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.

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?

Should cryptographers apply every imaginable simplification?

Replace GCM with ECB?

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

Should cryptographers apply every imaginable simplification?

Replace GCM with ECB?

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

Replace ECDH with FFDH?

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.

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."

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.

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.

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.

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 $a + b$.

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 $19q + r$.

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.

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

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

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.

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.

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".

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)
```

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

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.

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!"

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

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

"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".)

"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.