Simplicity

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

Joint work with:
Tanja Lange
Technische Universiteit Eindhoven

NIST's ECC standards
= NSA's prime choices
+ NSA's curve choices
+ NSA's coordinate choices
+ NSA's computation choices
+ NSA's protocol choices.

NIST's ECC standards create **unnecessary complexity in ECC implementations**.

This unnecessary complexity

- scares away implementors,
- reduces ECC adoption,
- interferes with optimization,
- keeps ECC out of small devices,
- scares away auditors,
- interferes with verification, and
- creates ECC security failures.

Simplicity

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

Joint work with:
Tanja Lange
Technische Universiteit Eindhoven

NIST's ECC standards
= NSA's prime choices
+ NSA's curve choices
+ NSA's coordinate choices
+ NSA's computation choices
+ NSA's protocol choices.

NIST's ECC standards create **unnecessary complexity in ECC implementations**.

This unnecessary complexity

- scares away implementors,
- reduces ECC adoption,
- interferes with optimization,
- keeps ECC out of small devices,
- scares away auditors,
- interferes with verification, and
- creates ECC security failures.

1992 Rivest: "*The poor user is given enough rope with which to hang himself—something a standard should not do.*"

ty

ernstein

ty of Illinois at Chicago &

che Universiteit Eindhoven

ork with:

ange

che Universiteit Eindhoven

ECC standards

s prime choices

s curve choices

s coordinate choices

s computation choices

s protocol choices.

NIST's ECC standards create **unnecessary complexity in ECC implementations**.

This unnecessary complexity

- scares away implementors,
- reduces ECC adoption,
- interferes with optimization,
- keeps ECC out of small devices,
- scares away auditors,
- interferes with verification, and
- creates ECC security failures.

1992 Rivest: "*The poor user is given enough rope with which to hang himself—something a standard should not do.*"

Should c

every im

Replace

is at Chicago &
siteit Eindhoven

siteit Eindhoven

---

lards
oices
oices
te choices
tion choices
choices.

NIST's ECC standards create **unnecessary complexity in ECC implementations**.

This unnecessary complexity

- scares away implementors,
- reduces ECC adoption,
- interferes with optimization,
- keeps ECC out of small devices,
- scares away auditors,
- interferes with verification, and
- creates ECC security failures.

1992 Rivest: "*The poor user is given enough rope with which to hang himself—something a standard should not do.*"

Should cryptograp
every imaginable s

Replace GCM with

ago &
hoven

hoven

es

NIST's ECC standards create **unnecessary complexity in ECC implementations**.

This unnecessary complexity

- scares away implementors,
- reduces ECC adoption,
- interferes with optimization,
- keeps ECC out of small devices,
- scares away auditors,
- interferes with verification, and
- creates ECC security failures.

1992 Rivest: "*The poor user is given enough rope with which to hang himself—something a standard should not do.*"

Should cryptographers apply every imaginable simplificati

Replace GCM with ECB?

NIST's ECC standards create **unnecessary complexity in ECC implementations**.

This unnecessary complexity

- scares away implementors,
- reduces ECC adoption,
- interferes with optimization,
- keeps ECC out of small devices,
- scares away auditors,
- interferes with verification, and
- creates ECC security failures.

1992 Rivest: "*The poor user is given enough rope with which to hang himself—something a standard should not do.*"

Should cryptographers apply every imaginable simplification?

Replace GCM with ECB?

NIST's ECC standards create **unnecessary complexity in ECC implementations**.

This unnecessary complexity

- scares away implementors,
- reduces ECC adoption,
- interferes with optimization,
- keeps ECC out of small devices,
- scares away auditors,
- interferes with verification, and
- creates ECC security failures.

1992 Rivest: "*The poor user is given enough rope with which to hang himself—something a standard should not do.*"

Should cryptographers apply every imaginable simplification?

Replace GCM with ECB?

No: ECB doesn't authenticate and doesn't securely encrypt.

NIST's ECC standards create **unnecessary complexity in ECC implementations**.

This unnecessary complexity

- scares away implementors,
- reduces ECC adoption,
- interferes with optimization,
- keeps ECC out of small devices,
- scares away auditors,
- interferes with verification, and
- creates ECC security failures.

1992 Rivest: "*The poor user is given enough rope with which to hang himself—something a standard should not do.*"

Should cryptographers apply every imaginable simplification?

Replace GCM with ECB?

No: ECB doesn't authenticate and doesn't securely encrypt.

Replace ECDH with FFDH?

NIST's ECC standards create **unnecessary complexity in ECC implementations**.

This unnecessary complexity

- scares away implementors,
- reduces ECC adoption,
- interferes with optimization,
- keeps ECC out of small devices,
- scares away auditors,
- interferes with verification, and
- creates ECC security failures.

1992 Rivest: "*The poor user is given enough rope with which to hang himself—something a standard should not do.*"

Should cryptographers apply every imaginable simplification?

Replace GCM with ECB?

No: ECB doesn't authenticate and doesn't securely encrypt.

Replace ECDH with FFDH?

No: FFDH is vulnerable to index calculus. Bigger keys; slower; much harder security analysis.

NIST's ECC standards create **unnecessary complexity in ECC implementations**.

This unnecessary complexity

- scares away implementors,
- reduces ECC adoption,
- interferes with optimization,
- keeps ECC out of small devices,
- scares away auditors,
- interferes with verification, and
- creates ECC security failures.

1992 Rivest: "*The poor user is given enough rope with which to hang himself—something a standard should not do.*"

Should cryptographers apply every imaginable simplification?

Replace GCM with ECB?

No: ECB doesn't authenticate and doesn't securely encrypt.

Replace ECDH with FFDH?

No: FFDH is vulnerable to index calculus. Bigger keys; slower; much harder security analysis.

Priority #1 is security.
Priority #2 is to meet the user's performance requirements.
Priority #3 is simplicity.

ECC standards create
ssary complexity
implementations.

necessary complexity
away implementors,
es ECC adoption,
eres with optimization,
ECC out of small devices,
away auditors,
eres with verification, and
s ECC security failures.

vest: "*The poor user is*
ough rope with which
himself—something
rd should not do."

Should cryptographers apply every imaginable simplification?

Replace GCM with ECB?

No: ECB doesn't authenticate and doesn't securely encrypt.

Replace ECDH with FFDH?

No: FFDH is vulnerable to index calculus. Bigger keys; slower; much harder security analysis.

Priority #1 is security.
Priority #2 is to meet the user's performance requirements.
Priority #3 is simplicity.

Wild ove
example

"Simplic

"Simplic

lards create
plexity
ntations.

complexity
lementors,
option,
ptimization,
of small devices,
itors,
erification, and
urity failures.

e poor user is
e with which
something
not do."

Should cryptographers apply every imaginable simplification?

Replace GCM with ECB?

No: ECB doesn't authenticate and doesn't securely encrypt.

Replace ECDH with FFDH?

No: FFDH is vulnerable to index calculus. Bigger keys; slower; much harder security analysis.

Priority #1 is security.
Priority #2 is to meet the user's performance requirements.
Priority #3 is simplicity.

Wild overgeneraliz
examples of oversi

"Simplicity damag

"Simplicity damag

Should cryptographers apply every imaginable simplification?

Replace GCM with ECB?

No: ECB doesn't authenticate and doesn't securely encrypt.

Replace ECDH with FFDH?

No: FFDH is vulnerable to index calculus. Bigger keys; slower; much harder security analysis.

Priority #1 is security.
Priority #2 is to meet the user's performance requirements.
Priority #3 is simplicity.

Wild overgeneralizations fro[m] examples of oversimplificati[on]

"Simplicity damages securit[y]

"Simplicity damages speed."

Should cryptographers apply every imaginable simplification?

Replace GCM with ECB?

No: ECB doesn't authenticate and doesn't securely encrypt.

Replace ECDH with FFDH?

No: FFDH is vulnerable to index calculus. Bigger keys; slower; much harder security analysis.

Priority #1 is security.
Priority #2 is to meet the user's performance requirements.
Priority #3 is simplicity.

Wild overgeneralizations from examples of oversimplification:

"Simplicity damages security."

"Simplicity damages speed."

Should cryptographers apply every imaginable simplification?

Replace GCM with ECB?

No: ECB doesn't authenticate and doesn't securely encrypt.

Replace ECDH with FFDH?

No: FFDH is vulnerable to index calculus. Bigger keys; slower; much harder security analysis.

Priority #1 is security.
Priority #2 is to meet the user's performance requirements.
Priority #3 is simplicity.

Wild overgeneralizations from examples of oversimplification:

"Simplicity damages security."

"Simplicity damages speed."

These overgeneralizations are often used to cover up deficient analyses of speed and security.

Should cryptographers apply every imaginable simplification?

Replace GCM with ECB?

No: ECB doesn't authenticate and doesn't securely encrypt.

Replace ECDH with FFDH?

No: FFDH is vulnerable to index calculus. Bigger keys; slower; much harder security analysis.

Priority #1 is security.
Priority #2 is to meet the user's performance requirements.
Priority #3 is simplicity.

Wild overgeneralizations from examples of oversimplification:

"Simplicity damages security."

"Simplicity damages speed."

These overgeneralizations are often used to cover up deficient analyses of speed and security.

In fact, many simplifications don't hurt security at all and don't hurt speed at all.

Should cryptographers apply every imaginable simplification?

Replace GCM with ECB?

No: ECB doesn't authenticate and doesn't securely encrypt.

Replace ECDH with FFDH?

No: FFDH is vulnerable to index calculus. Bigger keys; slower; much harder security analysis.

Priority #1 is security.
Priority #2 is to meet the user's performance requirements.
Priority #3 is simplicity.

Wild overgeneralizations from examples of oversimplification:

"Simplicity damages security."

"Simplicity damages speed."

These overgeneralizations are often used to cover up deficient analyses of speed and security.

In fact, many simplifications don't hurt security at all and don't hurt speed at all.

Next-generation ECC simplicity **contributes to security** and **contributes to speed**.

cryptographers apply
aginable simplification?

GCM with ECB?

3 doesn't authenticate
sn't securely encrypt.

ECDH with FFDH?

DH is vulnerable to index
. Bigger keys; slower;
arder security analysis.

#1 is security.
#2 is to meet the
erformance requirements.
#3 is simplicity.

Wild overgeneralizations from examples of oversimplification:

"Simplicity damages security."

"Simplicity damages speed."

These overgeneralizations are often used to cover up deficient analyses of speed and security.

In fact, many simplifications don't hurt security at all and don't hurt speed at all.

Next-generation ECC simplicity **contributes to security** and **contributes to speed**.

Constan

Imitate
Allocate
for each
Always p
on all bi

hers apply
implification?

ECB?

authenticate
ly encrypt.

th FFDH?

erable to index
eys; slower;
ity analysis.

rity.
eet the
requirements.
plicity.

Wild overgeneralizations from examples of oversimplification:

"Simplicity damages security."

"Simplicity damages speed."

These overgeneralizations are often used to cover up deficient analyses of speed and security.

In fact, many simplifications don't hurt security at all and don't hurt speed at all.

Next-generation ECC simplicity **contributes to security** and **contributes to speed**.

Constant-time Cu

Imitate hardware i
Allocate constant
for each integer.
Always perform ar
on all bits. Don't

y
on?
ate
t.
ndex
r;
is.
ents.

Wild overgeneralizations from examples of oversimplification:

"Simplicity damages security."

"Simplicity damages speed."

These overgeneralizations are often used to cover up deficient analyses of speed and security.

In fact, many simplifications don't hurt security at all and don't hurt speed at all.

Next-generation ECC simplicity **contributes to security** and **contributes to speed**.

Constant-time Curve25519

Imitate hardware in software
Allocate constant number of
for each integer.
Always perform arithmetic
on all bits. Don't skip bits.

Wild overgeneralizations from examples of oversimplification:

"Simplicity damages security."

"Simplicity damages speed."

These overgeneralizations are often used to cover up deficient analyses of speed and security.

In fact, many simplifications don't hurt security at all and don't hurt speed at all.

Next-generation ECC simplicity **contributes to security** and **contributes to speed**.

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.

Wild overgeneralizations from examples of oversimplification:

“Simplicity damages security.”

“Simplicity damages speed.”

These overgeneralizations are often used to cover up deficient analyses of speed and security.

In fact, many simplifications don’t hurt security at all and don’t hurt speed at all.

Next-generation ECC simplicity **contributes to security** and **contributes to speed**.

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.

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

Wild overgeneralizations from examples of oversimplification:

"Simplicity damages security."

"Simplicity damages speed."

These overgeneralizations are often used to cover up deficient analyses of speed and security.

In fact, many simplifications don't hurt security at all and don't hurt speed at all.

Next-generation ECC simplicity **contributes to security** and **contributes to speed**.

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.

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

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

ergeneralizations from
s of oversimplification:

ity damages security."

ity damages speed."

vergeneralizations are
ed to cover up deficient
of speed and security.

many simplifications
rt security at all
't hurt speed at all.

neration ECC simplicity
utes to security
tributes to speed.

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.

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

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 600 bi
Replace
$r = c$ m
same as
Allocate

zations from
mplification:

ges security."

ges speed."

izations are
er up deficient
and security.

plifications
y at all
eed at all.

CC simplicity
curity
o speed.

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.

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

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 600 bits are allo
Replace c with 19
$r = c \bmod 2^{255}$, q
same as c modulo
Allocate 350 bits f

m

on:

y.”

,

e

ient

ty.

s

city

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.

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

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 600 bits are allocated for
Replace c with $19q + r$ whe
$r = c \bmod 2^{255}$, $q = \lfloor c/2^{25}$
same as c modulo $p = 2^{255}$
Allocate 350 bits for $19q +$

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.

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

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 600 bits are allocated for c:
Replace c with $19q + r$ where $r = c \bmod 2^{255}$, $q = \lfloor c/2^{255} \rfloor$; same as c modulo $p = 2^{255} - 19$.
Allocate 350 bits for $19q + r$.

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.

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

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 600 bits are allocated for c:
Replace c with $19q + r$ where $r = c \bmod 2^{255}$, $q = \lfloor c/2^{255} \rfloor$; same as c modulo $p = 2^{255} - 19$.
Allocate 350 bits for $19q + r$.

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.

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

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 600 bits are allocated for c:
Replace c with $19q + r$ where $r = c \bmod 2^{255}$, $q = \lfloor c/2^{255} \rfloor$; same as c modulo $p = 2^{255} - 19$.
Allocate 350 bits for $19q + r$.

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 $c - p$.

t-time Curve25519

hardware in software.
constant number of bits
integer.
perform arithmetic
ts. Don't skip bits.

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

multiplying a by b,
6 bits allocated for a
bits allocated for b:
512 bits for ab.

If 600 bits are allocated for c:
Replace c with $19q + r$ where
$r = c \bmod 2^{255}$, $q = \lfloor c/2^{255} \rfloor$;
same as c modulo $p = 2^{255} - 19$.
Allocate 350 bits for $19q + r$.

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.

to b,

cated for a

ated for b:

or $a + b$.

ng a by b,

cated for a

ated for b:

or ab.

If 600 bits are allocated for c:
Replace c with $19q + r$ where
$r = c \bmod 2^{255}$, $q = \lfloor c/2^{255} \rfloor$;
same as c modulo $p = 2^{255} - 19$.
Allocate 350 bits for $19q + r$.

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_4
meaning $\sum_i A_i 2^{32}$

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

e.

f bits

If 600 bits are allocated for c:
Replace c with $19q + r$ where
$r = c \bmod 2^{255}$, $q = \lfloor c/2^{255} \rfloor$;
same as c modulo $p = 2^{255} - 19$.
Allocate 350 bits for $19q + r$.

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
$(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, A_{10}$
$A_8, A_7, A_6, A_5, A_4, A_3, A_2,$
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 600 bits are allocated for c:
Replace c with $19q + r$ where
$r = c \bmod 2^{255}$, $q = \lfloor c/2^{255} \rfloor$;
same as c modulo $p = 2^{255} - 19$.
Allocate 350 bits for $19q + r$.

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

ts are allocated for c:
c with $19q + r$ where
od 2^{255}, $q = \lfloor c/2^{255} \rfloor$;
c modulo $p = 2^{255} - 19$.
350 bits for $19q + r$.

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

cated for c:
$q + r$ where
$= \lfloor c/2^{255} \rfloor$;
$p = 2^{255} - 19$.
for $19q + r$.

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

Reduce modulo p
subtracting a few

c:

re

$^5\rfloor$;

$-\ 19$.

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

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

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?
presuma

ST P-256

p is

$+ 2^{96} - 1.$

specifies

re given

than p^2":

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

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

$^{i}.$

$D_1; D_2; D_3; D_4$

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

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**

presumably a secu

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**,

presumably a security proble

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

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**, presumably a security problem.

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

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**, presumably a security problem.

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**, presumably a security problem.

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**, presumably a security problem.

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**,
presumably a security problem.

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,x3
for i in
  bit =
  x2,x3
  z2,z3
  x3,z3

  x2,z2
    4*x2
  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**,
presumably a security problem.

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**,
presumably a security problem.

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**,
presumably a security problem.

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,

bly a security problem.

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,
rity problem.

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

em.

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 i

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

```
255)):

)

)

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!"

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!"

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".)

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:
out 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.

Many m

blog.cr
/201403

analyzes
designin

Unneces
ECDSA:
Weierstr
variable-

Next-ge
much si
much si
much si

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

Many more issues

blog.cr.yp.to
/20140323-ecdsa

analyzes choices m
designing ECC sig

Unnecessary comp
ECDSA: scalar inv
Weierstrass incom
variable-time NAF

Next-generation E
much simpler for i
much simpler for c
much simpler for a

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.

Many more issues

blog.cr.yp.to/20140323-ecdsa.html analyzes choices made in designing ECC signatures.

Unnecessary complexity in ECDSA: scalar inversion; Weierstrass incompleteness; variable-time NAF; et al.

Next-generation ECC is much simpler for implement much simpler for designers, much simpler for auditors, e

"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.

Many more issues

`blog.cr.yp.to/20140323-ecdsa.html` analyzes choices made in designing ECC signatures.

Unnecessary complexity in ECDSA: scalar inversion; Weierstrass incompleteness; variable-time NAF; et al.

Next-generation ECC is much simpler for implementors, much simpler for designers, much simpler for auditors, etc.