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.

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.

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.

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 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 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.

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.

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.

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.

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

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.

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”.

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)
```

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!"

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".)

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.

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.

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.

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.