Timing attacks

1970s: TENEX operating system
compares user-supplied string
against secret password
one character at a time,
stopping at first difference:

- `AAAAAA` vs. `SECRET`: stop at 1.
- `SAAAAA` vs. `SECRET`: stop at 2.
- `SEAAAA` vs. `SECRET`: stop at 3.

Attacker sees comparison time,
deduces position of difference.
A few hundred tries
reveal secret password.

How typical software checks
16-byte authenticator:

```
for (i = 0;i < 16;++i)
  if (x[i] != y[i]) return 0;
return 1;
```

Fix, eliminating information flow
from secrets to timings:

```
uint32 diff = 0;
for (i = 0;i < 16;++i)
  diff |= x[i] ^ y[i];
return 1 & ((diff-1) >> 8);
```

Notice that the language
makes the wrong thing simple
and the right thing complex.

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

Language designer's notion of "right" is too weak for security.

So mistakes continue to happen.

One of many current examples, part of the reference software for CAESAR candidate CLOC:

```
/* compare the tag */
int i;
for(i = 0;i < CRYPTO_ABYTES;i++)
  if(tag[i] != c[(*mlen) + i]){
    return RETURN_TAG_NO_MATCH;
  }
return RETURN_SUCCESS;
```

Do timing attacks really work?

Objection: "Timings are noisy!"

Do timing attacks really work?

Objection: "Timings are noisy!"

Answer #1:

Does noise stop *all* attacks?

To guarantee security, defender must block *all* information flow.

Do timing attacks really work?

Objection: "Timings are noisy!"

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender must block *all* information flow.

Answer #2: Attacker uses statistics to eliminate noise.

Do timing attacks really work?

Objection: "Timings are noisy!"

Answer #1:
Does noise stop *all* attacks?
To guarantee security, defender
must block *all* information flow.

Answer #2: Attacker uses
statistics to eliminate noise.

Answer #3, what the
1970s attackers actually did:
Cross page boundary,
inducing page faults,
to amplify timing signal.

Examples of successful attacks:

2005 Tromer–Osvik–Shamir:
65ms to steal Linux AES key
used for hard-disk encryption.

2013 AlFardan–Paterson "Lucky Thirteen: breaking the TLS and DTLS record protocols" steals plaintext using decryption timings.

2014 van de Pol–Smart–Yarom steals Bitcoin key from timings of 25 OpenSSL signatures.

2016 Yarom–Genkin–Heninger "CacheBleed" steals RSA secret key via timings of OpenSSL.

Constant-time ECC

ECDH computation: $a, P \mapsto aP$
where a is your secret key.

Key generation: $a \mapsto aB$.

Signing: $r \mapsto rB$.

All of these use secret data.
Does timing leak this data?

Are there any branches in
ECC ops? Point ops? Field ops?
Do the underlying machine insns
take variable time?

Recall left-to-right binary method to compute $n, P \mapsto nP$ using point addition:

```
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
```

Many branches here.

NAF etc. also use many branches.

Even if each point addition
takes the same amount of time
(certainly not true in Python),
total time depends on n.

If $2^{e-1} \leq n < 2^e$ and
n has exactly w bits set:
number of additions is $e + w - 2$.

Particularly fast total time
usually indicates very small n.
"Lattice attacks" on signatures
compute the secret key given
positions of very small nonces r.

Even worse:
CPUs do not try to protect
metadata regarding branches.

Actual time for a branch
affects, and is affected by,
detailed state of code cache,
branch predictor, etc.

Attacker interacts with this state,
often sees pattern of branches.
Exploited in, e.g., Bitcoin attack.

Even worse:
CPUs do not try to protect metadata regarding branches.

Actual time for a branch affects, and is affected by, detailed state of code cache, branch predictor, etc.

Attacker interacts with this state, often sees pattern of branches. Exploited in, e.g., Bitcoin attack.

Confidence-inspiring solution:
Avoid all data flow from secrets to branch conditions.

Double-and-add-always

Eliminate branches by
always computing both results:

```
def scalarmult(n,b,P):
  if b == 0: return 0
  R = scalarmult(n//2,b-1,P)
  R2 = R + R
  S = [R2,R2 + P]
  return S[n % 2]
```

Works for $0 \leq n < 2^b$.
Always takes $2b$ additions
(including b doublings).
Use public b: bits *allowed* in n.

Another big problem:
CPUs do not try to protect
metadata regarding *array indices*.

Actual time for `x[i]`
affects, and is affected by,
detailed state of data cache,
store-to-load forwarder, etc.

Exploited in, e.g., CacheBleed,
despite Intel and OpenSSL
claiming their code was safe.

Another big problem:
CPUs do not try to protect
metadata regarding *array indices*.

Actual time for `x[i]`
affects, and is affected by,
detailed state of data cache,
store-to-load forwarder, etc.

Exploited in, e.g., CacheBleed,
despite Intel and OpenSSL
claiming their code was safe.

Confidence-inspiring solution:
Avoid all data flow from
secrets to memory addresses.

Table lookups via arithmetic

Always read all table entries.

Use bit operations to select

the desired table entry:

```
def scalarmult(n,b,P):
  if b == 0: return 0
  R = scalarmult(n//2,b-1,P)
  R2 = R + R
  S = [R2,R2 + P]
  mask = -(n % 2)
  return S[0]^(mask&(S[1]^S[0]))
```

Width-2 unsigned fixed windows

```
def fixwin2(n,b,table):
  if b <= 0: return 0
  T = table[0]
  mask = (-(1 ^ (n % 4))) >> 2
  T ^= ~mask & (T^table[1])
  mask = (-(2 ^ (n % 4))) >> 2
  T ^= ~mask & (T^table[2])
  mask = (-(3 ^ (n % 4))) >> 2
  T ^= ~mask & (T^table[3])
  R = fixwin2(n//4,b-2,table)
  R = R + R
  R = R + R
  return R + T
```

```
def scalarmult(n,b,P):
  P2 = P+P
  table = [0,P,P2,P2+P]
  return fixwin2(n,b,table)
```

Public branches, public indices.

For $b \in 2\mathbf{Z}$:
Always b doublings.
Always $b/2$ additions of T.
Always 2 additions for table.

Can similarly protect
larger-width fixed windows.
Unsigned is slightly easier.
Signed is slightly faster.

Fixed-base scalar multiplication

Obvious way to handle keygen $a \mapsto aB$ and signing $r \mapsto rB$: reuse $n, P \mapsto nP$ from ECDH.

Fixed-base scalar multiplication

Obvious way to handle keygen $a \mapsto aB$ and signing $r \mapsto rB$: reuse $n, P \mapsto nP$ from ECDH.

Can do much better since B is a constant: standard base point.

e.g. For $b = 256$: Compute $(2^{128}n_1 + n_0)B$ as $n_1B_1 + n_0B$ using double-scalar fixed windows, after precomputing $B_1 = 2^{128}B$.

Fun exercise: For each k, try to minimize number of additions using k precomputed points.

Recall Chou timings:
57164 cycles for keygen,
63526 cycles for signature,
205741 cycles for verification,
159128 cycles for ECDH.

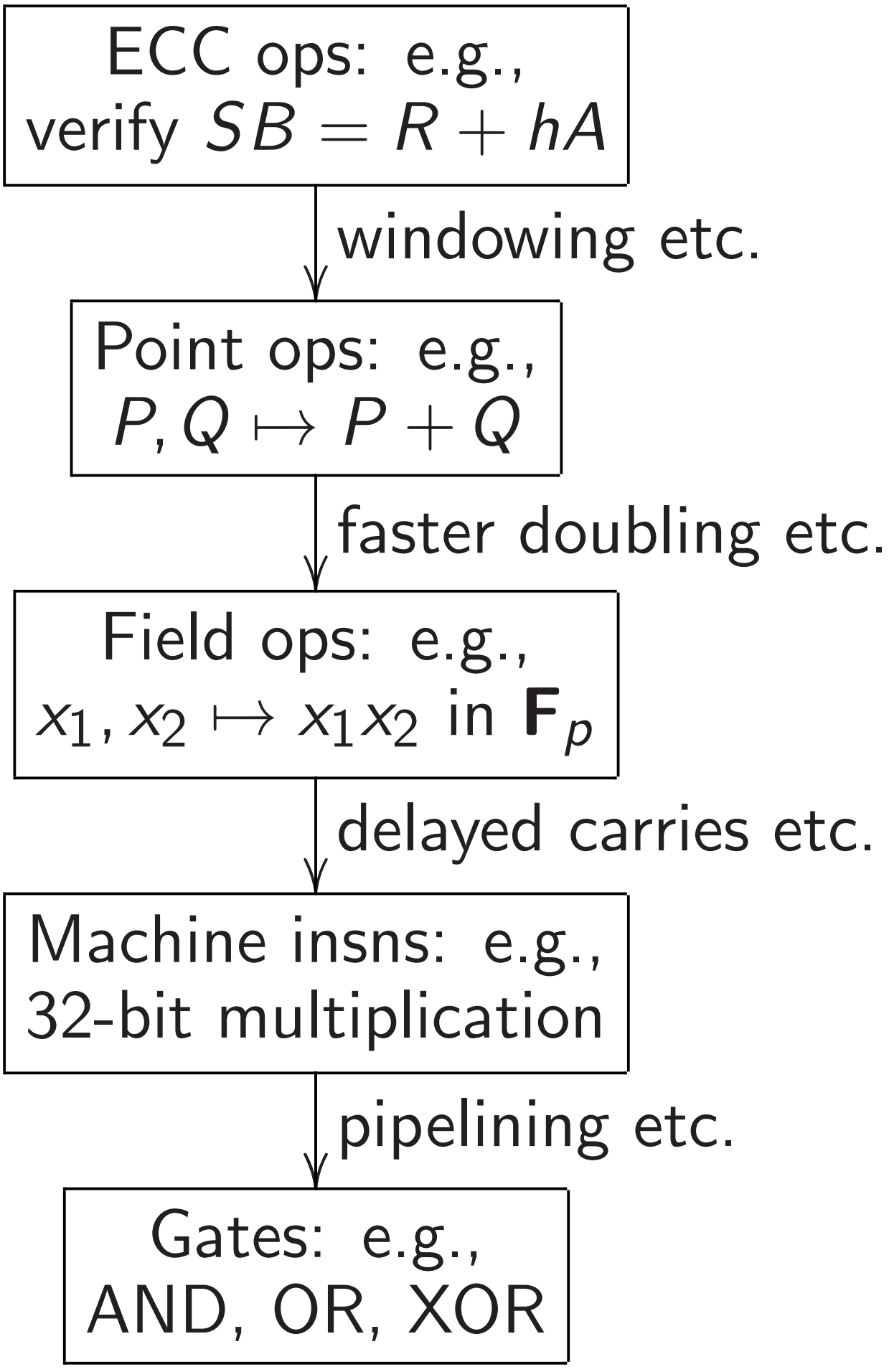
ECDH is single-scalar mult.

Verification is double-scalar mult,
somewhat slower than ECDH.
(But batch verification is faster.)

Keygen is fixed-base scalar mult,
much faster than ECDH.

Signing is keygen plus overhead
depending on message length.

Let's move down a level:



Eliminating divisions

Have to do many additions of curve points: $P, Q \mapsto P + Q$. How to efficiently decompose additions into field ops?

Addition $(x_1, y_1) + (x_2, y_2) =$
$((x_1y_2 + y_1x_2)/(1 + dx_1x_2y_1y_2),$
$(y_1y_2 - x_1x_2)/(1 - dx_1x_2y_1y_2))$
uses expensive divisions.

Eliminating divisions

Have to do many additions of curve points: $P, Q \mapsto P + Q$.
How to efficiently decompose additions into field ops?

Addition $(x_1, y_1) + (x_2, y_2) =$
$((x_1y_2 + y_1x_2)/(1 + dx_1x_2y_1y_2),$
$(y_1y_2 - x_1x_2)/(1 - dx_1x_2y_1y_2))$
uses expensive divisions.

Better: postpone divisions and work with fractions.
Represent (x, y) as $(X : Y : Z)$ with $x = X/Z$, $y = Y/Z$, $Z \neq 0$.

Addition now has to handle fractions as input:

$$\left(\frac{X_1}{Z_1}, \frac{Y_1}{Z_1}\right) + \left(\frac{X_2}{Z_2}, \frac{Y_2}{Z_2}\right) =$$

$$\left(\frac{\frac{X_1}{Z_1}\frac{Y_2}{Z_2} + \frac{Y_1}{Z_1}\frac{X_2}{Z_2}}{1 + d\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}}, \right.$$

$$\left. \frac{\frac{Y_1}{Z_1}\frac{Y_2}{Z_2} - \frac{X_1}{Z_1}\frac{X_2}{Z_2}}{1 - d\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}} \right) =$$

Addition now has to handle fractions as input:

$$\left(\frac{X_1}{Z_1}, \frac{Y_1}{Z_1}\right) + \left(\frac{X_2}{Z_2}, \frac{Y_2}{Z_2}\right) =$$

$$\left(\frac{\frac{X_1}{Z_1}\frac{Y_2}{Z_2} + \frac{Y_1}{Z_1}\frac{X_2}{Z_2}}{1 + d\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}}, \right.$$

$$\left. \frac{\frac{Y_1}{Z_1}\frac{Y_2}{Z_2} - \frac{X_1}{Z_1}\frac{X_2}{Z_2}}{1 - d\frac{X_1}{Z_1}\frac{X_2}{Z_2}\frac{Y_1}{Z_1}\frac{Y_2}{Z_2}} \right) =$$

$$\left(\frac{Z_1 Z_2 (X_1 Y_2 + Y_1 X_2)}{Z_1^2 Z_2^2 + d X_1 X_2 Y_1 Y_2}, \right.$$

$$\left. \frac{Z_1 Z_2 (Y_1 Y_2 - X_1 X_2)}{Z_1^2 Z_2^2 - d X_1 X_2 Y_1 Y_2} \right)$$

i.e. $\left(\frac{X_1}{Z_1}, \frac{Y_1}{Z_1}\right) + \left(\frac{X_2}{Z_2}, \frac{Y_2}{Z_2}\right)$

$= \left(\frac{X_3}{Z_3}, \frac{Y_3}{Z_3}\right)$

where

$F = Z_1^2 Z_2^2 - dX_1X_2Y_1Y_2,$

$G = Z_1^2 Z_2^2 + dX_1X_2Y_1Y_2,$

$X_3 = Z_1Z_2(X_1Y_2 + Y_1X_2)F,$

$Y_3 = Z_1Z_2(Y_1Y_2 - X_1X_2)G,$

$Z_3 = FG.$

Input to addition algorithm:

$X_1, Y_1, Z_1, X_2, Y_2, Z_2.$

Output from addition algorithm:

X_3, Y_3, Z_3. No divisions needed!

Eliminate common subexpressions to save multiplications:

$A = Z_1 \cdot Z_2$; $B = A^2$;
$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$E = d \cdot C \cdot D$;
$F = B - E$; $G = B + E$;
$X_3 = A \cdot F \cdot (X_1 \cdot Y_2 + Y_1 \cdot X_2)$;
$Y_3 = A \cdot G \cdot (D - C)$;
$Z_3 = F \cdot G$.

Cost: $11\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$ where $\mathbf{M}, \mathbf{S}$ are costs of mult, square. Choose small d for cheap $\mathbf{M}_d$.

Can do better: $10\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$.

Obvious $4\mathbf{M}$ method to compute product $C + Mt + Dt^2$ of polys $X_1 + Y_1 t$, $X_2 + Y_2 t$:

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = X_1 \cdot Y_2 + Y_1 \cdot X_2$.

Can do better: $10\mathbf{M} + 1\mathbf{S} + 1\mathbf{M}_d$.

Obvious $4\mathbf{M}$ method to compute product $C + Mt + Dt^2$ of polys $X_1 + Y_1 t$, $X_2 + Y_2 t$:

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = X_1 \cdot Y_2 + Y_1 \cdot X_2$.

Karatsuba's $3\mathbf{M}$ method:

$C = X_1 \cdot X_2$;
$D = Y_1 \cdot Y_2$;
$M = (X_1 + Y_1) \cdot (X_2 + Y_2) - C - D$.

Faster doubling

$$(x_1, y_1) + (x_1, y_1) =$$
$$((x_1y_1+y_1x_1)/(1+dx_1x_1y_1y_1),$$
$$(y_1y_1-x_1x_1)/(1-dx_1x_1y_1y_1)) =$$
$$((2x_1y_1)/(1 + dx_1^2y_1^2),$$
$$(y_1^2-x_1^2)/(1 - dx_1^2y_1^2)).$$

Faster doubling

$$(x_1, y_1) + (x_1, y_1) =$$
$$((x_1y_1+y_1x_1)/(1+dx_1x_1y_1y_1),$$
$$(y_1y_1-x_1x_1)/(1-dx_1x_1y_1y_1)) =$$
$$((2x_1y_1)/(1 + dx_1^2y_1^2),$$
$$(y_1^2-x_1^2)/(1 - dx_1^2y_1^2)).$$

$x_1^2 + y_1^2 = 1 + dx_1^2y_1^2$ so

$$(x_1, y_1) + (x_1, y_1) =$$
$$((2x_1y_1)/(x_1^2 + y_1^2),$$
$$(y_1^2-x_1^2)/(2 - x_1^2 - y_1^2)).$$

Faster doubling

$(x_1, y_1) + (x_1, y_1) =$
$((x_1y_1+y_1x_1)/(1+dx_1x_1y_1y_1),$
$(y_1y_1-x_1x_1)/(1-dx_1x_1y_1y_1)) =$
$((2x_1y_1)/(1 + dx_1^2y_1^2),$
$(y_1^2-x_1^2)/(1 - dx_1^2y_1^2)).$

$x_1^2 + y_1^2 = 1 + dx_1^2y_1^2$ so
$(x_1, y_1) + (x_1, y_1) =$
$((2x_1y_1)/(x_1^2 + y_1^2),$
$(y_1^2-x_1^2)/(2 - x_1^2 - y_1^2)).$

Again eliminate divisions using $(X : Y : Z)$: only $3\mathbf{M} + 4\mathbf{S}$.
Much faster than addition.

More addition strategies

Dual addition formula:

$(x_1, y_1) + (x_2, y_2) =$

$((x_1y_1 + x_2y_2)/(x_1x_2 + y_1y_2),$

$(x_1y_1 - x_2y_2)/(x_1y_2 - x_2y_1)).$

Low degree, no need for d.

More addition strategies

Dual addition formula:

$(x_1, y_1) + (x_2, y_2) =$

$((x_1y_1 + x_2y_2)/(x_1x_2 + y_1y_2),$

$(x_1y_1 - x_2y_2)/(x_1y_2 - x_2y_1)).$

Low degree, no need for d.

Warning: fails for doubling!

Is this really "addition"?

Most EC formulas have failures.

More addition strategies

Dual addition formula:
$(x_1, y_1) + (x_2, y_2) =$
$((x_1y_1 + x_2y_2)/(x_1x_2 + y_1y_2),$
$(x_1y_1 - x_2y_2)/(x_1y_2 - x_2y_1)).$
Low degree, no need for d.

Warning: fails for doubling!
Is this really "addition"?
Most EC formulas have failures.

Can test for failure cases.
Can produce constant-time code
by eliminating branches.
For some ECC ops, can prove
that failure cases never happen.

More coordinate systems: e.g.,

- inverted: $x = Z/X$, $y = Z/Y$.
- extended: $x = X/Z$, $y = Y/T$.
- completed: $x = X/Z$, $y = Y/Z$, $xy = T/Z$.

"-1-twisted Edwards curves" $-x^2 + y^2 = 1 + dx^2y^2$:
further speedups related to $-x^2 + y^2 = (y - x)(y + x)$.

Inside modern ECC operations:
$8\mathbf{M}$ for addition,
$3\mathbf{M} + 4\mathbf{S}$ for doubling.

NIST curves (e.g., P-256)
were standardized before
Edwards curves were published.

Much slower additions.

NIST curves (e.g., P-256)
were standardized before
Edwards curves were published.

Much slower additions.

Express as Edwards curves
using a field extension: slow.

NIST curves (e.g., P-256)
were standardized before
Edwards curves were published.

Much slower additions.

Express as Edwards curves
using a field extension: slow.

How did Curve25519 obtain
good speeds for ECDH?
"Montgomery curve with
the Montgomery ladder."

NIST curves (e.g., P-256)
were standardized before
Edwards curves were published.

Much slower additions.

Express as Edwards curves
using a field extension: slow.

How did Curve25519 obtain
good speeds for ECDH?
"Montgomery curve with
the Montgomery ladder."

Why did NIST not choose
Montgomery curves? Unclear.