, y_2) =$

$((x_1y_1 + x_2y_2)/(x_1x_2 + y_1y_2),$

$(x_1y_1 - x_2y_2)/(x_1y_2 - x_2y_1)).$

Low degree, no need for d.

Warning: fails for doubling!

Is this really “addition”?

Most EC formulas have failures.

Can test for failure cases.

Can produce constant-time code by eliminating branches.

For some ECC ops, can prove that failure cases never happen.

More coordinate s

- inverted: $x = Z$
- extended: $x = X$
- completed: $x =$

$xy =$

“-1-twisted Edwa

$-x^2 + y^2 = 1 + d$

further speedups r

$-x^2 + y^2 = (y -$

Inside modern EC

$8\mathbf{M}$ for addition,

$3\mathbf{M} + 4\mathbf{S}$ for doub

$_1),$

$_1)) =$

$+ 4\mathbf{S}.$

More addition strategies

Dual addition formula:

$(x_1, y_1) + (x_2, y_2) =$
$((x_1 y_1 + x_2 y_2)/(x_1 x_2 + y_1 y_2),$
$(x_1 y_1 - x_2 y_2)/(x_1 y_2 - x_2 y_1)).$

Low degree, no need for d.

Warning: fails for doubling!

Is this really "addition"?

Most EC formulas have failures.

Can test for failure cases.

Can produce constant-time code by eliminating branches.

For some ECC ops, can prove that failure cases never happen.

More coordinate systems: e.

- inverted: $x = Z/X$, $y = Z$
- extended: $x = X/Z$, $y =$
- completed: $x = X/Z$, $y =$
  $xy = T/Z$.

"-1-twisted Edwards curves
$-x^2 + y^2 = 1 + dx^2y^2$:
further speedups related to
$-x^2 + y^2 = (y - x)(y + x)$.

Inside modern ECC operatio
8$\mathbf{M}$ for addition,
3$\mathbf{M}$ + 4$\mathbf{S}$ for doubling.

More addition strategies

Dual addition formula:

$(x_1, y_1) + (x_2, y_2) =$
$((x_1 y_1 + x_2 y_2)/(x_1 x_2 + y_1 y_2),$
$(x_1 y_1 - x_2 y_2)/(x_1 y_2 - x_2 y_1)).$

Low degree, no need for d.

Warning: fails for doubling!
Is this really “addition”?
Most EC formulas have failures.

Can test for failure cases.
Can produce constant-time code by eliminating branches.
For some ECC ops, can prove that failure cases never happen.

More coordinate systems: e.g.,

- inverted: $x = Z/X$, $y = Z/Y$.
- extended: $x = X/Z$, $y = Y/T$.
- completed: $x = X/Z$, $y = Y/Z$, $xy = T/Z$.

“-1-twisted Edwards curves”
$-x^2 + y^2 = 1 + dx^2y^2$:
further speedups related to
$-x^2 + y^2 = (y - x)(y + x)$.

Inside modern ECC operations:
8**M** for addition,
3**M** + 4**S** for doubling.

dition strategies

dition formula:

$+ (x_2, y_2) =$

$\cdot x_2y_2)/(x_1x_2 + y_1y_2),$

$\cdot x_2y_2)/(x_1y_2 - x_2y_1)).$

gree, no need for d.

g: fails for doubling!

eally "addition"?

C formulas have failures.

t for failure cases.

duce constant-time code

nating branches.

e ECC ops, can prove

ure cases never happen.

More coordinate systems: e.g.,

- inverted: $x = Z/X$, $y = Z/Y$.
- extended: $x = X/Z$, $y = Y/T$.
- completed: $x = X/Z$, $y = Y/Z$, $xy = T/Z$.

"-1-twisted Edwards curves"
$-x^2 + y^2 = 1 + dx^2y^2$:
further speedups related to
$-x^2 + y^2 = (y - x)(y + x)$.

Inside modern ECC operations:
$8\mathbf{M}$ for addition,
$3\mathbf{M} + 4\mathbf{S}$ for doubling.

NIST cu

were sta

Edwards

Much sl

ategies

nula:

=

$_1x_2 + y_1y_2)$,

$_1y_2 - x_2y_1))$.

eed for d.

doubling!

tion"?

have failures.

e cases.

tant-time code

nches.

s, can prove

never happen.

More coordinate systems: e.g.,

- inverted: $x = Z/X$, $y = Z/Y$.
- extended: $x = X/Z$, $y = Y/T$.
- completed: $x = X/Z$, $y = Y/Z$, $xy = T/Z$.

"−1-twisted Edwards curves"
$-x^2 + y^2 = 1 + dx^2y^2$:
further speedups related to
$-x^2 + y^2 = (y - x)(y + x)$.

Inside modern ECC operations:
8**M** for addition,
3**M** + 4**S** for doubling.

NIST curves (e.g.,
were standardized
Edwards curves we

Much slower addit

),

)).

ures.

code

ve

pen.

More coordinate systems: e.g.,

- inverted: $x = Z/X$, $y = Z/Y$.
- extended: $x = X/Z$, $y = Y/T$.
- completed: $x = X/Z$, $y = Y/Z$,
  $xy = T/Z$.

"-1-twisted Edwards curves"
$-x^2 + y^2 = 1 + dx^2y^2$:
further speedups related to
$-x^2 + y^2 = (y - x)(y + x)$.

Inside modern ECC operations:
$8\mathbf{M}$ for addition,
$3\mathbf{M} + 4\mathbf{S}$ for doubling.

NIST curves (e.g., P-256)
were standardized before
Edwards curves were publish

Much slower additions.

More coordinate systems: e.g.,

- inverted: $x = Z/X$, $y = Z/Y$.
- extended: $x = X/Z$, $y = Y/T$.
- completed: $x = X/Z$, $y = Y/Z$,
  $xy = T/Z$.

"-1-twisted Edwards curves"
$-x^2 + y^2 = 1 + dx^2y^2$:
further speedups related to
$-x^2 + y^2 = (y - x)(y + x)$.

Inside modern ECC operations:
$8\mathbf{M}$ for addition,
$3\mathbf{M} + 4\mathbf{S}$ for doubling.

NIST curves (e.g., P-256)
were standardized before
Edwards curves were published.

Much slower additions.

More coordinate systems: e.g.,

- inverted: $x = Z/X$, $y = Z/Y$.
- extended: $x = X/Z$, $y = Y/T$.
- completed: $x = X/Z$, $y = Y/Z$, $xy = T/Z$.

"-1-twisted Edwards curves" $-x^2 + y^2 = 1 + dx^2y^2$: further speedups related to $-x^2 + y^2 = (y - x)(y + x)$.

Inside modern ECC operations: $8\mathbf{M}$ for addition, $3\mathbf{M} + 4\mathbf{S}$ for doubling.

NIST curves (e.g., P-256) were standardized before Edwards curves were published.

Much slower additions.

Express as Edwards curves using a field extension: slow.

More coordinate systems: e.g.,

- inverted: $x = Z/X$, $y = Z/Y$.
- extended: $x = X/Z$, $y = Y/T$.
- completed: $x = X/Z$, $y = Y/Z$, $xy = T/Z$.

"-1-twisted Edwards curves" $-x^2 + y^2 = 1 + dx^2y^2$: further speedups related to $-x^2 + y^2 = (y - x)(y + x)$.

Inside modern ECC operations: $8\mathbf{M}$ for addition, $3\mathbf{M} + 4\mathbf{S}$ for doubling.

NIST curves (e.g., P-256) were standardized before Edwards curves were published.

Much slower additions.

Express as Edwards curves using a field extension: slow.

How did Curve25519 obtain good speeds for ECDH? "Montgomery curve with the Montgomery ladder."

More coordinate systems: e.g.,

- inverted: $x = Z/X$, $y = Z/Y$.
- extended: $x = X/Z$, $y = Y/T$.
- completed: $x = X/Z$, $y = Y/Z$, $xy = T/Z$.

"-1-twisted Edwards curves" $-x^2 + y^2 = 1 + dx^2y^2$: further speedups related to $-x^2 + y^2 = (y - x)(y + x)$.

Inside modern ECC operations:
8**M** for addition,
3**M** + 4**S** for doubling.

NIST curves (e.g., P-256) were standardized before Edwards curves were published.

Much slower additions.

Express as Edwards curves using a field extension: slow.

How did Curve25519 obtain good speeds for ECDH? "Montgomery curve with the Montgomery ladder."

Why did NIST not choose Montgomery curves? Unclear.