Curve25519, Curve41417, E-521

D. J. Bernstein
University of Illinois at Chicago &
Technische Universiteit Eindhoven

Curve25519 mod $p = 2^{255} - 19$:
$y^2 = x^3 + 486662x^2 + x$.

Equivalent to Edwards curve
$x^2 + y^2 = 1 + (1 - 1/121666)x^2y^2$.

Curve41417 mod $2^{414} - 17$:
$x^2 + y^2 = 1 + 3617x^2y^2$.

E-521 mod $2^{521} - 1$:
$x^2 + y^2 = 1 - 376014x^2y^2$.

Curve25519

Introduced in ECC 2005 talk and PKC 2006 paper "New Diffie–Hellman speed records."

Main features listed in paper:
"extremely high speed";
"no time variability";
32-byte secret keys;
32-byte public keys;
"free key validation";
"short code".

The big picture:
Minimize tensions between speed, simplicity, security.

519, Curve41417, E-521

ernstein
ty of Illinois at Chicago &
che Universiteit Eindhoven

---

519 mod $p = 2^{255} - 19$:
$+ 486662x^2 + x$.

ent to Edwards curve
$= 1 + (1 - 1/121666)x^2y^2$.

417 mod $2^{414} - 17$:
$= 1 + 3617x^2y^2$.

od $2^{521} - 1$:
$= 1 - 376014x^2y^2$.

Curve25519

Introduced in ECC 2005 talk and PKC 2006 paper "New Diffie–Hellman speed records."

Main features listed in paper:
"extremely high speed";
"no time variability";
32-byte secret keys;
32-byte public keys;
"free key validation";
"short code".

The big picture:
Minimize tensions between speed, simplicity, security.

Tension:

How wil
compute

Many bo
Passes i
But **vari**
presuma

e41417, E-521

is at Chicago &
siteit Eindhoven

$p = 2^{255} - 19$:
$x^2 + x$.

ards curve
$- 1/121666)x^2y^2$.

$2^{414} - 17$:
$17x^2y^2$.

$- 1$:
$6014x^2y^2$.

Curve25519

Introduced in ECC 2005 talk and PKC 2006 paper "New Diffie–Hellman speed records."

Main features listed in paper:
"extremely high speed";
"no time variability";
32-byte secret keys;
32-byte public keys;
"free key validation";
"short code".

The big picture:
Minimize tensions between speed, simplicity, security.

Tension: a neutral

How will implemen
compute a/b mod

Many books recon
Passes interoperab
But **variable time**
presumably a secu

-521

ago &

hoven

- 19:

e

$56)x^2y^2$.

Curve25519

Introduced in ECC 2005 talk and PKC 2006 paper "New Diffie–Hellman speed records."

Main features listed in paper:
"extremely high speed";
"no time variability";
32-byte secret keys;
32-byte public keys;
"free key validation";
"short code".

The big picture:
Minimize tensions between speed, simplicity, security.

Tension: a neutral example

How will implementors compute $a/b \bmod p$?

Many books recommend Eu
Passes interoperability tests.
But **variable time**,
presumably a security proble

Curve25519

Introduced in ECC 2005 talk and PKC 2006 paper "New Diffie–Hellman speed records."

Main features listed in paper:
"extremely high speed";
"no time variability";
32-byte secret keys;
32-byte public keys;
"free key validation";
"short code".

The big picture:
Minimize tensions between speed, simplicity, security.

Tension: a neutral example

How will implementors compute $a/b \bmod p$?

Many books recommend Euclid.
Passes interoperability tests.
But **variable time**,
presumably a security problem.

Curve25519

Introduced in ECC 2005 talk and PKC 2006 paper "New Diffie–Hellman speed records."

Main features listed in paper:
"extremely high speed";
"no time variability";
32-byte secret keys;
32-byte public keys;
"free key validation";
"short code".

The big picture:
Minimize tensions between speed, simplicity, security.

Tension: a neutral example

How will implementors compute a/b mod p?

Many books recommend Euclid.
Passes interoperability tests.
But **variable time**,
presumably a security problem.

Defense 1: Encourage implementors to use ab^{p-2}.
Simpler than Euclid, fast enough.

Curve25519

Introduced in ECC 2005 talk and PKC 2006 paper "New Diffie–Hellman speed records."

Main features listed in paper:
"extremely high speed";
"no time variability";
32-byte secret keys;
32-byte public keys;
"free key validation";
"short code".

The big picture:
Minimize tensions between speed, simplicity, security.

Tension: a neutral example

How will implementors compute $a/b \bmod p$?

Many books recommend Euclid.
Passes interoperability tests.
But **variable time**,
presumably a security problem.

Defense 1: Encourage implementors to use ab^{p-2}.
Simpler than Euclid, fast enough.

But maybe implementor finds it simplest to use a Euclid library, and wants the Euclid speed.

519

ed in ECC 2005 talk
C 2006 paper "New
ellman speed records."

atures listed in paper:
ely high speed";
e variability";
secret keys;
public keys;
y validation";
ode".

picture:
ze tensions between
simplicity, security.

Tension: a neutral example

How will implementors compute $a/b \bmod p$?

Many books recommend Euclid. Passes interoperability tests. But **variable time**, presumably a security problem.

Defense 1: Encourage implementors to use ab^{p-2}. Simpler than Euclid, fast enough.

But maybe implementor finds it simplest to use a Euclid library, and wants the Euclid speed.

Defense
impleme
verify co
e.g. 201
Almeida

C 2005 talk
per “New
eed records.”

ed in paper:
peed”;
y”;
s;
s;
n”;

ns between
. security.

Tension: a neutral example

How will implementors
compute $a/b \bmod p$?

Many books recommend Euclid.
Passes interoperability tests.
But **variable time**,
presumably a security problem.

Defense 1: Encourage
implementors to use ab^{p-2}.
Simpler than Euclid, fast enough.

But maybe implementor finds it
simplest to use a Euclid library,
and wants the Euclid speed.

Defense 2: Encou
implementors to u
verify constant-tim
e.g. 2010 Langley
Almeida–Barbosa–

k

s."

r:

n

Tension: a neutral example

How will implementors compute $a/b \bmod p$?

Many books recommend Euclid. Passes interoperability tests. But **variable time**, presumably a security problem.

Defense 1: Encourage implementors to use ab^{p-2}. Simpler than Euclid, fast enough.

But maybe implementor finds it simplest to use a Euclid library, and wants the Euclid speed.

Defense 2: Encourage implementors to use tools to verify constant-time behavio e.g. 2010 Langley "ctgrind"; Almeida–Barbosa–Pinto–Vie

Tension: a neutral example

How will implementors compute $a/b \bmod p$?

Many books recommend Euclid. Passes interoperability tests. But **variable time**, presumably a security problem.

Defense 1: Encourage implementors to use ab^{p-2}. Simpler than Euclid, fast enough.

But maybe implementor finds it simplest to use a Euclid library, and wants the Euclid speed.

Defense 2: Encourage implementors to use tools to verify constant-time behavior. e.g. 2010 Langley "ctgrind"; 2013 Almeida–Barbosa–Pinto–Vieira.

Tension: a neutral example

How will implementors compute $a/b \bmod p$?

Many books recommend Euclid. Passes interoperability tests. But **variable time**, presumably a security problem.

Defense 1: Encourage implementors to use ab^{p-2}. Simpler than Euclid, fast enough.

But maybe implementor finds it simplest to use a Euclid library, and wants the Euclid speed.

Defense 2: Encourage implementors to use tools to verify constant-time behavior. e.g. 2010 Langley "ctgrind"; 2013 Almeida–Barbosa–Pinto–Vieira.

Defense 3: Encourage implementors to use fractions (e.g., "projective coordinates"). Then Euclid speedup is negligible.

Tension: a neutral example

How will implementors compute $a/b \bmod p$?

Many books recommend Euclid. Passes interoperability tests. But **variable time**, presumably a security problem.

Defense 1: Encourage implementors to use ab^{p-2}. Simpler than Euclid, fast enough.

But maybe implementor finds it simplest to use a Euclid library, and wants the Euclid speed.

Defense 2: Encourage implementors to use tools to verify constant-time behavior. e.g. 2010 Langley "ctgrind"; 2013 Almeida–Barbosa–Pinto–Vieira.

Defense 3: Encourage implementors to use fractions (e.g., "projective coordinates"). Then Euclid speedup is negligible.

Defense 4: Choose curves that naturally avoid *all* divisions.

Tension: a neutral example

How will implementors compute $a/b \bmod p$?

Many books recommend Euclid. Passes interoperability tests. But **variable time**, presumably a security problem.

Defense 1: Encourage implementors to use ab^{p-2}. Simpler than Euclid, fast enough.

But maybe implementor finds it simplest to use a Euclid library, and wants the Euclid speed.

Defense 2: Encourage implementors to use tools to verify constant-time behavior. e.g. 2010 Langley "ctgrind"; 2013 Almeida–Barbosa–Pinto–Vieira.

Defense 3: Encourage implementors to use fractions (e.g., "projective coordinates"). Then Euclid speedup is negligible.

Defense 4: Choose curves that naturally avoid *all* divisions. Seems incompatible with ECC.

Tension: a neutral example

How will implementors compute $a/b \bmod p$?

Many books recommend Euclid. Passes interoperability tests. But **variable time**, presumably a security problem.

Defense 1: Encourage implementors to use ab^{p-2}. Simpler than Euclid, fast enough.

But maybe implementor finds it simplest to use a Euclid library, and wants the Euclid speed.

Defense 2: Encourage implementors to use tools to verify constant-time behavior. e.g. 2010 Langley "ctgrind"; 2013 Almeida–Barbosa–Pinto–Vieira.

Defense 3: Encourage implementors to use fractions (e.g., "projective coordinates"). Then Euclid speedup is negligible.

Defense 4: Choose curves that naturally avoid *all* divisions. Seems incompatible with ECC. The good news: curve choice *can* resolve other tensions.

a neutral example

l implementors
e a/b mod p?

ooks recommend Euclid.
nteroperability tests.
iable time,
bly a security problem.

1: Encourage
entors to use ab^{p-2}.
than Euclid, fast enough.

ybe implementor finds it
to use a Euclid library,
ts the Euclid speed.

Defense 2: Encourage
implementors to use tools to
verify constant-time behavior.
e.g. 2010 Langley "ctgrind"; 2013
Almeida–Barbosa–Pinto–Vieira.

Defense 3: Encourage
implementors to use fractions
(e.g., "projective coordinates").
Then Euclid speedup is negligible.

Defense 4: Choose curves that
naturally avoid *all* divisions.
Seems incompatible with ECC.
The good news: curve choice
can resolve other tensions.

Constan

Imitate
Allocate
for each

Always
on all bi

e.g. If yo
with 255
and 255
allocate

e.g. If yo
with 256
and 256
allocate

l example

ntors

p?

mmend Euclid.

bility tests.

e,

rity problem.

rage

se ab^{p-2}.

id, fast enough.

nentor finds it

Euclid library,

clid speed.

Defense 2: Encourage implementors to use tools to verify constant-time behavior. e.g. 2010 Langley "ctgrind"; 2013 Almeida–Barbosa–Pinto–Vieira.

Defense 3: Encourage implementors to use fractions (e.g., "projective coordinates"). Then Euclid speedup is negligible.

Defense 4: Choose curves that naturally avoid *all* divisions. Seems incompatible with ECC. The good news: curve choice *can* resolve other tensions.

Constant-time Cur

Imitate hardware i

Allocate constant

for each integer.

Always perform ar

on all bits. Don't

e.g. If you're addin

with 255 bits alloc

and 255 bits alloca

allocate 256 bits fo

e.g. If you're mult

with 256 bits alloc

and 256 bits alloca

allocate 512 bits fo

clid.

em.

ough.

ds it

ary,

Defense 2: Encourage implementors to use tools to verify constant-time behavior. e.g. 2010 Langley “ctgrind”; 2013 Almeida–Barbosa–Pinto–Vieira.

Defense 3: Encourage implementors to use fractions (e.g., “projective coordinates”). Then Euclid speedup is negligible.

Defense 4: Choose curves that naturally avoid *all* divisions. Seems incompatible with ECC. The good news: curve choice *can* resolve other tensions.

Constant-time Curve25519

Imitate hardware in software
Allocate constant number of
for each integer.

Always perform arithmetic
on all bits. Don’t skip bits.

e.g. If you’re adding a to b,
with 255 bits allocated for a
and 255 bits allocated for b:
allocate 256 bits for $a + b$.

e.g. If you’re multiplying a b
with 256 bits allocated for a
and 256 bits allocated for b:
allocate 512 bits for ab.

Defense 2: Encourage implementors to use tools to verify constant-time behavior. e.g. 2010 Langley "ctgrind"; 2013 Almeida–Barbosa–Pinto–Vieira.

Defense 3: Encourage implementors to use fractions (e.g., "projective coordinates"). Then Euclid speedup is negligible.

Defense 4: Choose curves that naturally avoid *all* divisions. Seems incompatible with ECC. The good news: curve choice *can* resolve other tensions.

Constant-time Curve25519

Imitate hardware in software. Allocate constant number of bits for each integer.

Always perform arithmetic on all bits. Don't skip bits.

e.g. If you're adding a to b, with 255 bits allocated for a and 255 bits allocated for b: allocate 256 bits for $a + b$.

e.g. If you're multiplying a by b, with 256 bits allocated for a and 256 bits allocated for b: allocate 512 bits for ab.

2: Encourage
ntors to use tools to
nstant-time behavior.
0 Langley "ctgrind"; 2013
–Barbosa–Pinto–Vieira.

3: Encourage
ntors to use fractions
rojective coordinates").
uclid speedup is negligible.

4: Choose curves that
avoid *all* divisions.
ncompatible with ECC.
d news: curve choice
lve other tensions.

Constant-time Curve25519

Imitate hardware in software.
Allocate constant number of bits
for each integer.

Always perform arithmetic
on all bits. Don't skip bits.

e.g. If you're adding a to b,
with 255 bits allocated for a
and 255 bits allocated for b:
allocate 256 bits for $a + b$.

e.g. If you're multiplying a by b,
with 256 bits allocated for a
and 256 bits allocated for b:
allocate 512 bits for ab.

If (e.g.)
Replace
$r = c$ m
Allocate
This is t

Repeat s
350 bits
Small er

rage
se tools to
ne behavior.
"ctgrind"; 2013
-Pinto–Vieira.

rage
se fractions
coordinates").
lup is negligible.

e curves that
divisions.
le with ECC.
urve choice
tensions.

Constant-time Curve25519

Imitate hardware in software. Allocate constant number of bits for each integer.

Always perform arithmetic on all bits. Don't skip bits.

e.g. If you're adding a to b, with 255 bits allocated for a and 255 bits allocated for b: allocate 256 bits for $a + b$.

e.g. If you're multiplying a by b, with 256 bits allocated for a and 256 bits allocated for b: allocate 512 bits for ab.

If (e.g.) 600 bits a
Replace c with 19
$r = c \bmod 2^{255}$, q
Allocate 350 bits f
This is the same n

Repeat same comp
350 bits $\rightarrow$ 256 bi
Small enough for

o

or.

; 2013

eira.

ns

s”).

igible.

hat

CC.

ce

Constant-time Curve25519

Imitate hardware in software.
Allocate constant number of bits for each integer.

Always perform arithmetic on all bits. Don’t skip bits.

e.g. If you’re adding a to b, with 255 bits allocated for a and 255 bits allocated for b: allocate 256 bits for $a + b$.

e.g. If you’re multiplying a by b, with 256 bits allocated for a and 256 bits allocated for b: allocate 512 bits for ab.

If (e.g.) 600 bits allocated fo
Replace c with $19q + r$ wher
$r = c \bmod 2^{255}$, $q = \lfloor c/2^{25}$
Allocate 350 bits for $19q +$
This is the same modulo p.

Repeat same compression:
350 bits $\to$ 256 bits.
Small enough for next mult.

Constant-time Curve25519

Imitate hardware in software.
Allocate constant number of bits for each integer.

Always perform arithmetic on all bits. Don't skip bits.

e.g. If you're adding a to b, with 255 bits allocated for a and 255 bits allocated for b: allocate 256 bits for $a + b$.

e.g. If you're multiplying a by b, with 256 bits allocated for a and 256 bits allocated for b: allocate 512 bits for ab.

If (e.g.) 600 bits allocated for c:
Replace c with $19q + r$ where $r = c \bmod 2^{255}$, $q = \lfloor c/2^{255} \rfloor$.
Allocate 350 bits for $19q + r$.
This is the same modulo p.

Repeat same compression:
350 bits $\to$ 256 bits.
Small enough for next mult.

Constant-time Curve25519

Imitate hardware in software. Allocate constant number of bits for each integer.

Always perform arithmetic on all bits. Don't skip bits.

e.g. If you're adding a to b, with 255 bits allocated for a and 255 bits allocated for b: allocate 256 bits for $a + b$.

e.g. If you're multiplying a by b, with 256 bits allocated for a and 256 bits allocated for b: allocate 512 bits for ab.

If (e.g.) 600 bits allocated for c: Replace c with $19q + r$ where $r = c \bmod 2^{255}$, $q = \lfloor c/2^{255} \rfloor$. Allocate 350 bits for $19q + r$. This is the same modulo p.

Repeat same compression: 350 bits $\rightarrow$ 256 bits. Small enough for next mult.

To **completely** reduce 256 bits mod p, do two iterations of constant-time conditional sub.

One conditional sub: replace c with $c - (1 - s)p$ where s is sign bit in $c - p$.

t-time Curve25519

hardware in software.
constant number of bits
integer.

perform arithmetic
ts. Don't skip bits.

ou're adding a to b,
5 bits allocated for a
bits allocated for b:
256 bits for $a + b$.

ou're multiplying a by b,
5 bits allocated for a
bits allocated for b:
512 bits for ab.

If (e.g.) 600 bits allocated for c:
Replace c with $19q + r$ where
$r = c \bmod 2^{255}$, $q = \lfloor c/2^{255} \rfloor$.
Allocate 350 bits for $19q + r$.
This is the same modulo p.

Repeat same compression:
350 bits $\rightarrow$ 256 bits.
Small enough for next mult.

To **completely** reduce 256 bits
mod p, do two iterations of
constant-time conditional sub.

One conditional sub:
replace c with $c - (1 - s)p$
where s is sign bit in $c - p$.

Constan

NIST P-
$2^{256} - 2$

ECDSA
reductio
an integ

Write A
$(A_{15}, A_1$
$A_8, A_7,$
meaning

Define
$T; S_1; S_2$
as

rve25519

n software.

number of bits

ithmetic

skip bits.

ng a to b,

cated for a

ated for b:

or $a + b$.

iplying a by b,

cated for a

ated for b:

or ab.

If (e.g.) 600 bits allocated for c:
Replace c with $19q + r$ where
$r = c \bmod 2^{255}$, $q = \lfloor c/2^{255} \rfloor$.
Allocate 350 bits for $19q + r$.
This is the same modulo p.

Repeat same compression:
350 bits $\rightarrow$ 256 bits.
Small enough for next mult.

To **completely** reduce 256 bits mod p, do two iterations of constant-time conditional sub.

One conditional sub:
replace c with $c - (1 - s)p$
where s is sign bit in $c - p$.

Constant-time NIS

NIST P-256 prime
$2^{256} - 2^{224} + 2^{192}$

ECDSA standard s
reduction procedu
an integer "A less

Write A as
$(A_{15}, A_{14}, A_{13}, A_{12}$
A_8, A_7, A_6, A_5, A
meaning $\sum_i A_i 2^{32}$

Define
$T; S_1; S_2; S_3; S_4; D$
as

e.

f bits

by b,

If (e.g.) 600 bits allocated for c:
Replace c with $19q + r$ where
$r = c \bmod 2^{255}$, $q = \lfloor c/2^{255} \rfloor$.
Allocate 350 bits for $19q + r$.
This is the same modulo p.

Repeat same compression:
350 bits $\rightarrow$ 256 bits.
Small enough for next mult.

To **completely** reduce 256 bits mod p, do two iterations of constant-time conditional sub.

One conditional sub:
replace c with $c - (1 - s)p$
where s is sign bit in $c - p$.

Constant-time NIST P-256

NIST P-256 prime p is
$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$

ECDSA standard specifies reduction procedure given an integer "A less than p^2":

Write A as
$(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, A_{10}$
$A_8, A_7, A_6, A_5, A_4, A_3, A_2,$
meaning $\sum_i A_i 2^{32i}$.

Define
$T; S_1; S_2; S_3; S_4; D_1; D_2; D_3$
as

If (e.g.) 600 bits allocated for c:
Replace c with $19q + r$ where
$r = c \bmod 2^{255}$, $q = \lfloor c/2^{255} \rfloor$.
Allocate 350 bits for $19q + r$.
This is the same modulo p.

Repeat same compression:
350 bits $\to$ 256 bits.
Small enough for next mult.

To **completely** reduce 256 bits mod p, do two iterations of constant-time conditional sub.

One conditional sub:
replace c with $c - (1 - s)p$
where s is sign bit in $c - p$.

Constant-time NIST P-256

NIST P-256 prime p is
$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$.

ECDSA standard specifies reduction procedure given an integer "A less than p^2":

Write A as
$(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, A_{10}, A_9,$
$A_8, A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)$,
meaning $\sum_i A_i 2^{32i}$.

Define
$T; S_1; S_2; S_3; S_4; D_1; D_2; D_3; D_4$
as

600 bits allocated for c:

c with $19q + r$ where

od 2^{255}, $q = \lfloor c/2^{255} \rfloor$.

350 bits for $19q + r$.

he same modulo p.

same compression:

$\rightarrow$ 256 bits.

ough for next mult.

pletely reduce 256 bits

do two iterations of

t-time conditional sub.

ditional sub:

c with $c - (1 - s)p$

is sign bit in $c - p$.

Constant-time NIST P-256

NIST P-256 prime p is
$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$.

ECDSA standard specifies
reduction procedure given
an integer "A less than p^2":

Write A as
$(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, A_{10}, A_9,$
$A_8, A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)$,
meaning $\sum_i A_i 2^{32i}$.

Define
$T; S_1; S_2; S_3; S_4; D_1; D_2; D_3; D_4$
as

$(A_7, A_6,$

$(A_{15}, A_1$

$(0, A_{15},$

$(A_{15}, A_1$

$(A_8, A_{13}$

$(A_{10}, A_8$

$(A_{11}, A_9$

$(A_{12}, 0,$

$(A_{13}, 0,$

Comput

$S_4 - D_1$

Reduce

subtract

llocated for c:

$q + r$ where

$= \lfloor c/2^{255} \rfloor$.

for $19q + r$.

modulo p.

pression:

ts.

next mult.

duce 256 bits

rations of

ditional sub.

ub:

$(1 - s)p$

in $c - p$.

Constant-time NIST P-256

NIST P-256 prime p is
$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$.

ECDSA standard specifies reduction procedure given an integer "A less than p^2":

Write A as
$(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, A_{10}, A_9,$
$A_8, A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)$,
meaning $\sum_i A_i 2^{32i}$.

Define
$T; S_1; S_2; S_3; S_4; D_1; D_2; D_3; D_4$
as

$(A_7, A_6, A_5, A_4, A_3$

$(A_{15}, A_{14}, A_{13}, A_{12}$

$(0, A_{15}, A_{14}, A_{13},$

$(A_{15}, A_{14}, 0, 0, 0,$

$(A_8, A_{13}, A_{15}, A_{14},$

$(A_{10}, A_8, 0, 0, 0, A$

$(A_{11}, A_9, 0, 0, A_{15},$

$(A_{12}, 0, A_{10}, A_9, A$

$(A_{13}, 0, A_{11}, A_{10},$

Compute $T + 2S_1$
$S_4 - D_1 - D_2 -$

Reduce modulo p
subtracting a few

or c:

re

$\rfloor$.

r.

bits

ub.

Constant-time NIST P-256

NIST P-256 prime p is
$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$.

ECDSA standard specifies reduction procedure given an integer "A less than p^2":

Write A as
$(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, A_{10}, A_9,$
$A_8, A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)$,
meaning $\sum_i A_i 2^{32i}$.

Define
$T; S_1; S_2; S_3; S_4; D_1; D_2; D_3; D_4$
as

$(A_7, A_6, A_5, A_4, A_3, A_2, A_1,$

$(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, 0, 0$

$(0, A_{15}, A_{14}, A_{13}, A_{12}, 0, 0, 0$

$(A_{15}, A_{14}, 0, 0, 0, A_{10}, A_9, A_8$

$(A_8, A_{13}, A_{15}, A_{14}, A_{13}, A_{11},$

$(A_{10}, A_8, 0, 0, 0, A_{13}, A_{12}, A_1$

$(A_{11}, A_9, 0, 0, A_{15}, A_{14}, A_{13},$

$(A_{12}, 0, A_{10}, A_9, A_8, A_{15}, A_{14}$

$(A_{13}, 0, A_{11}, A_{10}, A_9, 0, A_{15},$

Compute $T + 2S_1 + 2S_2 +$
$S_4 - D_1 - D_2 - D_3 - D_4$.

Reduce modulo p "by addin
subtracting a few copies" of

Constant-time NIST P-256

NIST P-256 prime p is
$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$.

ECDSA standard specifies reduction procedure given an integer "A less than p^2":

Write A as
$(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, A_{10}, A_9,$
$A_8, A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)$,
meaning $\sum_i A_i 2^{32i}$.

Define
$T; S_1; S_2; S_3; S_4; D_1; D_2; D_3; D_4$
as

$(A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)$;
$(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, 0, 0, 0)$;
$(0, A_{15}, A_{14}, A_{13}, A_{12}, 0, 0, 0)$;
$(A_{15}, A_{14}, 0, 0, 0, A_{10}, A_9, A_8)$;
$(A_8, A_{13}, A_{15}, A_{14}, A_{13}, A_{11}, A_{10}, A_9)$;
$(A_{10}, A_8, 0, 0, 0, A_{13}, A_{12}, A_{11})$;
$(A_{11}, A_9, 0, 0, A_{15}, A_{14}, A_{13}, A_{12})$;
$(A_{12}, 0, A_{10}, A_9, A_8, A_{15}, A_{14}, A_{13})$;
$(A_{13}, 0, A_{11}, A_{10}, A_9, 0, A_{15}, A_{14})$.

Compute $T + 2S_1 + 2S_2 + S_3 +$
$S_4 - D_1 - D_2 - D_3 - D_4$.

Reduce modulo p "by adding or subtracting a few copies" of p.

t-time NIST P-256

-256 prime p is
$^{224} + 2^{192} + 2^{96} - 1$.

standard specifies
n procedure given
er "A less than p^2":

as
$_4, A_{13}, A_{12}, A_{11}, A_{10}, A_9,$
$A_6, A_5, A_4, A_3, A_2, A_1, A_0),$
$\sum_i A_i 2^{32i}$.

$_2; S_3; S_4; D_1; D_2; D_3; D_4$

$(A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)$;
$(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, 0, 0, 0)$;
$(0, A_{15}, A_{14}, A_{13}, A_{12}, 0, 0, 0)$;
$(A_{15}, A_{14}, 0, 0, 0, A_{10}, A_9, A_8)$;
$(A_8, A_{13}, A_{15}, A_{14}, A_{13}, A_{11}, A_{10}, A_9)$;
$(A_{10}, A_8, 0, 0, 0, A_{13}, A_{12}, A_{11})$;
$(A_{11}, A_9, 0, 0, A_{15}, A_{14}, A_{13}, A_{12})$;
$(A_{12}, 0, A_{10}, A_9, A_8, A_{15}, A_{14}, A_{13})$;
$(A_{13}, 0, A_{11}, A_{10}, A_9, 0, A_{15}, A_{14})$.

Compute $T + 2S_1 + 2S_2 + S_3 + S_4 - D_1 - D_2 - D_3 - D_4$.

Reduce modulo p "by adding or subtracting a few copies" of p.

What is
A loop?

ST P-256

p is

$+ 2^{96} - 1.$

specifies

re given

than p^2":

$, A_{11}, A_{10}, A_9,$

$_4, A_3, A_2, A_1, A_0),$

$^{2i}.$

$D_1; D_2; D_3; D_4$

$(A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0);$
$(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, 0, 0, 0);$
$(0, A_{15}, A_{14}, A_{13}, A_{12}, 0, 0, 0);$
$(A_{15}, A_{14}, 0, 0, 0, A_{10}, A_9, A_8);$
$(A_8, A_{13}, A_{15}, A_{14}, A_{13}, A_{11}, A_{10}, A_9);$
$(A_{10}, A_8, 0, 0, 0, A_{13}, A_{12}, A_{11});$
$(A_{11}, A_9, 0, 0, A_{15}, A_{14}, A_{13}, A_{12});$
$(A_{12}, 0, A_{10}, A_9, A_8, A_{15}, A_{14}, A_{13});$
$(A_{13}, 0, A_{11}, A_{10}, A_9, 0, A_{15}, A_{14}).$

Compute $T + 2S_1 + 2S_2 + S_3 + S_4 - D_1 - D_2 - D_3 - D_4$.

Reduce modulo p "by adding or subtracting a few copies" of p.

What is "a few co

A loop? **Variable**

1.

$, A_9,$

$A_1, A_0),$

$; D_4$

$(A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)$;
$(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, 0, 0, 0)$;
$(0, A_{15}, A_{14}, A_{13}, A_{12}, 0, 0, 0)$;
$(A_{15}, A_{14}, 0, 0, 0, A_{10}, A_9, A_8)$;
$(A_8, A_{13}, A_{15}, A_{14}, A_{13}, A_{11}, A_{10}, A_9)$;
$(A_{10}, A_8, 0, 0, 0, A_{13}, A_{12}, A_{11})$;
$(A_{11}, A_9, 0, 0, A_{15}, A_{14}, A_{13}, A_{12})$;
$(A_{12}, 0, A_{10}, A_9, A_8, A_{15}, A_{14}, A_{13})$;
$(A_{13}, 0, A_{11}, A_{10}, A_9, 0, A_{15}, A_{14})$.

Compute $T + 2S_1 + 2S_2 + S_3 + S_4 - D_1 - D_2 - D_3 - D_4$.

Reduce modulo p "by adding or subtracting a few copies" of p.

What is "a few copies"?

A loop? **Variable time**.

$(A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)$;
$(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, 0, 0, 0)$;
$(0, A_{15}, A_{14}, A_{13}, A_{12}, 0, 0, 0)$;
$(A_{15}, A_{14}, 0, 0, 0, A_{10}, A_9, A_8)$;
$(A_8, A_{13}, A_{15}, A_{14}, A_{13}, A_{11}, A_{10}, A_9)$;
$(A_{10}, A_8, 0, 0, 0, A_{13}, A_{12}, A_{11})$;
$(A_{11}, A_9, 0, 0, A_{15}, A_{14}, A_{13}, A_{12})$;
$(A_{12}, 0, A_{10}, A_9, A_8, A_{15}, A_{14}, A_{13})$;
$(A_{13}, 0, A_{11}, A_{10}, A_9, 0, A_{15}, A_{14})$.

Compute $T + 2S_1 + 2S_2 + S_3 + S_4 - D_1 - D_2 - D_3 - D_4$.

Reduce modulo p "by adding or subtracting a few copies" of p.

What is "a few copies"?

A loop? **Variable time**.

$(A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)$;
$(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, 0, 0, 0)$;
$(0, A_{15}, A_{14}, A_{13}, A_{12}, 0, 0, 0)$;
$(A_{15}, A_{14}, 0, 0, 0, A_{10}, A_9, A_8)$;
$(A_8, A_{13}, A_{15}, A_{14}, A_{13}, A_{11}, A_{10}, A_9)$;
$(A_{10}, A_8, 0, 0, 0, A_{13}, A_{12}, A_{11})$;
$(A_{11}, A_9, 0, 0, A_{15}, A_{14}, A_{13}, A_{12})$;
$(A_{12}, 0, A_{10}, A_9, A_8, A_{15}, A_{14}, A_{13})$;
$(A_{13}, 0, A_{11}, A_{10}, A_9, 0, A_{15}, A_{14})$.

Compute $T + 2S_1 + 2S_2 + S_3 + S_4 - D_1 - D_2 - D_3 - D_4$.

Reduce modulo p "by adding or subtracting a few copies" of p.

What is "a few copies"?
A loop? **Variable time**.

Correct but quite slow:
conditionally add $4p$,
conditionally add $2p$,
conditionally add p,
conditionally sub $4p$,
conditionally sub $2p$,
conditionally sub p.

$(A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)$;
$(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, 0, 0, 0)$;
$(0, A_{15}, A_{14}, A_{13}, A_{12}, 0, 0, 0)$;
$(A_{15}, A_{14}, 0, 0, 0, A_{10}, A_9, A_8)$;
$(A_8, A_{13}, A_{15}, A_{14}, A_{13}, A_{11}, A_{10}, A_9)$;
$(A_{10}, A_8, 0, 0, 0, A_{13}, A_{12}, A_{11})$;
$(A_{11}, A_9, 0, 0, A_{15}, A_{14}, A_{13}, A_{12})$;
$(A_{12}, 0, A_{10}, A_9, A_8, A_{15}, A_{14}, A_{13})$;
$(A_{13}, 0, A_{11}, A_{10}, A_9, 0, A_{15}, A_{14})$.

Compute $T + 2S_1 + 2S_2 + S_3 + S_4 - D_1 - D_2 - D_3 - D_4$.

Reduce modulo p "by adding or subtracting a few copies" of p.

What is "a few copies"?
A loop? **Variable time**.

Correct but quite slow:
conditionally add $4p$,
conditionally add $2p$,
conditionally add p,
conditionally sub $4p$,
conditionally sub $2p$,
conditionally sub p.

Delay until end of computation?
Trouble: "A less than p^2".

$(A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)$;
$(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, 0, 0, 0)$;
$(0, A_{15}, A_{14}, A_{13}, A_{12}, 0, 0, 0)$;
$(A_{15}, A_{14}, 0, 0, 0, A_{10}, A_9, A_8)$;
$(A_8, A_{13}, A_{15}, A_{14}, A_{13}, A_{11}, A_{10}, A_9)$;
$(A_{10}, A_8, 0, 0, 0, A_{13}, A_{12}, A_{11})$;
$(A_{11}, A_9, 0, 0, A_{15}, A_{14}, A_{13}, A_{12})$;
$(A_{12}, 0, A_{10}, A_9, A_8, A_{15}, A_{14}, A_{13})$;
$(A_{13}, 0, A_{11}, A_{10}, A_9, 0, A_{15}, A_{14})$.

Compute $T + 2S_1 + 2S_2 + S_3 + S_4 - D_1 - D_2 - D_3 - D_4$.

Reduce modulo p "by adding or subtracting a few copies" of p.

What is "a few copies"?
A loop? **Variable time**.

Correct but quite slow:
conditionally add $4p$,
conditionally add $2p$,
conditionally add p,
conditionally sub $4p$,
conditionally sub $2p$,
conditionally sub p.

Delay until end of computation?
Trouble: "A less than p^2".

Even worse: what about platforms where 2^{32} isn't best radix?

$A_5, A_4, A_3, A_2, A_1, A_0)$;

$_4, A_{13}, A_{12}, A_{11}, 0, 0, 0)$;

$A_{14}, A_{13}, A_{12}, 0, 0, 0)$;

$_4, 0, 0, 0, A_{10}, A_9, A_8)$;

$, A_{15}, A_{14}, A_{13}, A_{11}, A_{10}, A_9)$;

$, 0, 0, 0, A_{13}, A_{12}, A_{11})$;

$, 0, 0, A_{15}, A_{14}, A_{13}, A_{12})$;

$A_{10}, A_9, A_8, A_{15}, A_{14}, A_{13})$;

$A_{11}, A_{10}, A_9, 0, A_{15}, A_{14})$.

e $T + 2S_1 + 2S_2 + S_3 +$
$- D_2 - D_3 - D_4$.

modulo p "by adding or
ing a few copies" of p.

What is "a few copies"?
A loop? **Variable time**.

Correct but quite slow:
conditionally add $4p$,
conditionally add $2p$,
conditionally add p,
conditionally sub $4p$,
conditionally sub $2p$,
conditionally sub p.

Delay until end of computation?
Trouble: "A less than p^2".

Even worse: what about platforms
where 2^{32} isn't best radix?

The Mo

```
x2,z2,x
for i i
  bit =
  x2,x3
  z2,z3
  x3,z3

  x2,z2
    4*x
  x2,x3
  z2,z3
return 
```

$_3, A_2, A_1, A_0)$;
$_2, A_{11}, 0, 0, 0)$;
$A_{12}, 0, 0, 0)$;
$A_{10}, A_9, A_8)$;
$A_{13}, A_{11}, A_{10}, A_9)$;
$_{13}, A_{12}, A_{11})$;
$A_{14}, A_{13}, A_{12})$;
$_8, A_{15}, A_{14}, A_{13})$;
$A_9, 0, A_{15}, A_{14})$.

$+ 2S_2 + S_3 +$
$D_3 - D_4$.

"by adding or
copies" of p.

What is "a few copies"?
A loop? **Variable time**.

Correct but quite slow:
conditionally add $4p$,
conditionally add $2p$,
conditionally add p,
conditionally sub $4p$,
conditionally sub $2p$,
conditionally sub p.

Delay until end of computation?
Trouble: "A less than p^2".

Even worse: what about platforms
where 2^{32} isn't best radix?

The Montgomery

```
x2,z2,x3,z3 = 1,
for i in reverse
  bit = 1 & (n >
  x2,x3 = cswap(
  z2,z3 = cswap(
  x3,z3 = ((x2*x
         x1*(x2*z
  x2,z2 = ((x2^2
    4*x2*z2*(x2^
  x2,x3 = cswap(
  z2,z3 = cswap(
return x2*z2^(p-
```

A_0);
, 0);
);
);
A_{10}, A_9);
1);
A_{12});
., A_{13});
A_{14}).

S_3 +

g or
p.

What is “a few copies”?

A loop? **Variable time**.

Correct but quite slow:
conditionally add $4p$,
conditionally add $2p$,
conditionally add p,
conditionally sub $4p$,
conditionally sub $2p$,
conditionally sub p.

Delay until end of computation?
Trouble: “A less than p^2”.

Even worse: what about platforms
where 2^{32} isn't best radix?

The Montgomery ladder

```
x2,z2,x3,z3 = 1,0,x1,1
for i in reversed(range(2
  bit = 1 & (n >> i)
  x2,x3 = cswap(x2,x3,bit
  z2,z3 = cswap(z2,z3,bit
  x3,z3 = ((x2*x3-z2*z3)^
        x1*(x2*z3-z2*x3)^
  x2,z2 = ((x2^2-z2^2)^2,
    4*x2*z2*(x2^2+A*x2*z2
  x2,x3 = cswap(x2,x3,bit
  z2,z3 = cswap(z2,z3,bit
return x2*z2^(p-2)
```

What is “a few copies”?

A loop? **Variable time**.

Correct but quite slow:
conditionally add $4p$,
conditionally add $2p$,
conditionally add p,
conditionally sub $4p$,
conditionally sub $2p$,
conditionally sub p.

Delay until end of computation?
Trouble: “A less than p^2”.

Even worse: what about platforms where 2^{32} isn't best radix?

The Montgomery ladder

```
x2,z2,x3,z3 = 1,0,x1,1
for i in reversed(range(255)):
  bit = 1 & (n >> i)
  x2,x3 = cswap(x2,x3,bit)
  z2,z3 = cswap(z2,z3,bit)
  x3,z3 = ((x2*x3-z2*z3)^2,
           x1*(x2*z3-z2*x3)^2)
  x2,z2 = ((x2^2-z2^2)^2,
    4*x2*z2*(x2^2+A*x2*z2+z2^2))
  x2,x3 = cswap(x2,x3,bit)
  z2,z3 = cswap(z2,z3,bit)
return x2*z2^(p-2)
```

"a few copies"?

Variable time.

but quite slow:

nally add $4p$,

nally add $2p$,

nally add p,

nally sub $4p$,

nally sub $2p$,

nally sub p.

ntil end of computation?

"A less than p^2".

rse: what about platforms

^{32} isn't best radix?

The Montgomery ladder

```
x2,z2,x3,z3 = 1,0,x1,1
for i in reversed(range(255)):
  bit = 1 & (n >> i)
  x2,x3 = cswap(x2,x3,bit)
  z2,z3 = cswap(z2,z3,bit)
  x3,z3 = ((x2*x3-z2*z3)^2,
            x1*(x2*z3-z2*x3)^2)
  x2,z2 = ((x2^2-z2^2)^2,
    4*x2*z2*(x2^2+A*x2*z2+z2^2))
  x2,x3 = cswap(x2,x3,bit)
  z2,z3 = cswap(z2,z3,bit)
return x2*z2^(p-2)
```

Simple;

compute

on $y^2 =$

when A^2

pies"?

time.

slow:

$4p$,

$2p$,

p,

$4p$,

$2p$,

p.

computation?

han p^2".

about platforms

st radix?

The Montgomery ladder

```
x2,z2,x3,z3 = 1,0,x1,1
for i in reversed(range(255)):
  bit = 1 & (n >> i)
  x2,x3 = cswap(x2,x3,bit)
  z2,z3 = cswap(z2,z3,bit)
  x3,z3 = ((x2*x3-z2*z3)^2,
            x1*(x2*z3-z2*x3)^2)
  x2,z2 = ((x2^2-z2^2)^2,
    4*x2*z2*(x2^2+A*x2*z2+z2^2))
  x2,x3 = cswap(x2,x3,bit)
  z2,z3 = cswap(z2,z3,bit)
return x2*z2^(p-2)
```

Simple; fast; **alwa**

computes scalar m

on $y^2 = x^3 + Ax^2$

when $A^2 - 4$ is no

ion?

tforms

The Montgomery ladder

```
x2,z2,x3,z3 = 1,0,x1,1
for i in reversed(range(255)):
  bit = 1 & (n >> i)
  x2,x3 = cswap(x2,x3,bit)
  z2,z3 = cswap(z2,z3,bit)
  x3,z3 = ((x2*x3-z2*z3)^2,
            x1*(x2*z3-z2*x3)^2)
  x2,z2 = ((x2^2-z2^2)^2,
    4*x2*z2*(x2^2+A*x2*z2+z2^2))
  x2,x3 = cswap(x2,x3,bit)
  z2,z3 = cswap(z2,z3,bit)
return x2*z2^(p-2)
```


Simple; fast; **always**
computes scalar multiplicati
on $y^2 = x^3 + Ax^2 + x$
when $A^2 - 4$ is non-square.

The Montgomery ladder

```
x2,z2,x3,z3 = 1,0,x1,1
for i in reversed(range(255)):
  bit = 1 & (n >> i)
  x2,x3 = cswap(x2,x3,bit)
  z2,z3 = cswap(z2,z3,bit)
  x3,z3 = ((x2*x3-z2*z3)^2,
           x1*(x2*z3-z2*x3)^2)
  x2,z2 = ((x2^2-z2^2)^2,
    4*x2*z2*(x2^2+A*x2*z2+z2^2))
  x2,x3 = cswap(x2,x3,bit)
  z2,z3 = cswap(z2,z3,bit)
return x2*z2^(p-2)
```


Simple; fast; **always** computes scalar multiplication on $y^2 = x^3 + Ax^2 + x$ when $A^2 - 4$ is non-square.

The Montgomery ladder

```
x2,z2,x3,z3 = 1,0,x1,1
for i in reversed(range(255)):
  bit = 1 & (n >> i)
  x2,x3 = cswap(x2,x3,bit)
  z2,z3 = cswap(z2,z3,bit)
  x3,z3 = ((x2*x3-z2*z3)^2,
        x1*(x2*z3-z2*x3)^2)
  x2,z2 = ((x2^2-z2^2)^2,
    4*x2*z2*(x2^2+A*x2*z2+z2^2))
  x2,x3 = cswap(x2,x3,bit)
  z2,z3 = cswap(z2,z3,bit)
return x2*z2^(p-2)
```


Simple; fast; **always** computes scalar multiplication on $y^2 = x^3 + Ax^2 + x$ when $A^2 - 4$ is non-square.

With some extra lines can compute (x, y) output given (x, y) input.
But simpler to use just x, as proposed by 1985 Miller.

The Montgomery ladder

```
x2,z2,x3,z3 = 1,0,x1,1
for i in reversed(range(255)):
  bit = 1 & (n >> i)
  x2,x3 = cswap(x2,x3,bit)
  z2,z3 = cswap(z2,z3,bit)
  x3,z3 = ((x2*x3-z2*z3)^2,
        x1*(x2*z3-z2*x3)^2)
  x2,z2 = ((x2^2-z2^2)^2,
    4*x2*z2*(x2^2+A*x2*z2+z2^2))
  x2,x3 = cswap(x2,x3,bit)
  z2,z3 = cswap(z2,z3,bit)
return x2*z2^(p-2)
```


Simple; fast; **always** computes scalar multiplication on $y^2 = x^3 + Ax^2 + x$ when $A^2 - 4$ is non-square.

With some extra lines can compute (x, y) output given (x, y) input.
But simpler to use just x, as proposed by 1985 Miller.

Adaptations to NIST curves are much slower; not as simple; not proven to always work.
Other scalar-mult methods: proven but much more complex.

ntgomery ladder

```
3,z3 = 1,0,x1,1
n reversed(range(255)):
 1 & (n >> i)
 = cswap(x2,x3,bit)
 = cswap(z2,z3,bit)
 = ((x2*x3-z2*z3)^2,
 x1*(x2*z3-z2*x3)^2)
 = ((x2^2-z2^2)^2,
2*z2*(x2^2+A*x2*z2+z2^2))
 = cswap(x2,x3,bit)
 = cswap(z2,z3,bit)
x2*z2^(p-2)
```


Simple; fast; **always** computes scalar multiplication on $y^2 = x^3 + Ax^2 + x$ when $A^2 - 4$ is non-square.

With some extra lines can compute (x, y) output given (x, y) input.
But simpler to use just x, as proposed by 1985 Miller.

Adaptations to NIST curves are much slower; not as simple; not proven to always work.
Other scalar-mult methods: proven but much more complex.

"Hey, yo
that x_1

No need
Curve25

ladder

```
0,x1,1
d(range(255)):
> i)
x2,x3,bit)
z2,z3,bit)
3-z2*z3)^2,
3-z2*x3)^2)
-z2^2)^2,
2+A*x2*z2+z2^2))
x2,x3,bit)
z2,z3,bit)
2)
```


Simple; fast; **always** computes scalar multiplication on $y^2 = x^3 + Ax^2 + x$ when $A^2 - 4$ is non-square.

With some extra lines can compute (x, y) output given (x, y) input. But simpler to use just x, as proposed by 1985 Miller.

Adaptations to NIST curves are much slower; not as simple; not proven to always work. Other scalar-mult methods: proven but much more complex.

"Hey, you forgot t that x_1 is on the c

No need to check. Curve25519 is **twi**

```
255)):
.)
.)
2,
2)
2+z2^2))
.)
.)
```

Simple; fast; **always** computes scalar multiplication on $y^2 = x^3 + Ax^2 + x$ when $A^2 - 4$ is non-square.

With some extra lines can compute (x, y) output given (x, y) input. But simpler to use just x, as proposed by 1985 Miller.

Adaptations to NIST curves are much slower; not as simple; not proven to always work. Other scalar-mult methods: proven but much more complex.

"Hey, you forgot to check that x_1 is on the curve!"

No need to check. Curve25519 is **twist-secure**

Simple; fast; **always**
computes scalar multiplication
on $y^2 = x^3 + Ax^2 + x$
when $A^2 - 4$ is non-square.

With some extra lines
can compute (x, y) output
given (x, y) input.
But simpler to use just x,
as proposed by 1985 Miller.

Adaptations to NIST curves
are much slower; not as simple;
not proven to always work.
Other scalar-mult methods:
proven but much more complex.

"Hey, you forgot to check
that x_1 is on the curve!"

No need to check.
Curve25519 is **twist-secure**.

Simple; fast; **always** computes scalar multiplication on $y^2 = x^3 + Ax^2 + x$ when $A^2 - 4$ is non-square.

With some extra lines can compute (x, y) output given (x, y) input. But simpler to use just x, as proposed by 1985 Miller.

Adaptations to NIST curves are much slower; not as simple; not proven to always work. Other scalar-mult methods: proven but much more complex.

"Hey, you forgot to check that x_1 is on the curve!"

No need to check. Curve25519 is **twist-secure**.

"This textbook tells me to start the Montgomery ladder from the top bit *set* in n!" (Exploited in, e.g., 2011 Brumley–Tuveri "Remote timing attacks are still practical".)

The Curve25519 DH function takes $2^{254} \le n < 2^{255}$, so this is still constant-time.

fast; **always**
es scalar multiplication
$x^3 + Ax^2 + x$
$^2 - 4$ is non-square.

me extra lines
pute (x, y) output
$, y)$ input.
pler to use just x,
osed by 1985 Miller.

ions to NIST curves
h slower; not as simple;
ven to always work.
calar-mult methods:
but much more complex.

"Hey, you forgot to check that x_1 is on the curve!"

No need to check.
Curve25519 is **twist-secure**.

"This textbook tells me to start the Montgomery ladder from the top bit *set* in n!"
(Exploited in, e.g., 2011 Brumley–Tuveri "Remote timing attacks are still practical".)

The Curve25519 DH function takes $2^{254} \le n < 2^{255}$, so this is still constant-time.

Subsequ

More Cu

2007 Ga
Core 2,

2009 Co

2011 Be
Schwabe

2012 Be

2014 La
newer In

2014 Ma

2014 Sa

ys

ultiplication

$+ x$

n-square.

ines

) output

just x,

85 Miller.

ST curves

ot as simple;

ys work.

methods:

more complex.

"Hey, you forgot to check that x_1 is on the curve!"

No need to check. Curve25519 is **twist-secure**.

"This textbook tells me to start the Montgomery ladder from the top bit *set* in n!" (Exploited in, e.g., 2011 Brumley–Tuveri "Remote timing attacks are still practical".)

The Curve25519 DH function takes $2^{254} \le n < 2^{255}$, so this is still constant-time.

Subsequent develo

More Curve25519

2007 Gaudry–Tho

Core 2, Athlon 64.

2009 Costigan–Sc

2011 Bernstein–D

Schwabe–Yang: N

2012 Bernstein–Sc

2014 Langley–Mo

newer Intel chips.

2014 Mahé–Chau

2014 Sasdrich–Gü

on

ple;

plex.

"Hey, you forgot to check that x_1 is on the curve!"

No need to check. Curve25519 is **twist-secure**.

"This textbook tells me to start the Montgomery ladder from the top bit *set* in n!" (Exploited in, e.g., 2011 Brumley–Tuveri "Remote timing attacks are still practical".)

The Curve25519 DH function takes $2^{254} \le n < 2^{255}$, so this is still constant-time.

Subsequent developments

More Curve25519 implemen

2007 Gaudry–Thomé: tuned Core 2, Athlon 64.

2009 Costigan–Schwabe: Ce

2011 Bernstein–Duif–Lange–Schwabe–Yang: Nehalem et

2012 Bernstein–Schwabe: N

2014 Langley–Moon: variou newer Intel chips.

2014 Mahé–Chauvet: GPUs

2014 Sasdrich–Güneysu: FP

"Hey, you forgot to check that x_1 is on the curve!"

No need to check. Curve25519 is **twist-secure**.

"This textbook tells me to start the Montgomery ladder from the top bit *set* in n!" (Exploited in, e.g., 2011 Brumley–Tuveri "Remote timing attacks are still practical".)

The Curve25519 DH function takes $2^{254} \leq n < 2^{255}$, so this is still constant-time.

Subsequent developments

More Curve25519 implementations:

2007 Gaudry–Thomé: tuned for Core 2, Athlon 64.

2009 Costigan–Schwabe: Cell.

2011 Bernstein–Duif–Lange–Schwabe–Yang: Nehalem etc.

2012 Bernstein–Schwabe: NEON.

2014 Langley–Moon: various newer Intel chips.

2014 Mahé–Chauvet: GPUs.

2014 Sasdrich–Güneysu: FPGAs.

ou forgot to check
is on the curve!"

to check.
519 is **twist-secure**.

xtbook tells me
the Montgomery ladder
e top bit *set* in n!"
ed in, e.g., 2011
–Tuveri "Remote timing
are still practical".)

ve25519 DH function
$^{54} \leq n < 2^{255}$,
s still constant-time.

Subsequent developments

More Curve25519 implementations:

2007 Gaudry–Thomé: tuned for Core 2, Athlon 64.

2009 Costigan–Schwabe: Cell.

2011 Bernstein–Duif–Lange–Schwabe–Yang: Nehalem etc.

2012 Bernstein–Schwabe: NEON.

2014 Langley–Moon: various newer Intel chips.

2014 Mahé–Chauvet: GPUs.

2014 Sasdrich–Güneysu: FPGAs.

2011 Be
Schwabe
reusing (

2013 Be
Schwabe

2014 Ch
Tsai–Wa
"Verifyi

http://
/Curve2
lists App
TextSecu

Much lo
Nicolai E

o check
curve!"

st-secure.

lls me
gomery ladder
et in n!"
, 2011
Remote timing
actical".)

DH function
2^{255},
stant-time.

Subsequent developments

More Curve25519 implementations:

2007 Gaudry–Thomé: tuned for Core 2, Athlon 64.

2009 Costigan–Schwabe: Cell.

2011 Bernstein–Duif–Lange–Schwabe–Yang: Nehalem etc.

2012 Bernstein–Schwabe: NEON.

2014 Langley–Moon: various newer Intel chips.

2014 Mahé–Chauvet: GPUs.

2014 Sasdrich–Güneysu: FPGAs.

2011 Bernstein–Du
Schwabe–Yang: E
reusing Curve2551

2013 Bernstein–Ja
Schwabe: TweetN

2014 Chen–Hsu–L
Tsai–Wang–Yang–
"Verifying Curve2

http://en.wiki
/Curve25519#No
lists Apple's iOS,
TextSecure, Tor, e

Much longer list m
Nicolai Brown (IA

.

dder

ming

on

Subsequent developments

More Curve25519 implementations:

2007 Gaudry–Thomé: tuned for Core 2, Athlon 64.

2009 Costigan–Schwabe: Cell.

2011 Bernstein–Duif–Lange–Schwabe–Yang: Nehalem etc.

2012 Bernstein–Schwabe: NEON.

2014 Langley–Moon: various newer Intel chips.

2014 Mahé–Chauvet: GPUs.

2014 Sasdrich–Güneysu: FPGAs.

2011 Bernstein–Duif–Lange–
Schwabe–Yang: Ed25519,
reusing Curve25519 for signa

2013 Bernstein–Janssen–Lar
Schwabe: TweetNaCl.

2014 Chen–Hsu–Lin–Schwab
Tsai–Wang–Yang–Yang:
“Verifying Curve25519 softw

http://en.wikipedia.org
/Curve25519#Notable_use
lists Apple’s iOS, OpenSSH,
TextSecure, Tor, et al.

Much longer list maintained
Nicolai Brown (IANIX).

Subsequent developments

More Curve25519 implementations:

2007 Gaudry–Thomé: tuned for Core 2, Athlon 64.

2009 Costigan–Schwabe: Cell.

2011 Bernstein–Duif–Lange–Schwabe–Yang: Nehalem etc.

2012 Bernstein–Schwabe: NEON.

2014 Langley–Moon: various newer Intel chips.

2014 Mahé–Chauvet: GPUs.

2014 Sasdrich–Güneysu: FPGAs.

2011 Bernstein–Duif–Lange–Schwabe–Yang: Ed25519, reusing Curve25519 for signatures.

2013 Bernstein–Janssen–Lange–Schwabe: TweetNaCl.

2014 Chen–Hsu–Lin–Schwabe–Tsai–Wang–Yang–Yang: "Verifying Curve25519 software."

`http://en.wikipedia.org/wiki/Curve25519#Notable_uses` lists Apple's iOS, OpenSSH, TextSecure, Tor, et al.

Much longer list maintained by Nicolai Brown (IANIX).

ent developments

urve25519 implementations:

udry–Thomé: tuned for
Athlon 64.

stigan–Schwabe: Cell.

rnstein–Duif–Lange–
e–Yang: Nehalem etc.

rnstein–Schwabe: NEON.

ngley–Moon: various
ntel chips.

ahé–Chauvet: GPUs.

sdrich–Güneysu: FPGAs.

2011 Bernstein–Duif–Lange–Schwabe–Yang: Ed25519, reusing Curve25519 for signatures.

2013 Bernstein–Janssen–Lange–Schwabe: TweetNaCl.

2014 Chen–Hsu–Lin–Schwabe–Tsai–Wang–Yang–Yang: "Verifying Curve25519 software."

http://en.wikipedia.org/wiki/Curve25519#Notable_uses lists Apple's iOS, OpenSSH, TextSecure, Tor, et al.

Much longer list maintained by Nicolai Brown (IANIX).

2013.08:
requests
at highe

Bernstei
Now Sile

opments

implementations:

mé: tuned for

.

hwabe: Cell.

uif–Lange–

lehalem etc.

chwabe: NEON.

on: various

vet: GPUs.

neysu: FPGAs.

2011 Bernstein–Duif–Lange–Schwabe–Yang: Ed25519, reusing Curve25519 for signatures.

2013 Bernstein–Janssen–Lange–Schwabe: TweetNaCl.

2014 Chen–Hsu–Lin–Schwabe–Tsai–Wang–Yang–Yang: "Verifying Curve25519 software."

http://en.wikipedia.org/wiki/Curve25519#Notable_uses lists Apple's iOS, OpenSSH, TextSecure, Tor, et al.

Much longer list maintained by Nicolai Brown (IANIX).

2013.08: Silent Ci

requests non-NIST

at higher security

Bernstein–Lange:

Now Silent Circle's

tations:

l for

ell.

–

c.

IEON.

s

.

GAs.

2011 Bernstein–Duif–Lange–Schwabe–Yang: Ed25519, reusing Curve25519 for signatures.

2013 Bernstein–Janssen–Lange–Schwabe: TweetNaCl.

2014 Chen–Hsu–Lin–Schwabe–Tsai–Wang–Yang–Yang: "Verifying Curve25519 software."

http://en.wikipedia.org/wiki/Curve25519#Notable_uses lists Apple's iOS, OpenSSH, TextSecure, Tor, et al.

Much longer list maintained by Nicolai Brown (IANIX).

2013.08: Silent Circle requests non-NIST curve at higher security level.

Bernstein–Lange: Curve414

Now Silent Circle's default.

2011 Bernstein–Duif–Lange–Schwabe–Yang: Ed25519, reusing Curve25519 for signatures.

2013 Bernstein–Janssen–Lange–Schwabe: TweetNaCl.

2014 Chen–Hsu–Lin–Schwabe–Tsai–Wang–Yang–Yang: "Verifying Curve25519 software."

http://en.wikipedia.org/wiki/Curve25519#Notable_uses lists Apple's iOS, OpenSSH, TextSecure, Tor, et al.

Much longer list maintained by Nicolai Brown (IANIX).

2013.08: Silent Circle requests non-NIST curve at higher security level.

Bernstein–Lange: Curve41417. Now Silent Circle's default.

2011 Bernstein–Duif–Lange–Schwabe–Yang: Ed25519, reusing Curve25519 for signatures.

2013 Bernstein–Janssen–Lange–Schwabe: TweetNaCl.

2014 Chen–Hsu–Lin–Schwabe–Tsai–Wang–Yang–Yang: "Verifying Curve25519 software."

http://en.wikipedia.org/wiki/Curve25519#Notable_uses lists Apple's iOS, OpenSSH, TextSecure, Tor, et al.

Much longer list maintained by Nicolai Brown (IANIX).

2013.08: Silent Circle requests non-NIST curve at higher security level.

Bernstein–Lange: Curve41417. Now Silent Circle's default.

Bernstein–Lange, independently Hamburg, independently Aranha–Barreto–Pereira–Ricardini: E-521.

2011 Bernstein–Duif–Lange–Schwabe–Yang: Ed25519, reusing Curve25519 for signatures.

2013 Bernstein–Janssen–Lange–Schwabe: TweetNaCl.

2014 Chen–Hsu–Lin–Schwabe–Tsai–Wang–Yang–Yang: "Verifying Curve25519 software."

http://en.wikipedia.org/wiki/Curve25519#Notable_uses lists Apple's iOS, OpenSSH, TextSecure, Tor, et al.

Much longer list maintained by Nicolai Brown (IANIX).

2013.08: Silent Circle requests non-NIST curve at higher security level.

Bernstein–Lange: Curve41417. Now Silent Circle's default.

Bernstein–Lange, independently Hamburg, independently Aranha–Barreto–Pereira–Ricardini: E-521.

More options hurt simplicity; do they really help security? Note that typical claims regarding AES-ECC "balance" disregard multiple users; lucky attacks; quantum attacks.