Modern ECC signatures

2011 Bernstein–Duif–Lange–Schwabe–Yang:
Ed25519 signature scheme =
EdDSA using conservative
Curve25519 elliptic curve.
https://ed25519.cr.yp.to

32-byte public keys,
64-byte signatures,
$\approx 2^{125.8}$ security level.

Deployed in SSH, Signal,
many more applications:
https://ianix.com/pub
/ed25519-deployment.html

Many papers have explored Curve25519/Ed25519 speed.

e.g. 2015 Chou software:
on Intel Sandy Bridge (2011),
57164 cycles for keygen,
63526 cycles for signature,
205741 cycles for verification,
159128 cycles for ECDH.

Compare to, e.g., 2000 Brown–Hankerson–López–Menezes:
on Intel Pentium II (1997),
1920000 cycles for ECDH
using NIST P-256 curve.

A_C: cycles for alg A on CPU C.

Does $A_C < B_D$ prove that

A is better than B?

A_C: cycles for alg A on CPU C.
Does $A_C < B_D$ prove that
A is better than B?

No! Beware change in CPU.

Maybe $A_C > B_C$; $A_D > B_D$;
C does more work per cycle than
D, thanks to CPU manufacturer.

Sometimes people measure cost
in seconds instead of cycles.
Then they benefit
from more work per cycle and
from more cycles per second.

Better comparisons
(still raising many questions):

ECDH on Intel Pentium II/III
(still not exactly the same):
1920000 cycles for NIST P-256,
832457 cycles for Curve25519.

ECDH on Sandy Bridge:
374000 cycles for NIST P-256
(from 2013 Gueron–Krasnov),
159128 cycles for Curve25519.

Verification on Sandy Bridge:
529000 cycles for ECDSA-P-256,
205741 cycles for Ed25519.

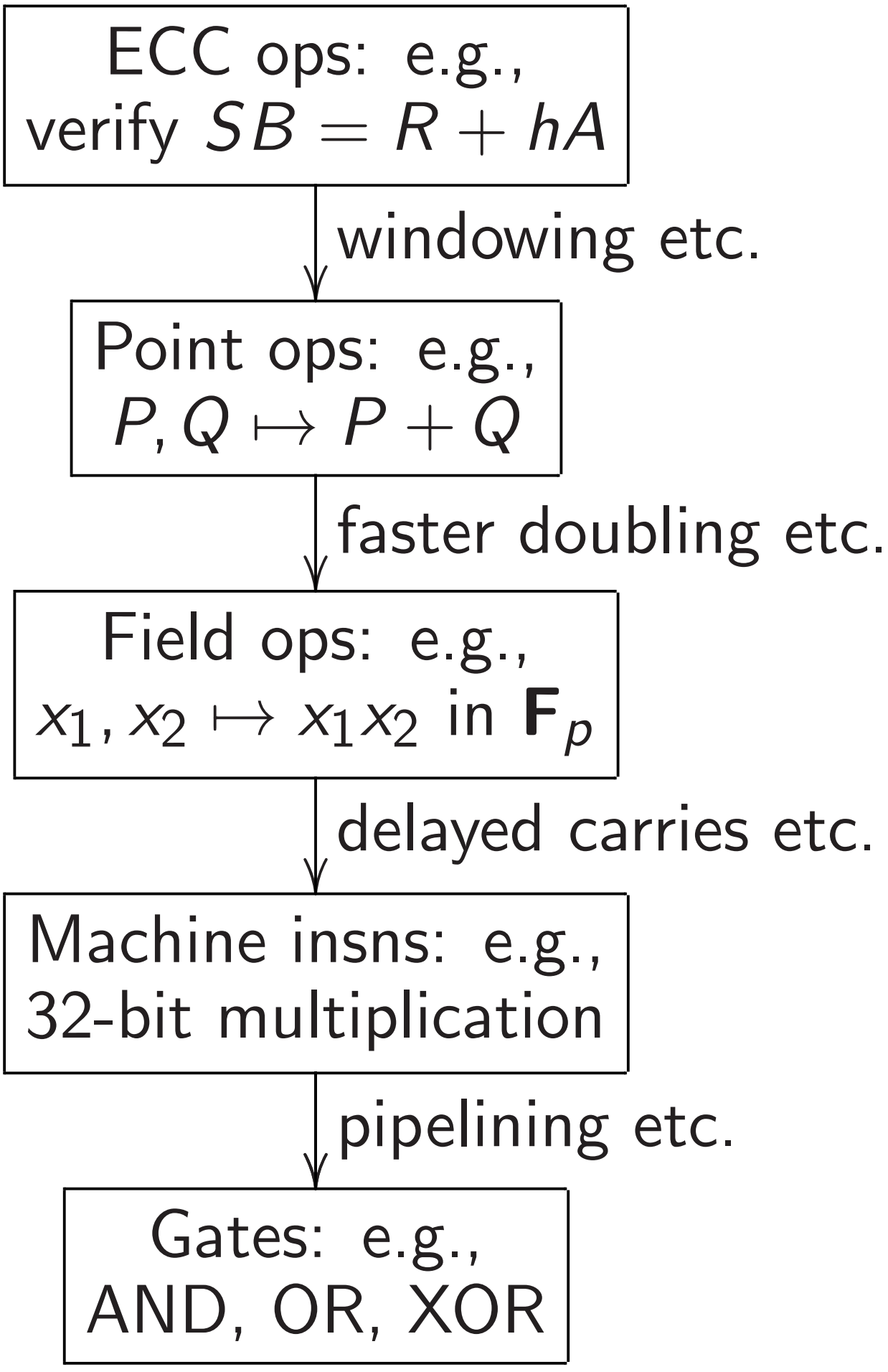
For each of these operations,
on each of these curves,
on each of these CPUs:

Simplest implementations
are much, much, much slower.

Questions in algorithm design
and software engineering:
How to build the fastest software
on, e.g., an ARM Cortex-A8 for
Ed25519 signature verification?

Answers feed back into crypto
design: e.g., choosing fast curves.

Several levels to optimize:



Single-scalar multiplication

Fundamental ECC operation:
$n, P \mapsto nP$.

Input n is integer in, e.g.,
$\{0, 1, \ldots, 2^{256} - 1\}$.

Input P is point on elliptic curve.

Will build $n, P \mapsto nP$
using additions $P, Q \mapsto P + Q$
and subtractions $P, Q \mapsto P - Q$.

Later will also look at
double-scalar multiplication
$m, P, n, Q \mapsto mP + nQ$.

Left-to-right binary method

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

Two Python notes:

- `n//2` in Python means $\lfloor n/2 \rfloor$.
- Recursion depth is limited.
  See `sys.setrecursionlimit`.

This recursion computes nP as

- $2\left(\frac{n}{2}P\right)$ if $n \in 2\mathbf{Z}$.

  e.g. $20P = 2 \cdot 10P$.

- $2\left(\frac{n-1}{2}P\right) + P$ if $n \in 1 + 2\mathbf{Z}$.

  e.g. $21P = 2 \cdot 10P + P$.

Base cases in recursion:

$0P = 0$. For Edwards: $0 = (0, 1)$.

$1P = P$. Could omit this case.

Assuming $n \geq 0$ for simplicity.

Otherwise use $nP = -(-n)P$.

If $0 \leq n < 2^b$ then

this algorithm uses

$\leq 2b - 2$ additions: specifically

$\leq b - 1$ doublings and

$\leq b - 1$ additions of P.

Example of worst case:

$31P = 2(2(2(2P+P)+P)+P)+P$.

$31 = (11111)_2$; $b = 5$;

4 doublings; 4 more additions.

Average case is better: e.g.

$35P = 2(2(2(2(2P))) + P) + P$.

$35 = (100011)_2$; $b = 6$;

5 doublings; 2 additions.

Non-adjacent form (NAF)

```
def scalarmult(n,P):
  if n == 0: return 0
  if n == 1: return P
  if n % 4 == 1:
    R = scalarmult((n-1)/4,P)
    R = R + R
    return (R + R) + P
  if n % 4 == 3:
    R = scalarmult((n+1)/4,P)
    R = R + R
    return (R + R) - P
  R = scalarmult(n/2,P)
  return R + R
```

Subtraction on the curve
is as cheap as addition.
NAF takes advantage of this.

$31P = 2(2(2(2(2P)))) - P$.
$31 = (10000\bar{1})_2$; $\bar{1}$ denotes -1.

$35P = 2(2(2(2(2P)) + P)) - P$.
$35 = (10010\bar{1})_2$.

"Non-adjacent": $\pm P$ ops are
separated by ≥ 2 doublings.

Worst case: $\approx b$ doublings
plus $\approx b/2$ additions of $\pm P$.
On average $\approx b/3$ additions.

Width-2 signed sliding windows

```
def window2(n,P,P3):
  if n == 0: return 0
  if n == 1: return P
  if n == 3: return P3
  if n % 8 == 1:
    R = window2((n-1)/8,P,P3)
    R = R + R
    R = R + R
    return (R + R) + P
  if n % 8 == 3:
    R = window2((n-3)/8,P,P3)
    R = R + R
    R = R + R
    return (R + R) + P3
```

```
  if n % 8 == 5:
    R = window2((n+3)/8,P,P3)
    R = R + R
    R = R + R
    return (R + R) - P3
  if n % 8 == 7:
    R = window2((n+1)/8,P,P3)
    R = R + R
    R = R + R
    return (R + R) - P
  R = window2(n/2,P,P3)
  return R + R

def scalarmult(n,P):
  return window2(n,P,P+P+P)
```

Worst case: $\approx b$ doublings plus $\approx b/3$ additions of $\pm P$ or $\pm 3P$.

On average $\approx b/4$ additions.

Worst case: $\approx b$ doublings plus
$\approx b/3$ additions of $\pm P$ or $\pm 3P$.
On average $\approx b/4$ additions.

Width-3 signed sliding windows:
Precompute $P, 3P, 5P, 7P$.
On average $\approx b/5$ additions.

Worst case: $\approx b$ doublings plus
$\approx b/3$ additions of $\pm P$ or $\pm 3P$.
On average $\approx b/4$ additions.

Width-3 signed sliding windows:
Precompute $P, 3P, 5P, 7P$.
On average $\approx b/5$ additions.

Width 4: Precompute
$P, 3P, 5P, 7P, 9P, 11P, 13P, 15P$.
On average $\approx b/6$ additions.

Worst case: $\approx b$ doublings plus
$\approx b/3$ additions of $\pm P$ or $\pm 3P$.
On average $\approx b/4$ additions.

Width-3 signed sliding windows:
Precompute $P, 3P, 5P, 7P$.
On average $\approx b/5$ additions.

Width 4: Precompute
$P, 3P, 5P, 7P, 9P, 11P, 13P, 15P$.
On average $\approx b/6$ additions.

Cost of precomputation
eventually outweighs savings.
Optimal: $\approx b$ doublings plus
roughly $b/\lg b$ additions.

Double-scalar multiplication

Want to quickly compute
$m, P, n, Q \mapsto mP + nQ$.

e.g. verify signature (R, S)
by computing $h = H(R, M)$,
computing $SB - hA$,
checking whether $R = SB - hA$.

Obvious approach:
Compute mP; compute nQ; add.

e.g. $b = 256$:
$\approx$256 doublings for mP,
$\approx$256 doublings for nQ,
$\approx$50 additions for mP,
$\approx$50 additions for nQ.

Joint doublings

Do much better by merging $2X + 2Y$ into $2(X + Y)$.

```
def scalarmult2(m,P,n,Q):
  if m == 0:
    return scalarmult(n,Q)
  if n == 0:
    return scalarmult(m,P)
  R = scalarmult2(m//2,P,n//2,Q)
  R = R + R
  if m % 2: R = R + P
  if n % 2: R = R + Q
  return R
```

For example: merge

$35P = 2(2(2(2(2P))) + P) + P,$

$31Q = 2(2(2(2Q+Q)+Q)+Q)+Q$

into $35P + 31Q =$

$2(2(2(2(2P+Q)+Q)+Q)+P+Q)$

$+P+Q.$

$\approx b$ doublings (merged!),

$\approx b/2$ additions of P,

$\approx b/2$ additions of Q.

Combine idea with windows: e.g.,

≈ 256 doublings for $b = 256$,

≈ 50 additions using P,

≈ 50 additions using Q.

Batch verification

Verifying many signatures:

need to be confident that

$S_1 B = R_1 + h_1 A_1$,

$S_2 B = R_2 + h_2 A_2$,

$S_3 B = R_3 + h_3 A_3$,

etc.

Obvious approach:

Check each equation separately.

Batch verification

Verifying many signatures:
need to be confident that
$S_1 B = R_1 + h_1 A_1$,
$S_2 B = R_2 + h_2 A_2$,
$S_3 B = R_3 + h_3 A_3$,
etc.

Obvious approach:
Check each equation separately.

Much faster approach:
Check random linear combination
of the equations.

Pick independent uniform random 128-bit $z_1, z_2, z_3, \ldots$.

Check whether

$(z_1S_1 + z_2S_2 + z_3S_3 + \cdots)B =$
$z_1R_1 + (z_1h_1)A_1 +$
$z_2R_2 + (z_2h_2)A_2 +$
$z_3R_3 + (z_3h_3)A_3 + \cdots$.

(If $\neq$: See 2012 Bernstein–Doumen–Lange–Oosterwijk.)

Easy to prove:
forgeries have probability $\leq 2^{-128}$ of fooling this check.

Multi-scalar multiplication

Review of asymptotic speeds:

1939 Brauer (windows):

$\approx (1 + 1/\lg b)b$
additions to compute
$P \mapsto nP$ if $n < 2^b$.

1964 Straus (joint doublings):

$\approx (1 + k/\lg b)b$
additions to compute
$P_1, \ldots, P_k \mapsto n_1 P_1 + \cdots + n_k P_k$
if $n_1, \ldots, n_k < 2^b$.

1976 Yao:

$\approx (1 + k/\lg b)b$
additions to compute
$P \mapsto n_1 P, \ldots, n_k P$
if $n_1, \ldots, n_k < 2^b$.

1976 Pippenger:

Similar asymptotics,
but replace $\lg b$ with $\lg(kb)$.
Faster than Straus and Yao
if k is large.

(Knuth says "generalization"
as if speed were the same.)

More generally, Pippenger's algorithm computes ℓ sums of multiples of k inputs.

$\approx \left(\min\{k, \ell\} + \dfrac{k\ell}{\lg(k\ell b)} \right) b$ adds if all coefficients are below 2^b.

Within $1 + \epsilon$ of optimal.

More generally, Pippenger's algorithm computes ℓ sums of multiples of k inputs.

$\approx \left(\min\{k, \ell\} + \dfrac{k\ell}{\lg(k\ell b)} \right) b$ adds if all coefficients are below 2^b.

Within $1 + \epsilon$ of optimal.

Various special cases of Pippenger's algorithm were reinvented and patented by 1993 Brickell–Gordon–McCurley–Wilson, 1995 Lim–Lee, etc.

Is that the end of the story?

No! 1989 Bos–Coster:

If $n_1 \geq n_2 \geq \cdots$ then
$n_1 P_1 + n_2 P_2 + n_3 P_3 + \cdots =$
$(n_1 - q n_2) P_1 + n_2 (q P_1 + P_2) +$
$n_3 P_3 + \cdots$ where $q = \lfloor n_1 / n_2 \rfloor$.

Remarkably simple;
competitive with Pippenger
for random choices of n_i's;
much better memory usage.

Example of Bos–Coster:

$000100000 = 32$

$000010000 = 16$

$100101100 = 300$

$010010010 = 146$

$001001101 = 77$

$000000010 = 2$

$000000001 = 1$

Goal: Compute $32P$, $16P$, $300P$, $146P$, $77P$, $2P$, $1P$.

Reduce largest row:

$000100000 = 32$

$000010000 = 16$

$010011010 = 154 \leftarrow$

$010010010 = 146$

$001001101 = 77$

$000000010 = 2$

$000000001 = 1$

Goal: Compute $32P$, $16P$, $154P$, $146P$, $77P$, $2P$, $1P$.

Plus one extra addition:

add $146P$ into $154P$,

obtaining $300P$.

Reduce largest row:

$000100000 = 32$

$000010000 = 16$

$000001000 = 8 \leftarrow$

$010010010 = 146$

$001001101 = 77$

$000000010 = 2$

$000000001 = 1$

plus 2 additions.

Reduce largest row:

$000100000 = 32$

$000010000 = 16$

$000001000 = 8$

$001000101 = 69 \leftarrow$

$001001101 = 77$

$000000010 = 2$

$000000001 = 1$

plus 3 additions.

Reduce largest row:

$000100000 = 32$

$000010000 = 16$

$000001000 = 8$

$001000101 = 69$

$000001000 = 8 \leftarrow$

$000000010 = 2$

$000000001 = 1$

plus 4 additions.

Reduce largest row:

$000100000 = 32$

$000010000 = 16$

$000001000 = 8$

$000100101 = 37 \leftarrow$

$000001000 = 8$

$000000010 = 2$

$000000001 = 1$

plus 5 additions.

Reduce largest row:

$000100000 = 32$

$000010000 = 16$

$000001000 = 8$

$000000101 = 5 \leftarrow$

$000001000 = 8$

$000000010 = 2$

$000000001 = 1$

plus 6 additions.

Reduce largest row:

$000010000 = 16 \leftarrow$

$000010000 = 16$

$000001000 = 8$

$000000101 = 5$

$000001000 = 8$

$000000010 = 2$

$000000001 = 1$

plus 7 additions.

Reduce largest row:

$000000000 = 0$

$000010000 = 16$

$000001000 = 8$

$000000101 = 5$

$000001000 = 8$

$000000010 = 2$

$000000001 = 1$

plus 7 additions.

Reduce largest row:

$000000000 = 0$

$000001000 = 8 \leftarrow$

$000001000 = 8$

$000000101 = 5$

$000001000 = 8$

$000000010 = 2$

$000000001 = 1$

plus 8 additions.

Reduce largest row:

$000000000 = 0$

$000000000 = 0 \leftarrow$

$000001000 = 8$

$000000101 = 5$

$000001000 = 8$

$000000010 = 2$

$000000001 = 1$

plus 8 additions.

Reduce largest row:

$000000000 = 0$

$000000000 = 0$

$000000000 = 0 \leftarrow$

$000000101 = 5$

$000001000 = 8$

$000000010 = 2$

$000000001 = 1$

plus 8 additions.

Reduce largest row:

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000101 = 5$

$000000011 = 3 \leftarrow$

$000000010 = 2$

$000000001 = 1$

plus 9 additions.

Reduce largest row:

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000010 = 2 \leftarrow$

$000000011 = 3$

$000000010 = 2$

$000000001 = 1$

plus 10 additions.

Reduce largest row:

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000010 = 2$

$000000001 = 1 \leftarrow$

$000000010 = 2$

$000000001 = 1$

plus 11 additions.

Reduce largest row:

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000000 = 0 \leftarrow$

$000000001 = 1$

$000000010 = 2$

$000000001 = 1$

plus 11 additions.

Reduce largest row:

$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0$$
$$000000001 = 1$$
$$000000001 = 1 \leftarrow$$
$$000000001 = 1$$

plus 12 additions.

Reduce largest row:

$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0 \leftarrow$$
$$000000001 = 1$$
$$000000001 = 1$$

plus 12 additions.

Reduce largest row:

$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0$$
$$000000000 = 0 \leftarrow$$
$$000000001 = 1$$

plus 12 additions.

Reduce largest row:

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000000 = 0$

$000000000 = 0 \leftarrow$

plus 12 additions.

Final addition chain: 1, 2, 3, 5, 8, 16, 32, 37, 69, 77, 146, 154, 300.

Short, no temporary storage, low two-operand complexity.

Revised goal: Compute
$32P_1 + 16P_2 + 300P_3 + 146P_4 +$
$77P_5 + 2P_6 + 1P_7$.

First compute $P'_4 = P_4 + P_3$
and then recursively compute
$32P_1 + 16P_2 + 154P_3 + 146P'_4 +$
$77P_5 + 2P_6 + 1P_7$.

Same scalars show up as before.

Ed25519 batch verification:
verify batch of 64 signatures
about twice as fast as
verifying each separately.