Cryptographic software engineering, part 2

Daniel J. Bernstein

Previous part:

- General software engineering.
- Using const-time instructions.

Software optimization

Almost all software is
much slower than it could be.

Software optimization

Almost all software is
much slower than it could be.

Is software applied to much data?
Usually not. Usually the
wasted CPU time is negligible.

Software optimization

Almost all software is
much slower than it could be.

Is software applied to much data?
Usually not. Usually the
wasted CPU time is negligible.

But *crypto software* should be
applied to all communication.

Crypto that's too slow
$\Rightarrow$ fewer users
$\Rightarrow$ fewer cryptanalysts
$\Rightarrow$ less attractive for everybody.

Typical situation:

X is a cryptographic system.

You have written a (const-time) reference implementation of X.

You want (const-time) software that computes X as efficiently as possible.

You have chosen a target CPU. (Can repeat for other CPUs.)

You measure performance of the implementation. Now what?

A simplified example

Target CPU: TI LM4F120H5QR microcontroller containing one ARM Cortex-M4F core.

Reference implementation:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;++i)
    result += x[i];
  return result;
}
```

Counting cycles:

```
static volatile unsigned int
  *const DWT_CYCCNT
  = (void *) 0xE0001004;
...

int beforesum = *DWT_CYCCNT;
int result = sum(x);
int aftersum = *DWT_CYCCNT;
UARTprintf("sum %d %d\n",
  result,aftersum-beforesum);
```

Output shows 8012 cycles.

Change 1000 to 500: 4012.

"Okay, 8 cycles per addition.
Um, are microcontrollers
really this slow at addition?"

"Okay, 8 cycles per addition.
Um, are microcontrollers
really this slow at addition?"

Bad practice:
Apply random "optimizations"
(and tweak compiler options)
until you get bored.
Keep the fastest results.

"Okay, 8 cycles per addition.
Um, are microcontrollers
really this slow at addition?"

Bad practice:
Apply random "optimizations"
(and tweak compiler options)
until you get bored.
Keep the fastest results.

Try `-Os`: 8012 cycles.

"Okay, 8 cycles per addition.
Um, are microcontrollers
really this slow at addition?"

Bad practice:
Apply random "optimizations"
(and tweak compiler options)
until you get bored.
Keep the fastest results.

Try `-Os`: 8012 cycles.
Try `-O1`: 8012 cycles.

"Okay, 8 cycles per addition.
Um, are microcontrollers
really this slow at addition?"

Bad practice:
Apply random "optimizations"
(and tweak compiler options)
until you get bored.
Keep the fastest results.

Try `-Os`: 8012 cycles.
Try `-O1`: 8012 cycles.
Try `-O2`: 8012 cycles.

"Okay, 8 cycles per addition.
Um, are microcontrollers
really this slow at addition?"

Bad practice:
Apply random "optimizations"
(and tweak compiler options)
until you get bored.
Keep the fastest results.

Try `-Os`: 8012 cycles.
Try `-O1`: 8012 cycles.
Try `-O2`: 8012 cycles.
Try `-O3`: 8012 cycles.

Try moving the pointer:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;++i)
    result += *x++;
  return result;
}
```

Try moving the pointer:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;++i)
    result += *x++;
  return result;
}
```

8010 cycles.

Try counting down:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 1000;i > 0;--i)
    result += *x++;
  return result;
}
```

Try counting down:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 1000;i > 0;--i)
    result += *x++;
  return result;
}
```

8010 cycles.

Try using an end pointer:

```
int sum(int *x)
{
  int result = 0;
  int *y = x + 1000;
  while (x != y)
    result += *x++;
  return result;
}
```

Try using an end pointer:

```
int sum(int *x)
{
  int result = 0;
  int *y = x + 1000;
  while (x != y)
    result += *x++;
  return result;
}
```

8010 cycles.

Back to original. Try unrolling:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;i += 2) {
    result += x[i];
    result += x[i + 1];
  }
  return result;
}
```

Back to original. Try unrolling:

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;i += 2) {
    result += x[i];
    result += x[i + 1];
  }
  return result;
}
```

5016 cycles.

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;i += 5) {
    result += x[i];
    result += x[i + 1];
    result += x[i + 2];
    result += x[i + 3];
    result += x[i + 4];
  }
  return result;
}
```

```
int sum(int *x)
{
  int result = 0;
  int i;
  for (i = 0;i < 1000;i += 5) {
    result += x[i];
    result += x[i + 1];
    result += x[i + 2];
    result += x[i + 3];
    result += x[i + 4];
  }
  return result;
}
```

4016 cycles. “Are we done yet?”

"Why is this bad practice?
Didn't we succeed
in making code twice as fast?"

"Why is this bad practice?
Didn't we succeed
in making code twice as fast?"

Yes, but CPU time is still
nowhere near optimal,
and human time was wasted.

"Why is this bad practice?
Didn't we succeed
in making code twice as fast?"

Yes, but CPU time is still
nowhere near optimal,
and human time was wasted.

Good practice:
Figure out lower bound for
cycles spent on arithmetic etc.
Understand gap between
lower bound and observed time.

Find “ARM Cortex-M4 Processor Technical Reference Manual”.

Rely on Wikipedia comment that M4F $=$ M4 $+$ floating-point unit.

Find “ARM Cortex-M4 Processor Technical Reference Manual”.

Rely on Wikipedia comment that M4F $=$ M4 $+$ floating-point unit.

Manual says that Cortex-M4 “implements the ARMv7E-M architecture profile”.

Find "ARM Cortex-M4 Processor Technical Reference Manual". Rely on Wikipedia comment that M4F $=$ M4 $+$ floating-point unit.

Manual says that Cortex-M4 "implements the ARMv7E-M architecture profile".

Points to the "ARMv7-M Architecture Reference Manual", which defines instructions: e.g., "ADD" for 32-bit addition.

First manual says that ADD takes just 1 cycle.

Inputs and output of ADD are “integer registers”. ARMv7-M has 16 integer registers, including special-purpose “stack pointer” and “program counter”.

Inputs and output of ADD are “integer registers”. ARMv7-M has 16 integer registers, including special-purpose “stack pointer” and “program counter”.

Each element of `x` array needs to be “loaded” into a register.

Inputs and output of ADD are “integer registers”. ARMv7-M has 16 integer registers, including special-purpose “stack pointer” and “program counter”.

Each element of `x` array needs to be “loaded” into a register.

Basic load instruction: LDR. Manual says 2 cycles but adds a note about “pipelining”. Then more explanation: if next instruction is also LDR (with address not based on first LDR) then it saves 1 cycle.

n consecutive LDRs
takes only $n + 1$ cycles
("more multiple LDRs can be pipelined together").

Can achieve this speed
in other ways (LDRD, LDM)
but nothing seems faster.

Lower bound for n LDR + n ADD:
$2n + 1$ cycles,
including n cycles of arithmetic.

Why observed time is higher:
non-consecutive LDRs;
costs of manipulating `i`.

```
int sum(int *x)
{
  int result = 0;
  int *y = x + 1000;
  int x0,x1,x2,x3,x4,
      x5,x6,x7,x8,x9;

  while (x != y) {
    x0 = 0[(volatile int *)x];
    x1 = 1[(volatile int *)x];
    x2 = 2[(volatile int *)x];
    x3 = 3[(volatile int *)x];
    x4 = 4[(volatile int *)x];
    x5 = 5[(volatile int *)x];
    x6 = 6[(volatile int *)x];
```

```
x7 = 7[(volatile int *)x];
x8 = 8[(volatile int *)x];
x9 = 9[(volatile int *)x];
result += x0;
result += x1;
result += x2;
result += x3;
result += x4;
result += x5;
result += x6;
result += x7;
result += x8;
result += x9;
x0 = 10[(volatile int *)x];
x1 = 11[(volatile int *)x];
```

```
x2 = 12[(volatile int *)x];
x3 = 13[(volatile int *)x];
x4 = 14[(volatile int *)x];
x5 = 15[(volatile int *)x];
x6 = 16[(volatile int *)x];
x7 = 17[(volatile int *)x];
x8 = 18[(volatile int *)x];
x9 = 19[(volatile int *)x];
x += 20;
result += x0;
result += x1;
result += x2;
result += x3;
result += x4;
result += x5;
```

```
    result += x6;
    result += x7;
    result += x8;
    result += x9;
  }

  return result;
}
```

```
    result += x6;
    result += x7;
    result += x8;
    result += x9;
  }

  return result;
}
```

2526 cycles. Even better in asm.

```
    result += x6;
    result += x7;
    result += x8;
    result += x9;
  }

  return result;
}
```

2526 cycles. Even better in asm.

Wikipedia: "By the late 1990s for even performance sensitive code, optimizing compilers exceeded the performance of human experts."

```
    result += x6;
    result += x7;
    result += x8;
    result += x9;
  }

  return result;
}
```

2526 cycles. Even better in asm.

Wikipedia: “By the late 1990s for even performance sensitive code, optimizing compilers exceeded the performance of human experts.”
— [citation needed]

A real example

Salsa20 reference software:
30.25 cycles/byte on this CPU.

Lower bound for arithmetic:
64 bytes require
$21 \cdot 16$ 1-cycle ADDs,
$20 \cdot 16$ 1-cycle XORs,
so at least 10.25 cycles/byte.

Also many rotations, but
ARMv7-M instruction set
includes free rotation
as part of XOR instruction.
(Compiler knows this.)

Detailed benchmarks show several cycles/byte spent on `load_littleendian` and `store_littleendian`.

Can replace with LDR and STR. (Compiler doesn't see this.)

Then observe 23 cycles/byte:
18 cycles/byte for rounds,
plus 5 cycles/byte overhead.
Still far above 10.25 cycles/byte.

Detailed benchmarks show several cycles/byte spent on `load_littleendian` and `store_littleendian`.

Can replace with LDR and STR. (Compiler doesn't see this.)

Then observe 23 cycles/byte: 18 cycles/byte for rounds, plus 5 cycles/byte overhead. Still far above 10.25 cycles/byte.

Gap is mostly loads, stores. Minimize load/store cost by choosing "spills" carefully.

Which of the 16 Salsa20 words
should be in registers?
Don't trust compiler to
optimize register allocation.

Which of the 16 Salsa20 words
should be in registers?
Don't trust compiler to
optimize register allocation.

Make loads consecutive?
Don't trust compiler to
optimize instruction scheduling.

Which of the 16 Salsa20 words
should be in registers?
Don't trust compiler to
optimize register allocation.

Make loads consecutive?
Don't trust compiler to
optimize instruction scheduling.

Spill to FPU instead of stack?
Don't trust compiler to
optimize instruction selection.

Which of the 16 Salsa20 words
should be in registers?
Don't trust compiler to
optimize register allocation.

Make loads consecutive?
Don't trust compiler to
optimize instruction scheduling.

Spill to FPU instead of stack?
Don't trust compiler to
optimize instruction selection.

On bigger CPUs,
selecting vector instructions
is critical for performance.

https://bench.cr.yp.to
includes 2392 implementations
of 614 cryptographic primitives.
$>$20 implementations of Salsa20.

Haswell: Reasonably simple `ref`
implementation compiled with
`gcc -O3 -fomit-frame-pointer`
is 6.15$\times$ slower than fastest
Salsa20 implementation.

https://bench.cr.yp.to
includes 2392 implementations
of 614 cryptographic primitives.
$>$20 implementations of Salsa20.

Haswell: Reasonably simple `ref`
implementation compiled with
`gcc -O3 -fomit-frame-pointer`
is 6.15$\times$ slower than fastest
Salsa20 implementation.

`merged` implementation
with "machine-independent"
optimizations and best of 121
compiler options: 4.52$\times$ slower.

Fast random permutations

Goal: Put list $(x_1, \ldots, x_n)$ into a random order.

Fast random permutations

Goal: Put list $(x_1, \ldots, x_n)$ into a random order.

One textbook strategy:
Sort $(Mr_1 + x_1, \ldots, Mr_n + x_n)$ for random $(r_1, \ldots, r_n)$, suitable M.

Fast random permutations

Goal: Put list $(x_1, \ldots, x_n)$
into a random order.

One textbook strategy:
Sort $(Mr_1 + x_1, \ldots, Mr_n + x_n)$ for
random $(r_1, \ldots, r_n)$, suitable M.

McEliece encryption example:
Randomly order 6960 bits
$(1, \ldots, 1, 0, \ldots, 0)$, weight 119.

Fast random permutations

Goal: Put list $(x_1, \ldots, x_n)$ into a random order.

One textbook strategy: Sort $(Mr_1 + x_1, \ldots, Mr_n + x_n)$ for random $(r_1, \ldots, r_n)$, suitable M.

McEliece encryption example: Randomly order 6960 bits $(1, \ldots, 1, 0, \ldots, 0)$, weight 119.

NTRU encryption example: Randomly order 761 trits $(\pm 1, \ldots, \pm 1, 0, \ldots, 0)$, wt 286.

Simulate uniform random r_i
using RNG: e.g., stream cipher.

Simulate uniform random r_i using RNG: e.g., stream cipher.

How many bits in r_i? Negligible collisions? Occasional collisions?

Simulate uniform random r_i using RNG: e.g., stream cipher.

How many bits in r_i? Negligible collisions? Occasional collisions?

Restart on collision?
Uniform distribution; some cost.

Simulate uniform random r_i
using RNG: e.g., stream cipher.

How many bits in r_i? Negligible
collisions? Occasional collisions?

Restart on collision?
Uniform distribution; some cost.

Example: $n = 6960$ bits;
weight 119; 31-bit r_i; no restart.
Any output is produced in
$\le 119!(n-119)!\binom{2^{31}+n-1}{n}$ ways;
i.e., $< 1.02 \cdot 2^{31n}/\binom{n}{119}$ ways.
Factor $<$1.02 increase in
attacker's chance of winning.

Which sorting algorithm?

Reference bubblesort code does $n(n-1)/2$ `minmax` operations.

Which sorting algorithm?

Reference bubblesort code does $n(n-1)/2$ `minmax` operations.

Many standard algorithms use fewer operations: mergesort, quicksort, heapsort, radixsort, etc.

But these algorithms rely on secret branches and secret indices.

Which sorting algorithm?

Reference bubblesort code does $n(n-1)/2$ `minmax` operations.

Many standard algorithms use fewer operations: mergesort, quicksort, heapsort, radixsort, etc.

But these algorithms rely on secret branches and secret indices.

Exercise: convert mergesort into constant-time mergesort using $\Theta(n^2)$ operations.

Converting bubblesort into constant-time bubblesort loses only a constant factor: cost of constant-time `minmax`.

Converting bubblesort into constant-time bubblesort loses only a constant factor: cost of constant-time `minmax`.

"Sorting network": sorting algorithm built as constant sequence of `minmax` operations ("comparators").

Converting bubblesort into constant-time bubblesort loses only a constant factor: cost of constant-time `minmax`.

"Sorting network": sorting algorithm built as constant sequence of `minmax` operations ("comparators").

Sorting network on next slide: Batcher's merge-exchange sort. $\Theta(n(\log n)^2)$ `minmax` operations; $(1/4)(e^2 - e + 4)n - 1$ for $n = 2^e$.

```
void sort(int32 *x,long long n)
{ long long t,p,q,i;
  t = 1; if (n < 2) return;
  while (t < n-t) t += t;
  for (p = t;p > 0;p >>= 1) {
    for (i = 0;i < n-p;++i)
      if (!(i & p))
        minmax(x+i,x+i+p);
    for (q = t;q > p;q >>= 1)
      for (i = 0;i < n-q;++i)
        if (!(i & p))
          minmax(x+i+p,x+i+q);
  }
}
```

How many cycles on, e.g.,
Intel Haswell CPU core?

Every cycle: a vector of 8 32-bit "min" operations and a vector of 8 32-bit "max" operations.

How many cycles on, e.g., Intel Haswell CPU core?

Every cycle: a vector of 8 32-bit “min” operations and a vector of 8 32-bit “max” operations.

$\geq$3008 cycles for $n = 1024$.
Current software: 7328 cycles.

How many cycles on, e.g.,
Intel Haswell CPU core?

Every cycle: a vector of 8 32-bit "min" operations and a vector of 8 32-bit "max" operations.

$\geq$3008 cycles for $n = 1024$.
Current software: 7328 cycles.
(Can gap be narrowed?)

How many cycles on, e.g., Intel Haswell CPU core?

Every cycle: a vector of 8 32-bit "min" operations and a vector of 8 32-bit "max" operations.

$\geq$3008 cycles for $n = 1024$.
Current software: 7328 cycles.
(Can gap be narrowed?)

This is fastest available sorting software. Much faster than, e.g., Intel's "Integrated Performance Primitives" software library.

Constant-time code faster than "optimized" non-constant-time code? How is this possible?

Constant-time code faster than "optimized" non-constant-time code? How is this possible?

People optimize algorithms for a naive model of CPUs:

- Branches are fast.
- Random access is fast.

Constant-time code faster than "optimized" non-constant-time code? How is this possible?

People optimize algorithms for a naive model of CPUs:

- Branches are fast.
- Random access is fast.

CPUs are evolving farther and farther away from this naive model.
Fundamental hardware costs of constant-time arithmetic are much lower than random access.

Modular arithmetic

Basic ECC operations:
add, sub, mul of, e.g.,
integers mod $2^{255} - 19$.

(Basic NTRU operations:
add, sub, mul of, e.g.,
polynomials mod $x^{761} - x - 1$.)

Modular arithmetic

Basic ECC operations:
add, sub, mul of, e.g.,
integers mod $2^{255} - 19$.

(Basic NTRU operations:
add, sub, mul of, e.g.,
polynomials mod $x^{761} - x - 1$.)

Typical "big-integer library":
a variable-length `uint32` string
$(f_0, f_1, \ldots, f_{\ell-1})$ represents
the nonnegative integer
$f_0 + 2^{32} f_1 + \cdots + 2^{32(\ell-1)} f_{\ell-1}$.
Uniqueness: $\ell = 0$ or $f_{\ell-1} \neq 0$.

Library provides functions acting on this representation: (1) $f, g \mapsto fg$; (2) $f, g \mapsto f \bmod g$; etc.

Library provides functions acting on this representation: (1) $f, g \mapsto fg$; (2) $f, g \mapsto f \bmod g$; etc.

ECC implementor using library: multiply $f, g \bmod 2^{255} - 19$ by (1) multiplying f by g; (2) reducing mod $2^{255} - 19$.

Library provides functions acting on this representation: (1) $f, g \mapsto fg$; (2) $f, g \mapsto f \bmod g$; etc.

ECC implementor using library:
multiply f, g mod $2^{255} - 19$
by (1) multiplying f by g;
(2) reducing mod $2^{255} - 19$.

But these functions take variable time to ensure uniqueness!

Library provides functions acting on this representation: (1) $f, g \mapsto fg$; (2) $f, g \mapsto f \bmod g$; etc.

ECC implementor using library: multiply $f, g \bmod 2^{255} - 19$ by (1) multiplying f by g; (2) reducing mod $2^{255} - 19$.

But these functions take variable time to ensure uniqueness!

Need a different representation for constant-time arithmetic. Can also gain speed this way.

Constant-time bigint library:
a constant-length `uint32` string
$(f_0, f_1, \ldots, f_{\ell-1})$ represents
the nonnegative integer
$f_0 + 2^{32} f_1 + \cdots + 2^{32(\ell-1)} f_{\ell-1}$.

Adding two ℓ-limb integers:
always allocate $\ell + 1$ limbs.
Don't remove top zero limb.

Constant-time bigint library:
a constant-length `uint32` string
$(f_0, f_1, \ldots, f_{\ell-1})$ represents
the nonnegative integer
$f_0 + 2^{32} f_1 + \cdots + 2^{32(\ell-1)} f_{\ell-1}$.

Adding two ℓ-limb integers:
always allocate $\ell + 1$ limbs.
Don't remove top zero limb.

Can also track bounds more
refined than $2^0, 2^{32}, 2^{64}, 2^{96}, \ldots$;
but no limbs$\rightarrow$bounds data flow.

Constant-time bigint library:
a constant-length `uint32` string
$(f_0, f_1, \ldots, f_{\ell-1})$ represents
the nonnegative integer
$f_0 + 2^{32} f_1 + \cdots + 2^{32(\ell-1)} f_{\ell-1}$.

Adding two ℓ-limb integers:
always allocate $\ell + 1$ limbs.
Don't remove top zero limb.

Can also track bounds more
refined than $2^0, 2^{32}, 2^{64}, 2^{96}, \ldots$;
but no limbs$\to$bounds data flow.

$f \bmod p$ is as short as p.

Usually faster representation: `uint32` string $(f_0, f_1, \ldots, f_9)$ represents $f_0 + 2^{26} f_1 + 2^{51} f_2 + 2^{77} f_3 + 2^{102} f_4 + 2^{128} f_5 + 2^{153} f_6 + 2^{179} f_7 + 2^{204} f_8 + 2^{230} f_9$.

Constant bound on each f_i.

More limbs than before, but save time by avoiding overflows and delaying carries.

After multiplication, replace 2^{255} with 19.

Usually faster representation:
`uint32` string $(f_0, f_1, \ldots, f_9)$
represents $f_0 + 2^{26} f_1 + 2^{51} f_2 + 2^{77} f_3 + 2^{102} f_4 + 2^{128} f_5 + 2^{153} f_6 + 2^{179} f_7 + 2^{204} f_8 + 2^{230} f_9$.

Constant bound on each f_i.

More limbs than before,
but save time by avoiding
overflows and delaying carries.

After multiplication,
replace 2^{255} with 19.

Slightly faster on some CPUs:
`int32` string $(f_0, f_1, \ldots, f_9)$.

```
int32 f7_2 = 2 * f7;
int32 g7_19 = 19 * g7;
...
int64 f0g4 = f0 * (int64) g4;
int64 f7g7_38 =
  f7_2 * (int64) g7_19;
...
int64 h4 = f0g4 + f1g3_2
         + f2g2 + f3g1_2
         + f4g0 + f5g9_38
         + f6g8_19 + f7g7_38
         + f8g6_19 + f9g5_38;
...
c4 = (h4 + (int64)(1<<25)) >> 26;
h5 += c4; h4 -= c4 << 26;
```

Initial computation of `h0`, . . . , `h9` is polynomial multiplication modulo $x^{10} - 19$.

Exercise: Which polynomials are being multiplied?

Initial computation of `h0`, . . . , `h9` is polynomial multiplication modulo $x^{10} - 19$.

Exercise: Which polynomials are being multiplied?

Reduction modulo $x^{10} - 19$ and carries such as `h4`$\rightarrow$`h5` **squeeze** the product into limited-size representation suitable for next multiplication.

Initial computation of `h0`, . . . , `h9` is polynomial multiplication modulo $x^{10} - 19$.
Exercise: Which polynomials are being multiplied?

Reduction modulo $x^{10} - 19$ and carries such as `h4`$\to$`h5` **squeeze** the product into limited-size representation suitable for next multiplication.

At end of computation: **freeze** representation into unique representation suitable for network transmission.

Much more about ECC speed:
see, e.g., 2015 Chou.

Much more about ECC speed:
see, e.g., 2015 Chou.

Verifying constant time:
increasingly automated.

Much more about ECC speed:
see, e.g., 2015 Chou.

Verifying constant time:
increasingly automated.

Testing can miss rare bugs
that attacker might trigger.
Fix: prove that software
matches mathematical spec;
have computer check proofs.

Much more about ECC speed:
see, e.g., 2015 Chou.

Verifying constant time:
increasingly automated.

Testing can miss rare bugs
that attacker might trigger.
Fix: prove that software
matches mathematical spec;
have computer check proofs.

Progress in deploying proven
fast software: see, e.g., 2015
Bernstein–Schwabe "gfverif";
2017 HACL* X25519 in Firefox.

gfverif has verified `ref10` implementation of X25519, plus occasional annotations, against the following specification:

```
p = 2**255-19
A = 486662
x2,z2,x3,z3 = 1,0,x1,1
for i in reversed(range(255)):
  ni = bit(n,i)
  x2,x3 = cswap(x2,x3,ni)
  z2,z3 = cswap(z2,z3,ni)
  x3,z3 = (4*(x2*x3-z2*z3)**2,
   4*x1*(x2*z3-z2*x3)**2)
  x2,z2 = ((x2**2-z2**2)**2,
   4*x2*z2*(x2**2+A*x2*z2+z2**2))
```

```
  x3,z3 = (x3%p,z3%p)
  x2,z2 = (x2%p,z2%p)
  cut(x2)
  cut(x3)
  cut(z2)
  cut(z3)
  x2,x3 = cswap(x2,x3,ni)
  z2,z3 = cswap(z2,z3,ni)
cut(x2)
cut(z2)
return x2*pow(z2,p-2,p)
```

What's verified: output of `ref10` is the same as spec mod p, and is between 0 and $p-1$.

"What a difference a prime makes"

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”?

Variable-time loop is unsafe.

What is “a few copies”?
Variable-time loop is unsafe.

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”?
Variable-time loop is unsafe.

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”?
Variable-time loop is unsafe.

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?

There are many more ways that cryptographic design choices affect difficulty of building fast correct constant-time software.

e.g. ECDSA needs divisions of scalars. EdDSA doesn't.

e.g. ECDSA splits elliptic-curve additions into several cases. EdDSA uses complete formulas.

There are many more ways that cryptographic design choices affect difficulty of building fast correct constant-time software.

e.g. ECDSA needs divisions of scalars. EdDSA doesn't.

e.g. ECDSA splits elliptic-curve additions into several cases. EdDSA uses complete formulas.

What's better use of time: implementing ECDSA, or upgrading protocol to EdDSA?