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.
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.

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
⇒ fewer users
⇒ fewer cryptanalysts
⇒ less attractive for everybody.
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
⇒ fewer users
⇒ fewer cryptanalysts
⇒ 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?
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
⇒ fewer users
⇒ fewer cryptanalysts
⇒ less attractive for everybody.
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
⇒ fewer users
⇒ fewer cryptanalysts
⇒ less attractive for everybody.
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 ⇒ fewer users
⇒ fewer cryptanalysts
⇒ less attractive for everybody.
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 ⇒ fewer users ⇒ fewer cryptanalysts ⇒ 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:

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

Almost all software is much slower than it could be.

Is software applied to too 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 ⇒ fewer users ⇒ fewer cryptanalysts ⇒ 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:

```c
int sum(int *x) {
    int result = 0;
    int i;
    for (i = 0; i < 1000; ++i)
        result += x[i];
    return result;
}
```
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 ⇒ fewer users ⇒ fewer cryptanalysts ⇒ 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:

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

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

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

Reference implementation:

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

Counting cycles:

```c
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
", result, aftersum-beforesum);
```

Output shows 8012 cycles.

Change 1000 to 500: 4012.
A simplified example

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

Reference implementation:

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

Counting cycles:

```c
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.
A simplified example

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

Reference implementation:

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

Counting cycles:

```c
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.
```
A simplified example

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

Reference implementation:

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

Counting cycles:

```c
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.
A simplified example

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

Reference implementation:

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

Counting cycles:

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

```c
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?”
A simplified example

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

Reference implementation:

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

Counting cycles:

```c
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
", 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?”
Counting cycles:

```c
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?”
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?”
Counting cycles:

```c
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?”

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

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

Try -Os: 8012 cycles.
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?”

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.
Counting cycles:

```c
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?”

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.
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?”

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.
Counting cycles:

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

```c
int beforesum = *DWT_CYCCNT;
int result = sum(x);
int aftersum = *DWT_CYCCNT;
UARTprintf("sum %d %d
", 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?”

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:

```c
int sum(int *x)
{
int result = 0;
int i;
for (i = 0;i < 1000;++i)
result += *x++;
return result;
}
```
“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;
}
```
“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 -01: 8012 cycles.
Try -02: 8012 cycles.
Try -03: 8012 cycles.

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

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

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

8010 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:

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

8010 cycles.

Try counting down:

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

```c
int sum(int *x)
{
    int result = 0;
    int i;
    for (i = 0; i < 1000; ++i)
        result += *x++;
    return result;
}
```
8010 cycles.

Try counting down:

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

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

8010 cycles.
```

Try counting down:

```c
int sum(int *x)
{
    int result = 0;
    int i;
    for (i = 1000; i > 0; --i)
        result += *x++;
    return result;
}
```
Try moving the pointer:

```c
int sum(int *x)
{
    int result = 0;
    int i;
    for (i = 0; i < 1000; ++i)
        result += *x++;
    return result;
}
8010 cycles.
```

Try counting down:

```c
int sum(int *x)
{
    int result = 0;
    int i;
    for (i = 1000; i > 0; --i)
        result += *x++;
    return result;
}
Try moving the pointer:

```c
int sum(int *x)
{
    int result = 0;
    int i;
    for (i = 0; i < 1000; ++i)
        result += *x++;
    return result;
}
8010 cycles.
```

Try counting down:

```c
int sum(int *x)
{
    int result = 0;
    int i;
    for (i = 1000; i > 0; --i)
        result += *x++;
    return result;
}
8010 cycles.
```
Try moving the pointer:

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

8010 cycles.

Try counting down:

```c
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:

```c
int sum(int *x)
{
    int result = 0;
    int *y = x + 1000;
    while (x != y)
        result += *x++;
    return result;
}
```
Try moving the pointer:

```c
int sum(int *x)
{
    int result = 0;
    int i;
    for (i = 0; i < 1000; ++i)
        result += *x++;
    return result;
}
```
8010 cycles.

Try counting down:

```c
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:

```c
int sum(int *x)
{
    int result = 0;
    int *y = x + 1000;
    while (x != y)
        result += *x++;
    return result;
}
```
Try counting down:

```c
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:

```c
int sum(int *x)
{
    int result = 0;
    int *y = x + 1000;
    while (x != y)
        result += *x++;
    return result;
}
```
Try counting down:

```c
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:

```c
int sum(int *x)
{
    int result = 0;
    int *y = x + 1000;
    while (x != y)
        result += *x++;
    return result;
}
```
Try counting down:

```c
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:

```c
int sum(int *x)
{
    int result = 0;
    int *y = x + 1000;
    while (x != y)
        result += *x++;
    return result;
}
8010 cycles.
```
Try counting down:

```c
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:

```c
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:

```c
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;
}
```
Try counting down:

```c
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:

```c
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:

```c
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;
}
```
Try using an end pointer:

```c
int sum(int *x)
{
    int result = 0;
    int *y = x + 1000;
    while (x != y)
    {
        result += *x++;
        result += *x++;
    }
    return result;
}
8010 cycles.
```

Back to original. Try unrolling:

```c
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;
}
Try using an end pointer:

```c
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:

```c
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;
}
```
Try using an end pointer:

```c
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:

```c
int sum(int *x)
{
    int result = 0;
    int i;
    for (i = 0; i < 1000; i += 2) {
        result += *x++;
        result += (*x++);
    }
    return result;
}
5016 cycles.
```
Try using an end pointer:

```c
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:

```c
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.
Try using an end pointer:

```c
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:

```c
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.
Back to original. Try unrolling:

```c
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.
Back to original. Try unrolling:

```c
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.
```

```c
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;
}
```
Back to original. Try unrolling:

```c
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;
}
4016 cycles. “Are we done yet?”
```
```c
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.
```

```c
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?”
Try unrolling:

```c
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;
}
```

4016 cycles. “Are we done yet?”

“Why is this bad practice? Didn’t we succeed in making code twice as fast?”
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;
}

4016 cycles. “Are we done yet?”

“Why is this bad practice?
Didn’t we succeed in making code twice as fast?”
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?”
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?”

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

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.
int sum(int *x) {
    int result = 0;
    for (int 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?”

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.

Rely on Wikipedia comment that M4F = M4 + floating-point unit.
```
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?”

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.

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

“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.

“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.

“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.

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.
Why is this bad practice?

No, we didn’t succeed making code twice as fast.

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

Good practice:

Find lower bound for cycles spent on arithmetic etc. and gap between lower bound and observed time.


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.
“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:
Find lower bound for cycles spent on arithmetic etc.
Understand gap between lower bound and observed time.

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”.

“integer registers” are named 0 to 15. ARMv7-M has 16 integer registers, including special-purpose “stack pointer” and “program counter”.

First manual says that ADD takes just 1 cycle.
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.

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”.

etc.

e.g., etc.
in etc.

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

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”.

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

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”.

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.

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”.

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 \( i \).
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.

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 \( i \).

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”.

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 $n \text{ LDR} + n \text{ ADD}$: $2n + 1$ cycles, including $n$ cycles of arithmetic.

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

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 $n \text{ LDR} + n \text{ ADD}$: $2n + 1$ cycles, including $n$ cycles of arithmetic.

Why observed time is higher: non-consecutive LDRs; costs of manipulating $i$. 
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 about “pipelining”.

More explanation: if next instruction is also LDR (with address not based on first LDR) 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 $i$.

```c
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];
    }
}
```
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.

Operation: LDR.

But adds "pipelining".

Operation: if next 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 \( i \).

```c
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];
        ...
    }
    return result;
}
```
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 $i$.

```c
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];
    }
}```
$n$ 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 $i$.

```c
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];
    }
}
```
consecutive LDRs
takes only \( n + 1 \) cycles
(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];
        result += x0;
        result += x1;
        result += x2;
        result += x3;
        result += x4;
        result += x5;
        result += x6;
        x0 = 10[(volatile int *)x];
        x1 = 11[(volatile int *)x];
    }
}
```
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:
\[
2n + 1 \text{ cycles, including } n \text{ cycles of arithmetic.}
\]

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

```c
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];
        result += x0;
        result += x1;
        result += x2;
        result += x3;
        result += x4;
        result += x5;
        result += x6;
        result += x7;
        result += x8;
        result += x9;
    }
    return result;
}
```
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.

```c
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];
    }
    return result;
}
```
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];
        result += x0;
        result += x1;
        result += x2;
        result += x3;
        result += x4;
        result += x5;
        x0 = 10[(volatile int *)x];
        x1 = 11[(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];
        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];
    }
}

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;
}
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];
        result += x0;
        result += x1;
        result += x2;
        result += x3;
        result += x4;
        result += x5;
        result += x6;
        result += x7;
        result += x8;
        result += x9;
        x0 = 20;
        x1 = 21;
        x2 = 22;
        x3 = 23;
        x4 = 24;
        x5 = 25;
        x6 = 26;
        x7 = 27;
        x8 = 28;
        x9 = 29;
        result += x0;
        result += x1;
        result += x2;
        result += x3;
        result += x4;
        result += x5;
        result += x6;
        result += x7;
        result += x8;
        result += x9;
        x0 = 30;
        x1 = 31;
        x2 = 32;
        x3 = 33;
        x4 = 34;
        x5 = 35;
        x6 = 36;
        x7 = 37;
        x8 = 38;
        x9 = 39;
        result += x0;
        result += x1;
        result += x2;
        result += x3;
        result += x4;
        result += x5;
        result += x6;
        result += x7;
        result += x8;
        result += x9;
        x0 = 40;
        x1 = 41;
        x2 = 42;
        x3 = 43;
        x4 = 44;
        x5 = 45;
        x6 = 46;
        x7 = 47;
        x8 = 48;
        x9 = 49;
        result += x0;
        result += x1;
        result += x2;
        result += x3;
        result += x4;
        result += x5;
        result += x6;
        result += x7;
        result += x8;
        result += x9;
        x0 = 50;
        x1 = 51;
        x2 = 52;
        x3 = 53;
        x4 = 54;
        x5 = 55;
        x6 = 56;
        x7 = 57;
        x8 = 58;
        x9 = 59;
        result += x0;
        result += x1;
        result += x2;
        result += x3;
        result += x4;
        result += x5;
        result += x6;
        result += x7;
        result += x8;
        result += x9;
        x0 = 60;
        x1 = 61;
        x2 = 62;
        x3 = 63;
        x4 = 64;
        x5 = 65;
        x6 = 66;
        x7 = 67;
        x8 = 68;
        x9 = 69;
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;
        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;
    }
    return result;
}

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;
17
\[
x[7] = 7[(\text{volatile int }*)x];
\]
\[
x[8] = 8[(\text{volatile int }*)x];
\]
\[
x[9] = 9[(\text{volatile int }*)x];
\]
\[
\text{result} += x[0];
\]
\[
\text{result} += x[1];
\]
\[
\text{result} += x[2];
\]
\[
\text{result} += x[3];
\]
\[
\text{result} += x[4];
\]
\[
\text{result} += x[5];
\]
\[
\text{result} += x[6];
\]
\[
\text{result} += x[7];
\]
\[
\text{result} += x[8];
\]
\[
\text{result} += x[9];
\]
\[
x[10] = 10[(\text{volatile int }*)x];
\]
\[
x[11] = 11[(\text{volatile int }*)x];
\]
\[
x[12] = 12[(\text{volatile int }*)x];
\]
\[
x[13] = 13[(\text{volatile int }*)x];
\]
\[
x[14] = 14[(\text{volatile int }*)x];
\]
\[
x[15] = 15[(\text{volatile int }*)x];
\]
\[
x[16] = 16[(\text{volatile int }*)x];
\]
\[
x[17] = 17[(\text{volatile int }*)x];
\]
\[
x[18] = 18[(\text{volatile int }*)x];
\]
\[
x[19] = 19[(\text{volatile int }*)x];
\]
\[
x[20] += 20;
\]
\[
\text{result} += x[0];
\]
\[
\text{result} += x[1];
\]
\[
\text{result} += x[2];
\]
\[
\text{result} += x[3];
\]
\[
\text{result} += x[4];
\]
\[
\text{result} += x[5];
\]
\[
\text{result} += x[6];
\]
\[
\text{result} += x[7];
\]
\[
\text{result} += x[8];
\]
\[
\text{result} += x[9];
\]
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;
}
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;
}
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;
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;
}

2526 cycles. Even better in asm.
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;

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 += x0;
result += x1;
result += x2;
result += x3;
result += x4;
result += x5;
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]
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]
volatile int *)x];
volatile int *)x];
volatile int *)x];
volatile int *)x];
volatile int *)x];
volatile int *)x];
volatile int *)x];
volatile int *)x];

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

Lower bound for arithmetic:
64 bytes require
21 · 16 1-cycle ADDs,
20 · 16 1-cycle XORs,
so at least 10.25 cycles.

Also many rotations, but
ARMv7-M instruction set includes free rotation as part of XOR instruction.
(Compiler knows this.)
\texttt{result += x6;}
\texttt{result += x7;}
\texttt{result += x8;}
\texttt{result += x9;}
\}
\}

\texttt{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]
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 · 16 1-cycle ADDs,
20 · 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.
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.
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.
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.
A real example

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

Lower bound for arithmetic: 64 bytes require
21 · 16 1-cycle ADDs,
20 · 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.)


Gap is mostly loads, stores. Minimize load/store cost by choosing “spills” carefully.
A real example
Salsa20 reference software:
30.25 cycles/byte on this CPU.

Lower bound for arithmetic:
64 bytes require
21 · 16 1-cycle ADDs,
20 · 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.

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.

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.
A real example

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

Lower bound for arithmetic:
64 bytes require
21 · 16 1-cycle ADDs,
20 · 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.

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.
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.

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

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


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

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


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. Make loads consecutive? Don’t trust compiler to optimize instruction scheduling.
Detailed benchmarks show several cycles/byte spent on load\_littleendian and store\_littleendian.

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


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.

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.
Detailed benchmarks show several cycles/byte spent on load_littleendian and store_littleendian.
Can replace with LDR and STR. (Compiler doesn’t see this.)
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.
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.
Detailed benchmarks show several cycles/byte spent on load_littleendian and store_littleendian. Can replace with LDR and STR. (Compiler doesn’t see this.)

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.

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 × slower than fastest Salsa20 implementation.
Detailed benchmarks show several cycles/byte spent on `load_littleendian` and `store_littleendian`. Can replace with LDR and STR. (Compiler doesn’t see this.)


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.

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: Reasonable implementation compiled with gcc -O3 -fomit-frame-pointer is 6.15× slower than fastest Salsa20 implementation.

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 implementation compiled with gcc -O3 -fomit-frame-pointer is 6.15× slower than fastest Salsa20 implementation.
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.
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× slower than fastest Salsa20 implementation.

merged implementation with “machine-independent” optimizations and best of 121 compiler options: 4.52× slower.
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× slower than fastest Salsa20 implementation.

merged implementation with “machine-independent” optimizations and best of 121 compiler options: 4.52× slower.

Fast random permutations
Goal: Put list (x₁; : : : ; xₙ) into a random order.
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.

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.
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× slower than fastest Salsa20 implementation.

merged implementation with “machine-independent” optimizations and best of 121 compiler options: 4.52× slower.

Fast random permutations

Goal: Put list \((x_1, \ldots, x_n)\) into a random order.
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× slower than fastest
Salsa20 implementation.

merged implementation
with “machine-independent”
optimizations and best of 121
compiler options: 4.52× slower.

Fast random permutations

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

One textbook strategy:
Sort $(Mr_1 + x_1, \ldots, Mr_n + x_n)$ for random $(r_1, \ldots, r_n)$, suitable $M$. 
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× slower than fastest Salsa20 implementation.

merged implementation with “machine-independent” optimizations and best of 121 compiler options: 4.52× slower.

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.
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× slower than fastest Salsa20 implementation.

merged implementation with “machine-independent” optimizations and best of 121 compiler options: 4.52× slower.

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.
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.
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.
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.
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.
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.
How many bits in \(r_i\)? Negligible collisions? Occasional collisions?
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.

How many bits in \(r_i\)? Negligible collisions? Occasional collisions?
Restart on collision?
Uniform distribution; some cost.
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.

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
\[\leq 119!(n - 119)!(2^{31} + n - 1)\] ways;
i.e., \(< 1.02 \cdot 2^{31n}/\binom{n}{119}\) ways.
Factor \(< 1.02\) increase in attacker’s chance of winning.
Random permutations

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

One textbook strategy:
Sort \((r_1 + x_1, \ldots, M r_n + x_n)\) for \((r_1, \ldots, r_n)\), 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.

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 \(\leq 119!(n-119)!\left(\frac{2^{31}+n-1}{n}\right)\) 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 \(\min\)max operations.
Fast random permutations

Goal: Put list \((x_1; : : : ; x_n)\) into a random order.

One textbook strategy:
Sort \((M_r + x_1; : : : ; M_r + x_n)\) for random \((r_1; : : : ; r_n)\), suitable \(M_r\).

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

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

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 \(\leq 119!(n - 119)!(2^{31} + n - 1)\) ways;
\(\leq 119!(n - 119)!(2^{31} + n - 1)\) 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.
Fast random permutations

Goal: Put list \((x_1; : : : ; x_n)\) into a random order.

One textbook strategy:
Sort \((M_1 + x_1; : : : ; M_n + x_n)\) for random \((r_1; : : : ; r_n)\), suitable \(M_i\).

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

NTRU encryption example:
Randomly order 761 trits \((-1; : : : ; -1; 0; : : : ; 0)\), wt 286.

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
\[\leq 119!(n - 119)!\left(\frac{2^{31} + n - 1}{n}\right)\] ways;
i.e., \(< 1.02 \cdot 2^{31.57}/(\binom{n}{119})\) ways.

Factor \(< 1.02\) increase in attacker’s chance of winning.

Which sorting algorithm?

Reference bubblesort code does \(n(n - 1)/2\) minmax operations.
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
$\leq 119!(n - 119)!\left(2^{31} + n - 1\right)$ 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.
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

$$\leq 119!(n - 119)!(2^{31} + n - 1) \text{ ways;}$$

i.e., $< 1.02 \cdot 2^{31n}/(\binom{n}{119}) \text{ ways.}$

Factor $< 1.02$ increase in attacker’s chance of winning.

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.
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
\[ \leq 119!(n - 119)! \left( \frac{2^{31} + n - 1}{n} \right) \] 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.
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.
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.

Output is produced in $\binom{n-119}{31+n-1} \cdot 2^{31} + n - 1$ ways; $2^{31} \cdot 2^{31n/119}$ ways. 

$\leq 1.02$ increase in attacker’s chance of winning.

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.
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 $\leq \binom{n}{119} \cdot 2^{31-n}$ ways; i.e., $< 1 : 0.2 \cdot 2^{31-n} = \binom{n}{119}$ ways. Factor $< 1 : 0.2$ increase in attacker's chance of winning.

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.
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 $\leq 119!(n-119)! \approx 2^{31}n$ ways; i.e., $< 1 : 2^{31} \cdot 2^{31-n} = n^{119}$ ways. Factor $< 1 : 2$ increase in attacker's chance of winning.

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.
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.
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.

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

“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 \( n = 2^e \).
Which sorting algorithm?

Reference bubblesort code does \( (n - 1) = 2 \) minmax operations.

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

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.

“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 \( n = 2^e \).

```c
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);
}
}```
Which sorting algorithm?
Reference bubblesort code does \((n - 1) = 2\) 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.

“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 \(n = 2^e\).

```c
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);
    }
}
```
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 \(n = 2^e\).

```c
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);
    }
}
```
Converting bubblesort into constant-time bubblesort loses only a constant factor: cost of constant-time \texttt{minmax}.

“Sorting network”: sorting algorithm built as constant sequence of \texttt{minmax} operations (“comparators”).

Sorting network on next slide: Batcher’s merge-exchange sort.

\begin{equation*}
\Theta(n(\log n)^2) \text{ minmax operations; } \left(\frac{1}{4}\right)(e^2 - e + 4)n - 1 \text{ for } n = 2^e.
\end{equation*}

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

"Sorting network":
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;
$(1^2 - e + 4)n - 1$ for $n = 2^e$.

```c
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.
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").

In next slide: Batcher's merge-exchange sort.

\[ \Theta( n (\log_2 n)^2) \] minmax operations;

\[ (1 = 4)( e^{\frac{e}{2}} - e + 4) n - 1 \] for \( n = 2^e \).

```c
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.
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.

\[ \Theta(n \log^2 n) \] minmax operations;
\[ (1 = 4)(e^2 - e + 4)n - 1 \] for \( n = 2^e \).

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.

```c
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);
  }
}
```
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.
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);
    }
}
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);
    }
}
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.

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

≥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?
```c
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.

\( \geq 3008 \) 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.
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.

≥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?
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.

≥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?
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.

≥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?

People optimize algorithms for a naive model of CPUs:

- Branches are fast.
- Random access is fast.
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.

≥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?

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.
How many cycles on, e.g., Intel Haswell CPU core?
Every cycle: a vector of 8 32-bit “min” operations and a vector of “max” operations.
≥ 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?
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 \( x^{761} - x - 1 \).)
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.
≥ 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?
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 \( x^{761} - x - 1 \).)
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 $x^{761} - x - 1$.)
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 $x^{761} - x - 1$.)
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 $x^{761} - x - 1$.)

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 $f_{\ell-1} \neq 0$. 
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 far 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 $x^{761} - x - 1$.)

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) $f, g \mapsto f \mod g$; etc.
30
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.

31
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) $f, g \mapsto f g$; (2) $f, g \mapsto f \mod g$; etc.
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 $x^{761} - x - 1$.)

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) $f, g \mapsto f \mod g$; etc.
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 \texttt{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) $f, g \mapsto f \mod g$; etc.
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) $f, g \mapsto f \mod g$; etc.

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

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

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

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

“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) $f, g \mapsto f \mod g$; 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!

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 $\ell$-limb integers: always allocate $\ell + 1$ limbs. Don’t remove top zero limb.
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; : : : ; f_{\ell} − 1)
represents
the nonnegative integer
f_0 + 2^{32}f_1 + \cdots + 2^{32(\ell − 1)}f_{\ell − 1}.

Uniqueness: \(' = 0 or f_{\ell − 1} \neq 0.

Library provides functions acting
on this representation: (1) \( f, g \mapsto fg \); (2) \( f, g \mapsto f \mod g \); 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!

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; : : : ; f_{\ell − 1}) represents
the nonnegative integer
f_0 + 2^{32}f_1 + \cdots + 2^{32(\ell − 1)}f_{\ell − 1}.

Adding two \( \ell \)-limb integers:
always allocate \( \ell + 1 \) limbs.
Don’t remove top zero limb.
Library provides functions acting on this representation: (1) \(f, g \mapsto fg\); (2) \(f, g \mapsto f \mod g\); 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!

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 \(\ell\)-limb integers: always allocate \(\ell + 1\) limbs. Don't remove top zero limb.
Library provides functions acting on this representation: (1) \( f, g \mapsto fg \); (2) \( f, g \mapsto f \mod g \); 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!

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 \( \ell \)-limb integers: always allocate \( \ell + 1 \) limbs. Don’t remove top zero limb.
Library provides functions acting on this representation: (1) \( f, g \mapsto f \cdot g \); (2) \( f, g \mapsto f \mod g \); 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!

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 \( \ell \)-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 \( \mapsto \) bounds data flow.
Library provides functions acting on this representation: (1) \( f, g \mapsto fg \); (2) \( f, g \mapsto f \mod g \); 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!

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 \( \ell \)-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.

\( f \mod p \) is as short as \( p \).
Library provides functions acting on this representation: (1) $f, g \mapsto f \cdot g$; (2) $f, g \mapsto f \mod g$; 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!

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 $\ell$-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.

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

Usually faster representation: uint32 string $(f_0; f_1; \ldots; f_9)$ represents $2^{77}f_3 + 2^{179}f_7 + 2^{204}f_8 + 2^{230}f_9$.

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 $19$.

After multiplication, replace $2^{255}$ with $19$. 
Library provides functions acting on this representation: (1) \( f, g \mapsto f g \); (2) \( f, g \mapsto f \mod g \); etc.

ECC implementor using library:
- multiply \( f, g \mod 2^{255} - 19 \) by (1) multiplying \( f \) by \( g \);
- 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 \( \text{uint32} \) 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 \( \ell \)-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 \( \mapsto \) bounds data flow.

\( f \mod p \) is as short as \( p \).

Usually faster representation:
\( \text{uint32} \) string \((f_0, f_1, \ldots, f_9)\) represents \( f_0 + 2^{26} f_1 + 2^{51} f_2 + 2^{77} f_3 + 2^{102} f_4 + 2^{128} f_5 + 2^{179} f_6 + 2^{204} f_7 + 2^{230} f_8 + 2^{255} f_9 \).

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

Adding two $\ell$-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.

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

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

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.
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 \( \ell \)-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.

\( f \mod p \) 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.
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 \(\ell\)-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.

\(f\) mod \(p\) 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.

Slightly faster on some CPUs: int32 string \((f_0, f_1, \ldots, f_9)\).
Constant-time bigint library:

- A constant-length uint32 string 
  \( f_0, f_1, \ldots, f_{\ell-1} \) represents
  a nonnegative integer

\[
  f_0 + 2^{32} f_1 + 2^{64} f_2 + 2^{96} f_3 + 2^{102} f_4 + 2^{128} f_5 + 2^{153} f_6 + 2^{179} f_7 + 2^{204} f_8 + 2^{230} f_9.
\]

- Adding two \( \ell \)-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.

- \( f \text{ mod } p \) is as short as \( p \).

Usually faster representation:

- A 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:

- An int32 string \( (f_0, f_1, \ldots, f_9) \).

\[\text{Example code snippet:}\]

\[
\text{int32 } f_7_2 = 2 \times f_7;
\]

\[
\text{int32 } g_7_19 = 19 \times g_7;
\]

\[
\text{...}
\]

\[
\text{int64 } f_0 g_4 = f_0 \times (\text{int64} g_4);
\]

\[
\text{int64 } f_7 g_7_38 = f_7_2 \times (\text{int64} g_7_19);
\]

\[
\text{...}
\]

\[
\text{int64 } h_4 = f_0 g_4 + f_1 g_3_2 + f_2 g_2 + f_3 g_1_2 + f_4 g_0 + f_5 g_9_38 + f_6 g_8_19 + f_7 g_7_38 + f_8 g_6_19 + f_9 g_5_38;
\]

\[
\text{...}
\]

\[
c_4 = (h_4 + (\text{int64}(1 << 25))) \gg 26;
\]

\[
h_5 += c_4; h_4 -= c_4 \ll 26;
\]

\[
\text{...}
\]

\[
\text{...}
\]
Constant-time bigint library:
A constant-length uint32 string
\( (f_0; f_1; \ldots; f_{\ell-1}) \) represents
the nonnegative integer
\[ f_0 + 2^{32(f_1-1)} f_{\ell-1}. \]
Adding two \( \ell\)-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 → bounds data flow.
\( f \mod p \) 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.
Slightly faster on some CPUs:
int32 string \( (f_0, f_1, \ldots, f_9) \).

\[
\begin{align*}
\text{int32 } f7_2 &= 2 \times f7; \\
\text{int32 } g7_19 &= 19 \times g7; \\
\ldots \\
\text{int64 } f0g4 &= f0 \times (\text{int64}) g4; \\
\text{int64 } f7g7_38 &= f7_2 \times (\text{int64}) g7_19; \\
\ldots \\
\text{int64 } h4 &= f0g4 + f2g2 + f4g0 + f6g8 + f8g6; \\
\text{c4} &= (h4 + (\text{int64})(1<<25)) \gg 26; \\
\text{h5} &= h5 + c4; h4 &= h4 - c4 \ll 26;
\end{align*}
\]
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 \((f_0, f_1, \ldots, f_9)\).
Usually faster representation:

\[
\text{uint32 string } (f_0, f_1, \ldots, f_9) \text{ represents } f_0 + 2^{26} f_1 + 2^{51} f_2 + \ldots + 2^{204} f_8 + 2^{230} f_9.
\]

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:

\[
\text{int32 string } (f_0, f_1, \ldots, f_9).
\]

Initial computation of \(h_0, \ldots, h_9\) is polynomial multiplication modulo \(x^{10} - 19\).

Exercise: Which polynomials are being multiplied?

```c
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;
```
Representation:
\[ f_0, f_1, \ldots, f_9 \]
\[ 2^{26} f_1 + 2^{51} f_2 + 2^{128} f_5 + 2^{153} f_6 + 2^{230} f_9. \]

In each \( f_i \).

Before, avoiding delaying carries.

In,

19.

Some CPUs:
\[ f_1, \ldots, f_9. \]

\begin{verbatim}
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;
\end{verbatim}

Exercise: Which polynomials are being multiplied?
Usually faster representation:
\[
\text{uint32 string ( } f_0; f_1; \ldots; f_9 \text{ )}
\]
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:
\[
\text{int32 string ( } f_0; f_1; \ldots; f_9 \text{ ).}
\]

Initial computation of \( h_0, \ldots, h_9 \) is polynomial multiplication modulo \( x^{10} - 19 \).
Exercise: Which polynomials are being multiplied?

```c
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;
...
int64 c4 = (h4 + (int64)(1<<25)) >> 26;
int64 h5 = h5 + c4; h4 = h4 - c4 << 26;
```
Initial computation of $h_0, \ldots, h_9$ is polynomial multiplication modulo $x^{10} - 19$.
Exercise: Which polynomials are being multiplied?
Initial computation of $h_0, \ldots, h_9$ is polynomial multiplication modulo $x^{10} - 19$.

Exercise: Which polynomials are being multiplied?

Reduction modulo $x^{10} - 19$ and carries such as $h_4 \rightarrow h_5$ squeeze the product into limited-size representation suitable for next multiplication.
Initial computation of $h_0, \ldots, h_9$ is polynomial multiplication modulo $x^{10} - 19$.

Exercise: Which polynomials are being multiplied?

Reduction modulo $x^{10} - 19$ and carries such as $h_4 \rightarrow h_5$ **squeeze** the product into limited-size representation suitable for next multiplication.

At end of computation: **freeze** representation into unique representation suitable for network transmission.
Initial computation of $h_0$, $\ldots$, $h_9$ is polynomial multiplication modulo $x^{10} - 19$.

Exercise: Which polynomials are being multiplied?

Reduction modulo $x^{10} - 19$ and carries such as $h_4 \rightarrow h_5$ 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.
Initial computation of $h_0, \ldots, h_9$ is polynomial multiplication modulo $x^{10} - 19$.

Exercise: Which polynomials are being multiplied?

Reduction modulo $x^{10} - 19$ and carries such as $h_4 \rightarrow h_5$ 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.
Initial computation of \( h_0, \ldots, h_9 \) is polynomial multiplication modulo \( x^{10} - 19 \).

Exercise: Which polynomials are being multiplied?

Reduction modulo \( x^{10} - 19 \) and carries such as \( h_4 \rightarrow h_5 \) 

**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.
Initial computation of \( h_0, \ldots, h_9 \) is polynomial multiplication modulo \( x^{10} - 19 \).

Exercise: Which polynomials are being multiplied?

Reduction modulo \( x^{10} - 19 \) and carries such as \( h_4 \rightarrow h_5 \) 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.
Initial computation of $h_0, \ldots, h_9$ is polynomial multiplication modulo $x^{10} - 19$.
Exercise: Which polynomials are being multiplied?

Reduction modulo $x^{10} - 19$ and carries such as $h_4 \rightarrow h_5$
**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.

Verifying constant time: increasingly automated.
Initial computation of \( h_0, \ldots, h_9 \) is polynomial multiplication modulo \( x^{10} - 19 \).

Exercise: Which polynomials are being multiplied?

Reduction modulo \( x^{10} - 19 \) and carries such as \( h_4 \rightarrow h_5 \) **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*.

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.
Initial computation of $h_0, \ldots, h_9$ is polynomial multiplication modulo $x^{10} - 19$.

Exercise: Which polynomials are being multiplied?

Reduction modulo $x^{10} - 19$ and carries such as $h_4 \rightarrow h_5$ 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.

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.
Initial computation of $h_0, \ldots, h_9$ is polynomial multiplication modulo $x^{10} - 19$.

Exercise: Which polynomials are being multiplied?

Reduction modulo $x^{10} - 19$ and carries such as $h_4 \rightarrow h_5$ 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.

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:

$$p = 2^{255} - 19$$

$$A = 486662$$

$$x_2, z_2, x_3, z_3 = 1, 0, x_1, 1$$

for i in reversed(range(255)):
    ni = bit(n,i)
    x_2, x_3 = cswap(x_2, x_3, ni)
    z_2, z_3 = cswap(z_2, z_3, ni)
    x_3, z_3 = (4*(x_2*x_3-z_2*z_3)**2, 4*x_1*(x_2*z_3-z_2*x_3)**2)
    x_2, z_2 = ((x_2**2-z_2**2)**2, 4*x_2*z_2*(x_2**2+A*x_2*z_2+z_2**2))
Initial computation of $h_0, \ldots, h_9$ is polynomial multiplication modulo $x^{10} - 19$.

Exercise: Which polynomials are being multiplied?

Reduction modulo $x^{10} - 19$ and carries such as $h_4 \rightarrow h_5$ 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.

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 implementation of X25519, plus occasional annotations, against the following specification:

- $p = 2^{255} - 19$
- $A = 486662$
- $x^2, z^2, x^3, z^3 = 1, 0, x^1, 1$

for $i$ in reversed(range(255)):

```python
ni = bit(n, i)
x_2, x_3 = cswap(x_2, x_3, ni)
z_2, z_3 = cswap(z_2, z_3, ni)
x_3, z_3 = (4*(x_2*x_3 - z_2*z_3)**2, 4*x_1*(x_2*z_3 - z_2*x_3)**2)
x_2, z_2 = ((x_2**2 - z_2**2)**2, 4*x_2*z_2*(x_2**2 + A*x_2*z_2 + z_2**2))
```
Initial computation of \( h_0, \ldots, h_9 \) is polynomial multiplication modulo \( x^{10} - 19 \).

Exercise: Which polynomials are being multiplied?

Reduction modulo \( x^{10} - 19 \) and carries such as \( h_4 \rightarrow h_5 \) 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.

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:

\[
p = 2^{255}-19
\]

\[
A = 486662
\]

\[
x_2, z_2, x_3, z_3 = 1, 0, x_1, 1
\]

for \( i \) in reversed(range(255)):

\[
\begin{align*}
ni &= \text{bit}(n, i) \\
x_2, x_3 &= \text{cswap}(x_2, x_3, ni) \\
z_2, z_3 &= \text{cswap}(z_2, z_3, ni) \\
x_3, z_3 &= (4(x_2x_3-z_2z_3)^2, 4x_1(x_2z_3-z_2x_3)^2) \\
x_2, z_2 &= ((x_2^2-z_2^2)^2, 4x_2z_2(x_2^2+A*x_2z_2+z_2^2))
\end{align*}
\]
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:

\[
p = 2^{255} - 19 \\
A = 486662 \\
x_2, z_2, x_3, z_3 = 1, 0, x_1, 1
\]

for \( i \in \text{reversed(range(255))} \):

\[
i = \text{bit}(n, i) \\
x_2, x_3 = \text{cswap}(x_2, x_3, ni) \\
z_2, z_3 = \text{cswap}(z_2, z_3, ni) \\
x_3, z_3 = (4*(x_2*x_3-z_2*z_3)**2, \\
4*x_1*(x_2*z_3-z_2*x_3)**2) \\
x_2, z_2 = ((x_2**2-z_2**2)**2, \\
4*x_2*z_2*(x_2**2+A*x_2*z_2+z_2**2))
\]
Much more about ECC speed: see, e.g., 2015 Chou.

Verifying constant time: increasingly automated.

Testing can miss rare bugs 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:

\[ p = 2^{255}-19 \]
\[ A = 486662 \]
\[ x_2, z_2, x_3, z_3 = 1, 0, x_1, 1 \]

for \( i \) in reversed(range(255)):

\[ n_i = \text{bit}(n, i) \]
\[ x_2, x_3 = \text{cswap}(x_2, x_3, n_i) \]
\[ z_2, z_3 = \text{cswap}(z_2, z_3, n_i) \]
\[ x_3, z_3 = (4*(x_2*x_3-z_2*z_3)**2, 4*x_1*(x_2*z_3-z_2*x_3)**2) \]
\[ x_2, z_2 = ((x_2**2-z_2**2)**2, 4*x_2*z_2*(x_2**2+A*x_2*z_2+z_2**2)) \]
\[ x_3, z_3 = (x_3%p, z_3%p) \]
\[ x_2, z_2 = (x_2%p, z_2%p) \]
\[ \text{cut}(x_2) \]
\[ \text{cut}(x_3) \]
\[ \text{cut}(z_2) \]
\[ \text{cut}(z_3) \]
\[ x_2, x_3 = \text{cswap}(x_2, x_3, n_i) \]
\[ z_2, z_3 = \text{cswap}(z_2, z_3, n_i) \]
\[ \text{cut}(x_2) \]
\[ \text{cut}(z_2) \]
return \( x_2^{\text{pow}(z_2, p-2, p)} \)

What’s verified: output of ref10 is the same as spec mod \( p \), and is between 0 and \( p - 1 \).
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:

\[
p = 2^{255}-19
\]

\[
A = 486662
\]

\[
x_2, z_2, x_3, z_3 = 1, 0, x_1, 1
\]

for \( i \) in reversed(range(255)):

\[
\text{ni} = \text{bit}(n, i)
\]

\[
x_2, x_3 = \text{cswap}(x_2, x_3, \text{ni})
\]

\[
z_2, z_3 = \text{cswap}(z_2, z_3, \text{ni})
\]

\[
x_3, z_3 = (4*(x_2*x_3-z_2*z_3)^2, 4*x_1*(x_2*z_3-z_2*x_3)^2)
\]

\[
x_2, z_2 = ((x_2**2-z_2**2)^2, 4*x_2*z_2*(x_2**2+A*x_2*z_2+z_2**2))
\]

x_3, z_3 = (x_3%p, z_3%p)

x_2, z_2 = (x_2%p, z_2%p)

\[
\text{cut}(x_2)
\]

\[
\text{cut}(z_2)
\]

\[
\text{cut}(x_3)
\]

\[
\text{cut}(z_3)
\]

\[
x_2, x_3 = \text{cswap}(x_2, x_3, \text{ni})
\]

\[
z_2, z_3 = \text{cswap}(z_2, z_3, \text{ni})
\]

\[
\text{cut}(x_2)
\]

\[
\text{cut}(z_2)
\]

\[
\text{return } x_2*\text{pow}(z_2, p-2, p)
\]

What’s verified: output of ref10 is the same as spec mod \( p \),
and is between 0 and \( p-1 \).
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:

\[
\begin{align*}
p &= 2^{255} - 19 \\
A &= 486662 \\
x_2, z_2, x_3, z_3 &= 1, 0, x_1, 1 \\
\text{for } i \text{ in reversed(range(255))}:
\quad \text{ni} &= \text{bit}(n, i) \\
\quad x_2, x_3 &= \text{cswap}(x_2, x_3, \text{ni}) \\
\quad z_2, z_3 &= \text{cswap}(z_2, z_3, \text{ni}) \\
\quad x_3, z_3 &= (4(x_2x_3-z_2z_3)^2, 4x_1(x_2z_3-z_2x_3)^2) \\
\quad x_2, z_2 &= ((x_2^2-z_2^2)^2, 4x_2z_2(x_2^2+A*x_2z_2+z_2^2)) \\
\quad x_2, z_2 &= (x_2\%p, z_2\%p) \\
\quad x_3, z_3 &= (x_3\%p, z_3\%p) \\
\quad \text{cut}(x_2) &\text{ cut}(x_3) \\
\quad \text{cut}(z_2) &\text{ cut}(z_3) \\
\quad \text{cut}(x_2) &\text{ return } x_2\text{pow}(z_2, p-2, p)
\end{align*}
\]

What’s verified: output of ref10
is the same as spec mod \( p \),
and is between 0 and \( p - 1 \).
gfverif has verified ref10 implementation of X25519, plus occasional annotations, against the following specification:

\[
p = 2^{255}-19
\]

\[
A = 486662
\]

\[
x_2, z_2, x_3, z_3 = 1,0,x_1,1
\]

for \( i \) in reversed(range(255)):

\[
i = \text{bit}(n, i)
\]

\[
x_2, x_3 = \text{cswap}(x_2, x_3, i)
\]

\[
z_2, z_3 = \text{cswap}(z_2, z_3, i)
\]

\[
x_3, z_3 = (4*(x_2*x_3-z_2*z_3)**2, 4*x_1*(x_2*z_3-z_2*x_3)**2)
\]

\[
x_2, z_2 = ((x_2**2-z_2**2)**2, 4*x_2*z_2*(x_2**2+A*x_2*z_2+z_2**2))
\]

\[
(x_3\mod p, z_3\mod p)
\]

\[
(x_2\mod p, z_2\mod p)
\]

\[
\text{cut}(x_2)
\]

\[
\text{cut}(x_3)
\]

\[
\text{cut}(z_2)
\]

\[
\text{cut}(z_3)
\]

\[
x_2, x_3 = \text{cswap}(x_2, x_3, i)
\]

\[
z_2, z_3 = \text{cswap}(z_2, z_3, i)
\]

\[
\text{cut}(x_2)
\]

\[
\text{cut}(z_2)
\]

return \( x_2*\text{pow}(z_2, p-2, p) \)

What's verified: output of ref10 is the same as spec mod \( p \), and is between 0 and \( p - 1 \).
gfverif has verified ref10 implementation of X25519, plus occasional annotations, against the following specification:

\[ p = 2^{255} - 19 \]
\[ A = 486662 \]
\[ x_2, z_2, x_3, z_3 = 1, 0, x_1, 1 \]

for \( i \) in reversed(range(255)):

\[ n_i = \text{bit}(n, i) \]
\[ x_2, x_3 = \text{cswap}(x_2, x_3, n_i) \]
\[ z_2, z_3 = \text{cswap}(z_2, z_3, n_i) \]
\[ x_3, z_3 = \text{mod}(x_3, p) \]
\[ x_2, z_2 = \text{mod}(x_2, p) \]
\[ \text{cut}(x_2) \]
\[ \text{cut}(x_3) \]
\[ \text{cut}(z_2) \]
\[ \text{cut}(z_3) \]

What’s verified: output of ref10 is the same as spec mod \( p \), and is between 0 and \( p - 1 \).

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

\[ P_i A_i 2^{32 i} \]

Define \( T; S_1; S_2; S_3; S_4; D_1; D_2; D_3; D_4 \) as
gfverif has verified implementation of X25519, plus occasional annotations, against the following specification:

\[ p = 2^{255} - 19 \]
\[ A = 486662 \]
\[ (x_2, z_2, x_3, z_3) = (1, 0, x_1, 1) \]
\[ \text{for } i \text{ in reversed(range(255))}: \]
\[ n_i = \text{bit}(n, i) \]
\[ (x_2, x_3) = \text{cswap}(x_2, x_3, n_i) \]
\[ (z_2, z_3) = \text{cswap}(z_2, z_3, n_i) \]
\[ (x_3, z_3) = (4(x_2 x_3 - z_2 z_3)^2, 4x_1(x_2 z_3 - z_2 x_3)^2) \]
\[ (x_2, z_2) = ((x_2^2 - z_2^2)^2, 4x_2 z_2(x_2^2 + A x_2 z_2 + z_2^2)) \]
\[ x_3, z_3 = (x_3 \% p, z_3 \% p) \]
\[ x_2, z_2 = (x_2 \% p, z_2 \% p) \]
\[ \text{cut}(x_2) \]
\[ \text{cut}(x_3) \]
\[ \text{cut}(z_2) \]
\[ \text{cut}(z_3) \]
\[ x_2, x_3 = \text{cswap}(x_2, x_3, n_i) \]
\[ z_2, z_3 = \text{cswap}(z_2, z_3, n_i) \]
\[ \text{cut}(x_2) \]
\[ \text{cut}(z_2) \]

return \( x_2 \cdot \text{pow}(z_2, p-2, p) \)

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 \( p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 \).

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

\[ p = 2^{255} - 19 \]
\[ A = 486662 \]
\[ x_2, z_2, x_3, z_3 = 1,0, x_1, 1 \]

for \( i \) in reversed(range(255)):

\[ n_i = \text{bit}(n, i) \]
\[ x_2, x_3 = \text{cswap}(x_2, x_3, n_i) \]
\[ z_2, z_3 = \text{cswap}(z_2, z_3, n_i) \]
\[ x_3, z_3 = (4(x_2 \cdot x_3 - z_2 \cdot z_3)^2, 4x_1(x_2 \cdot z_3 - z_2 \cdot x_3)^2) \]
\[ x_2, z_2 = ((x_2^2 - z_2^2)^2, 4x_2 \cdot z_2(x_2^2 + A \cdot x_2 \cdot z_2 + z_2^2)) \]

\[ x_3, z_3 = (x_3 \mod p, z_3 \mod p) \]
\[ x_2, z_2 = (x_2 \mod p, z_2 \mod p) \]
\[ \text{cut}(x_2) \]
\[ \text{cut}(x_3) \]
\[ \text{cut}(z_2) \]
\[ \text{cut}(z_3) \]
\[ x_2, x_3 = \text{cswap}(x_2, x_3, n_i) \]
\[ z_2, z_3 = \text{cswap}(z_2, z_3, n_i) \]
\[ \text{cut}(x_2) \]
\[ \text{cut}(z_2) \]
\[ \text{return } x_2 \cdot \text{pow}(z_2, p-2, p) \]

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 \( T; S_1; S_2; S_3; S_4; D_1; D_2; D_3 \) as
\[ x_3, z_3 = (x_3 \% p, z_3 \% p) \]
\[ x_2, z_2 = (x_2 \% p, z_2 \% p) \]
\[ \text{cut}(x_2) \]
\[ \text{cut}(x_3) \]
\[ \text{cut}(z_2) \]
\[ \text{cut}(z_3) \]
\[ x_2, x_3 = \text{cswap}(x_2, x_3, ni) \]
\[ z_2, z_3 = \text{cswap}(z_2, z_3, ni) \]
\[ \text{cut}(x_2) \]
\[ \text{cut}(z_2) \]
\[ \text{return } x_2 \text{pow}(z_2, p-2, p) \]

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
“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

$$x2*\text{pow}(z2,p-2,p)$$

verified: output of ref10 same as spec mod $p$, between 0 and $p - 1$. 

Reduce modulo $p$ “by adding or subtracting a few copies” of $p$. 

$$(A_7, A_6, A_5; A_4, A_3, A_2, A_1, A_0);$$

$$(A_{15}, A_{14}, A_{13}; A_{12}, A_{11}, A_{10}; A_9);$$

$$(A_{15}, A_{14}, 0; A_{13}, A_{12}, A_{11}; A_{10});$$

$$(A_{10}, A_8, A_7; A_6, A_5, A_4; A_3);$$

$$(A_{11}, A_9, 0; A_8, A_7, A_6; A_5);$$

$$(A_{12}, 0, A_7; A_6, A_5, A_4; A_3);$$

$$(A_{13}, 0, 0; A_8, A_7, A_6; A_5).$$

Compute

$$T + 2S_1 + 2S_2 + S_3 + S_4 - D_1 - D_2 - D_3 - D_4.$$
“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

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

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

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

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

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

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

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

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

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

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

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 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 

\( \text{Compute } T + 2S_1 + 2S_2 + S_3 + S_4 - D_1 - D_2 - D_3 - D_4. \)

Reduce modulo \( p \) “by adding or subtracting a few copies” of \( p \).
“What 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 a difference a prime makes

NIST P-256 prime $p$ is

$$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1.$$  

ECDSA standard specifies

a reduction procedure given

an integer “$A$ less than $p^2$”:

Write $A$ as

$$(A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0),$$

meaning

$$P_i A_i 2^{32i}.$$  

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 a difference a prime makes”

The NIST P-256 prime $p$ is

\[ p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1. \]

ECDSA standard specifies reduction procedure given an integer "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 $P_i A_i 2^{32i}$.

Define $T; S_1; S_2; S_3; S_4; D_1; D_2; D_3; D_4$ as

\[
\begin{align*}
(A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0); \\
(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, 0, 0, 0); \\
(0, A_{15}, A_{14}, A_{13}, A_{12}, 0, 0, 0); \\
(A_{15}, A_{14}, 0, 0, 0, A_{10}, A_9, A_8); \\
(A_8, A_{13}, A_{15}, A_{14}, A_{13}, A_{11}, A_{10}, A_9); \\
(A_{10}, A_8, 0, 0, A_{13}, A_{12}, A_{11}); \\
(A_{11}, A_9, 0, 0, A_{15}, A_{14}, A_{13}, A_{12}); \\
(A_{12}, 0, A_{10}, A_9, A_8, A_{15}, A_{14}, A_{13}); \\
(A_{13}, 0, A_{11}, A_{10}, A_9, 0, A_{15}, A_{14}).
\end{align*}
\]

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 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 "less than $p^2$": Write $A$ as $(A_{15}; A_{14}; A_{13}; A_{12}; A_{11}; 0; 0; 0)$, meaning $P_i A_i 2^{32i}$.

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.
(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 \( p \).
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.

Correct but quite slow: conditionally add $4p$, conditionally add $2p$, conditionally add $p$, conditionally sub $4p$, conditionally sub $2p$, conditionally sub $p$. 
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.

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 $p^2$”. 
(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.

Correct but quite slow:
conditionally add 4\( p \),
conditionally add 2\( p \),
conditionally add \( p \),
conditionally sub 4\( p \),
conditionally sub 2\( p \),
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?
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.

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.

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.
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; A_13; A_12; A_11; 0; 0; 0).  

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.

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.
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.
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.

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