Cryptographic software engineering, part 2

Daniel J. Berntein

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:

```
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:

```
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:

```
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:

```
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:

```
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.

raphic
engineering,

. Bernstein

e:
al software engineering.
const-time instructions.
aring time to lower bound.

e: Adding 1000 integers
ex-M4F. Lower bound:
cycles for n LDR + n ADD.
not knowing this . . .

Reference implementation:

```
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 mov

```
int sum
{
  int re
  int i
  for (i
    resu
  returr
}
```

ng,

n

---

e engineering.

e instructions.

to lower bound.

1000 integers

ower bound:

n LDR + n ADD.

ing this . . .

Reference implementation:

```
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 po

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

ng.
ons.
pound.
gers
nd:
ADD.
.

Reference implementation:

```
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:

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

Reference implementation:

```
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:

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

Reference implementation:

```
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:

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

8010 cycles.

ce implementation:

```
(int *x)

esult = 0;
;
i = 0;i < 1000;++i)
ult += x[i];
n result;
```

: 8012 cycles.

: 8012 cycles.

: 8012 cycles.

: 8012 cycles.

Try moving the pointer:

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

8010 cycles.

Try coun

```
int sum
{
  int re
  int i
  for (i
    resu
  returr
}
```

entation:

```
;

 1000;++i)
i];
```

cles.

cles.

cles.

cles.

Try moving the pointer:

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

8010 cycles.

Try counting dow

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

Try moving the pointer:

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

8010 cycles.

Try counting down:

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

Try moving the pointer:

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

8010 cycles.

Try counting down:

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

Try moving the pointer:

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

8010 cycles.

Try counting down:

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

8010 cycles.

ving the pointer:

```
(int *x)

esult = 0;
;
i = 0;i < 1000;++i)
ult += *x++;
n result;
```

cles.

Try counting down:

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

8010 cycles.

Try usin

```
int sum
{
  int re
  int *
  while
    resu
  retur
}
```

ointer:

```
;
 1000;++i)
++;
```


Try counting down:

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

8010 cycles.

Try using an end p

```
int sum(int *x)
{
  int result = 0
  int *y = x + 1
  while (x != y)
    result += *x
  return result;
}
```

Try counting down:

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

8010 cycles.

Try using an end pointer:

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

Try counting down:

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

8010 cycles.

Try using an end pointer:

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

Try counting down:

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

8010 cycles.

Try using an end pointer:

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

8010 cycles.

nting down:

```
(int *x)

esult = 0;
;
i = 1000;i > 0;--i)
ult += *x++;
n result;
```

cles.

Try using an end pointer:

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

8010 cycles.

Back to

```
int sum
{
  int re
  int i
  for (
    resu
    resu
  }
  return
}
```

n:

```
;

i > 0;--i)
++;
```


Try using an end pointer:

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

8010 cycles.

Back to original. T

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

)

Try using an end pointer:

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

8010 cycles.

Back to original. Try unrolli

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

Try using an end pointer:

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

8010 cycles.

Back to original. Try unrolling:

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

Try using an end pointer:

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

8010 cycles.

Back to original. Try unrolling:

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

5016 cycles.

g an end pointer:

```
(int *x)

esult = 0;
y = x + 1000;
 (x != y)
ult += *x++;
n result;
```

cles.

Back to original. Try unrolling:

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

5016 cycles.

```
int sum
{
  int re
  int i
  for (i
    resu
    resu
    resu
    resu
    resu
  }
  retur
}
```

pointer:

```
;
000;
++;
```


Back to original. Try unrolling:

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

5016 cycles.

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

5

Back to original. Try unrolling:

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

5016 cycles.

6

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

```
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;
}
```

Back to original. Try unrolling:

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

5016 cycles.


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

4016 cycles. “Are we done yet?”

Back to original. Try unrolling:

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

5016 cycles.


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

4016 cycles. "Are we done yet?"

No. Use the lower bound . . .

original. Try unrolling:

```
(int *x)

esult = 0;
;
i = 0;i < 1000;i += 2) {
ult += x[i];
ult += x[i + 1];

n result;
```

cles.


```
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 re
  int *y
  int x0
      x5

  while
    x0 =
    x1 =
    x2 =
    x3 =
    x4 =
    x5 =
    x6 =
```

Try unrolling:

```
;

 1000;i += 2) {
i];
i + 1];
```



```
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 + 1
  int x0,x1,x2,x
      x5,x6,x7,x

  while (x != y)
    x0 = 0[(vola
    x1 = 1[(vola
    x2 = 2[(vola
    x3 = 3[(vola
    x4 = 4[(vola
    x5 = 5[(vola
    x6 = 6[(vola
```

ng:

= 2) {


```
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
    x1 = 1[(volatile int
    x2 = 2[(volatile int
    x3 = 3[(volatile int
    x4 = 4[(volatile int
    x5 = 5[(volatile int
    x6 = 6[(volatile int
```

```
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];
```

```
(int *x)

esult = 0;
;
i = 0;i < 1000;i += 5) {
ult += x[i];
ult += x[i + 1];
ult += x[i + 2];
ult += x[i + 3];
ult += x[i + 4];

n result;
```

cles. “Are we done yet?”
e 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 =
x8 =
x9 =
resu
resu
resu
resu
resu
resu
resu
resu
resu
resu
x0 =
x1 =
```

```
;

 1000;i += 5) {
i];
i + 1];
i + 2];
i + 3];
i + 4];
```

we done yet?"

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[(vola
    x8 = 8[(vola
    x9 = 9[(vola
    result += x0
    result += x1
    result += x2
    result += x3
    result += x4
    result += x5
    result += x6
    result += x7
    result += x8
    result += x9
    x0 = 10[(vol
    x1 = 11[(vol
```

= 5) {

yet?”

.


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

```
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)

esult = 0;
y = x + 1000;
0,x1,x2,x3,x4,
5,x6,x7,x8,x9;

 (x != y) {
= 0[(volatile int *)x];
= 1[(volatile int *)x];
= 2[(volatile int *)x];
= 3[(volatile int *)x];
= 4[(volatile int *)x];
= 5[(volatile int *)x];
= 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 =
x3 =
x4 =
x5 =
x6 =
x7 =
x8 =
x9 =
x +=
resu
resu
resu
resu
resu
resu
```

```
;
000;
3,x4,
8,x9;

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

```
*)x];
*)x];
*)x];
*)x];
*)x];
*)x];
```

8

```
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];
```

9

```
x2 = 12[(volatile int
x3 = 13[(volatile int
x4 = 14[(volatile int
x5 = 15[(volatile int
x6 = 16[(volatile int
x7 = 17[(volatile int
x8 = 18[(volatile int
x9 = 19[(volatile int
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];
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;
```

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

```
    resu
    resu
    resu
    resu
  }

  returr
}
```

```
tile int *)x];
tile int *)x];
tile int *)x];
;
;
;
;
;
;
;
;
;
;
atile int *)x];
atile int *)x];
```



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

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

  return result;
}
```

```
*)x];
*)x];
*)x];

 *)x];
 *)x];
```



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

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

  return result;
}
```

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



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

  return result;
}
```

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



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

  return result;
}
```

2526 cycles. Even better in asm.

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



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

  return result;
}
```

2526 cycles. Even better in asm.

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

```
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."
— [citation needed]

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



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

  return result;
}
```

2526 cycles. Even better in asm.

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

A real ex

Salsa20

30.25 cy

Lower bo

64 bytes

21 · 16 1

20 · 16 1

so at lea

Also ma

ARMv7-

includes

as part o

(Compil

```
atile int *)x];
atile int *)x];
atile int *)x];
atile int *)x];
atile int *)x];
atile int *)x];
atile int *)x];
atile int *)x];

;
;
;
;
;
;
```



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

  return result;
}
```

2526 cycles. Even better in asm.

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

A real example

Salsa20 reference
30.25 cycles/byte

Lower bound for a
64 bytes require
$21 \cdot 16$ 1-cycle AD
$20 \cdot 16$ 1-cycle XO
so at least 10.25 c

Also many rotatio
ARMv7-M instruct
includes free rotat
as part of XOR ins
(Compiler knows t

```
*)x];
*)x];
*)x];
*)x];
*)x];
*)x];
*)x];
*)x];
```

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

  return result;
}
```

2526 cycles. Even better in asm.

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

A real example

Salsa20 reference software:
30.25 cycles/byte on this CP

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
$21 \cdot 16$ 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.)

```
ult += x6;
ult += x7;
ult += x8;
ult += x9;

n result;
```

cles. Even better in asm.

ia: "By the late 1990s for
rformance sensitive code,
ng compilers exceeded the
ance of human experts."
tion needed]

A real example

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

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

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

Detailed
several c
`load_li`
`store_li`

Can repl
(Compil

Then ob
18 cycles
plus 5 cy
Still far

;

;

;

;

better in asm.

ne late 1990s for
sensitive code,
ers exceeded the
man experts."
d]

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 benchma
several cycles/byte
`load_littleendia`
`store_littleendi`

Can replace with L
(Compiler doesn't

Then observe 23 c
18 cycles/byte for
plus 5 cycles/byte
Still far above 10.2

asm.

0s for
code,
ed the
rts."

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 S
(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/

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.

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

xample

reference software:
cles/byte on this CPU.

ound for arithmetic:
require
-cycle ADDs,
-cycle XORs,
st 10.25 cycles/byte.

ny rotations, but
M instruction set
free rotation
of XOR instruction.
er 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 o
should b
Don't tr
optimize

software:
on this CPU.

rithmetic:

Ds,
Rs,
cycles/byte.

ns, but
tion set
ion
struction.
his.)

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 S
should be in regist
Don't trust compi
optimize register a

PU.

e.

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 wo
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.)

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.)

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.

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.)

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.

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

benchmarks show
cycles/byte spent on
ttleendian and
ittleendian.

ace with LDR and STR.
er doesn't see this.)

oserve 23 cycles/byte:
s/byte for rounds,
ycles/byte overhead.
above 10.25 cycles/byte.

nostly loads, stores.
e load/store cost by
g "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:/
includes
of 614 c
>20 imp

Haswell:
impleme
gcc -O3
is 6.15×
Salsa20

rks show
e spent on
n and
an.

LDR and STR.
see this.)

cycles/byte:
rounds,
overhead.
25 cycles/byte.

ds, stores.
re cost by
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.c
includes 2392 imp
of 614 cryptograp
$>$20 implementati

Haswell: Reasonab
implementation co
`gcc -O3 -fomit-`
is 6.15$\times$ slower th
Salsa20 implement

STR.

e:

byte.

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 implementatio
of 614 cryptographic primiti
>20 implementations of Sal

Haswell: Reasonably simple
implementation compiled wi
`gcc -O3 -fomit-frame-po`
is 6.15× slower than fastest
Salsa20 implementation.

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

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

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

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

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

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

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

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

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

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

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

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

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

f the 16 Salsa20 words
e in registers?
ust compiler to
e register allocation.

ads consecutive?
ust compiler to
e instruction scheduling.

FPU instead of stack?
ust compiler to
e instruction selection.

er CPUs,
g vector instructions
l 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 ran

Goal: Pu
into a ra

alsa20 words
ters?
ler to
llocation.

cutive?
ler to
on scheduling.

ad of stack?
ler to
on selection.

structions
rmance.

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

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

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

Fast random perm

Goal: Put list (x_1,
into a random ord

rds

ing.

k?

n.

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

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

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

Fast random permutations

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

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

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

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

Fast random permutations

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

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

Haswell: Reasonably simple `ref`
implementation compiled with
`gcc -O3 -fomit-frame-pointer`
is 6.15× 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.

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.

//bench.cr.yp.to

2392 implementations
ryptographic primitives.
plementations of Salsa20.

Reasonably simple `ref`
ntation compiled with
`-fomit-frame-pointer`
slower than fastest
implementation.

implementation
achine-independent"
ntions and best of 121
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.

Simulate
using R

cr.yp.to
lementations
hic primitives.
ons of Salsa20.

ply simple `ref`
ompiled with
`frame-pointer`
an fastest
tation.

tation
lependent"
best of 121
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.

Simulate uniform r
using RNG: e.g., s

ons
ves.
sa20.

ref
th
inter

y
21
wer.

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 ciph

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
$(\pm1, \ldots, \pm1, 0, \ldots, 0)$, 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 $\le 119!(n-119)!\binom{2^{31}+n-1}{n}$ ways;
i.e., $< 1.02 \cdot 2^{31n}/\binom{n}{119}$ ways.
Factor $<$1.02 increase in attacker's chance of winning.

dom permutations

ut list $(x_1, \ldots, x_n)$
ndom order.

tbook strategy:
$r_1 + x_1, \ldots, Mr_n + x_n)$ for
$(r_1, \ldots, r_n)$, suitable M.

e encryption example:
ly order 6960 bits
., $0, \ldots, 0)$, weight 119.

ncryption example:
ly order 761 trits
, $\pm 1, 0, \ldots, 0)$, 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
$\le 119!(n-119)!\binom{2^{31}+n-1}{n}$ ways;
i.e., $< 1.02 \cdot 2^{31n}/\binom{n}{119}$ ways.
Factor <1.02 increase in attacker's chance of winning.

Which s

Referenc
$n(n-1)$

utations

. . . , x_n)

er.

tegy:

. , $Mr_n + x_n$) for

), suitable M.

on example:

960 bits

, weight 119.

example:

61 trits

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

Which sorting algo

Reference bubbles

$n(n-1)/2$ `minma`

x_n) for
e M.

e:

19.

86.

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

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

Restart on collision?
Uniform distribution; some cost.

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

Which sorting algorithm?

Reference bubblesort code d
$n(n-1)/2$ `minmax` operatio

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

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

Restart on collision?
Uniform distribution; some cost.

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

Which sorting algorithm?

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

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

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

Restart on collision? Uniform distribution; some cost.

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

Which sorting algorithm?

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

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

Which sorting algorithm?

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

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.

e uniform random r_i
NG: e.g., stream cipher.

ny bits in r_i? Negligible
s? Occasional collisions?

on collision?
distribution; some cost.

e: $n = 6960$ bits;
.19; 31-bit r_i; no restart.
put is produced in
$n - 119)!\binom{2^{31}+n-1}{n}$ ways;
$.02 \cdot 2^{31n}/\binom{n}{119}$ ways.
<1.02 increase in
'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.

Converti
constant
loses onl
cost of c

random r_i
tream cipher.

r_i? Negligible
onal collisions?

n?
on; some cost.

0 bits;
r_i; no restart.
duced in
$\binom{2^{31}+n-1}{n}$ ways;
$\binom{n}{119}$ ways.
ease in
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 bubble
constant-time bub
loses only a consta
cost of constant-ti

her.

gible

ons?

cost.

tart.

ways;

s.

g.

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 `minma`

Which sorting algorithm?

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

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

But these algorithms rely on secret branches and secret indices.

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

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

Which sorting algorithm?

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

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

But these algorithms rely on secret branches and secret indices.

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

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

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

Which sorting algorithm?

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

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

But these algorithms rely on secret branches and secret indices.

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

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

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

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

orting algorithm?

ce bubblesort code does
)/2 minmax operations.

andard algorithms use
perations: mergesort,
t, heapsort, radixsort, etc.

se algorithms rely on
ranches and secret indices.

: convert mergesort
stant-time mergesort
(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 $n = 2^e$.

```
void so
{ long l
  t = 1
  while
  for (p
    for
      i
    for
      fo
    }
  }
}
```

orithm?

ort code does
x operations.

gorithms use
mergesort,
t, radixsort, etc.

ms rely on
d secret indices.

mergesort
e mergesort
tions.

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

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

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

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

oes
ns.
se
,
t, etc.
n
ndices.
t

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

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

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


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

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

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

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

$\Theta(n(\log n)^2)$ `minmax` operations;
$(1/4)(e^2 - e + 4)n - 1$ for $n = 2^e$.


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

ng bubblesort into
t-time bubblesort
ly a constant factor:
constant-time minmax.

; network":
algorithm built as
t sequence of minmax
ns ("comparators").

network on next slide:
's merge-exchange sort.
$n)^2)$ minmax operations;
$^2 - e + 4)n - 1$ for $n = 2^e$.


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

How ma
Intel Ha

Every cy
"min" o
8 32-bit

sort into
blesort
ant factor:
me minmax.

:
built as
of minmax
parators").

n next slide:
xchange sort.
ax operations;
$n-1$ for $n=2^e$.


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

How many cycles
Intel Haswell CPU

Every cycle: a vec
"min" operations
8 32-bit "max" op

ax.

x

e:

ort.

ions;

$n = 2^e$.


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

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

Every cycle: a vector of 8 32
"min" operations and a vect
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.

```
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.

Some gap, but already 5$\times$ faster than Intel's Integrated Performance Primitives library.

```
rt(int32 *x,long long n)
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)
f (!(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.

$\geq$3008 cycles for $n = 1024$. Current software (from 2017 Bernstein–Chuengsatiansup–Lange–van Vredendaal “NTRU Prime”): 26692 cycles.

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

Constan
“optimiz
code? H

```
*x,long long n)
q,i;
 2) return;
) t += t;
 0;p >>= 1) {
 < n-p;++i)
p))
+i,x+i+p);
 > p;q >>= 1) {
;i < n-q;++i)
& p))
(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.

Some gap, but already 5$\times$ faster than Intel's Integrated Performance Primitives library.

Constant-time coc
"optimized" non-c
code? How is this

```
ong n)

n;

1) {
i)

;
= 1) {
++i)

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.

Some gap, but already 5$\times$ faster than Intel's Integrated Performance Primitives library.

Constant-time code faster th
"optimized" non-constant-ti
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.

$\geq$3008 cycles for $n = 1024$. Current software (from 2017 Bernstein–Chuengsatiansup–Lange–van Vredendaal "NTRU Prime"): 26692 cycles.

Some gap, but already 5$\times$ 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.

$\geq$3008 cycles for $n = 1024$. Current software (from 2017 Bernstein–Chuengsatiansup–Lange–van Vredendaal "NTRU Prime"): 26692 cycles.

Some gap, but already 5$\times$ 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.

$\geq$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?

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.

ny cycles on, e.g.,
swell CPU core?

cle: a vector of 8 32-bit
perations and a vector of
"max" operations.

cycles for $n = 1024$.
software (from 2017
n–Chuengsatiansup–
an Vredendaal "NTRU
: 26692 cycles.

ap, but already 5×
an Intel's Integrated
ance 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

Basic EC
add, sub
integers

(Basic N
add, sub
polynom

on, e.g.,
core?

tor of 8 32-bit
and a vector of
erations.

$n = 1024$.
from 2017
satiansup–
daal “NTRU
ycles.

eady 5×
Integrated
itives 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 arithmeti

Basic ECC operati
add, sub, mul of, e
integers mod 2^{255}

(Basic NTRU oper
add, sub, mul of, e
polynomials mod x

2-bit
tor of
RU
ry.

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 $x^{761} - x -$

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 $f_{\ell-1} \neq 0$.

t-time code faster than
zed" non-constant-time
low is this possible?

ptimize algorithms
ve model of CPUs:
hes are fast.
m access is fast.

e evolving
nd farther away
s naive model.
ental hardware costs
ant-time arithmetic are
wer 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 p
on this r
fg; (2)

de faster than
constant-time
possible?

lgorithms
of CPUs:
st.
is fast.

r away
odel.
ware costs
rithmetic are
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 fu
on this representat
fg; (2) $f, g \mapsto f$ r

han
me

s
are
cess.

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 ac
on this representation: (1) f
fg; (2) $f, g \mapsto f \bmod g$; 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) $f, g \mapsto f \bmod g$; 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) $f, g \mapsto f \bmod g$; etc.

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

Modular arithmetic

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

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

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

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

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

But these functions take variable
time to ensure uniqueness!

Modular arithmetic

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

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

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

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

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

But these functions take variable time to ensure uniqueness!

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

r arithmetic

CC operations:
, mul of, e.g.,
mod $2^{255} - 19$.

ITRU operations:
, mul of, e.g.,
ials mod $x^{761} - x - 1$.)

“big-integer library”:
le-length `uint32` string
. . , $f_{\ell-1}$) represents
negative integer
$f_1 + \cdots + 2^{32(\ell-1)} f_{\ell-1}$.
ess: $\ell = 0$ 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 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.

Constan
a consta
($f_0, f_1, \ldots$
the nonn
$f_0 + 2^{32}$

Adding t
always a
Don't re

c

ons:

e.g.,

$- 19$.

rations:

e.g.,

$x^{761} - x - 1$.)

er library”:

uint32 string

epresents

nteger

$2^{32(\ell-1)} f_{\ell-1}$.

) 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 big

a constant-length

$(f_0, f_1, \ldots, f_{\ell-1})$ re

the nonnegative in

$f_0 + 2^{32} f_1 + \cdots +$

Adding two ℓ-limb

always allocate ℓ +

Don’t remove top

$-$ 1.)

:

ring

-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` st $(f_0, f_1, \ldots, f_{\ell-1})$ represents the nonnegative integer $f_0 + 2^{32} f_1 + \cdots + 2^{32(\ell-1)} f_{\ell}$

Adding two ℓ-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) $f, g \mapsto f \bmod g$; etc.

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

But these functions take variable time to ensure uniqueness!

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

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

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

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

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

But these functions take variable time to ensure uniqueness!

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

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

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

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

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

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

But these functions take variable time to ensure uniqueness!

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

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

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

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

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

provides functions acting
representation: (1) $f, g \mapsto$
$f, g \mapsto f \bmod g$; etc.

plementor using library:
$f, g \bmod 2^{255} - 19$
multiplying f by g;
cing mod $2^{255} - 19$.

se functions take variable
ensure uniqueness!

different representation
tant-time arithmetic.
gain speed this way.

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

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

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

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

Usually
`uint32`
represen
$2^{77} f_3 +$
$2^{179} f_7 +$

Constan

More lim
but save
overflow

After mu
replace

unctions acting
tion: (1) $f, g \mapsto$
mod g; etc.

using library:
$2^{255} - 19$
f by g;
$2^{255} - 19$.

ns take variable
queness!

epresentation
arithmetic.
ed this way.

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

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

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 repr
`uint32` string $(f_0,$
represents $f_0 + 2^{2}$
$2^{77} f_3 + 2^{102} f_4 + 2$
$2^{179} f_7 + 2^{204} f_8 +$

Constant bound o

More limbs than b
but save time by a
overflows and dela

After multiplicatio
replace 2^{255} with

cting

$f, g \mapsto$

t.

ary:

.

riable

ion

.

y.

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

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

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

f mod p is as short as p.

Usually faster representation
`uint32` string $(f_0, f_1, \ldots, f_9)$
represents $f_0 + 2^{26} f_1 + 2^{51} f$
$2^{77} f_3 + 2^{102} f_4 + 2^{128} f_5 + 2^{1}$
$2^{179} f_7 + 2^{204} f_8 + 2^{230} f_9$.

Constant bound on each f_i.

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

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

Can also track bounds more
refined than $2^0, 2^{32}, 2^{64}, 2^{96}, \ldots$;
but no limbs→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 ℓ-limb integers:
always allocate $\ell + 1$ limbs.
Don't remove top zero limb.

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

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

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

Constant bound on each f_i.

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

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

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

t-time bigint library:
nt-length uint32 string
. ., $f_{\ell-1}$) represents
negative integer
$f_1 + \cdots + 2^{32(\ell-1)} f_{\ell-1}$.

two ℓ-limb integers:
llocate $\ell + 1$ limbs.
emove top zero limb.

track bounds more
than $2^0, 2^{32}, 2^{64}, 2^{96}, \ldots$;
imbs→bounds data flow.

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

```
int32 f
int32 g
...
int64 f
int64 f
  f7_2
...
int64 h
...
c4 = (h
h5 += c
```

int library:
uint32 string
epresents
nteger
$2^{32(\ell-1)} f_{\ell-1}$.

integers:
+ 1 limbs.
zero limb.

unds more
$^2, 2^{64}, 2^{96}, \ldots$;
unds data flow.

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

```
int32 f7_2 = 2 *
int32 g7_19 = 19
...
int64 f0g4 = f0
int64 f7g7_38 =
  f7_2 * (int64)
...
int64 h4 = f0g4
         + f2g2
         + f4g0
         + f6g8_
         + f8g6_
...
c4 = (h4 + (int6
h5 += c4; h4 -=
```

tring

$-1\cdot$

,...;
flow.

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

Constant bound on each f_i.

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

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

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

```
int32 f7_2 = 2 * f7;
int32 g7_19 = 19 * g7;
...
int64 f0g4 = f0 * (int64)
int64 f7g7_38 =
  f7_2 * (int64) g7_19;
...
int64 h4 = f0g4 + f1g3_2
         + f2g2 + f3g1_2
         + f4g0 + f5g9_38
         + f6g8_19 + f7g7
         + f8g6_19 + f9g5
...
c4 = (h4 + (int64)(1<<25)
h5 += c4; h4 -= c4 << 26;
```

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

Constant bound on each f_i.

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

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

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


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

faster representation:
string $(f_0, f_1, \ldots, f_9)$
ts $f_0 + 2^{26} f_1 + 2^{51} f_2 +$
$2^{102} f_4 + 2^{128} f_5 + 2^{153} f_6 +$
$2^{204} f_8 + 2^{230} f_9$.

t bound on each f_i.

nbs than before,
time by avoiding
s and delaying carries.

ultiplication,
2^{255} with 19.

faster on some CPUs:
tring $(f_0, f_1, \ldots, f_9)$.


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

Initial co
is polyno
modulo
Exercise
are being

esentation:

$f_1, \ldots, f_9)$

${}^{6} f_1 + 2^{51} f_2 +$

${}^{128} f_5 + 2^{153} f_6 +$

$2^{230} f_9$.

n each f_i.

efore,
avoiding
ying carries.

n,
19.

some CPUs:
$_1, \ldots, f_9)$.


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

Initial computatio
is polynomial mult
modulo $x^{10} - 19$.
Exercise: Which p
are being multiplie

n:

)

f_2 +

$^{53} f_6$ +

es.

ls:


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

Initial computation of h0, . .

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;
```


Initial computation of `h0`, . . . , `h9` 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;
```


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

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

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


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

Exercise: Which polynomials are being multiplied?

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

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

```
7_2 = 2 * f7;
7_19 = 19 * g7;

0g4 = f0 * (int64) g4;
7g7_38 =
* (int64) g7_19;

4 = f0g4 + f1g3_2
  + f2g2 + f3g1_2
  + f4g0 + f5g9_38
  + f6g8_19 + f7g7_38
  + f8g6_19 + f9g5_38;

4 + (int64)(1<<25)) >> 26;
4; h4 -= c4 << 26;
```


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

Exercise: Which polynomials are being multiplied?

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

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

Much m
see, e.g.

```
 f7;
 * g7;

* (int64) g4;

 g7_19;

+ f1g3_2
+ f3g1_2
+ f5g9_38
19 + f7g7_38
19 + f9g5_38;

4)(1<<25)) >> 26;
c4 << 26;
```


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

Exercise: Which polynomials are being multiplied?

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

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

Much more about

see, e.g., 2015 Ch

```
 g4;
3
_38
_38;
) >> 26;
```


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

Exercise: Which polynomials are being multiplied?

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

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

Much more about ECC spee

see, e.g., 2015 Chou.

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

Exercise: Which polynomials are being multiplied?

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

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

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

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

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

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

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

Verifying constant time: increasingly automated.

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

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

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

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

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

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

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

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

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.

omputation of h0, . . . , h9
omial multiplication
$x^{10} - 19$.
: Which polynomials
g multiplied?

on modulo $x^{10} - 19$
ies such as h4→h5
e the product
ted-size representation
for next multiplication.

of computation:
epresentation
que representation
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 h
impleme
plus occ
against t

```
p = 2**2
A = 4866
x2,z2,x3
for i in
  ni = 
  x2,x3
  z2,z3
  x3,z3
   4*x1*
  x2,z2
   4*x2*
```

n of h0, . . . , h9
tiplication

olynomials
ed?

$x^{10} - 19$
s h4→h5
uct
epresentation
nultiplication.

ation:
tion
entation
rk transmission.

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

Verifying constant time: increasingly automated.

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

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

gfverif has verified
implementation of
plus occasional an
against the followi

```
p = 2**255-19
A = 486662
x2,z2,x3,z3 = 1,
for i in reverse
  ni = bit(n,i)
  x2,x3 = cswap(
  z2,z3 = cswap(
  x3,z3 = (4*(x2
   4*x1*(x2*z3-z
  x2,z2 = ((x2**
   4*x2*z2*(x2**
```

., h9

s

on

on.

ssion.

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 specifi

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

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

Verifying constant time: increasingly automated.

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

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

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

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

ore about ECC speed:
, 2015 Chou.

g constant time:
ngly automated.

can miss rare bugs
acker might trigger.
ve that software
mathematical spec;
mputer check proofs.

s in deploying proven
ware: see, e.g., 2015
n–Schwabe "gfverif";
ACL* X25519 in Firefox.

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

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

```
  x3,z3
  x2,z2
  cut(x
  cut(x
  cut(z
  cut(z
  x2,x3
  z2,z3
cut(x2)
cut(z2)
return
```

What's
is the sa
and is b

ECC speed:
ou.

time:
nated.

are bugs
nt trigger.
ftware
tical spec;
eck proofs.

ing proven
e.g., 2015
e "gfverif";
519 in Firefox.

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

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

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

What's verified: o
is the same as spe
and is between 0 a

ed:

.

n

5

';

efox.

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

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

What's verified: output of r

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



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

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

as verified ref10
entation of X25519,
asional annotations,
the following specification:

```
255-19
662
3,z3 = 1,0,x1,1
n reversed(range(255)):
bit(n,i)
 = cswap(x2,x3,ni)
 = cswap(z2,z3,ni)
 = (4*(x2*x3-z2*z3)**2,
*(x2*z3-z2*x3)**2)
 = ((x2**2-z2**2)**2,
*z2*(x2**2+A*x2*z2+z2**2))
```



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

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

"What a

NIST P-
$2^{256} - 2$

ECDSA
reductio
an integ

Write A
$(A_{15}, A_1$
$A_8, A_7,$
meaning

Define
$T; S_1; S_2$
as

ref10

X25519,

notations,

ng specification:

```
0,x1,1
d(range(255)):

x2,x3,ni)
z2,z3,ni)
*x3-z2*z3)**2,
2*x3)**2)
2-z2**2)**2,
2+A*x2*z2+z2**2))
```



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

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

"What a differenc

NIST P-256 prime

$2^{256} - 2^{224} + 2^{192}$

ECDSA standard

reduction procedu

an integer "A less

Write A as

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

A_8, A_7, A_6, A_5, A

meaning $\sum_i A_i 2^{32}$

Define

$T; S_1; S_2; S_3; S_4; D$

as

cation:

```
255)):
)**2,
*2,
+z2**2))
```



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

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

"What a difference a prime

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 $(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, A_{10}$ $A_8, A_7, A_6, A_5, A_4, A_3, A_2,$ meaning $\sum_i A_i 2^{32i}$.

Define $T; S_1; S_2; S_3; S_4; D_1; D_2; D_3$ 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 $(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, A_{10}, A_9, A_8, A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)$, 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%p,z3%p)
 = (x2%p,z2%p)
2)
3)
2)
3)
 = cswap(x2,x3,ni)
 = cswap(z2,z3,ni)

x2*pow(z2,p-2,p)
```

verified: output of `ref10`
me as spec mod p,
etween 0 and $p-1$.

"What a difference a prime makes"

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

ECDSA standard specifies reduction procedure given an integer "A less than p^2":

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

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

$(A_7, A_6,$
$(A_{15}, A_1$
$(0, A_{15},$
$(A_{15}, A_1$
$(A_8, A_{13}$
$(A_{10}, A_8$
$(A_{11}, A_9$
$(A_{12}, 0,$
$(A_{13}, 0,$

Comput
$S_4 - D_1$

Reduce
subtract

```
z3%p)
z2%p)
```

```
x2,x3,ni)
z2,z3,ni)
```

```
,p-2,p)
```

utput of ref10
c mod p,
and $p - 1$.

“What a difference a prime makes”

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

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

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

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

$(A_7, A_6, A_5, A_4, A_3$
$(A_{15}, A_{14}, A_{13}, A_{12}$
$(0, A_{15}, A_{14}, A_{13},$
$(A_{15}, A_{14}, 0, 0, 0,$
$(A_8, A_{13}, A_{15}, A_{14},$
$(A_{10}, A_8, 0, 0, 0, A$
$(A_{11}, A_9, 0, 0, A_{15},$
$(A_{12}, 0, A_{10}, A_9, A$
$(A_{13}, 0, A_{11}, A_{10},$

Compute $T + 2S_1$
$S_4 - D_1 - D_2 - D$

Reduce modulo p
subtracting a few

ef10

“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
$(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, 0, 0$
$(0, A_{15}, A_{14}, A_{13}, A_{12}, 0, 0, 0$
$(A_{15}, A_{14}, 0, 0, 0, A_{10}, A_9, A_8$
$(A_8, A_{13}, A_{15}, A_{14}, A_{13}, A_{11},$
$(A_{10}, A_8, 0, 0, 0, A_{13}, A_{12}, A_1$
$(A_{11}, A_9, 0, 0, A_{15}, A_{14}, A_{13},$
$(A_{12}, 0, A_{10}, A_9, A_8, A_{15}, A_{14}$
$(A_{13}, 0, A_{11}, A_{10}, A_9, 0, A_{15},$

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

Reduce modulo p “by addin
subtracting a few copies” of

"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.

a difference a prime makes"

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

standard specifies
n procedure given
er "A less than p^2":

as
$_4, A_{13}, A_{12}, A_{11}, A_{10}, A_9,$
$A_6, A_5, A_4, A_3, A_2, A_1, A_0)$,
$\sum_i A_i 2^{32i}$.

$_2; S_3; S_4; D_1; D_2; D_3; D_4$

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

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
Variable-

e a prime makes”

e p is

$^2 + 2^{96} - 1$.

specifies

re given

than p^2”:

$_2, A_{11}, A_{10}, A_9,$

$_4, A_3, A_2, A_1, A_0)$,

^{2i}.

$D_1; D_2; D_3; D_4$

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

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 co

Variable-time loop

makes”

1.

, A_9,

A_1, A_0),

; D_4

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

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

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

What is “a few copies”?

Variable-time loop is unsafe.

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

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

Reduce modulo p "by adding or subtracting a few copies" of p.

What is "a few copies"?

Variable-time loop is unsafe.

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

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

Reduce modulo p "by adding or subtracting a few copies" of p.

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.

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

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

Delay until end of computation?
Trouble: "A less than p^2".

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

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

Reduce modulo p "by adding or subtracting a few copies" of p.

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

Correct but quite slow:
conditionally add $4p$,
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?

$A_5, A_4, A_3, A_2, A_1, A_0)$;
$_4, A_{13}, A_{12}, A_{11}, 0, 0, 0)$;
$A_{14}, A_{13}, A_{12}, 0, 0, 0)$;
$_4, 0, 0, 0, A_{10}, A_9, A_8)$;
$, A_{15}, A_{14}, A_{13}, A_{11}, A_{10}, A_9)$;
$, 0, 0, 0, A_{13}, A_{12}, A_{11})$;
$, 0, 0, A_{15}, A_{14}, A_{13}, A_{12})$;
$A_{10}, A_9, A_8, A_{15}, A_{14}, A_{13})$;
$A_{11}, A_{10}, A_9, 0, A_{15}, A_{14})$.

e $T + 2S_1 + 2S_2 + S_3 +$
$- D_2 - D_3 - D_4$.

modulo p "by adding or
ing a few copies" of p.

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

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

Delay until end of computation?
Trouble: "A less than p^2".

Even worse: what about platforms
where 2^{32} isn't best radix?

There ar
cryptogr
affect di
correct c

e.g. ECD
of scalar

e.g. ECD
addition
EdDSA

$_3, A_2, A_1, A_0)$;

$_2, A_{11}, 0, 0, 0)$;

$A_{12}, 0, 0, 0)$;

$A_{10}, A_9, A_8)$;

$A_{13}, A_{11}, A_{10}, A_9)$;

$_{13}, A_{12}, A_{11})$;

$A_{14}, A_{13}, A_{12})$;

$_8, A_{15}, A_{14}, A_{13})$;

$A_9, 0, A_{15}, A_{14})$.

$+ 2S_2 + S_3 +$
$D_3 - D_4$.

"by adding or
copies" of p.

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

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

Delay until end of computation?
Trouble: "A less than p^2".

Even worse: what about platforms
where 2^{32} isn't best radix?

There are many m
cryptographic desi
affect difficulty of
correct constant-ti

e.g. ECDSA needs
of scalars. EdDSA

e.g. ECDSA splits
additions into seve
EdDSA uses comp

A_0);
, 0);
);
);
A_{10}, A_9);
$_1$);
A_{12});
., A_{13});
A_{14}).

S_3 +

g or
p.

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

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

Delay until end of computation?
Trouble: "A less than p^2".

Even worse: what about platforms
where 2^{32} isn't best radix?

There are many more ways t
cryptographic design choices
affect difficulty of building fa
correct constant-time softwa

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

e.g. ECDSA splits elliptic-cu
additions into several cases.
EdDSA uses complete formu

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?