Cryptographic software engineering, part 2

Daniel J. Bernstein

Last time:
• General software engineering.
• Using const-time instructions.
• Comparing time to lower bound.

Example: Adding 1000 integers on Cortex-M4F. Lower bound: $2n + 1$ cycles for $n \text{ LDR} + n \text{ ADD}$.

Imagine not knowing this . . .

Reference implementation:

```c
int sum(int *x)
{
    int result = 0;
    int i;
    for (i = 0; i < 1000; ++i)
        result += x[i];
    return result;
}
```
Cryptographic software engineering, part 2

Daniel J. Bernstein

Last time:
• General software engineering.
• Using const-time instructions.
• Comparing time to lower bound.

Example: Adding 1000 integers on Cortex-M4F. Lower bound: 
2n + 1 cycles for n LDR + n ADD. Imagine not knowing this . . .

Reference implementation:

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

Try -Os: 8012 cycles.
Cryptographic software engineering, part 2

Daniel J. Bernstein

Last time:
- General software engineering.
- Using const-time instructions.
- Comparing time to lower bound.

Example: Adding 1000 integers on Cortex-M4F. Lower bound: $2n + 1$ cycles for $n$ LDR + $n$ ADD. Imagine not knowing this . . .

Reference implementation:

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

Try -Os: 8012 cycles.
Try -O1: 8012 cycles.
Cryptographic software engineering, part 2

Daniel J. Bernstein

Last time:
- General software engineering.
- Using const-time instructions.
- Comparing time to lower bound.

Example: Adding 1000 integers on Cortex-M4F. Lower bound: \(2n + 1\) cycles for \(n\ LDR + n\ ADD\).

Imagine not knowing this...

Reference implementation:

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

Try -Os: 8012 cycles.
Try -O1: 8012 cycles.
Try -O2: 8012 cycles.
Cryptographic software engineering, part 2

Daniel J. Bernstein

Last time:
• General software engineering.
• Using const-time instructions.
• Comparing time to lower bound.

Example: Adding 1000 integers on Cortex-M4F. Lower bound: \(2n + 1\) cycles for \(n\) LDR + \(n\) ADD. Imagine not knowing this . . .

Reference implementation:

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

Try -0s: 8012 cycles.
Try -01: 8012 cycles.
Try -02: 8012 cycles.
Try -03: 8012 cycles.
Cryptographic software engineering, part 2

Daniel J. Bernstein

Last time:
• General software engineering.
• Using const-time instructions.
• Comparing time to lower bound.

Example: Adding 1000 integers on Cortex-M4F. Lower bound:
\[2^n + 1\] cycles for \(n\) LDR + \(n\) ADD.

Imagine not knowing this . . .

Reference implementation:

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

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;
}
```
Reference implementation:

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

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;
}
```
Reference implementation:

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

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;
}
```
Reference implementation:

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

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;
}
```
Reference implementation:

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

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.
Reference implementation:

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

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

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;
}
```

8010 cycles.
Try moving the pointer:

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

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++;
    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[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.
```
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 += 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.

```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.

```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.

```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?”
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;
}
```

4016 cycles. “Are we done yet?”

No. Use the lower bound ...
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?”

No. Use the lower bound ...

int sum(int *x)
{
    int result = 0;
    int i;
    int *y = x + 1000;
    int x0, x1, x2, x3, x4,
        x5, x6, x7, x8, x9;
    while (x != y) {
        x0 = *(volatile int *)x;
        x1 = *(volatile int *)x;
        x2 = *(volatile int *)x;
        x3 = *(volatile int *)x;
        x4 = *(volatile int *)x;
        x5 = *(volatile int *)x;
        x6 = *(volatile int *)x;
        result += x[i];
        result += x[i + 1];
        result += x[i + 2];
        result += x[i + 3];
        result += x[i + 4];
    }
    return result;
}
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?”
No. Use the lower bound . . .

```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?”
No. Use the lower bound . . .

```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];
    }
    return result;
}
```
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?"
No. Use the lower bound ...
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?"
No. Use the lower bound ...
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?”

No. Use the lower bound ...

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];
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?"
No. Use the lower bound ...
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];
    }
    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];
(int *x)

result = 0;
y = x + 1000;
0,1,2,3,4,
5,6,7,8,9;

if (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;
x0 = 20[(volatile int *)x];
x1 = 21[(volatile int *)x];
x2 = 22[(volatile int *)x];
x3 = 23[(volatile int *)x];
x4 = 24[(volatile int *)x];
x5 = 25[(volatile int *)x];
x6 = 26[(volatile int *)x];
x7 = 27[(volatile int *)x];
x8 = 28[(volatile int *)x];
x9 = 29[(volatile int *)x];
x += 30;
result += x0;
result += x1;
result += x2;
result += x3;
result += x4;
result += x5;
result += x6;
result += x7;
result += x8;
result += x9;
}
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 = *(volatile int *)x;
        x1 = *(volatile int *)x;
        x2 = *(volatile int *)x;
        x3 = *(volatile int *)x;
        x4 = *(volatile int *)x;
        x5 = *(volatile int *)x;
        x6 = *(volatile int *)x;
        x7 = *(volatile int *)x;
        x8 = *(volatile int *)x;
        x9 = *(volatile int *)x;
        result += x0;
        result += x1;
        result += x2;
        result += x3;
        result += x4;
        result += x5;
        result += x6;
        result += x7;
        result += x8;
        result += x9;
    }
    x0 = *(volatile int *)x;
    x1 = *(volatile int *)x;
    x2 = *(volatile int *)x;
    x3 = *(volatile int *)x;
    x4 = *(volatile int *)x;
    x5 = *(volatile int *)x;
    x6 = *(volatile int *)x;
    x7 = *(volatile int *)x;
    x8 = *(volatile int *)x;
    x9 = *(volatile int *)x;
    result += x0;
    result += x1;
    result += x2;
    result += x3;
    result += x4;
    result += x5;
    x += 20;
}

```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];
        result += x0;
        x1 = 1[(volatile int *)x];
        result += x1;
        x2 = 2[(volatile int *)x];
        result += x2;
        x3 = 3[(volatile int *)x];
        result += x3;
        x4 = 4[(volatile int *)x];
        result += x4;
        x5 = 5[(volatile int *)x];
        result += x5;
        x6 = 6[(volatile int *)x];
        result += x6;
        x7 = 7[(volatile int *)x];
        result += x7;
        x8 = 8[(volatile int *)x];
        result += x8;
        x9 = 9[(volatile int *)x];
        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;
    x += 20;
    result += x0;
    result += x1;
    result += x2;
    result += x3;
    result += x4;
    result += x5;
    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;
`x7 = 7[(volatile int *)x];
x8 = 8[(volatile int *)x];
x9 = 9[(volatile int *)x];`

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

```c
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;
```

```c
return result;
```
tile int *(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;  
x0 = 10[(volatile int *)x];  
x1 = 11[(volatile int *)x];  
result += x6;  
result += x7;  
result += x8;  
result += x9;  
}
return result;  
}
9
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];

10
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 += x2;
result += x3;
result += x4;
result += x5;
x += 20;
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]
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]

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.)
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.)
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 \cdot 16 1-cycle ADDs,
20 \cdot 16 1-cycle XORs,
so at least 10\frac{1}{4} 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.
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.

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.
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 10²5 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
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 \text{ 1-cycle ADDs},

20 \cdot 16 \text{ 1-cycle XORs},

so at least 10

\[ 25 \text{ cycles/byte}. \]

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.)

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.

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.

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 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× 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.
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.
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.
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.
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.

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.
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 \(\leq 119!(n − 119)!\left(2^{31n−1}\right)\) ways; i.e., \(< 1.02 \cdot 2^{31n}/\binom{n}{119}\) ways.

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

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

One textbook strategy:
Sort \((r_1 + x_1, \ldots, Mr_n + x_n)\) 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 \((-1, \ldots, -1, 0, \ldots, 0)\), 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.

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

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

One textbook strategy:
Sort \((\text{Mr}_1 + x_1; \ldots; \text{Mr}_n + x_n)\) 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 \((-1; \ldots; -1; 0; \ldots; 0)\), 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 \(n(n - 1)/2\) 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
\( \leq 119!(n - 119)! \left( \frac{2^{31} + n - 1}{n} \right) \) 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.
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

\[ \leq 119!(n - 119)! \binom{2^{31} + n - 1}{n} \] ways;

\[ \text{i.e., } < 1.02 \cdot 2^{31n}/\binom{n}{119} \] 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.
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.

Input is produced in $\frac{n!}{(n - 119)! \left(2^{31} + n - 1\right)}$ ways;
$\frac{1.02 \cdot 2^{31n}}{\binom{n}{119}}$ ways.

$< 1 : 0.02$ 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.

Converting bubblesort into constant-time bubblesort loses only a constant factor: cost of constant-time minmax.
Simulate uniform random $r_i$ using RNG: e.g., stream cipher.


Example: $n = 6960$ bits; weight 119; 31-bit $r_i$; no restart.

Any output is produced in $\leq \frac{2^{31} + n - 1}{n}$ ways; $\leq \binom{n}{119}$ ways. Decrease 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.
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 $\leq \frac{n!}{(n-119)!}$ ways; i.e., $< 1 : 02 \cdot 2^{31} n = n^{119}$ 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.

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 \( n(n - 1)/2 \) \text{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 \text{minmax}.

“Sorting network”: sorting algorithm built as constant sequence of \text{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 \( n(n-1) = 2 \) 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);
        }
    }
}
```
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 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; $(1/e^2 - e + 4)n - 1$ 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:
the cost of constant-time \( \min\max \).

"Sorting network": sorting algorithm built as a constant sequence of \( \min\max \) operations ("comparators").  

Sorting network on next slide: Batchers merge-exchange sort.  

\[
\Theta \left( n \left( \log^2 n \right) \right) \ \text{\( \min\max \) operations;}
\]

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

```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; \( n - 1 \) for \( n = 2^e \).
Converting bubblesort into constant-time bubblesort loses only a constant factor: cost of constant-time minmax.

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

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);
        }
    }
}
```

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);
    }
}

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);
    }
}

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 (from 2017
Bernstein–Chuengsatiansup–
Lange–van Vredendaal “NTRU
Prime”): 26692 cycles.
```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.

≥3008 cycles for \( n = 1024 \).


Some gap, but already 5× faster than Intel’s Integrated Performance Primitives library.
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) {
        or (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.


Some gap, but already 5× faster than Intel’s Integrated Performance Primitives library.

Constant-time code faster than “optimized” non-constant-time code? How is this possible?
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 (from 2017
Bernstein–Chuengsatiansup–
Lange–van Vredendaal “NTRU
Prime”): 26692 cycles.

Some gap, but already 5×
faster than Intel’s Integrated
Performance Primitives 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 $n = 1024$.


Some gap, but already 5× faster than Intel’s Integrated Performance Primitives library.
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 \( n = 1024 \).


Some gap, but already 5× faster than Intel’s Integrated Performance Primitives 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 $n = 1024$.


Some gap, but already 5× faster than Intel’s Integrated Performance Primitives 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.


Some gap, but already 5× faster than Intel’s Integrated Performance Primitives 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 \( n = 1024 \).


Some gap, but already 5× faster than Intel’s Integrated Performance Primitives 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., an Intel Haswell CPU core?

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

\[ \geq 3008 \text{ cycles for } n = 1024. \]


Some gap, but already 5 × faster than Intel’s Integrated Performance Primitives 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.)
21 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 \( n = 1024 \).
Some gap, but already 5× faster than Intel’s Integrated Performance Primitives library.

22 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.

23 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) $f, g \mapsto f g$; (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.
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 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 \(\text{uint32}\) string
\((f_0, f_1, \ldots, f_{\ell-1})\) represents
the nonnegative integer
\(f_0 + 2^{32}f_1 + \cdots + 2^{32(\ell-1)}f_{\ell-1}\).
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 $2^{255} - 19$.

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

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

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

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

But these functions take variable
time to ensure uniqueness!

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

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

Adding two `$\ell$-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 \(\text{uint32}\) string \((f_0; f_1; \ldots; f_{\ell-1})\) represents the nonnegative integer \(f_0 + 2^{32}f_1 + \cdots + 2^{32(\ell-1)}f_{\ell-1}\). Uniqueness: \(f_0 \neq 0\) 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!

But these functions take variable time to ensure uniqueness!

Adding two \(\ell\)-limb integers: always allocate \(\ell + 1\) limbs.

Don’t remove top zero limb.

Constant-time bigint library: a constant-length \(\text{uint32}\) string \((f_0, f_1, \ldots, f_{\ell-1})\) represents the nonnegative integer \(f_0 + 2^{32}f_1 + \cdots + 2^{32(\ell-1)}f_{\ell-1}\). Can also gain speed this way.
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) \( f, g \mapsto fg \); (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 \( f_0 + 2^{32} f_1 + \ldots + 2^{32(\ell-1)} f_{\ell-1} \).

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 \( \mapsto \) bounds data flow.
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 \texttt{uint32} string \((f_0, f_1, \ldots, f_{\ell-1})\) represents the nonnegative integer \( f_0 + 2^{32}f_1 + \cdots + 2^{32}(\ell-1)f_{\ell-1} \).

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: multiplying \( f \) by \( g \); reducing mod \( 2^{255} - 19 \).

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 \( f_0 + 2^{26} f_1 + 2^{51} f_2 + \cdots + 2^{179} f_7 + 2^{204} f_8 + 2^{230} f_9 \).

Constant bounds on each \( f_i \).
More limbs than before, but save time by avoiding overflows and delaying carries.

After multiplication, replace \( 2^{255} \) with \( 19 \).
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 $f \cdot g \mod 2^{255} - 19$ 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. 

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

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

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 \(\rightarrow\) 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 → 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 the nonnegative integer \(f_0 + 2^{26} f_1 + 2^{51} f_2 + 2^{77} f_3 + 2^{102} f_4 + 2^{128} f_5 + 2^{153} f_6 + 2^{179} f_7 + 2^{204} f_8 + 2^{230} f_9\).

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.

Slightly faster on some CPUs: int32 string \((f_0, f_1, \ldots, f_9)\).
Constant-time bigint library:
a constant-length \( \text{uint32} \) string
\((f_0; f_1; \ldots; f_{\ell-1})\) represents
the nonnegative integer
\( f_0 + 2^{32(f_1)} + 2^{32(f_2)} + \cdots + 2^{32(f_{\ell-1})} \).

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

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 \((f_0, f_1, \ldots, f_9)\).

\begin{align*}
\text{int32} f7_2 &= 2 * f7; \\
\text{int32} g7_19 &= 19 * g7; \\
\ldots \\
\text{int64} f0g4 &= f0 * (\text{int64}) g4; \\
\text{int64} f7g7_38 &= f7_2 * (\text{int64}) g7_19; \\
\ldots \\
\text{int64} h4 &= f0g4 + f2g2 + f4g0 + f6g8_19 + f8g6_19; \\
c4 &= (h4 + (\text{int64})(1<<25)) >> 26; \\
h5 &= c4; h4 &= h4 - c4 << 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:
\[
\text{uint32 string (} f_0, f_1, \ldots, f_9) \text{ represents } f_0 + 2^{26} f_1 + 2^{51} f_2 + 2^{77} f_3 + 2^{102} f_4 + 2^{128} f_5 + 2^{153} f_6 + 2^{179} f_7 + 2^{204} f_8 + 2^{230} f_9.
\]

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

```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;
```
Usually faster representation:

uint32 string \( f_0, f_1, \ldots, f_9 \)

represents \( f_0 + 2^{26} f_1 + 2^{51} f_2 + 2^{102} f_4 + 2^{128} f_5 + 2^{153} f_6 + 2^{204} f_8 + 2^{230} f_9 \).

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 \( f_0, f_1, \ldots, f_9 \).

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

Exercise: Which polynomials are being multiplied?

```
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;
```
Usually faster representation:

```
uint32 string (f_0; f_1; : : : ; f_9)
represents f_0 + 2^{26} f_1 + 2^{51} f_2 + 2^{128} f_5 + 2^{153} f_6 + 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; : : : ; f_9).
```

Initial computation of $h_0$, : : :, $h_9$ 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;
```
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?
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.
\[ f_7_2 = 2 \times f_7; \]
\[ f_7_19 = 19 \times g_7; \]
\[ g_0g_4 = f_0 \times (\text{int64}) \, g_4; \]
\[ g_7g_7_38 = f_7_2 \times (\text{int64}) \, g_7_19; \]
\[ h_4 = f_0g_4 + f_1g_3_2 + f_2g_2 + f_3g_1_2 + f_4g_0 + f_5g_9_38 + f_6g_8_19 + f_7g_7_38 + f_8g_6_19 + f_9g_5_38; \]
\[ c_4 = (h_4 + (\text{int64})(1 \ll 25)) \gg 26; \]
\[ h_5 += c_4; h_4 -= c_4 \ll 26; \]

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 \( h_4 \to h_5 \) 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.
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)):
  - $ni = \text{bit}(n,i)$
  - $x_2, x_3 = \text{cswap}(x_2, x_3, ni)$
  - $z_2, z_3 = \text{cswap}(z_2, z_3, ni)$
  - $x_3, z_3 = (4*(x_2*x_3-z_2*z_3)**2, 4*x_1*(x_2*z_3-z_2*x_3)**2)$
  - $x_2, z_2 = ((x_2**2-z_2**2)**2, 4*x_2*z_2*(x_2**2+A*x_2*z_2+z_2**2))$
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 mathematicalspec; 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)):

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

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

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

$x_3, z_3 = (4*(x_2*x_3-z_2*z_3)^2, 4*x_1*(x_2*z_3-z_2*x_3)^2)$

$x_2, z_2 = ((x_2^2-z_2^2)^2, 4*x_2*z_2*(x_2^2+A*x_2*z_2+z_2^2))$
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)):
  - $n_i = \text{bit}(n, i)$
  - $x_2, x_3 = \text{cswap}(x_2, x_3, n_i)$
  - $z_2, z_3 = \text{cswap}(z_2, z_3, n_i)$
  - $x_3, z_3 = (4(x_2x_3 - z_2z_3)^2, 4x_1(x_2z_3 - z_2x_3)^2)$
  - $x_2, z_2 = ((x_2^2 - z_2^2)^2, 4x_2z_2(x_2^2 + Ax_2z_2 + z_2^2))$
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)):

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

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

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

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

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

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

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)):

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

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:

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

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

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

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)):

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

\[ x_2, x_3 = \text{cswap}(x_2, x_3, n_i) \]
\[ z_2, z_3 = \text{cswap}(z_2, z_3, n_i) \]
\[ x_2, z_2 = (x_2^2-z_2^2)^2, 4*x_2*z_2*(x_2^2+A*x_2*z_2+z_2^2)) \]

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

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

\[
i \text{ bit}(n, i)
\]
\[
= \text{ cswap}(x_2, x_3, ni)
\]
\[
= \text{ cswap}(z_2, z_3, ni)
\]
\[
= (4 \cdot (x_2 \cdot x_3 - z_2 \cdot z_3) \cdot **2, \)
\]
\[
= (x_2 \cdot z_3 - z_2 \cdot x_3) \cdot **2)
\]
\[
= ((x_2**2 - z_2**2) **2, \)
\]
\[
z2*(x2**2+A*x2*z2+z2**2))
\]

\[
x_2, x_3 = \text{ cswap}(x_2, x_3, ni)
\]
\[
z2, z3 = \text{ cswap}(z2, z3, ni)
\]
\[
x_2, z_2 = ((x_2^2 - z_2^2)^2, \)
\]
\[
4 \cdot x_2 \cdot z_2 \cdot (x_2^2 + A \cdot x_2 \cdot z_2 + z_2^2)
\]
\[
x_3, z_3 = (x_3 \% p, z_3 \% p)
\]
\[
x_2, z_2 = (x_2 \% p, z_2 \% p)
\]
\[
cut(x_2)
\]
\[
cut(x_3)
\]
\[
cut(z_2)
\]
\[
cut(z_3)
\]

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

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

NIST P-256 prime
\(2^{256} - 2^{224} + 2^{192} + 2^{96} - 1\).

ECDSA standard specifies
reduction procedure given
an integer \(A < p^2\):

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
\(P_i = A_i \cdot 2^{32 \cdot i}\).

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"
ref10 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)):

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

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

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

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

\[ x_2, z_2 = \left((x_2^2 - z_2^2)^2, 4x_2 \cdot z_2 (x_2^2 + A \cdot x_2 \cdot z_2 + z_2^2)\right) \]

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

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

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

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

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

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

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

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

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

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

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

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 \( T; S_1; S_2; S_3; S_4; D_1; D_2; D_3; D_4 \) 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)):

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

---

“What a difference a prime makes”

NIST P-256 prime $p$ is
\[ 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1. \]

ECDSA standard specifies reduction procedure given an integer “$A$ less than $p^2$”:

Write $A$ as
\[ A = (A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, A_{10}, A_9, A_8, A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0), \]
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'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 $\sum_i A_i 2^{32i}$.

Define

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

$$(A_7, A_6, 0, A_5, A_4, 0, A_3, A_2, A_1, A_0);$$

$$(A_{15}, A_{14}, 0, A_{13}, A_{12}, 0, A_{11}, A_{10}, A_9, A_8);$$

$$(A_{15}, A_{14}, 0, A_{13}, 0, A_{12}, A_{11}, A_{10}, A_9, A_8);$$

$$(A_{15}, A_{14}, 0, A_{13}, A_{12}, 0, A_{11}, A_{10}, A_9, A_8);$$

$$(A_{15}, A_{14}, A_{13}, 0, A_{12}, 0, A_{11}, A_{10}, A_9, A_8).$$

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

Reduce “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

\[
(A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)
\]

\[
(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, A_{10}, A_9, A_8, A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)
\]

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

\[
(A_8, A_{13}, A_{15}, A_{14}, A_{12}, A_{11}, A_{10}, A_9, A_8, A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)
\]

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

\[
(A_{11}, A_9, 0, 0, 0, A_{13}, A_{12}, A_{11}, A_{10}, A_9, A_8, A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)
\]

\[
(A_{12}, 0, A_{10}, A_9, A_8, A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)
\]

\[
(A_{13}, 0, A_{11}, A_{10}, A_9, A_8, A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)
\]

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

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

\[(0, A_{15}, A_{14}, A_{13}, A_{12}, 0, 0, 0, 0, A_{10}, A_9, A_8, A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)\]

\[(A_{15}, A_{14}, 0, 0, 0, A_{10}, A_9, A_8, A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)\]

\[(A_8, A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, A_{10}, A_9, A_8, A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)\]

\[(A_{10}, A_8, 0, 0, 0, A_{13}, A_{12}, A_{11}, A_{10}, A_9, A_8, A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)\]

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

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

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

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

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

ECDSA standard specifies

a reduction procedure given

an integer “$A$ less than $p^2$”:

write $A$ as

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

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

$$(A_{15}, A_{14}, 0, 0, 0, A_{10}, A_9, A_8),$$

$$(A_8, A_{13}, A_{15}, A_{14}, A_{13}, A_{11}, A_{10}, A_9),$$

$$(A_{10}, A_8, 0, 0, 0, A_{13}, A_{12}, A_{11}),$$

$$(A_{11}, A_9, 0, 0, A_{15}, A_{14}, A_{13}, A_{12}),$$

$$(A_{12}, 0, A_{10}, A_9, A_8, A_{15}, A_{14}, A_{13}),$$

$$(A_{13}, 0, A_{11}, A_{10}, A_9, 0, A_{15}, A_{14}).$$

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

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

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

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 reduction procedure given an integer \( A \) less than $p^2$:

Write \( A \) as $$(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, 0, 0, 0),$$ meaning $P_i A_i 2^{32 i}$.

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

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 \( p \).
\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, 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.

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?
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?
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 $p$.

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?
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?