SORBONNE UNIVERSITÉ

cnrs

Ph.D. Defense
Usuba, Optimizing Bitslicing Compiler

Darius MERCADIER

Jury:

- Pierre-Évariste DAGAND (Advisor)
- Gilles MULLER (Supervisor)
- Karthik BHARGAVAN (Reviewer)
- Sandrine BLAZY (Reviewer)
- Caroline COLLANGE
- Xavier LEROY
- Thomas PORNIN
- Damien VERGNAUD

Defended on:
November 20, 2020

LIP6

Inria
INVENTEURS DU MONDE NUMÉRIQUE

Cryptography

Are you working on your PhD?

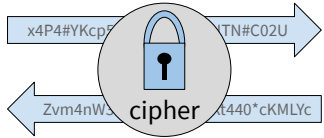
Nah, mostly playing video games

Cryptography

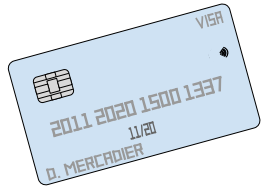
Are you working on your PhD?

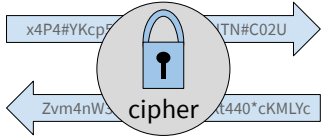
Nah, mostly playing video games

Cryptography

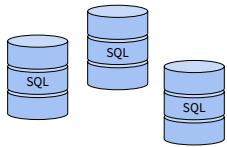


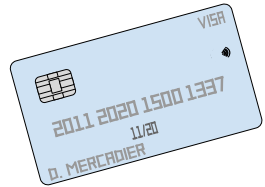
Cryptography

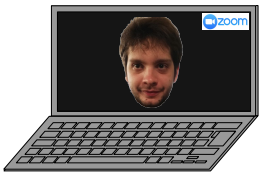


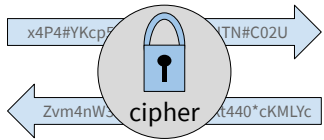


Cryptography

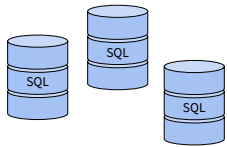


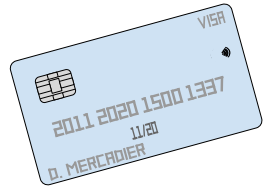


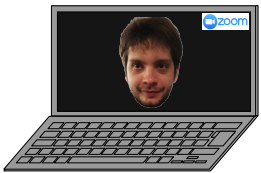


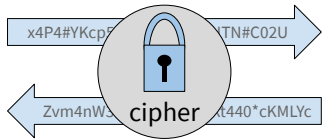


Cryptography

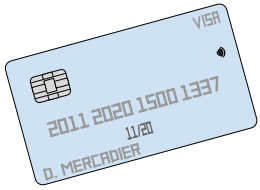


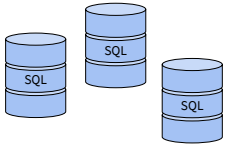


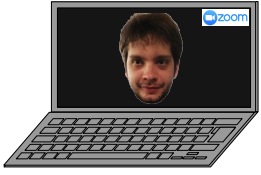


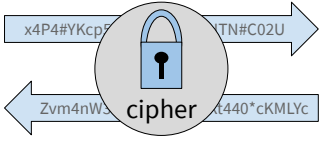


Cryptography

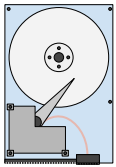


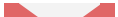


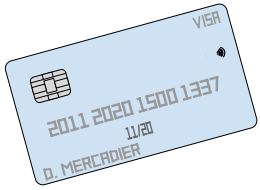


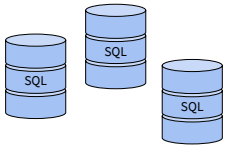


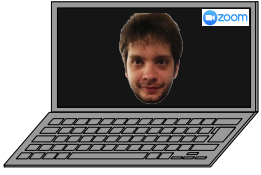
Cryptography

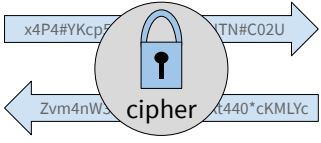




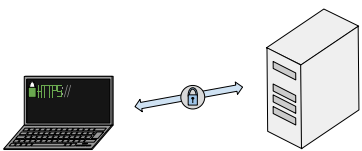




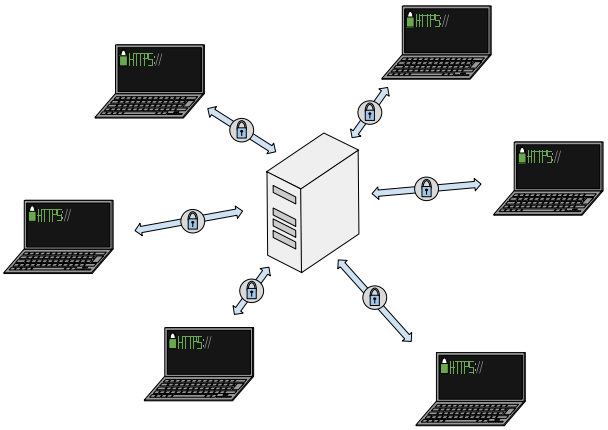




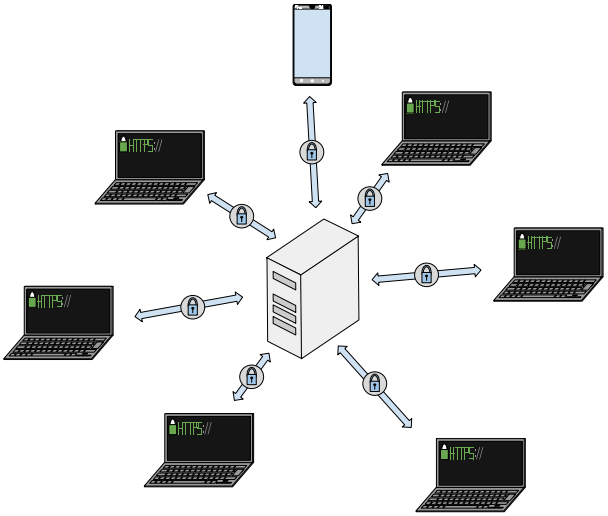
Cryptography: need for speed



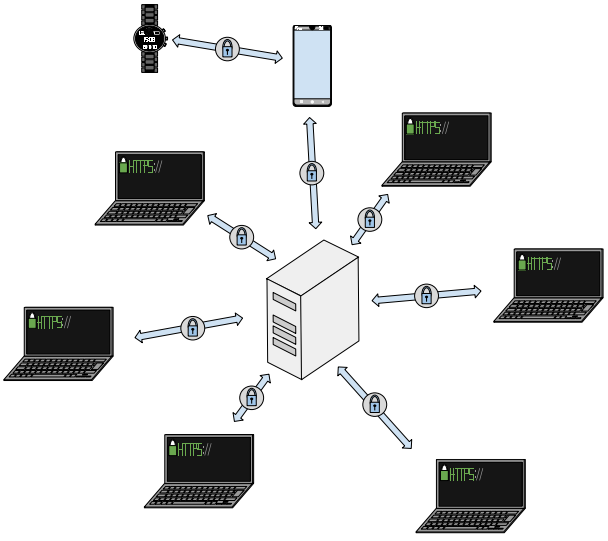
Cryptography: need for speed



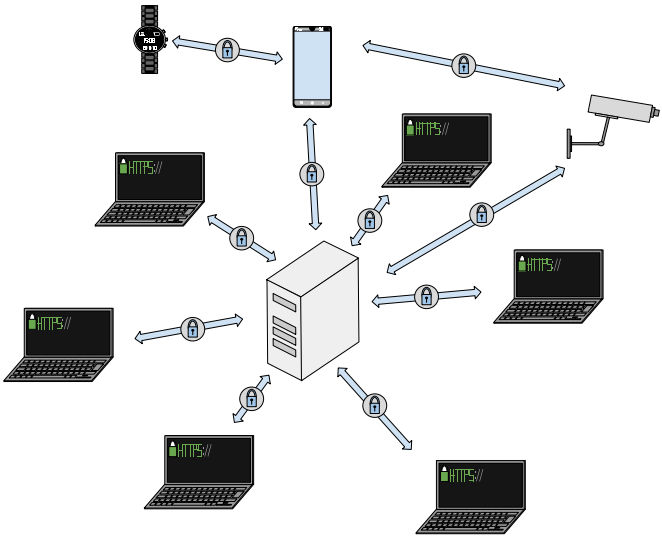
Cryptography: need for speed



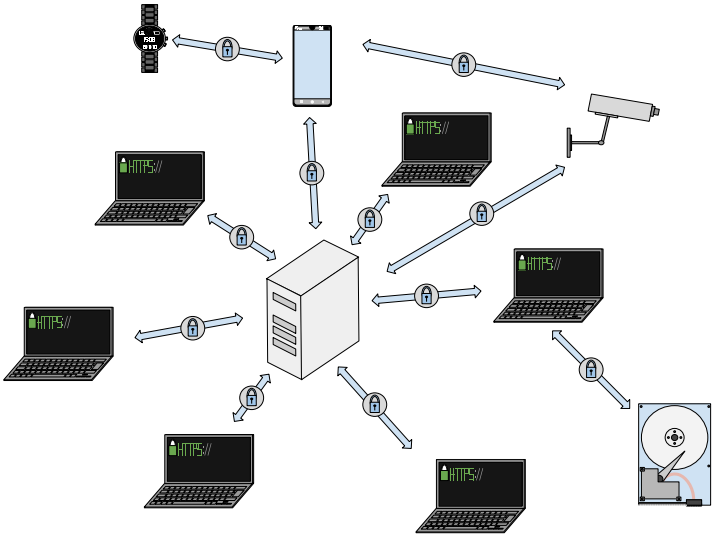
Cryptography: need for speed



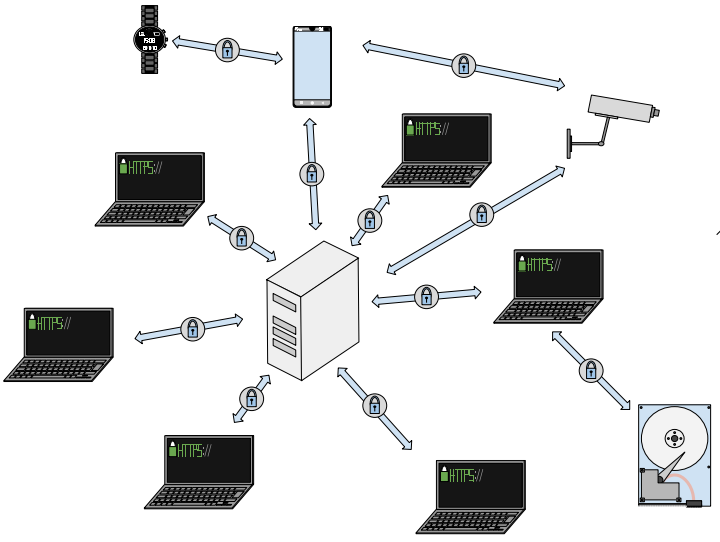
Cryptography: need for speed



Cryptography: need for speed

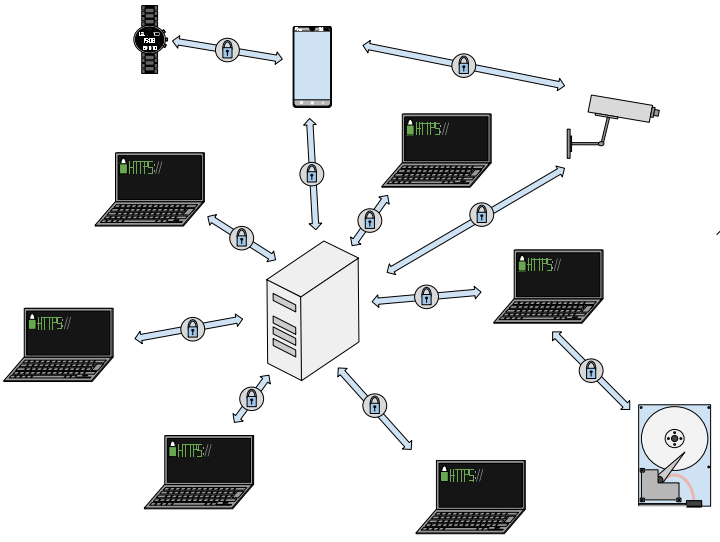


Cryptography: need for speed



Hardware

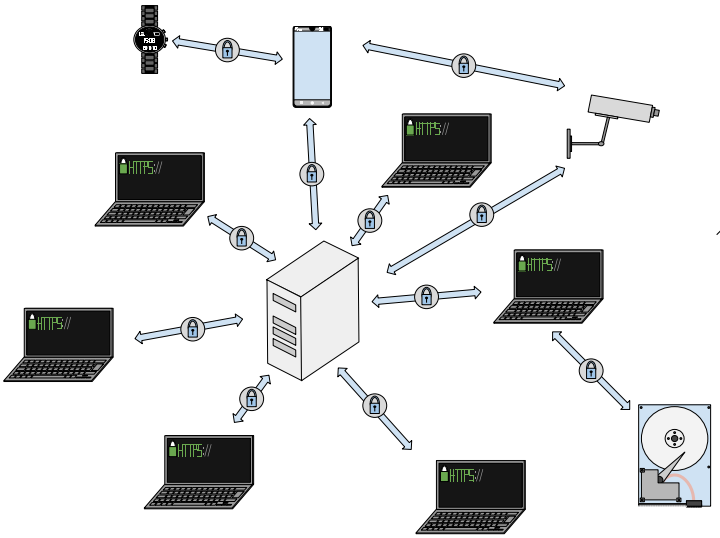
Cryptography: need for speed



Hardware

- cipher-specific

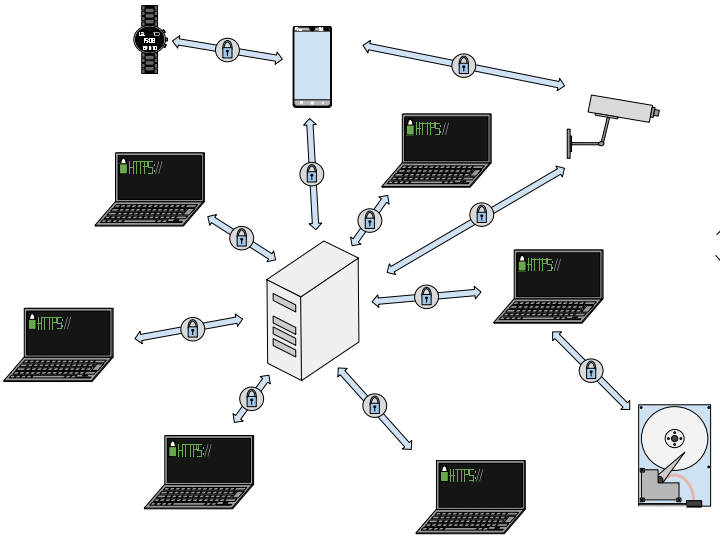
Cryptography: need for speed



Hardware

- cipher-specific
- set in silicon

Cryptography: need for speed

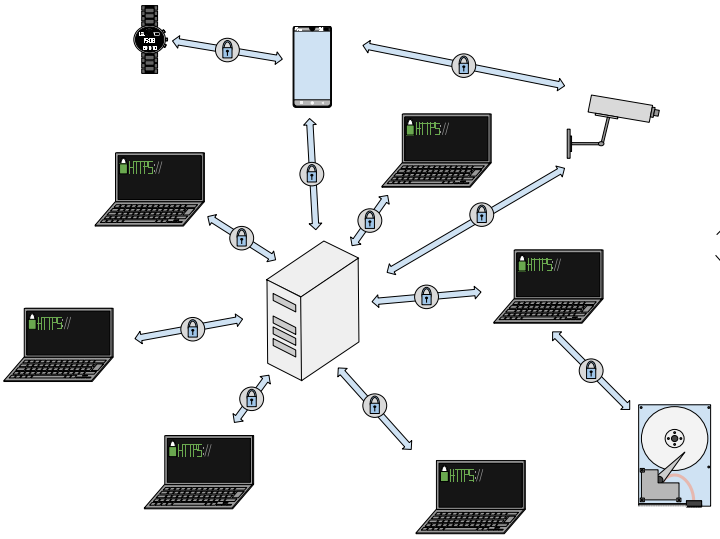


Hardware

- cipher-specific
- set in silicon

Bitslicing (software)

Cryptography: need for speed



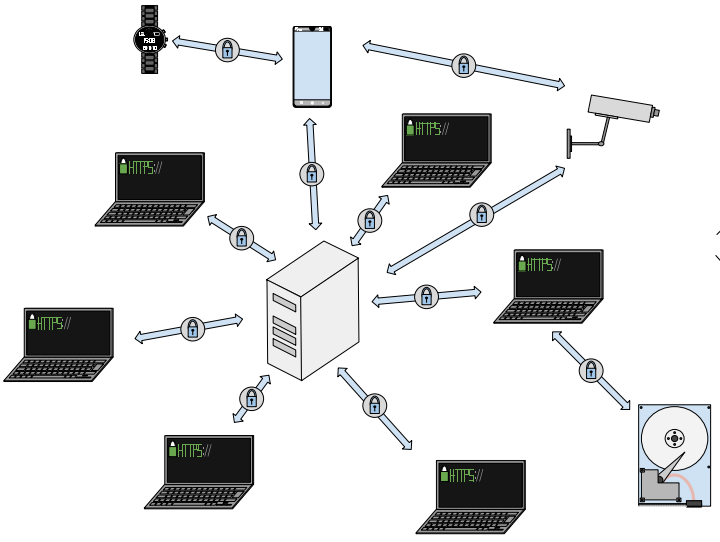
Hardware

- cipher-specific
- set in silicon

Bitslicing (software)

- any cipher

Cryptography: need for speed



Hardware

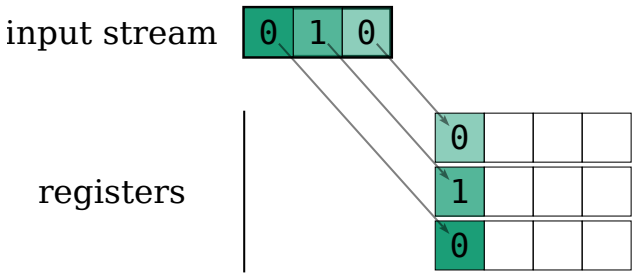
- cipher-specific
- set in silicon

Bitslicing (software)

- any cipher
- fixable

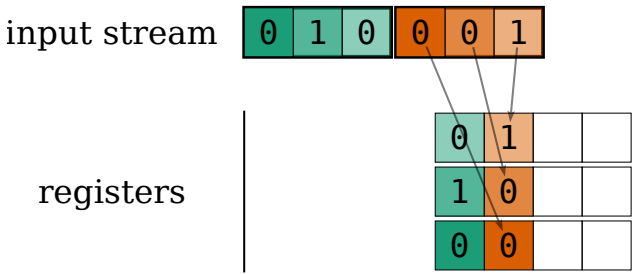
Bitslicing

High-throughput software circuits



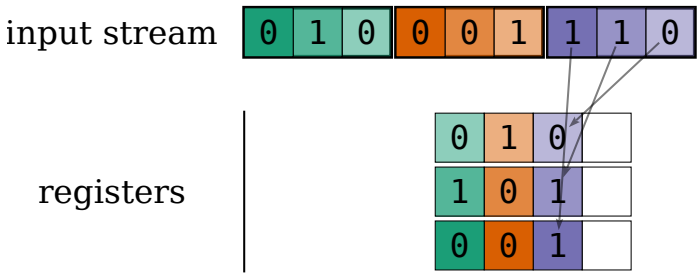
Bitslicing

High-throughput software circuits



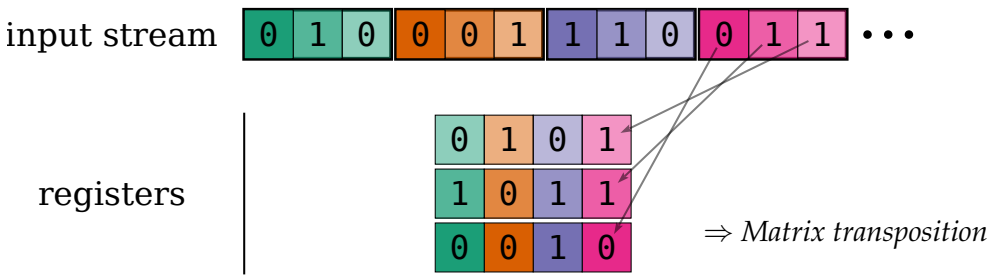
Bitslicing

High-throughput software circuits



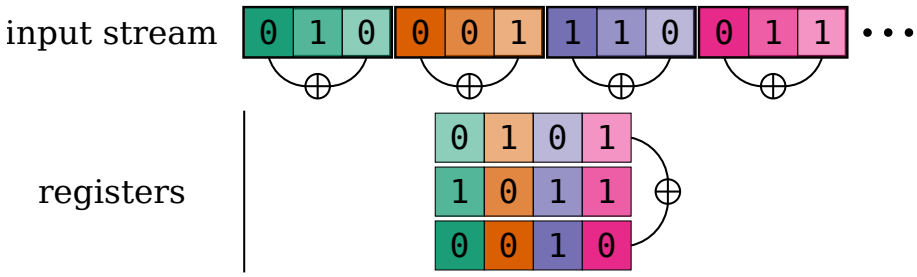
Bitslicing

High-throughput software circuits



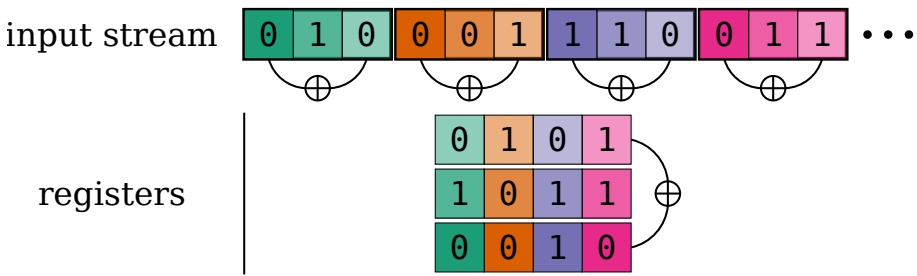
Bitslicing

High-throughput software circuits



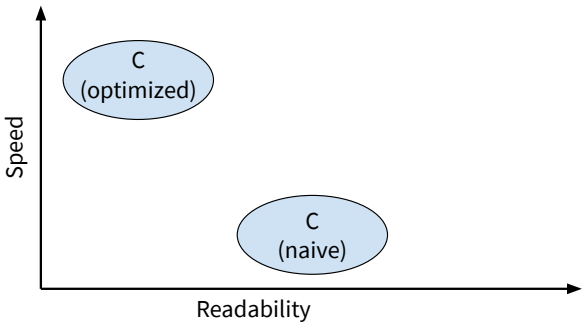
Bitslicing

High-throughput software circuits

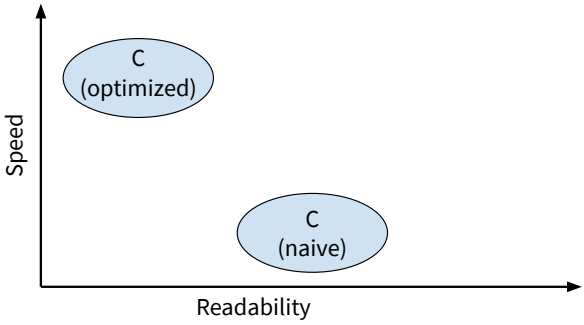


Wider registers ⇒ More parallelism
(SSE, AVX, AVX2, AVX-512, . . .)

Writing bitsliced code

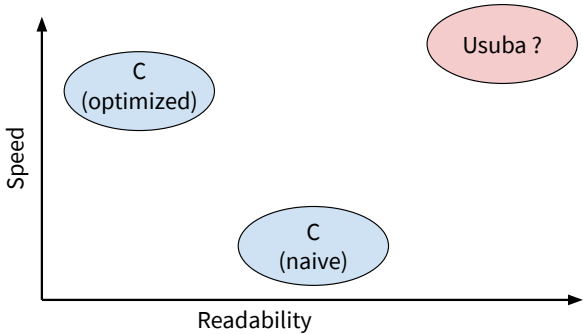


Writing bitsliced code



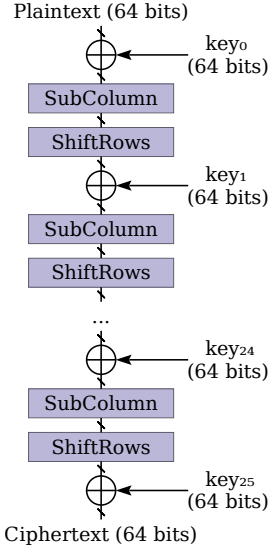

```
s2 (13 ^ k[5], 14 ^ k[41], 15 ^ k[32], 16 ^ k[26], 17 ^ k[17],
  18 ^ k[54], &r12, &r27, &r1, &r17);
s3 (17 ^ k[6], 18 ^ k[3], 19 ^ k[11], 110 ^ k[12], 111 ^ k[27],
  112 ^ k[40], &r23, &r15, &r29, &r5);
s4 (111 ^ k[39], 112 ^ k[33], 113 ^ k[34], 114 ^ k[10], 115 ^ k[18],
  116 ^ k[55], &r25, &r19, &r9, &r0);
s5 (115 ^ k[16], 116 ^ k[7], 117 ^ k[1], 118 ^ k[43], 119 ^ k[31],
  120 ^ k[28], &r7, &r13, &r24, &r2);
s6 (119 ^ k[49], 120 ^ k[9], 121 ^ k[0], 122 ^ k[44], 123 ^ k[15],
  124 ^ k[38], &r3, &r28, &r10, &r18);
s7 (123 ^ k[37], 124 ^ k[45], 125 ^ k[2], 126 ^ k[35], 127 ^ k[22],
  128 ^ k[14], &r31, &r11, &r21, &r6);
s8 (127 ^ k[51], 128 ^ k[23], 129 ^ k[52], 130 ^ k[36], 131 ^ k[42],
  10 ^ k[8], &r4, &r26, &r14, &r20);
s1 (r31 ^ k[39], r0 ^ k[3], r1 ^ k[18], r2 ^ k[27], r3 ^ k[5],
  r4 ^ k[33], &l8, &l16, &l22, &l30);
s2 (r3 ^ k[19], r4 ^ k[55], r5 ^ k[46], r6 ^ k[40], r7 ^ k[6],
  r8 ^ k[11], &l12, &l27, &l1, &l17);
s3 (r7 ^ k[20], r8 ^ k[17], r9 ^ k[25], r10 ^ k[26], r11 ^ k[41],
  r12 ^ k[54], &l23, &l15, &l29, &l5);
s4 (r11 ^ k[53], r12 ^ k[47], r13 ^ k[48], r14 ^ k[24], r15 ^ k[32],
  r16 ^ k[12], &l25, &l19, &l9, &l10);
s5 (r15 ^ k[30], r16 ^ k[21], r17 ^ k[15], r18 ^ k[2], r19 ^ k[45],
  r20 ^ k[42], &l7, &l13, &l24, &l12);
s6 (r19 ^ k[8], r20 ^ k[23], r21 ^ k[14], r22 ^ k[31], r23 ^ k[29],
  r24 ^ k[52], &l3, &l28, &l10, &l18);
s7 (r23 ^ k[51], r24 ^ k[0], r25 ^ k[16], r26 ^ k[49], r27 ^ k[36],
  r28 ^ k[28], &l31, &l11, &l21, &l6);
s8 (r27 ^ k[38], r28 ^ k[37], r29 ^ k[7], r30 ^ k[50], r31 ^ k[1],
  r0 ^ k[22], &l4, &l26, &l14, &l20);
s1 (131 ^ k[53], 10 ^ k[17], 11 ^ k[32], 12 ^ k[41], 13 ^ k[19],
  14 ^ k[47], &r8, &r16, &r22, &r30);
s2 (13 ^ k[33], 14 ^ k[12], 15 ^ k[3], 16 ^ k[54], 17 ^ k[20],
  18 ^ k[25], &r12, &r27, &r1, &r17);
s3 (17 ^ k[34], 18 ^ k[6], 19 ^ k[39], 110 ^ k[40], 111 ^ k[55],
  112 ^ k[11], &r23, &r15, &r29, &r5);
s4 (111 ^ k[10], 112 ^ k[4], 113 ^ k[5], 114 ^ k[13], 115 ^ k[46],
  116 ^ k[26], &r25, &r19, &r9, &r0);
s5 (115 ^ k[44], 116 ^ k[35], 117 ^ k[29], 118 ^ k[16], 119 ^ k[0],
  120 ^ k[1], &r7, &r13, &r24, &r2);
s6 (119 ^ k[22], 120 ^ k[37], 121 ^ k[28], 122 ^ k[45], 123 ^ k[43],
  124 ^ k[7], &r3, &r28, &r10, &r18);
```

Writing bitsliced code

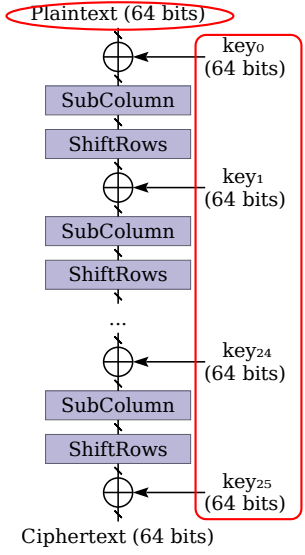



```
s2 (13 ^ k[5], 14 ^ k[41], 15 ^ k[32], 16 ^ k[26], 17 ^ k[17],
  18 ^ k[54], &r12, &r27, &r1, &r17);
s3 (17 ^ k[6], 18 ^ k[3], 19 ^ k[11], 110 ^ k[12], 111 ^ k[27],
  112 ^ k[40], &r23, &r15, &r29, &r5);
s4 (111 ^ k[39], 112 ^ k[33], 113 ^ k[34], 114 ^ k[10], 115 ^ k[18],
  116 ^ k[55], &r25, &r19, &r9, &r0);
s5 (115 ^ k[16], 116 ^ k[7], 117 ^ k[1], 118 ^ k[43], 119 ^ k[31],
  120 ^ k[28], &r7, &r13, &r24, &r2);
s6 (119 ^ k[49], 120 ^ k[9], 121 ^ k[0], 122 ^ k[44], 123 ^ k[15],
  124 ^ k[38], &r3, &r28, &r10, &r18);
s7 (123 ^ k[37], 124 ^ k[45], 125 ^ k[2], 126 ^ k[35], 127 ^ k[22],
  128 ^ k[14], &r31, &r11, &r21, &r6);
s8 (127 ^ k[51], 128 ^ k[23], 129 ^ k[52], 130 ^ k[36], 131 ^ k[42],
  10 ^ k[8], &r4, &r26, &r14, &r20);
s1 (r31 ^ k[39], r0 ^ k[3], r1 ^ k[18], r2 ^ k[27], r3 ^ k[5],
  r4 ^ k[33], &l8, &l16, &l22, &l30);
s2 (r3 ^ k[19], r4 ^ k[55], r5 ^ k[46], r6 ^ k[40], r7 ^ k[6],
  r8 ^ k[11], &l12, &l27, &l1, &l17);
s3 (r7 ^ k[20], r8 ^ k[17], r9 ^ k[25], r10 ^ k[26], r11 ^ k[41],
  r12 ^ k[54], &l23, &l15, &l29, &l5);
s4 (r11 ^ k[53], r12 ^ k[47], r13 ^ k[48], r14 ^ k[24], r15 ^ k[32],
  r16 ^ k[12], &l25, &l19, &l9, &l0);
s5 (r15 ^ k[30], r16 ^ k[21], r17 ^ k[15], r18 ^ k[2], r19 ^ k[45],
  r20 ^ k[42], &l7, &l13, &l24, &l2);
s6 (r19 ^ k[8], r20 ^ k[23], r21 ^ k[14], r22 ^ k[31], r23 ^ k[29],
  r24 ^ k[52], &l3, &l28, &l10, &l18);
s7 (r23 ^ k[51], r24 ^ k[0], r25 ^ k[16], r26 ^ k[49], r27 ^ k[36],
  r28 ^ k[28], &l31, &l11, &l21, &l6);
s8 (r27 ^ k[38], r28 ^ k[37], r29 ^ k[7], r30 ^ k[50], r31 ^ k[1],
  r0 ^ k[22], &l4, &l26, &l14, &l20);
s1 (131 ^ k[53], 10 ^ k[17], 11 ^ k[32], 12 ^ k[41], 13 ^ k[19],
  14 ^ k[47], &r8, &r16, &r22, &r30);
s2 (13 ^ k[33], 14 ^ k[12], 15 ^ k[3], 16 ^ k[54], 17 ^ k[20],
  18 ^ k[25], &r12, &r27, &r1, &r17);
s3 (17 ^ k[34], 18 ^ k[6], 19 ^ k[39], 110 ^ k[40], 111 ^ k[55],
  112 ^ k[11], &r23, &r15, &r29, &r5);
s4 (111 ^ k[10], 112 ^ k[4], 113 ^ k[5], 114 ^ k[13], 115 ^ k[46],
  116 ^ k[26], &r25, &r19, &r9, &r0);
s5 (115 ^ k[44], 116 ^ k[35], 117 ^ k[29], 118 ^ k[16], 119 ^ k[0],
  120 ^ k[1], &r7, &r13, &r24, &r2);
s6 (119 ^ k[22], 120 ^ k[37], 121 ^ k[28], 122 ^ k[45], 123 ^ k[43],
  124 ^ k[7], &r3, &r28, &r10, &r18);
```

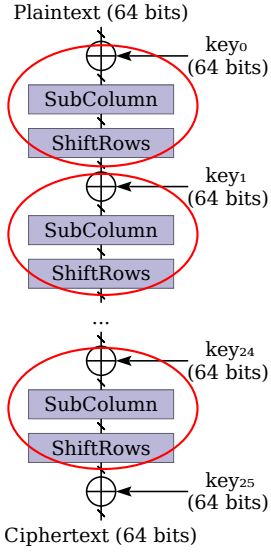
Anatomy of a block cipher: Rectangle



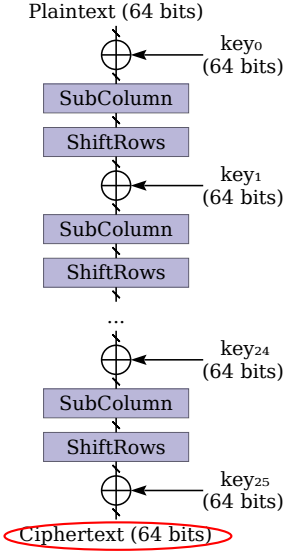
Anatomy of a block cipher: Rectangle



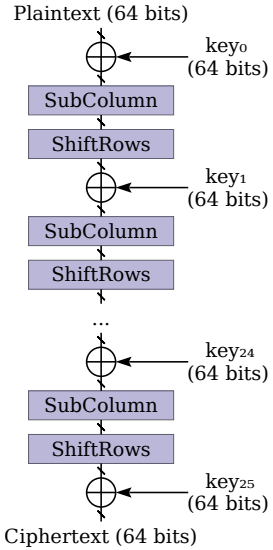
Anatomy of a block cipher: Rectangle



Anatomy of a block cipher: Rectangle



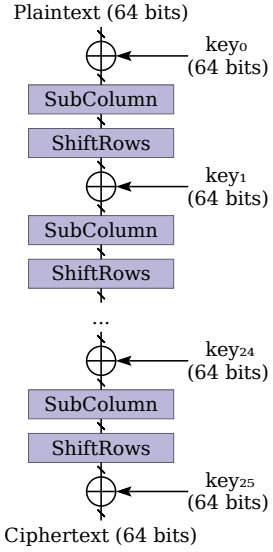
Anatomy of a block cipher: Rectangle



Usuba

```
node Rectangle (plain:b64,
                key  :b64[26])
       returns (cipher:b64)
vars
    state : b64
let
    state = plain;
    forall i in [0,24] {
      state :=
         ShiftRows(
           SubColumn(
             state ^ key[i]
           )
         )
    }
    cipher = state ^ key[25]
tel
```

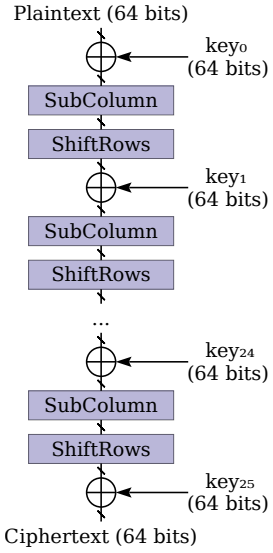
Anatomy of a block cipher: Rectangle



Usuba

```
node Rectangle (plain:b64
                key  :b64[26])
       returns (cipher:b64)
vars
    state : b64
let
    state = plain;
    forall i in [0,24] {
      state :=
         ShiftRows(
           SubColumn(
             state ^ key[i]
           )
         )
    }
    cipher = state ^ key[25]
tel
```

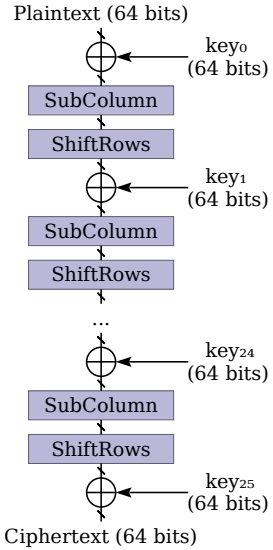
Anatomy of a block cipher: Rectangle



Usuba

```
node Rectangle (plain:b64,
                key  :b64[26])
       returns (cipher:b64)
vars
    state : b64
let
    state = plain;
    forall i in [0,24] {
      state :=
        ShiftRows(
          SubColumn(
            state ^ key[i]
          )
        )
    }
    cipher = state ^ key[25]
tel
```

Anatomy of a block cipher: Rectangle

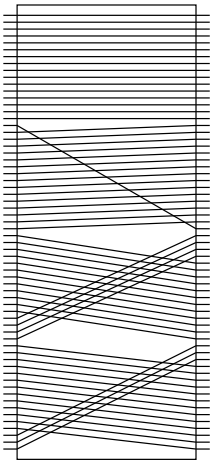


Usuba

```
node Rectangle (plain:b64,
                key  :b64[26])
       returns (cipher:b64)
vars
    state : b64
let
    state = plain;
    forall i in [0,24] {
      state :=
         ShiftRows(
           SubColumn(
             state^key[i]
           )
         )
    }
    cipher = state^key[25]
tel
```

Anatomy of a block cipher: Rectangle

ShiftRows



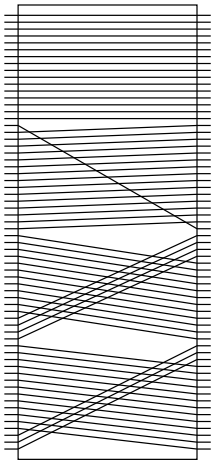
```
void ShiftRows(int a[64], int b[64]) {
  b[0]  = a[0];
  b[1]  = a[1];
  b[2]  = a[2];
  b[3]  = a[3];
  b[4]  = a[4];
  b[5]  = a[5];
  ...
  b[59] = a[56];
  b[60] = a[57];
  b[61] = a[58];
  b[62] = a[59];
  b[63] = a[60];
}
```

Anatomy of a block cipher: Rectangle

ShiftRows

Copy propagation

```
a = b
x = f(a)
```



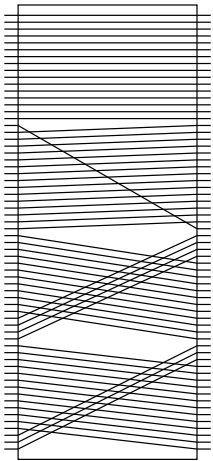
```
void ShiftRows(int a[64], int b[64]) {
  b[0]  = a[0];
  b[1]  = a[1];
  b[2]  = a[2];
  b[3]  = a[3];
  b[4]  = a[4];
  b[5]  = a[5];
  ...
  b[59] = a[56];
  b[60] = a[57];
  b[61] = a[58];
  b[62] = a[59];
  b[63] = a[60];
}
```

Anatomy of a block cipher: Rectangle

ShiftRows

Copy propagation

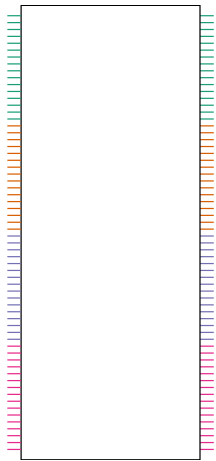
```
a = b                    x = f(b)
x = f(a)    ⟶
```



```
void ShiftRows(int a[64], int b[64]) {
  b[0]  = a[0];
  b[1]  = a[1];
  b[2]  = a[2];
  b[3]  = a[3];
  b[4]  = a[4];
  b[5]  = a[5];
  ...
  b[59] = a[56];
  b[60] = a[57];
  b[61] = a[58];
  b[62] = a[59];
  b[63] = a[60];
}
```

Anatomy of a block cipher: Rectangle

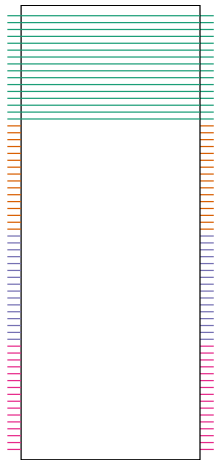
ShiftRows



```
node ShiftRows (input:u16x4)
   returns     (out:u16x4)
```

Anatomy of a block cipher: Rectangle

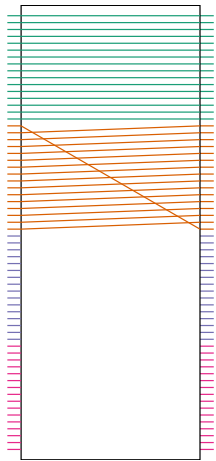
ShiftRows



```
node ShiftRows (input:u16x4)
   returns      (out:u16x4)
let
    out[0] = input[0];



tel
```

Anatomy of a block cipher: Rectangle

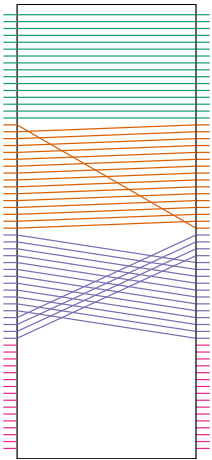
ShiftRows



```
node ShiftRows (input:u16x4)
   returns      (out:u16x4)
let
    out[0] = input[0];
    out[1] = input[1] <<< 1;


tel
```

Anatomy of a block cipher: Rectangle

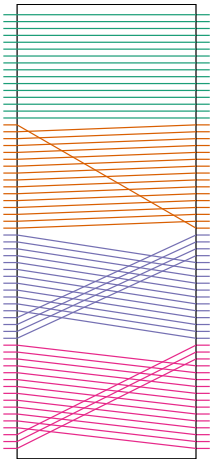
ShiftRows



```
node ShiftRows (input:u16x4)
   returns      (out:u16x4)
let
    out[0] = input[0];
    out[1] = input[1] <<< 1;
    out[2] = input[2] <<< 12;

tel
```

Anatomy of a block cipher: Rectangle

ShiftRows



```
node ShiftRows (input:u16x4)
   returns     (out:u16x4)
let
    out[0] = input[0];
    out[1] = input[1] <<< 1;
    out[2] = input[2] <<< 12;
    out[3] = input[3] <<< 13
tel
```

Anatomy of a block cipher: Rectangle

SubColumn

The S-box used in RECTANGLE is a 4-bit to 4-bit S-box $S : F_2^4 \rightarrow F_2^4$. The action of this S-box in hexadecimal notation is given by the following table.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x)$	6	5	C	A	1	E	7	9	B	0	3	D	8	F	4	2

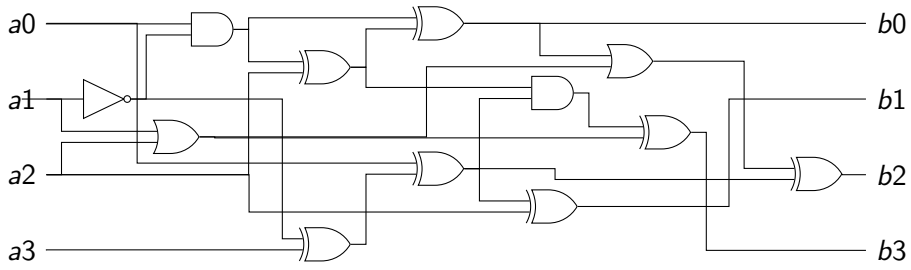
Caution: lookup tables are vulnerable to **cache-timing attacks**!

Anatomy of a block cipher: Rectangle

SubColumn

The S-box used in RECTANGLE is a 4-bit to 4-bit S-box $S : F_2^4 \rightarrow F_2^4$. The action of this S-box in hexadecimal notation is given by the following table.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x)$	6	5	C	A	1	E	7	9	B	0	3	D	8	F	4	2



Anatomy of a block cipher: Rectangle

SubColumn

The S-box used in RECTANGLE is a 4-bit to 4-bit S-box $S : F_2^4 \rightarrow F_2^4$. The action of this S-box in hexadecimal notation is given by the following table.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x)$	6	5	C	A	1	E	7	9	B	0	3	D	8	F	4	2

```
void SubColumn(int *a0, int *a1,
               int *a2, int *a3) {
    int t1, t2, t3, t5, t6, t8, t9, t11;
    int a0_ = *a0;       int a1_ = *a1;
    t1 = ~*a1;           t2 = *a0 & t1;       t3 = *a2 ^ *a3;
    *a0 = t2 ^ t3;       t5 = *a3 | t1;       t6 = a0_ ^ t5;
    *a1 = *a2 ^ t6;      t8 = a1_ ^ *a2;      t9 = t3 & t6;
    *a3 = t8 ^ t9;       t11 = *a0 | t8;      *a2 = t6 ^ t11;
}
```

Anatomy of a block cipher: Rectangle

SubColumn

The S-box used in RECTANGLE is a 4-bit to 4-bit S-box $S : F_2^4 \rightarrow F_2^4$. The action of this S-box in hexadecimal notation is given by the following table.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x)$	6	5	C	A	1	E	7	9	B	0	3	D	8	F	4	2

```
table SubColumn (a:v4)
        returns (b:v4) {
  6,  5, 12, 10, 1, 14, 7, 9,
  11, 0,  3, 13, 8, 15, 4, 2
}
```

Anatomy of a block cipher: Rectangle

SubColumn

The S-box used in RECTANGLE is a 4-bit to 4-bit S-box $S : F_2^4 \rightarrow F_2^4$. The action of this S-box in hexadecimal notation is given by the following table.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x)$	6	5	C	A	1	E	7	9	B	0	3	D	8	F	4	2

```
table SubColumn (a:v4)
        returns (b:v4) {
  6,  5, 12, 10, 1, 14, 7, 9,
  11, 0,  3, 13, 8, 15, 4, 2
}
```

```
node SubColumn (a:v4)
       returns (b:v4)
let
    ...
tel
```

Anatomy of a block cipher: Rectangle

SubColumn

The S-box used in RECTANGLE is a 4-bit to 4-bit S-box $S : F_2^4 \rightarrow F_2^4$. The action of this S-box in hexadecimal notation is given by the following table.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x)$	6	5	C	A	1	E	7	9	B	0	3	D	8	F	4	2

```
table SubColumn (a:v4)
        returns (b:v4) {
  6,  5, 12, 10, 1, 14, 7, 9,
  11, 0,  3, 13, 8, 15, 4, 2
}
```

database

```
node SubColumn (a:v4)
       returns (b:v4)
let
    ...
tel
```

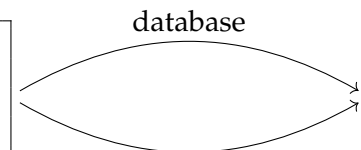
Anatomy of a block cipher: Rectangle

SubColumn

The S-box used in RECTANGLE is a 4-bit to 4-bit S-box $S : F_2^4 \rightarrow F_2^4$. The action of this S-box in hexadecimal notation is given by the following table.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x)$	6	5	C	A	1	E	7	9	B	0	3	D	8	F	4	2

```
table SubColumn (a:v4)
        returns (b:v4) {
  6,  5, 12, 10, 1, 14, 7, 9,
  11, 0,  3, 13, 8, 15, 4, 2
}
```



[Pornin01]

```
node SubColumn (a:v4)
       returns (b:v4)
let
    ...
tel
```

Anatomy of a block cipher

The whole cipher

```
node ShiftRows (input:u16x4)
       returns (out:u16x4)
let
    out[0] = input[0];
    out[1] = input[1] <<< 1;
    out[2] = input[2] <<< 12;
    out[3] = input[3] <<< 13
tel
```

```
table SubColumn (input:v4)
       returns  (out:v4) {
    6, 5, 12, 10, 1, 14, 7, 9,
    11, 0, 3, 13, 8, 15, 4, 2
}
```

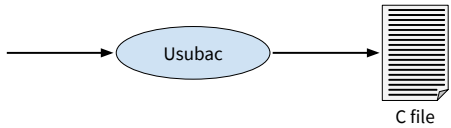
```
node Rectangle (plain:u16x4,
                key  :u16x4[26])
       returns (cipher:u16x4)
vars
    state : u16x4
let
    state = plain;
    forall i in [0,24] {
      state :=
         ShiftRows(
           SubColumn(
             state ^ key[i]
           )
         )
    }
    cipher = state ^ key[25]
tel
```

Usuba

```
node ShiftRows (input:u16x4)
       returns (out:u16x4)
let
    out[0] = input[0];
    out[1] = input[1] <<< 1;
    out[2] = input[2] <<< 12;
    out[3] = input[3] <<< 13
tel
```

```
table SubColumn (input:v4)
       returns  (out:v4) {
    6, 5, 12, 10, 1, 14, 7, 9,
    11, 0, 3, 13, 8, 15, 4, 2
}
```

```
node Rectangle (plain:u16x4,
                key  :u16x4[26])
       returns (cipher:u16x4)
vars
    state : u16x4
let
    state = plain;
    forall i in [0,24] {
      state :=
          ShiftRows(
            SubColumn(
              state ^ key[i]
            )
          )
    }
    cipher = state ^ key[25]
tel
```

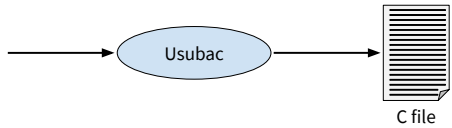


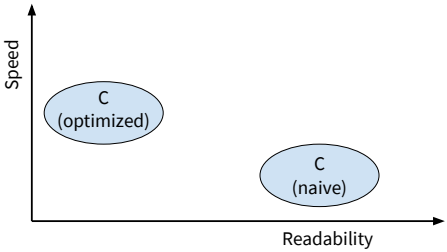
Usuba

```
node ShiftRows (input:u16x4)
       returns (out:u16x4)
let
    out[0] = input[0];
    out[1] = input[1] <<< 1;
    out[2] = input[2] <<< 12;
    out[3] = input[3] <<< 13
tel
```

```
table SubColumn (input:v4)
       returns  (out:v4) {
    6, 5, 12, 10, 1, 14, 7, 9,
    11, 0, 3, 13, 8, 15, 4, 2
}
```

```
node Rectangle (plain:u16x4,
                key  :u16x4[26])
       returns (cipher:u16x4)
vars
    state : u16x4
let
    state = plain;
    forall i in [0,24] {
      state :=
          ShiftRows(
            SubColumn(
              state ^ key[i]
            )
          )
    }
    cipher = state ^ key[25]
tel
```



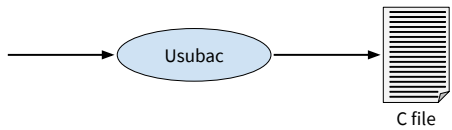


Usuba

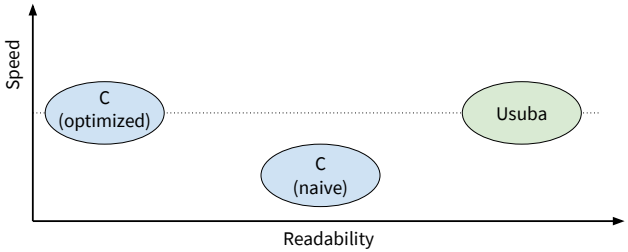
```
node ShiftRows (input:u16x4)
       returns (out:u16x4)
let
    out[0] = input[0];
    out[1] = input[1] <<< 1;
    out[2] = input[2] <<< 12;
    out[3] = input[3] <<< 13
tel
```

```
table SubColumn (input:v4)
       returns  (out:v4) {
    6, 5, 12, 10, 1, 14, 7, 9,
    11, 0, 3, 13, 8, 15, 4, 2
}
```

```
node Rectangle (plain:u16x4,
                key  :u16x4[26])
       returns (cipher:u16x4)
vars
    state : u16x4
let
    state = plain;
    forall i in [0,24] {
      state :=
          ShiftRows(
            SubColumn(
              state ^ key[i]
            )
          )
    }
    cipher = state ^ key[25]
tel
```



- High-level
- Fast

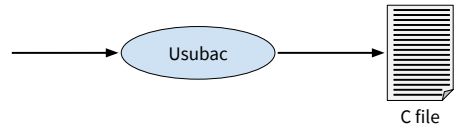


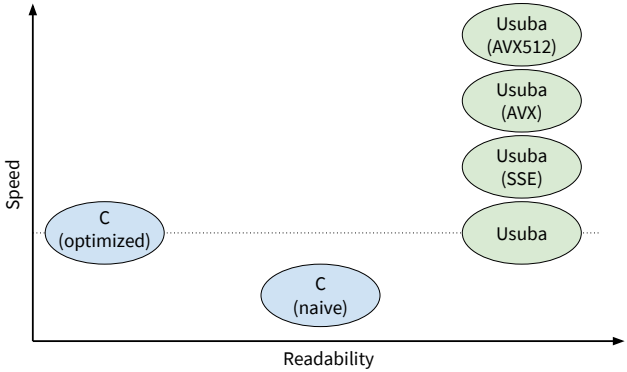
Usuba

```
node ShiftRows (input:u16x4)
       returns (out:u16x4)
let
    out[0] = input[0];
    out[1] = input[1] <<< 1;
    out[2] = input[2] <<< 12;
    out[3] = input[3] <<< 13
tel
```

```
table SubColumn (input:v4)
       returns  (out:v4) {
    6, 5, 12, 10, 1, 14, 7, 9,
    11, 0, 3, 13, 8, 15, 4, 2
}
```

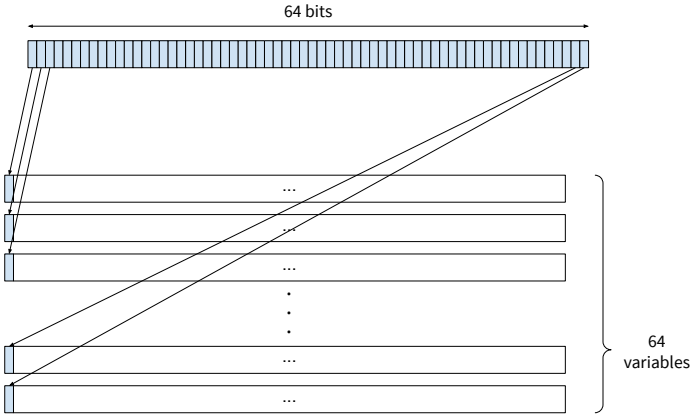
```
node Rectangle (plain:u16x4,
                key  :u16x4[26])
       returns (cipher:u16x4)
vars
    state : u16x4
let
    state = plain;
    forall i in [0,24] {
      state :=
          ShiftRows(
            SubColumn(
              state ^ key[i]
            )
          )
    }
    cipher = state ^ key[25]
tel
```



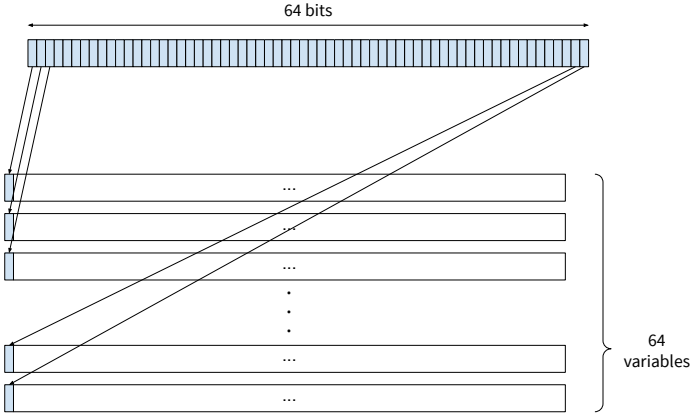


- High-level
- Fast
- Generic

Limitation of bitslicing: spilling

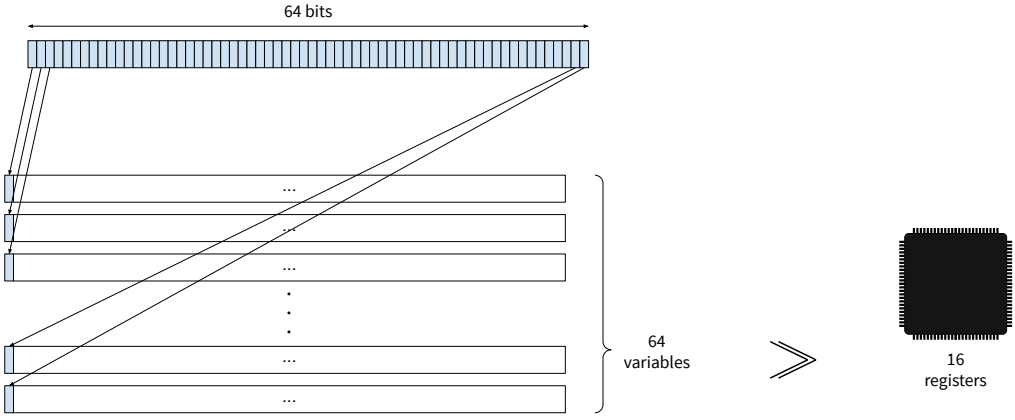


Limitation of bitslicing: spilling

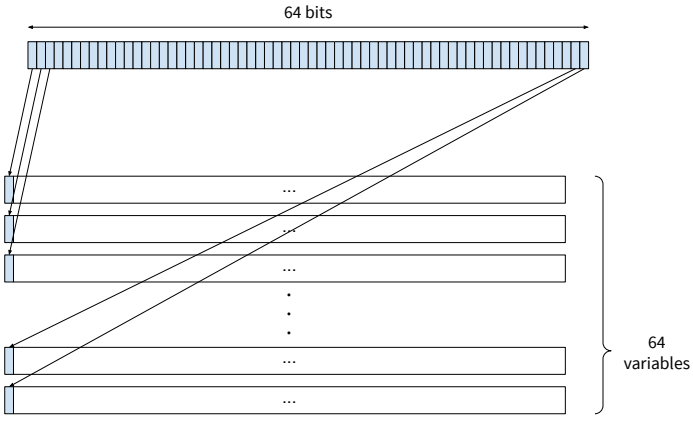


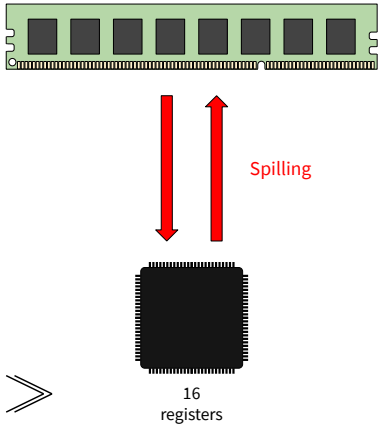
16
registers

Limitation of bitslicing: spilling



Limitation of bitslicing: spilling



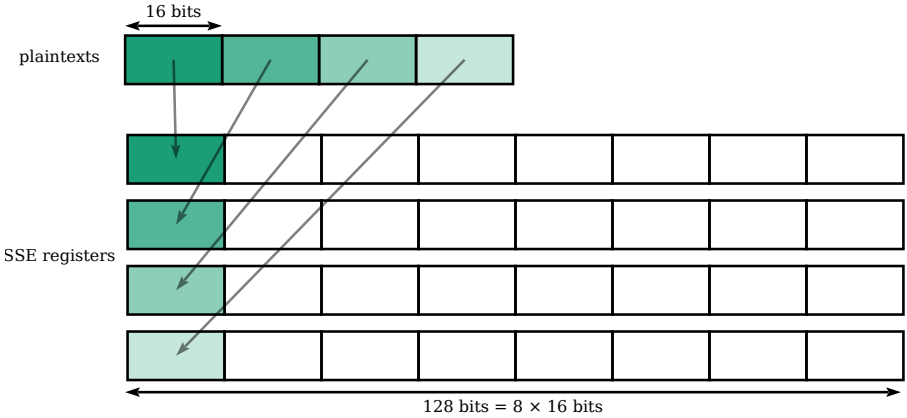


*m*slicing

*v*slicing

ShiftRows in vertical mode

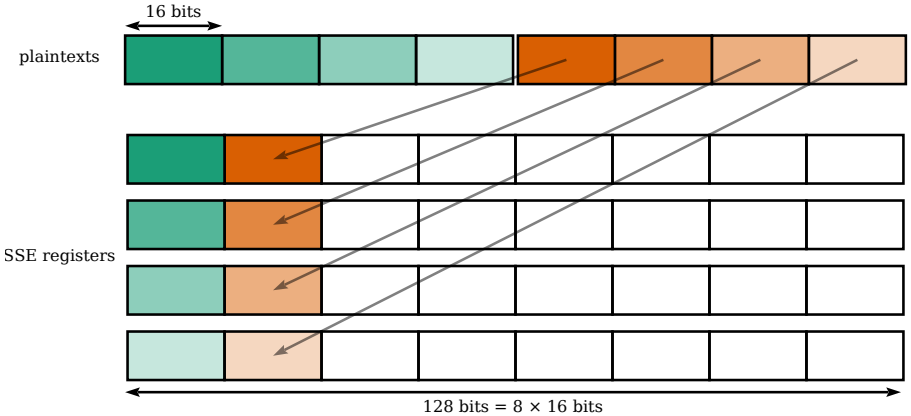
```
node ShiftRows (input:u16x4) : (out:u16x4)
let    out[0] = input[0];
       out[1] = input[1] <<< 1;
       out[2] = input[2] <<< 12;
       out[3] = input[3] <<< 13;        tel
```



*v*slicing

ShiftRows in vertical mode

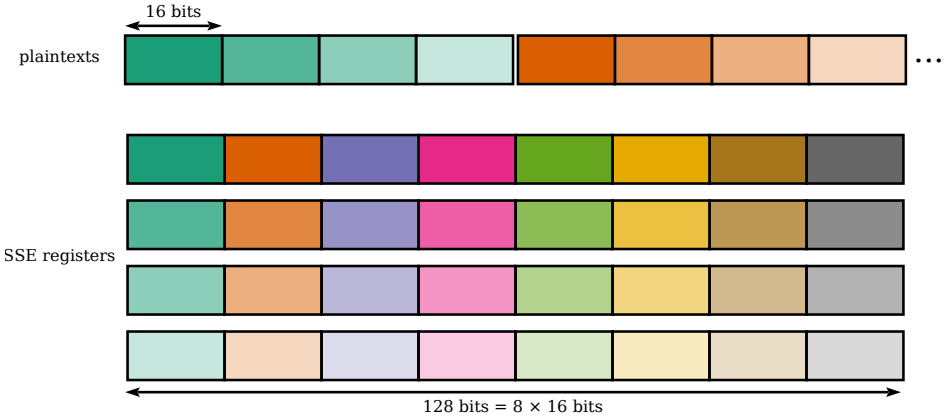
```
node ShiftRows (input:u16x4) : (out:u16x4)
let    out[0] = input[0];
       out[1] = input[1] <<< 1;
       out[2] = input[2] <<< 12;
       out[3] = input[3] <<< 13;        tel
```



*v*slicing

ShiftRows in vertical mode

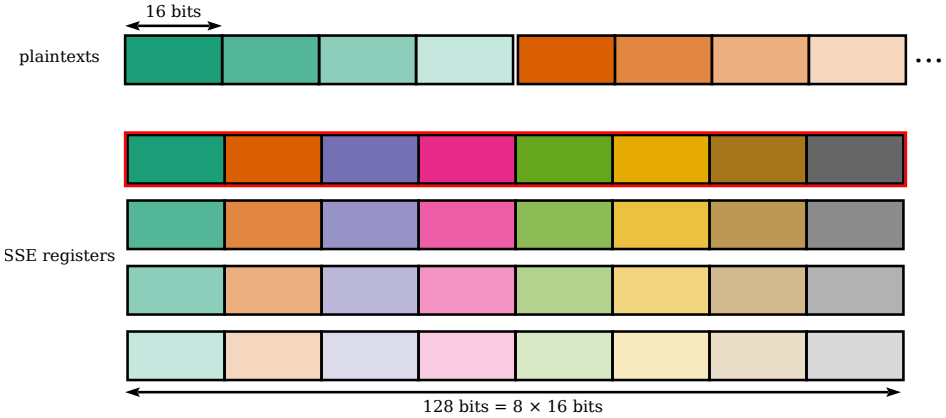
```
node ShiftRows (input:u16x4) : (out:u16x4)
let    out[0] = input[0];
       out[1] = input[1] <<< 1;
       out[2] = input[2] <<< 12;
       out[3] = input[3] <<< 13;        tel
```



*v*slicing

ShiftRows in vertical mode

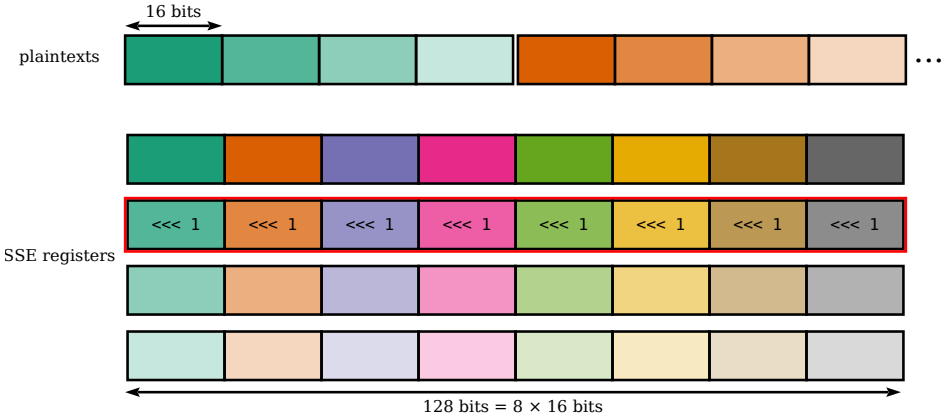
```
node ShiftRows (input:u16x4) : (out:u16x4)
let    out[0] = input[0];
       out[1] = input[1] <<< 1;
       out[2] = input[2] <<< 12;
       out[3] = input[3] <<< 13;        tel
```



*v*slicing

ShiftRows in vertical mode

```
node ShiftRows (input:u16x4) : (out:u16x4)
let    out[0] = input[0];
       out[1] = input[1] <<< 1;
       out[2] = input[2] <<< 12;
       out[3] = input[3] <<< 13;        tel
```

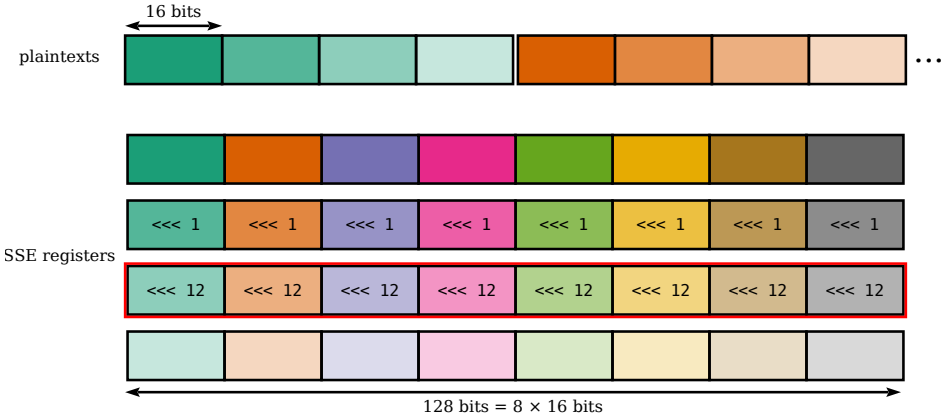


SSE instructions: `_mm_slli_epi16`, `_mm_or_si128`, `__mm_srli_epi16`

*v*slicing

ShiftRows in vertical mode

```
node ShiftRows (input:u16x4) : (out:u16x4)
let    out[0] = input[0];
       out[1] = input[1] <<< 1;
       out[2] = input[2] <<< 12;
       out[3] = input[3] <<< 13;        tel
```

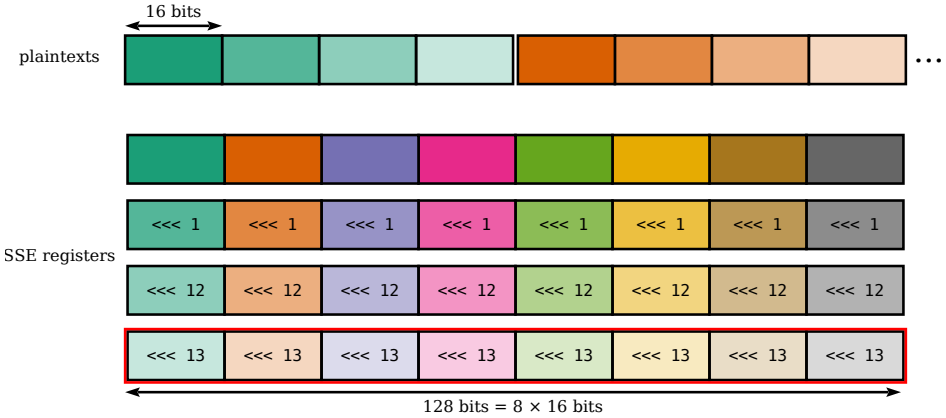


SSE instructions: `_mm_slli_epi16`, `_mm_or_si128`, `__mm_srli_epi16`

*v*slicing

ShiftRows in vertical mode

```
node ShiftRows (input:u16x4) : (out:u16x4)
let    out[0] = input[0];
       out[1] = input[1] <<< 1;
       out[2] = input[2] <<< 12;
       out[3] = input[3] <<< 13;        tel
```

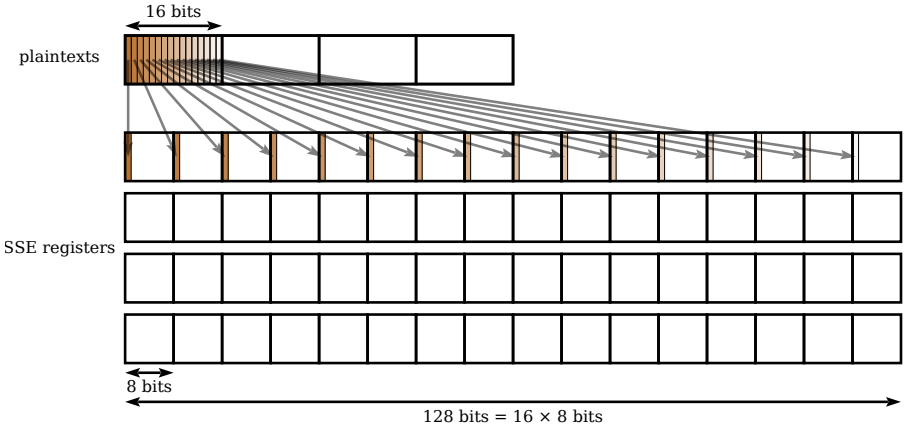


SSE instructions: `_mm_slli_epi16`, `_mm_or_si128`, `__mm_srli_epi16`

*h*slicing [Kasper09]

ShiftRows in horizontal mode

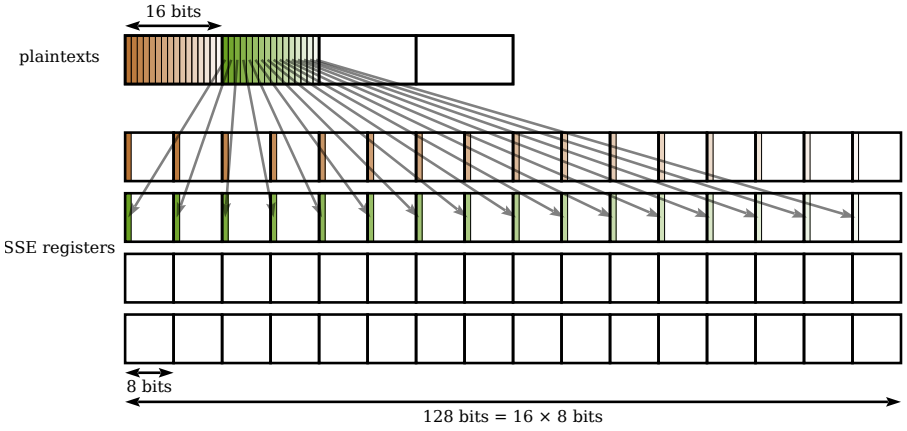
```
node ShiftRows (input:u16x4) : (out:u16x4)
let    out[0] = input[0];
       out[1] = input[1] <<< 1;
       out[2] = input[2] <<< 12;
       out[3] = input[3] <<< 13;        tel
```



*h*slicing [Kasper09]

ShiftRows in horizontal mode

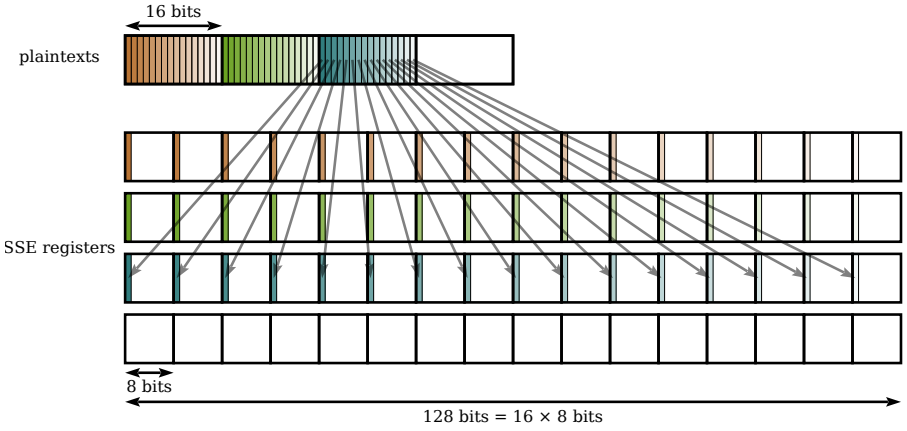
```
node ShiftRows (input:u16x4) : (out:u16x4)
let    out[0] = input[0];
       out[1] = input[1] <<< 1;
       out[2] = input[2] <<< 12;
       out[3] = input[3] <<< 13;        tel
```



*h*slicing [Kasper09]

ShiftRows in horizontal mode

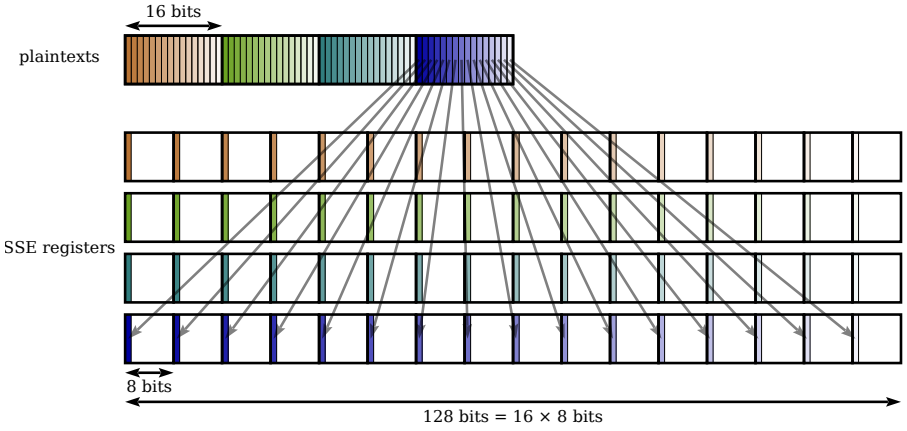
```
node ShiftRows (input:u16x4) : (out:u16x4)
let    out[0] = input[0];
       out[1] = input[1] <<< 1;
       out[2] = input[2] <<< 12;
       out[3] = input[3] <<< 13;        tel
```



*h*slicing [Kasper09]

ShiftRows in horizontal mode

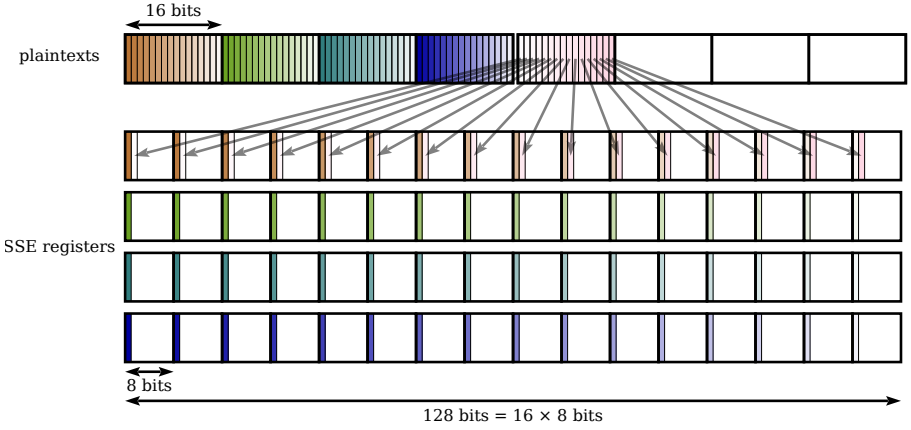
```
node ShiftRows (input:u16x4) : (out:u16x4)
let    out[0] = input[0];
       out[1] = input[1] <<< 1;
       out[2] = input[2] <<< 12;
       out[3] = input[3] <<< 13;        tel
```



*h*slicing [Kasper09]

ShiftRows in horizontal mode

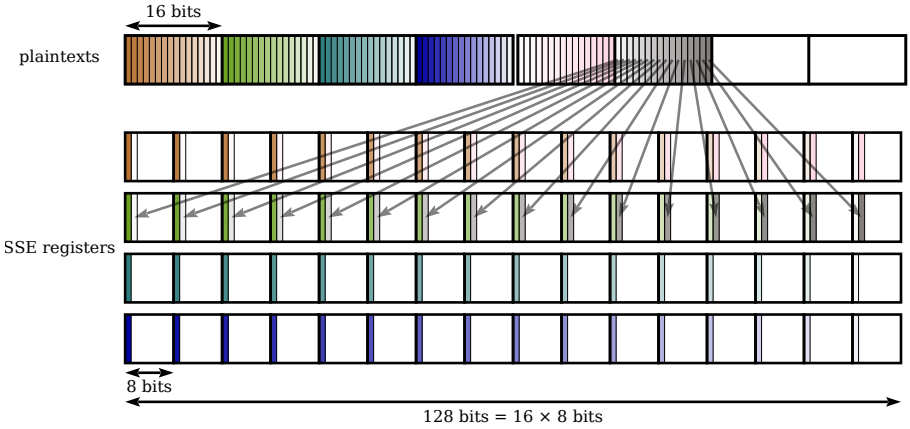
```
node ShiftRows (input:u16x4) : (out:u16x4)
let    out[0] = input[0];
       out[1] = input[1] <<< 1;
       out[2] = input[2] <<< 12;
       out[3] = input[3] <<< 13;        tel
```



*h*slicing [Kasper09]

ShiftRows in horizontal mode

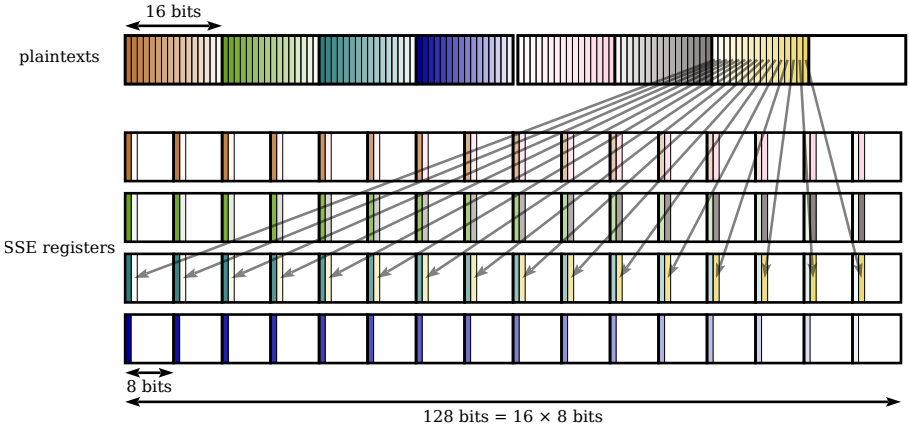
```
node ShiftRows (input:u16x4) : (out:u16x4)
let    out[0] = input[0];
       out[1] = input[1] <<< 1;
       out[2] = input[2] <<< 12;
       out[3] = input[3] <<< 13;        tel
```



*h*slicing [Kasper09]

ShiftRows in horizontal mode

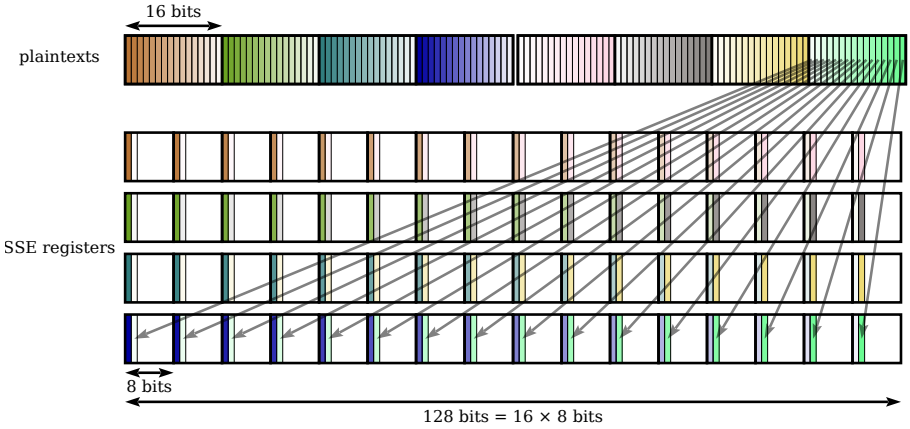
```
node ShiftRows (input:u16x4) : (out:u16x4)
let    out[0] = input[0];
       out[1] = input[1] <<< 1;
       out[2] = input[2] <<< 12;
       out[3] = input[3] <<< 13;        tel
```



*h*slicing [Kasper09]

ShiftRows in horizontal mode

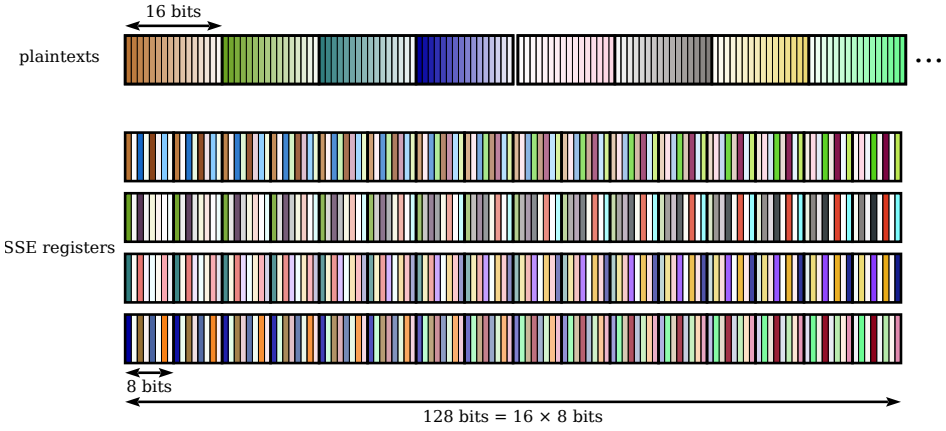
```
node ShiftRows (input:u16x4) : (out:u16x4)
let    out[0] = input[0];
       out[1] = input[1] <<< 1;
       out[2] = input[2] <<< 12;
       out[3] = input[3] <<< 13;        tel
```



*h*slicing [Kasper09]

ShiftRows in horizontal mode

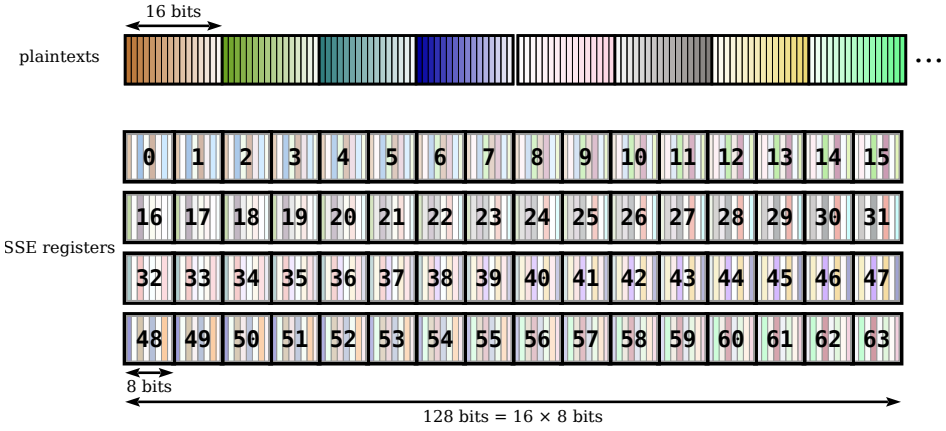
```
node ShiftRows (input:u16x4) : (out:u16x4)
let    out[0] = input[0];
       out[1] = input[1] <<< 1;
       out[2] = input[2] <<< 12;
       out[3] = input[3] <<< 13;        tel
```



*h*slicing [Kasper09]

ShiftRows in horizontal mode

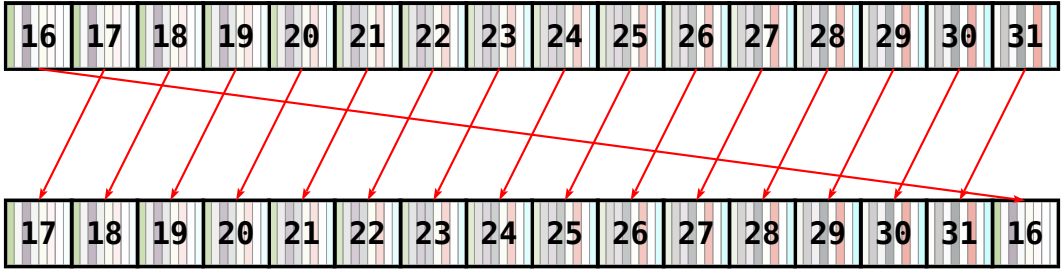
```
node ShiftRows (input:u16x4) : (out:u16x4)
let    out[0] = input[0];
       out[1] = input[1] <<< 1;
       out[2] = input[2] <<< 12;
       out[3] = input[3] <<< 13;        tel
```



*h*slicing [Kasper09]

ShiftRows in horizontal mode

```
node ShiftRows (input:u16x4) : (out:u16x4)
let    out[0] = input[0];
       out[1] = input[1] <<< 1;
       out[2] = input[2] <<< 12;
       out[3] = input[3] <<< 13;        tel
```

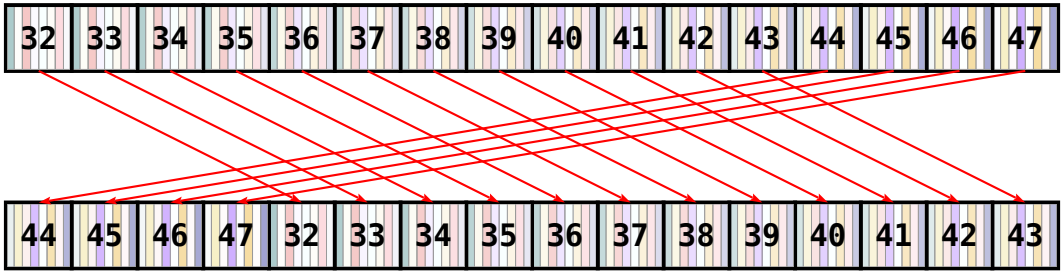


SSE instruction: `__m128i _mm_shuffle_epi8 (__m128i input, __m128i pattern)`

*h*slicing [Kasper09]

ShiftRows in horizontal mode

```
node ShiftRows (input:u16x4) : (out:u16x4)
let    out[0] = input[0];
       out[1] = input[1] <<< 1;
       out[2] = input[2] <<< 12;
       out[3] = input[3] <<< 13;        tel
```

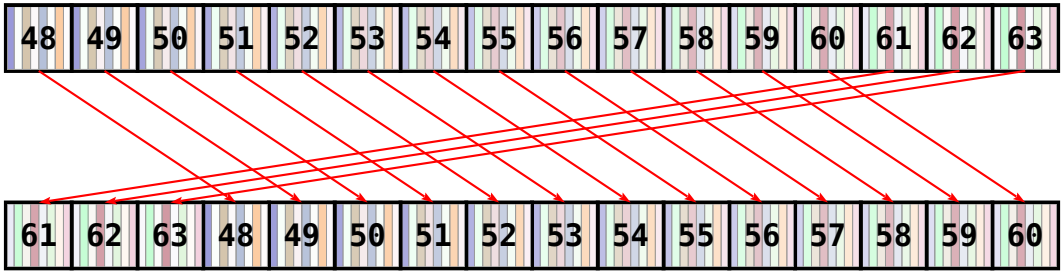


SSE instruction: `__m128i _mm_shuffle_epi8 (__m128i input, __m128i pattern)`

*h*slicing [Kasper09]

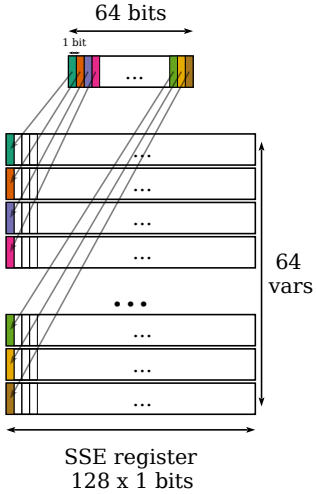
ShiftRows in horizontal mode

```
node ShiftRows (input:u16x4) : (out:u16x4)
let    out[0] = input[0];
       out[1] = input[1] <<< 1;
       out[2] = input[2] <<< 12;
       out[3] = input[3] <<< 13;        tel
```



SSE instruction: `__m128i _mm_shuffle_epi8 (__m128i input, __m128i pattern)`

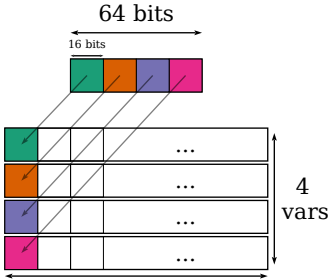
Slicing in Usuba



bitslicing

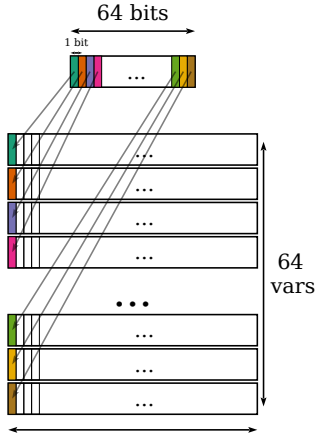
b64

Slicing in Usuba



*v*slicing

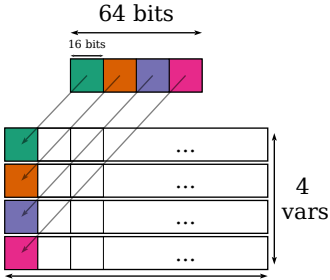
`u`$_V$`16x4`



bitslicing

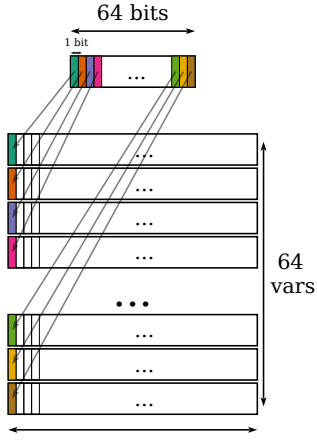
`b64`

Slicing in Usuba



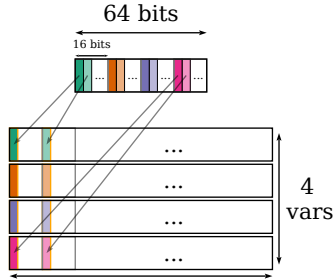
*v*slicing

`u`$_V$`16x4`



bitslicing

`b64`



*h*slicing

`u`$_H$`16x4`

Usuba: polymorphism

```
node ShiftRows (input:u16x4)
     returns (output:u16x4)
let
    out[0] = input[0];
    out[1] = input[1] <<< 1;
    out[2] = input[2] <<< 12;
    out[3] = input[3] <<< 13
tel
```

Usuba: polymorphism

```
node ShiftRows (input:u16x4)
     returns (output:u16x4)
let
    out[0] = input[0];
    out[1] = input[1] <<< 1;
    out[2] = input[2] <<< 12;
    out[3] = input[3] <<< 13
tel
```

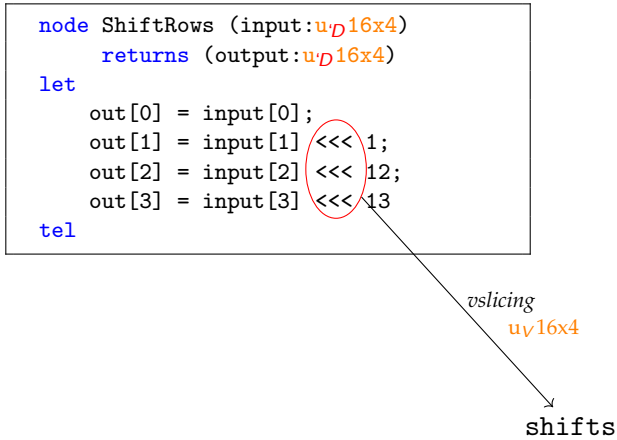
Usuba: polymorphism

```
node ShiftRows (input:u'D16x4)
     returns (output:u'D16x4)
let
    out[0] = input[0];
    out[1] = input[1] <<< 1;
    out[2] = input[2] <<< 12;
    out[3] = input[3] <<< 13
tel
```

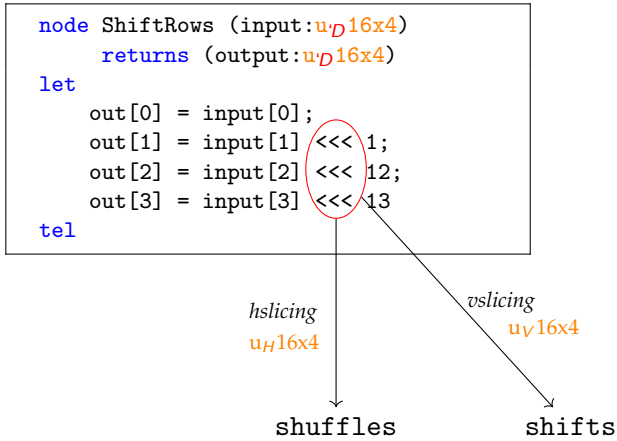
Usuba: polymorphism

```
node ShiftRows (input:u'D16x4)
     returns (output:u'D16x4)
let
    out[0] = input[0];
    out[1] = input[1] <<< 1;
    out[2] = input[2] <<< 12;
    out[3] = input[3] <<< 13
tel
```

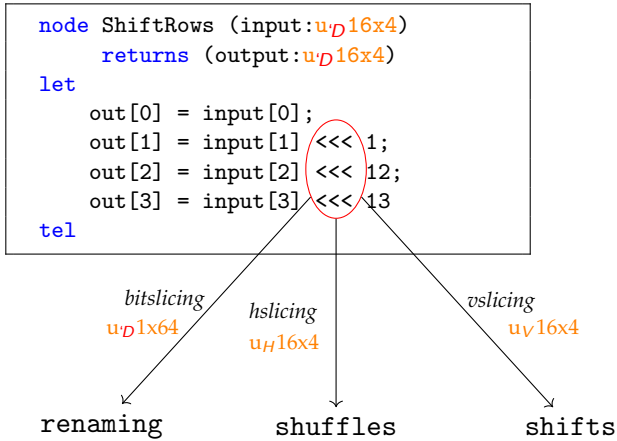
Usuba: polymorphism



Usuba: polymorphism



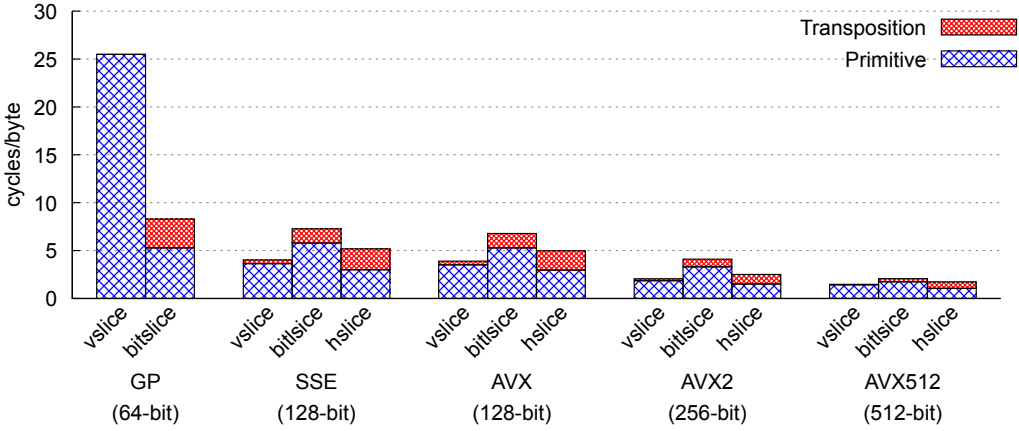
Usuba: polymorphism



Monomorphization

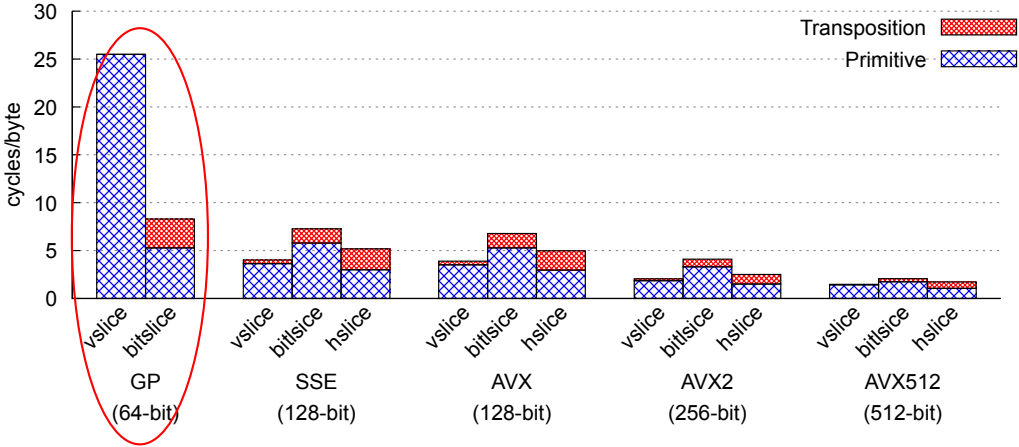
```
node Rectangle (plain : u16x4, key : u16x4[26]) returns (cipher : u16x4)
```

Monomorphization



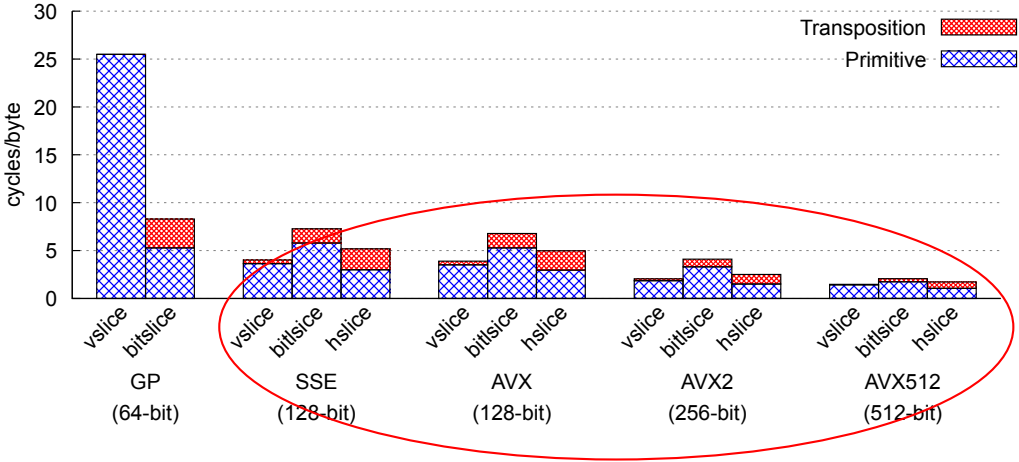

```
node Rectangle (plain : u16x4, key : u16x4[26]) returns (cipher : u16x4)
```

Monomorphization



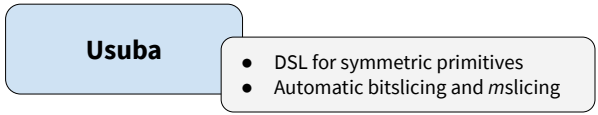

```
node Rectangle (plain : u16x4, key : u16x4[26]) returns (cipher : u16x4)
```

Monomorphization

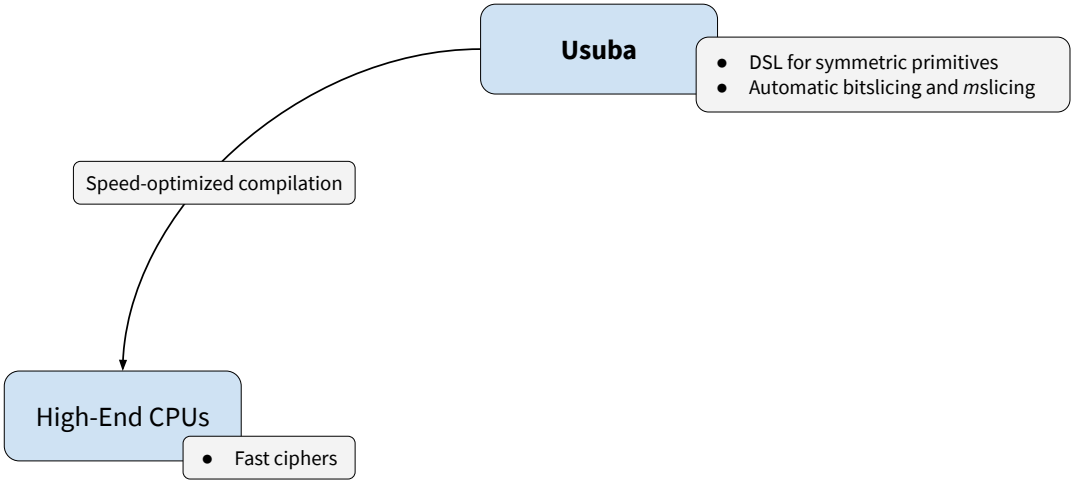



```
node Rectangle (plain : u16x4, key : u16x4[26]) returns (cipher : u16x4)
```

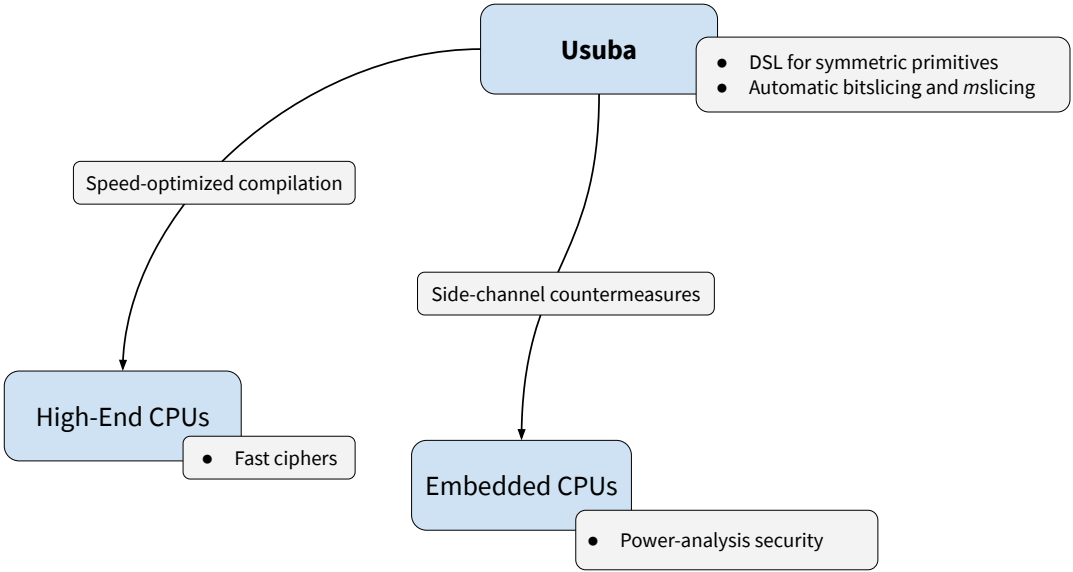
Usuba: what now?



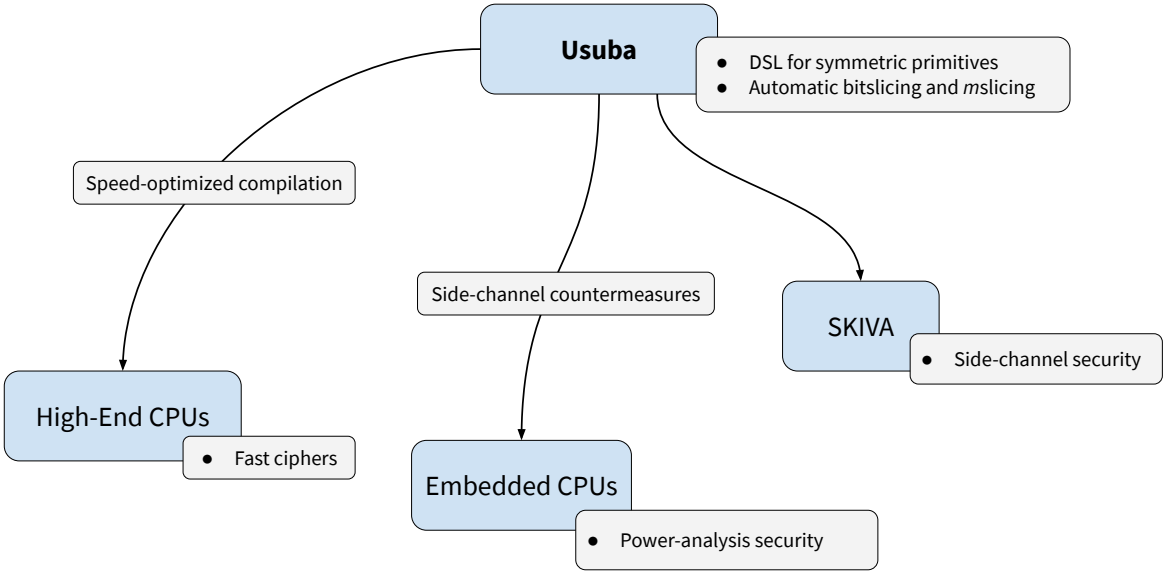
Usuba: what now?



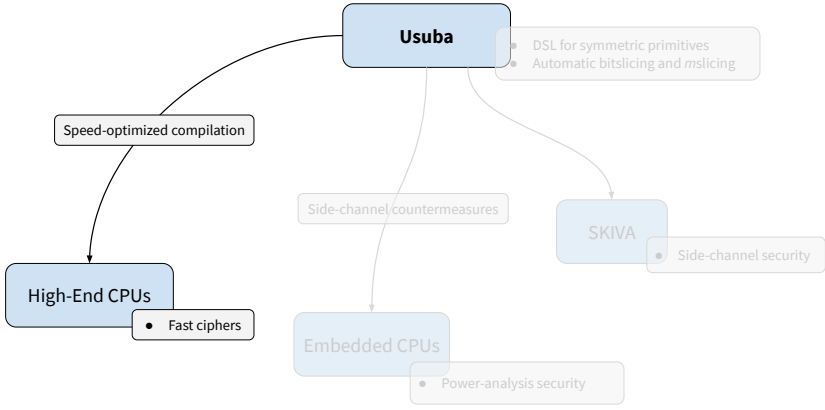
Usuba: what now?



Usuba: what now?



Compiling for High-end SIMD Architectures

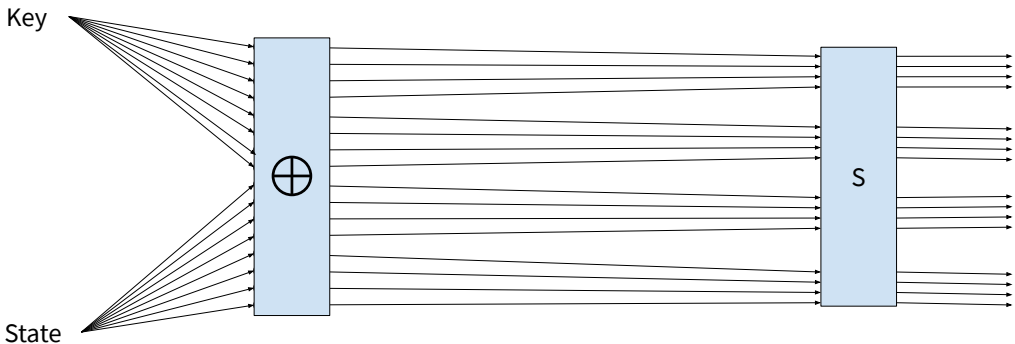


Scheduling bitsliced code

Reducing register pressure

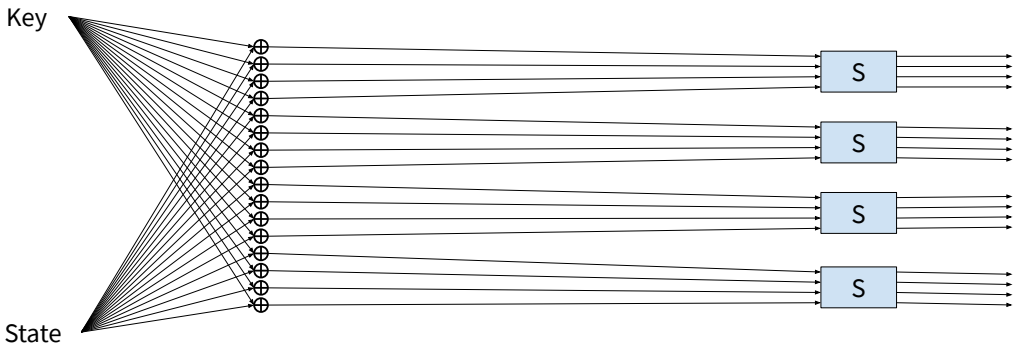
Scheduling bitsliced code

Reducing register pressure



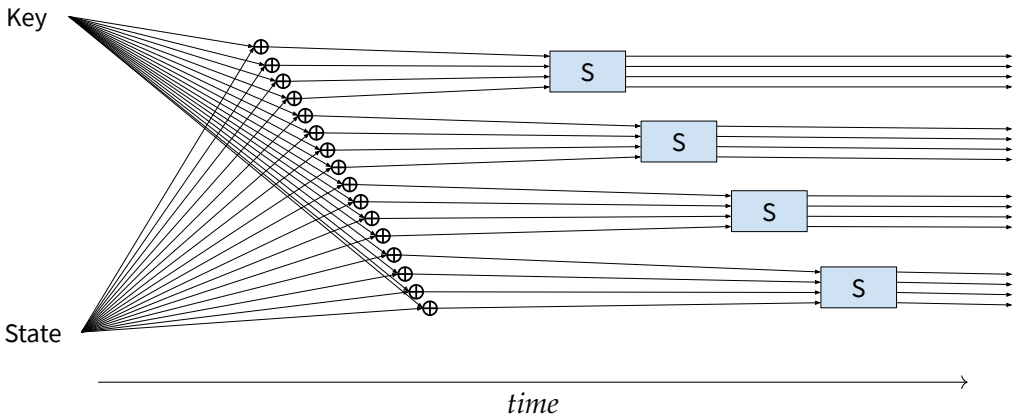
Scheduling bitsliced code

Reducing register pressure



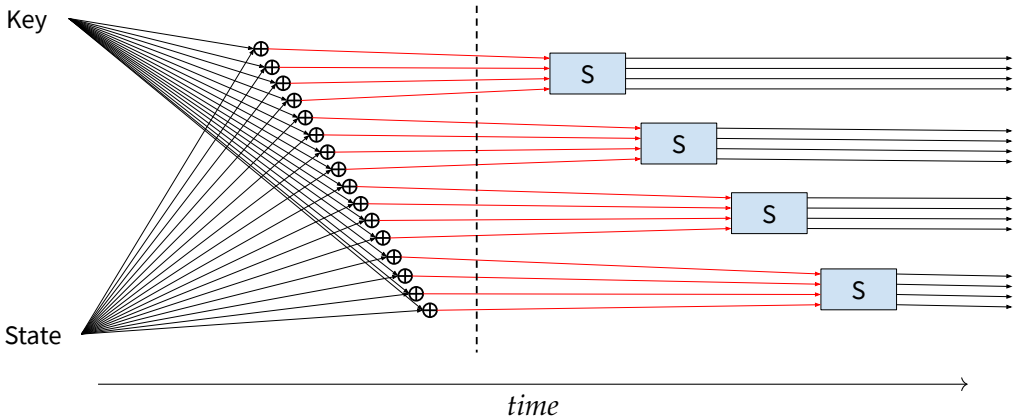
Scheduling bitsliced code

Reducing register pressure



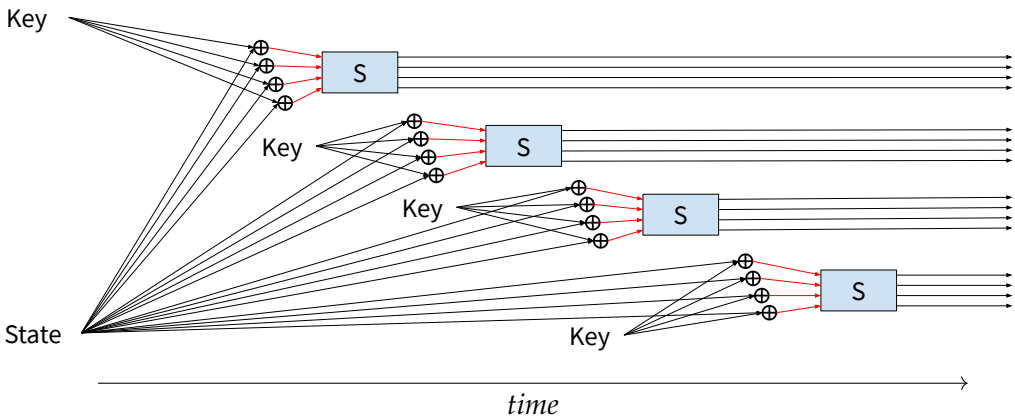
Scheduling bitsliced code

Reducing register pressure

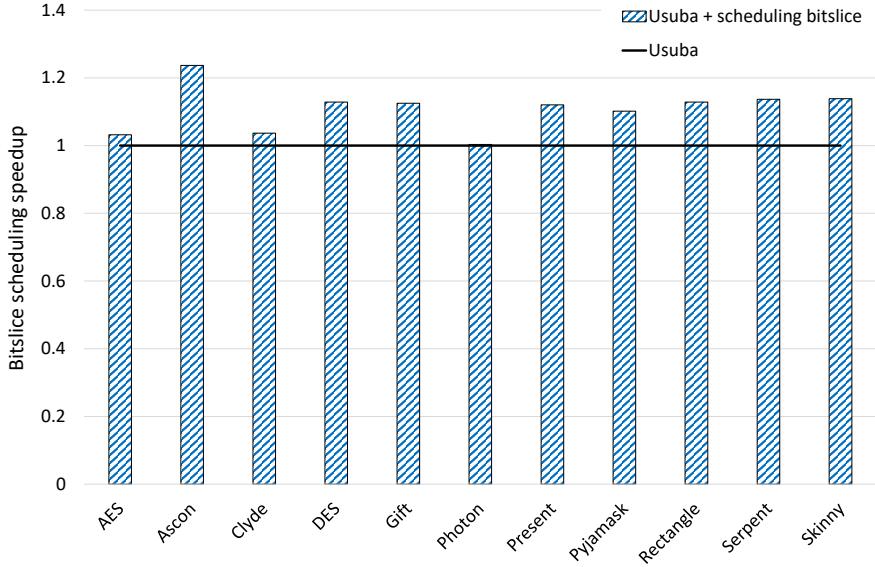


Scheduling bitsliced code

Reducing register pressure



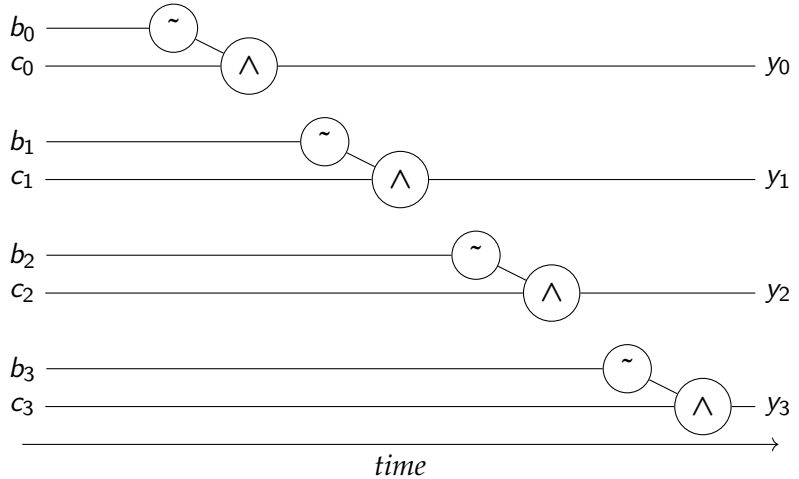
Scheduling bitsliced code: evaluation



Scheduling *m*sliced code

Maximizing instruction-level parallelism

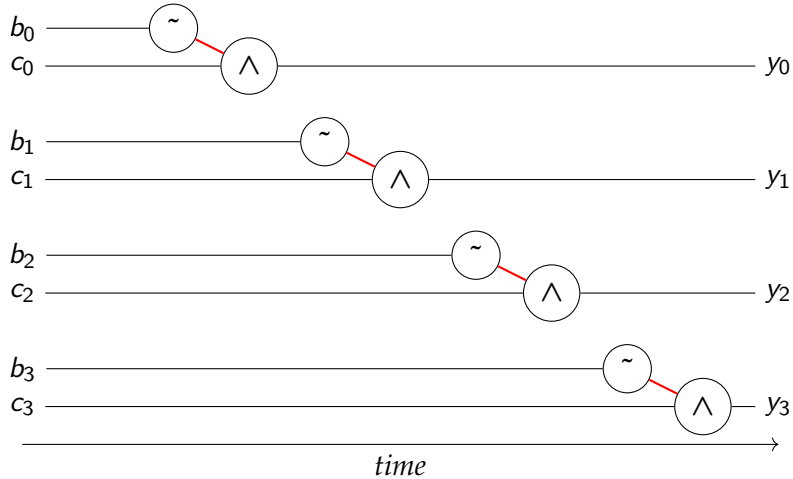
```
node my_cipher (a,b:b4) returns (y:b4)
let   forall i in [0, 3] {
         tmp  = ~ a[i];
         y[i] = tmp  ^ b[i];
      }                                tel
```



Scheduling *m*sliced code

Maximizing instruction-level parallelism

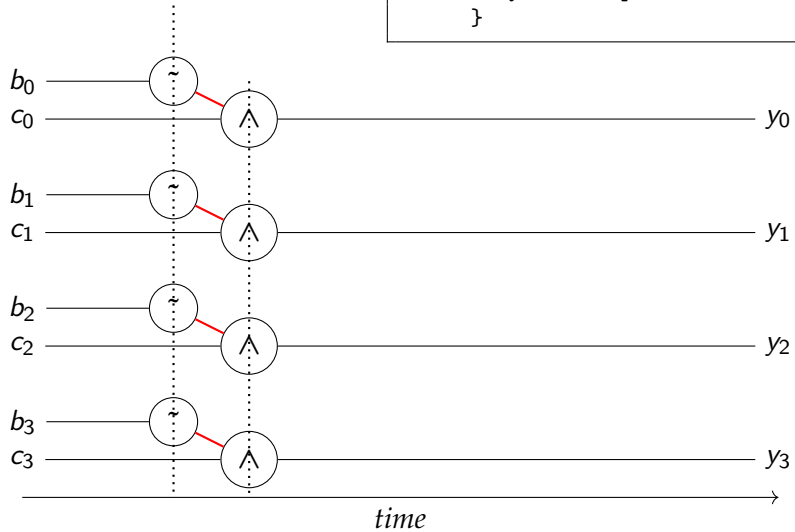
```
node my_cipher (a,b:b4) returns (y:b4)
let   forall i in [0, 3] {
         tmp  = ~ a[i];
         y[i] = tmp  ^ b[i];
      }                                tel
```



Scheduling *m*sliced code

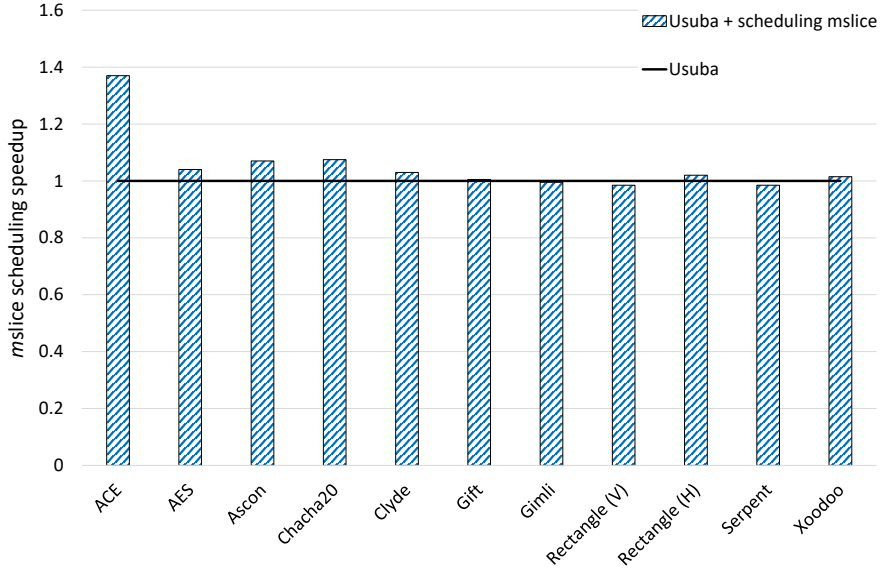
Maximizing instruction-level parallelism

```
node my_cipher (a,b:b4) returns (y:b4)
let   forall i in [0, 3] {
         tmp  = ~ a[i];
         y[i] = tmp  ^ b[i];
      }                                tel
```



Scheduling *m*sliced code: evaluation

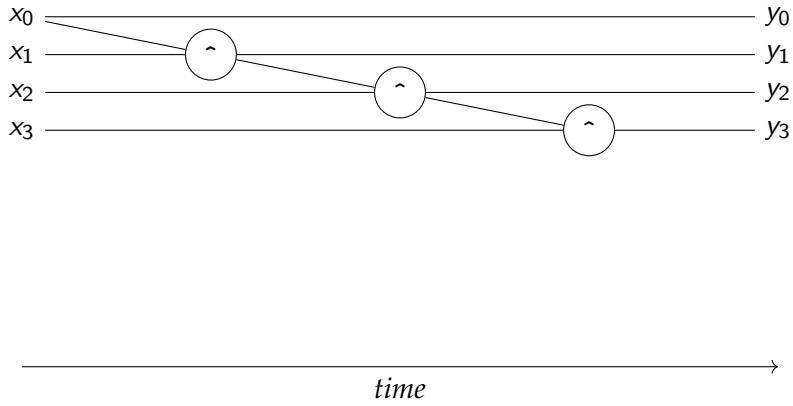
Higher is better



Interleaving

Exploiting superscalar CPUs

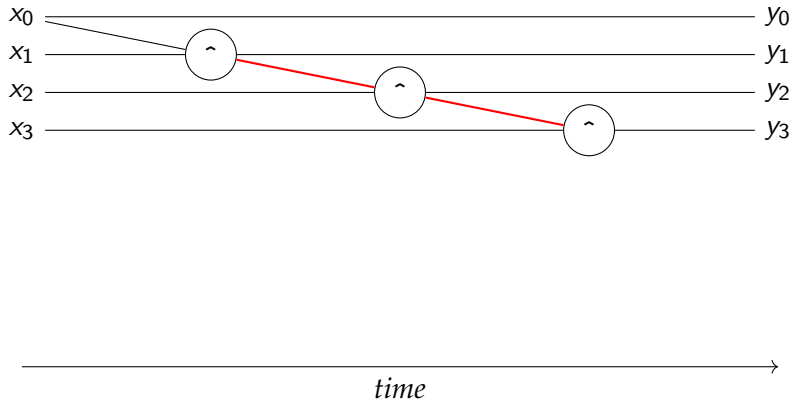
```
node my_cipher (x:b4) returns (y:b4)
let     y[0] = x[0];
        y[1] = y[0] ^ x[1];
        y[2] = y[1] ^ x[2];
        y[3] = y[2] ^ x[3];          tel
```



Interleaving

Exploiting superscalar CPUs

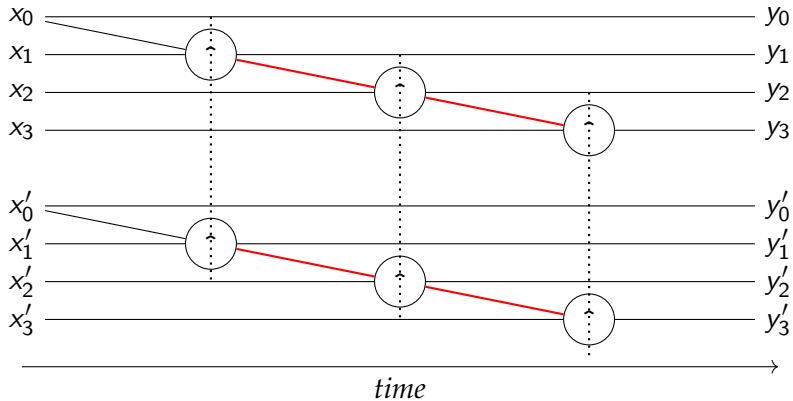
```
node my_cipher (x:b4) returns (y:b4)
let     y[0] = x[0];
        y[1] = y[0] ^ x[1];
        y[2] = y[1] ^ x[2];
        y[3] = y[2] ^ x[3];          tel
```



Interleaving

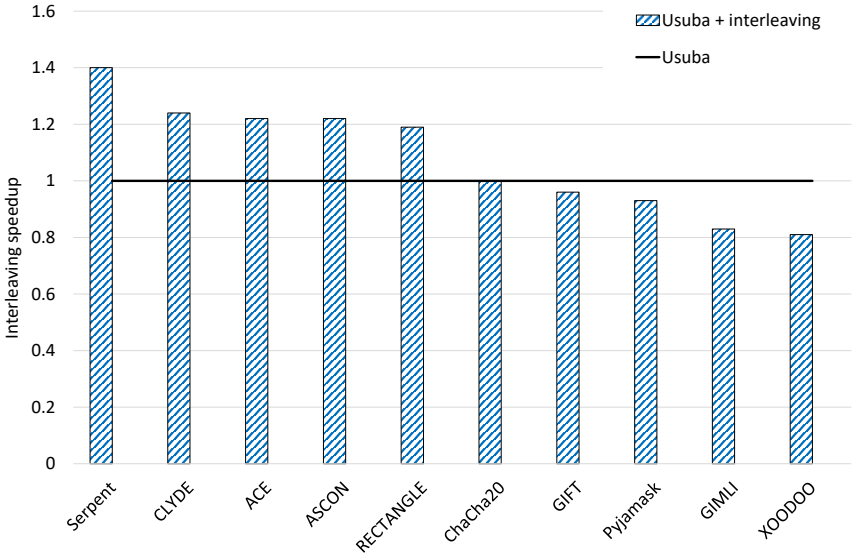
Exploiting superscalar CPUs

```
node my_cipher (x:b4) returns (y:b4)
let     y[0] = x[0];
        y[1] = y[0] ^ x[1];
        y[2] = y[1] ^ x[2];
        y[3] = y[2] ^ x[3];          tel
```



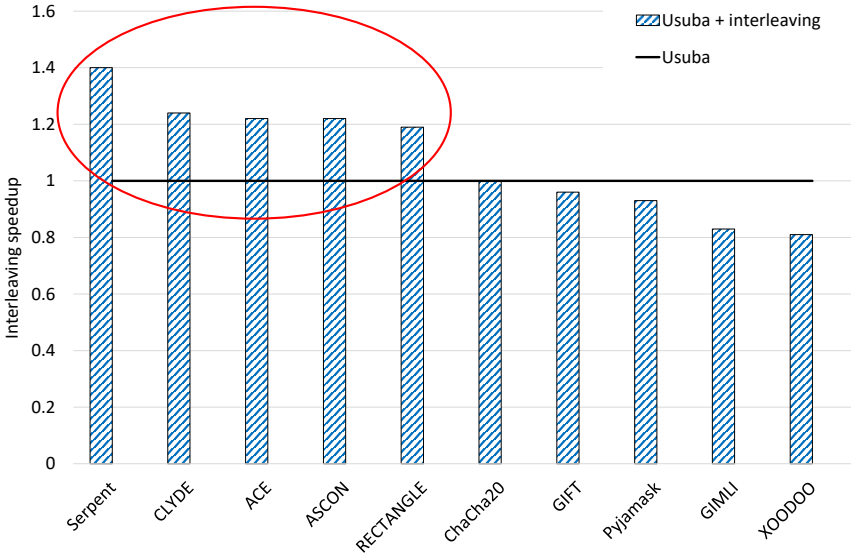
Interleaving: evaluation

Higher is better



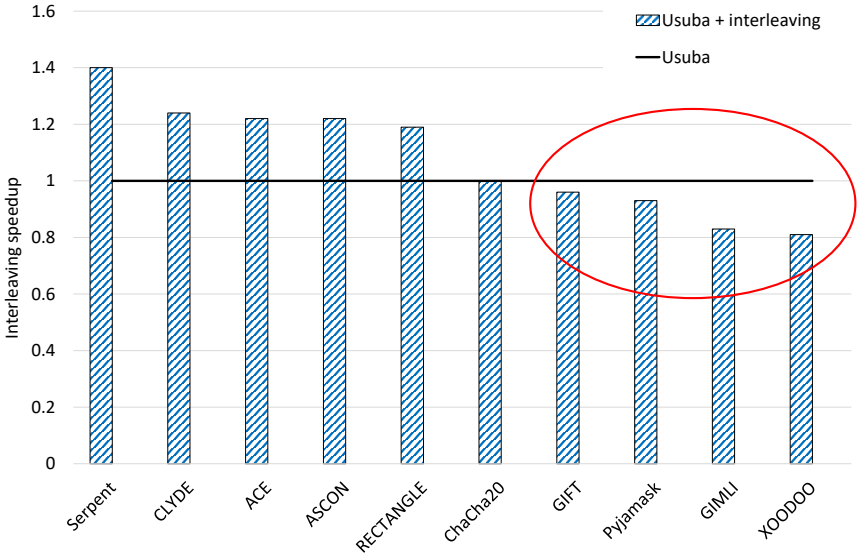
Interleaving: evaluation

Higher is better



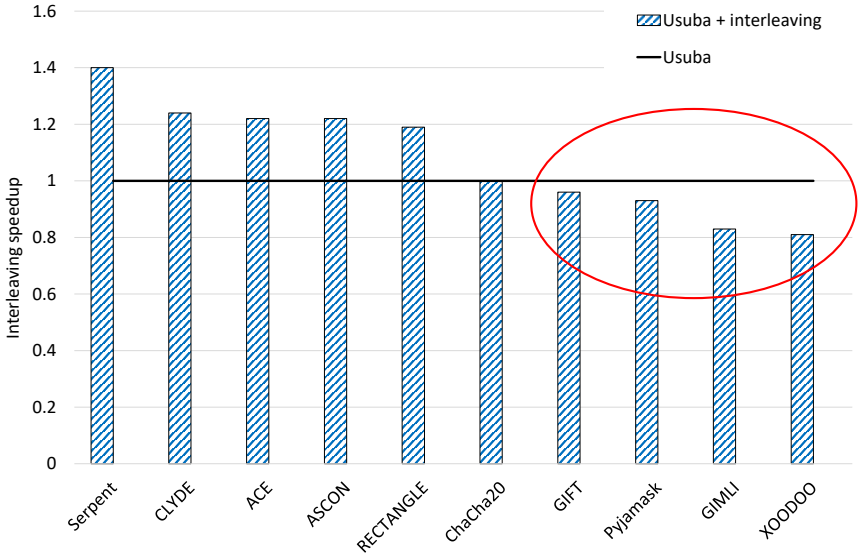
Interleaving: evaluation

Higher is better



Interleaving: evaluation

Higher is better



Slicing
$\Longrightarrow$
constant-time

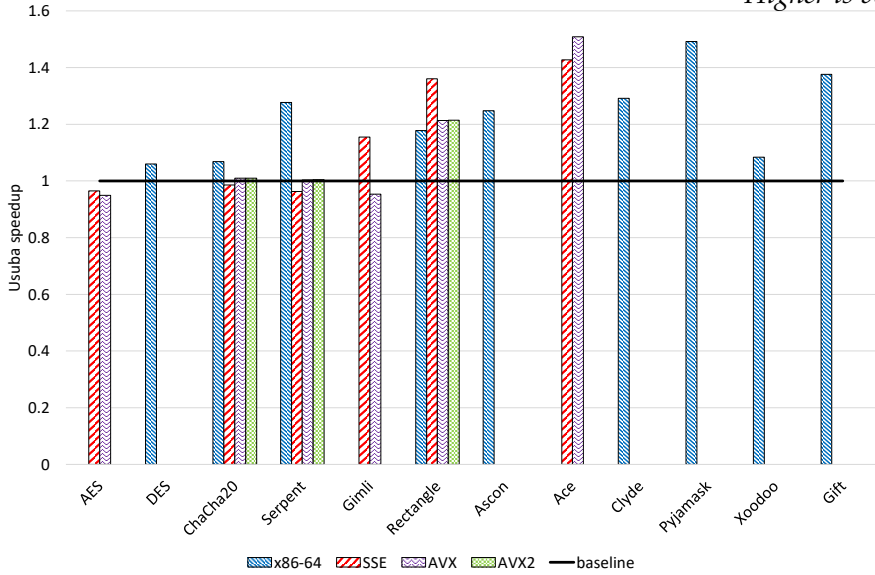
Interleaving: evaluation

Higher is better

Slicing
$\Longrightarrow$
constant-time

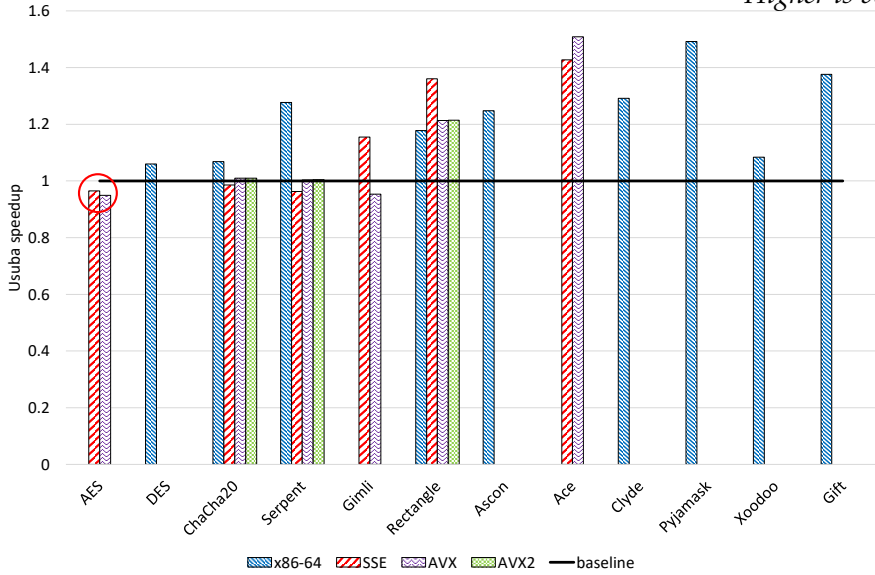
Evaluation: Usuba *vs.* Reference

Higher is better



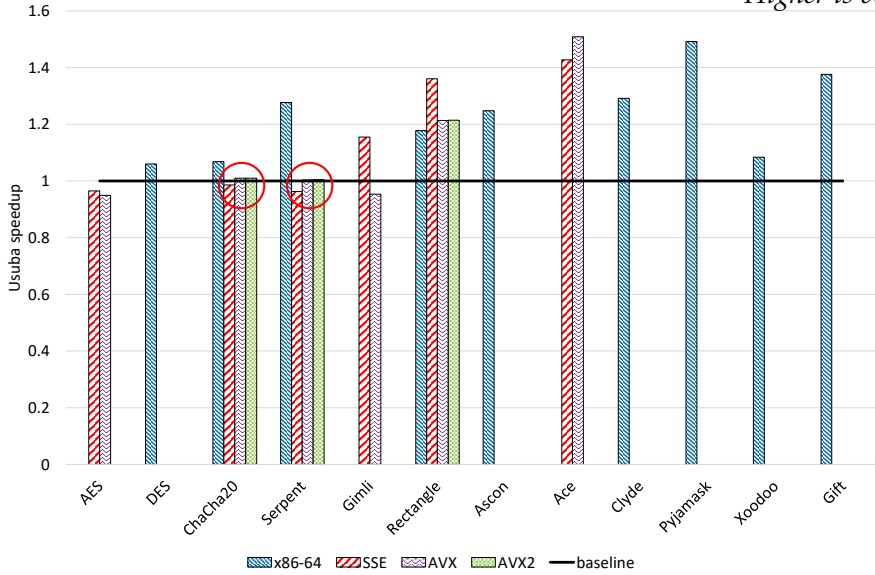
Evaluation: Usuba *vs.* Reference

Higher is better

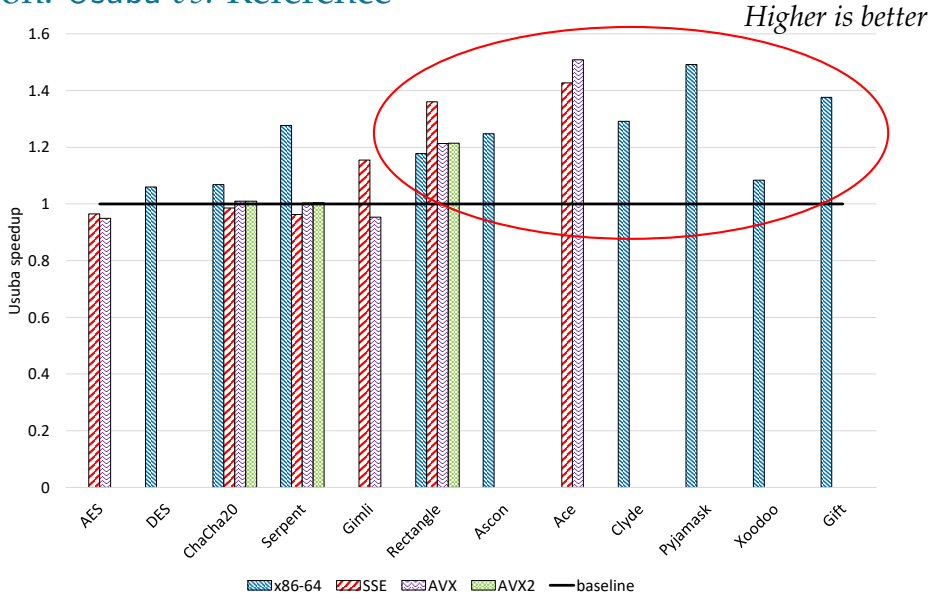


Evaluation: Usuba *vs.* Reference

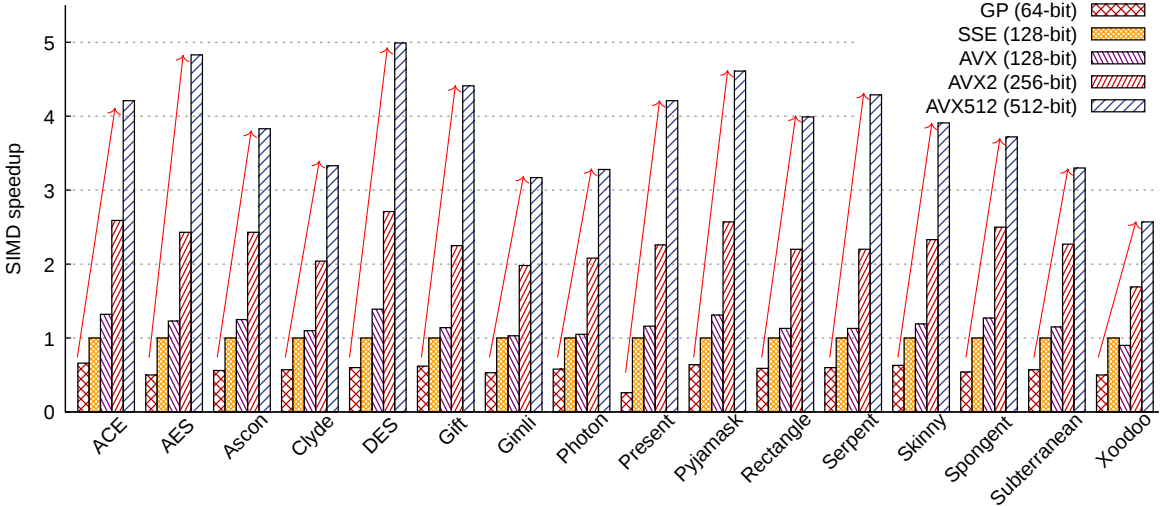
Higher is better



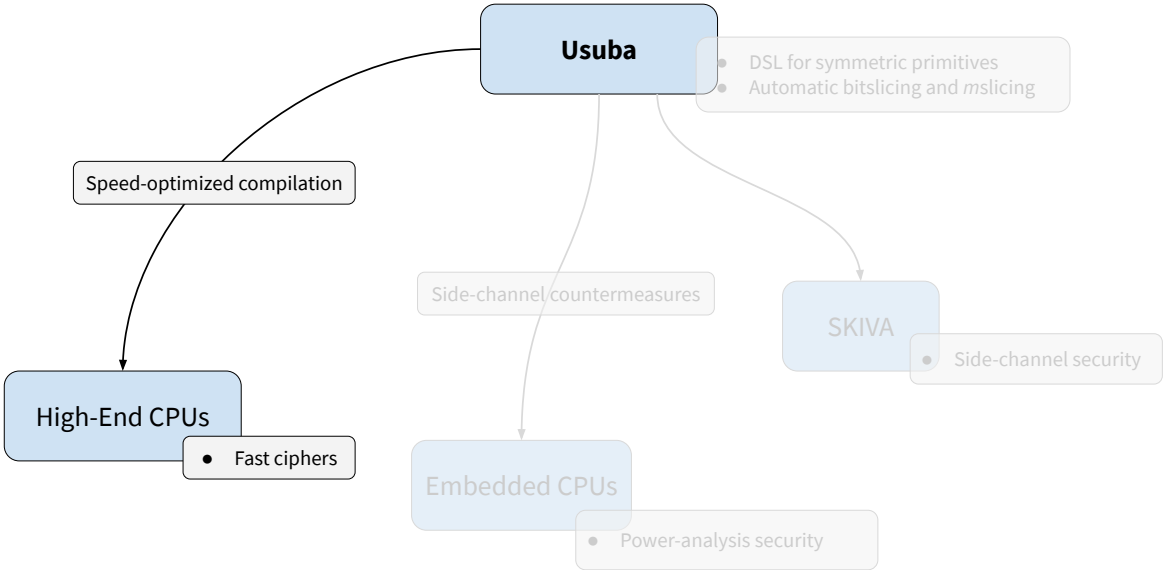
Evaluation: Usuba *vs.* Reference



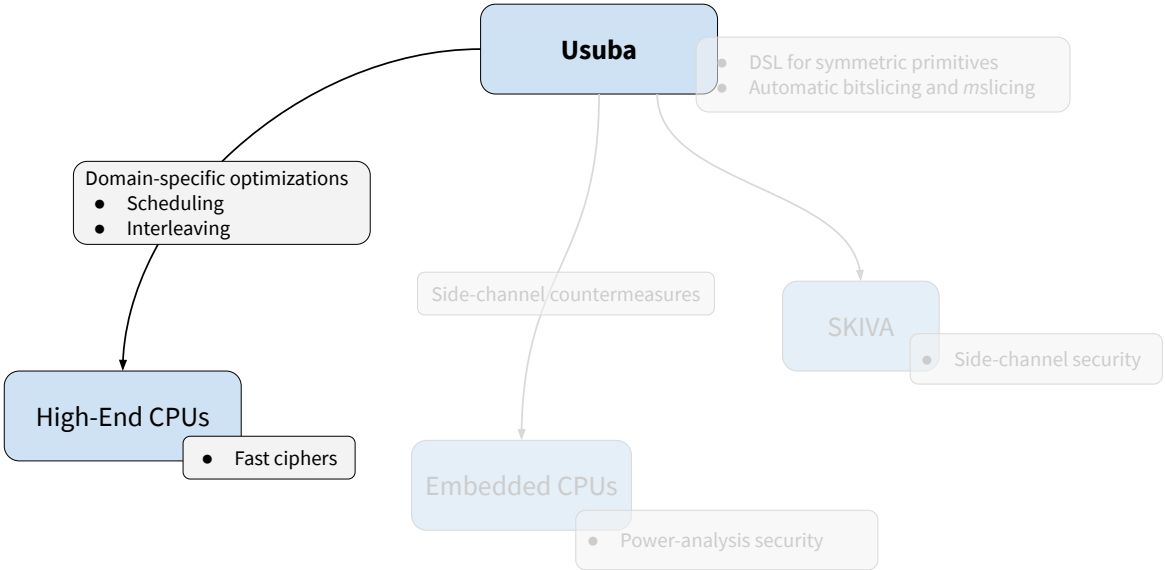
Evaluation: bitslicing scaling



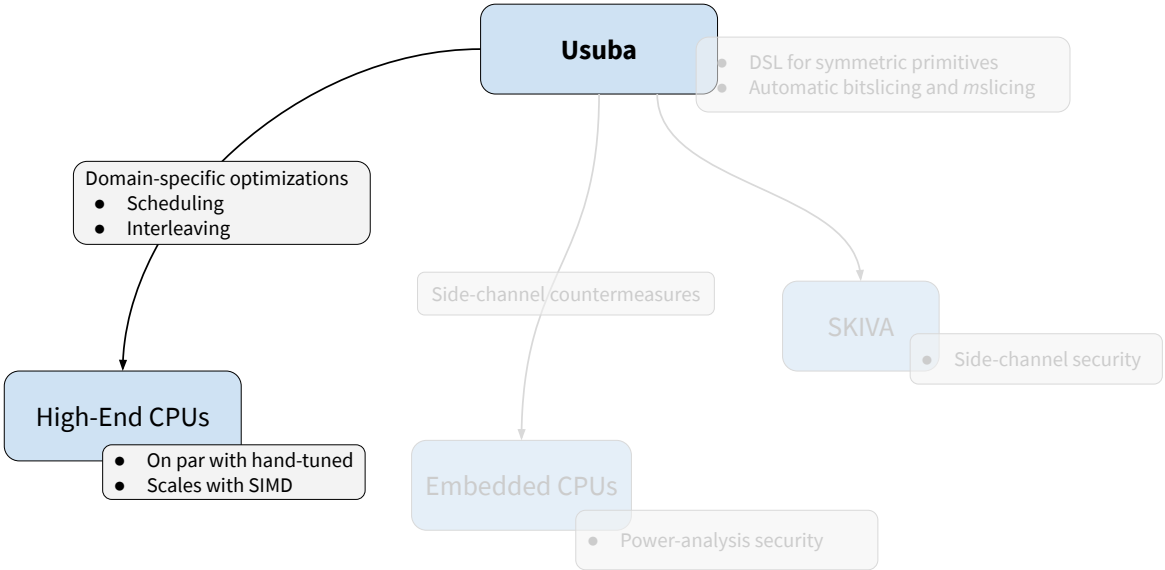
High-End CPUS: conclusion



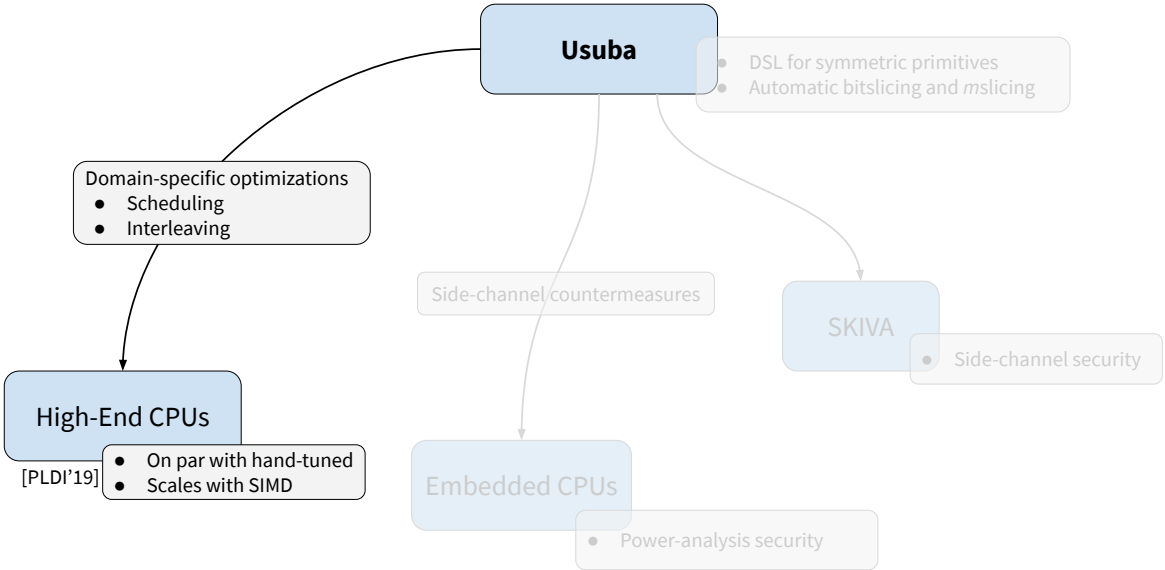
High-End CPUS: conclusion



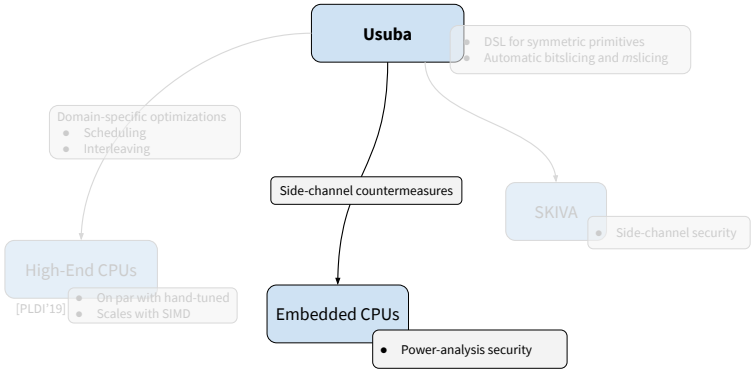
High-End CPUS: conclusion



High-End CPUS: conclusion



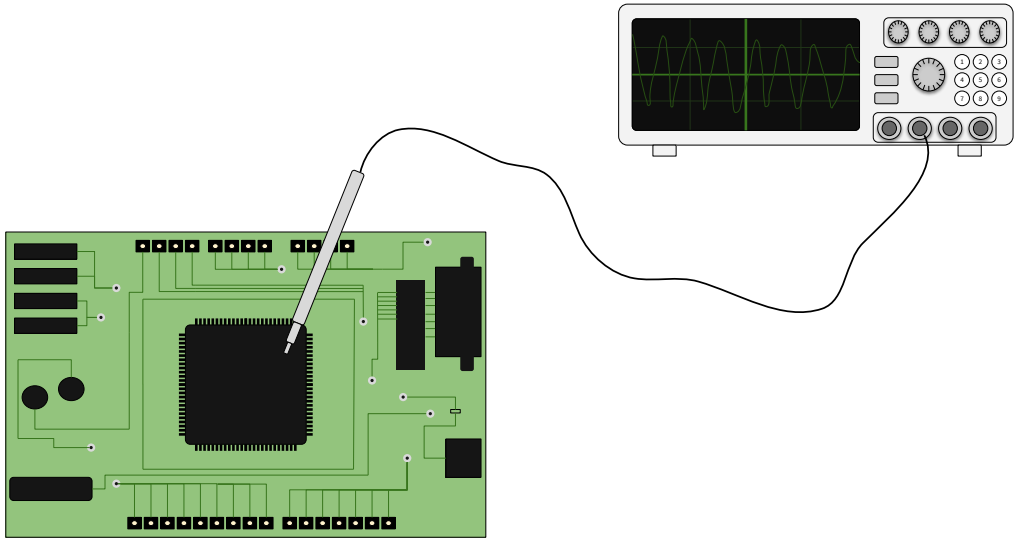
Thwarting Power-Based Side-channel Attacks



Collaboration with Raphaël Wintersdorff, Sonia Belaïd and Matthieu Rivain (CryptoExperts)

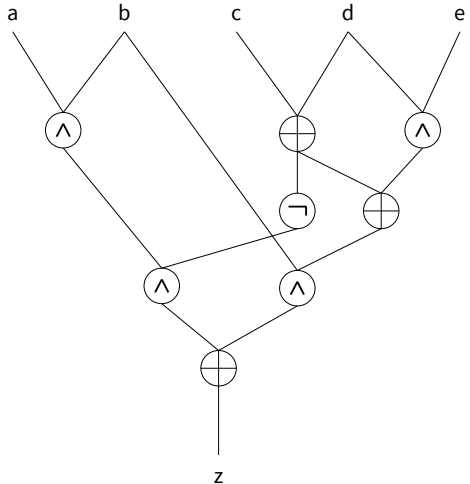
Side-channel attacks

Power-based attacks



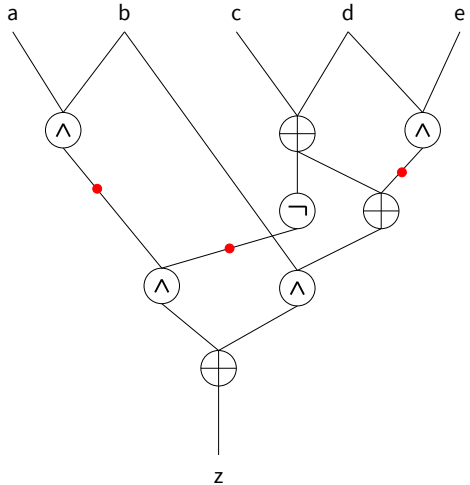
Probing model [Ishai03]

Modeling power-based side-channel attacks



Probing model [Ishai03]

Modeling power-based side-channel attacks



Boolean masking

Random values (“shares”)

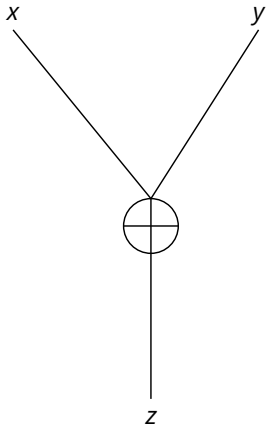
$x = (x_0, x_1, ..., x_n)$

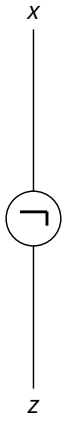
such that

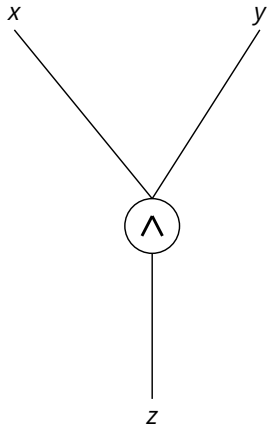
$$x_0 \oplus x_1 \oplus ... \oplus x_n = x$$

Masking a circuit

Gadgets

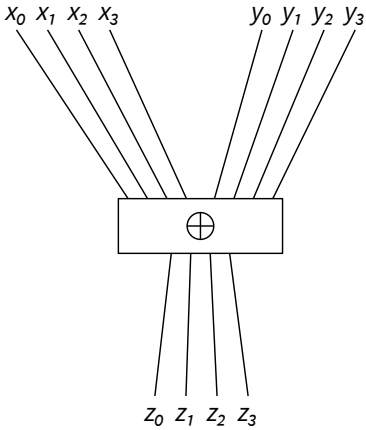


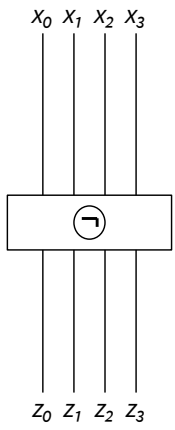


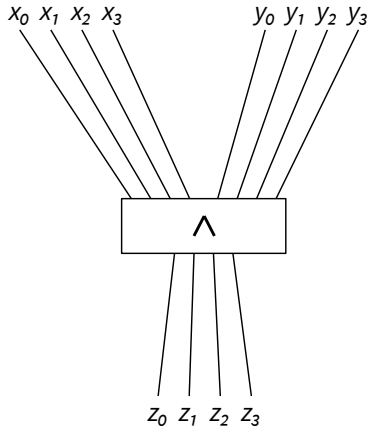


Masking a circuit

Gadgets

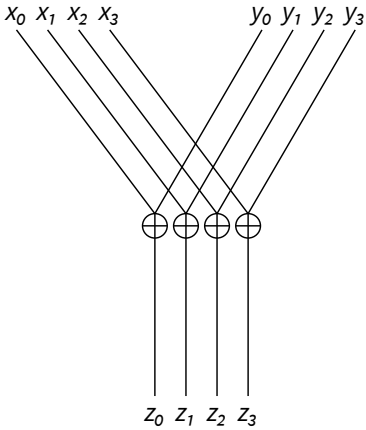


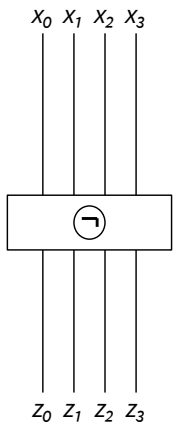


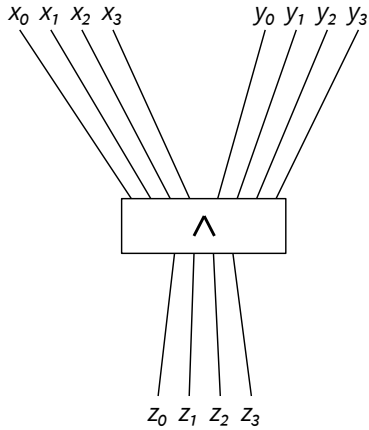


Masking a circuit

Gadgets

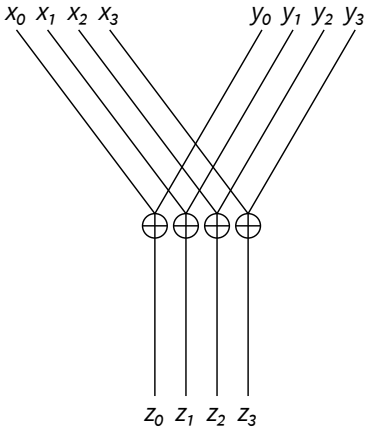


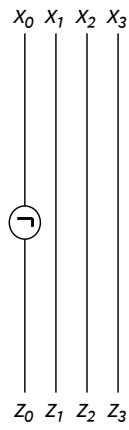


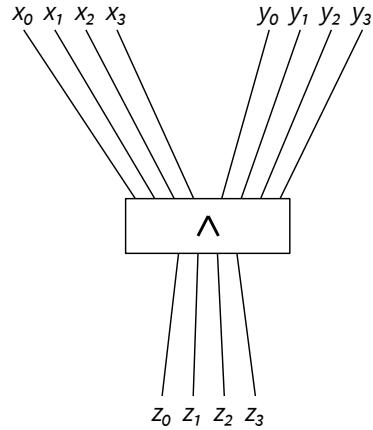


Masking a circuit

Gadgets

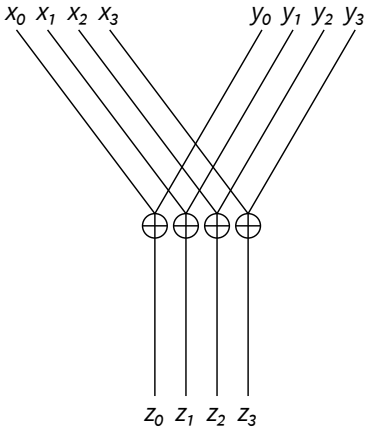


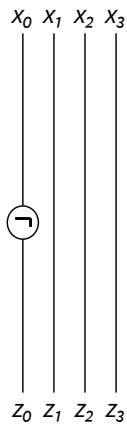


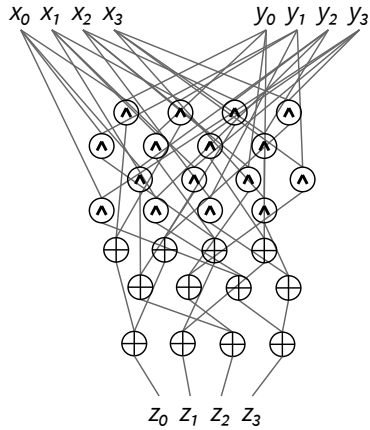


Masking a circuit

Gadgets

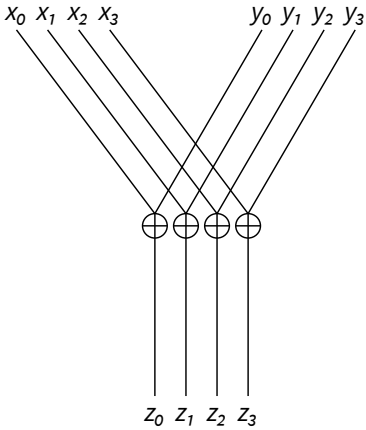


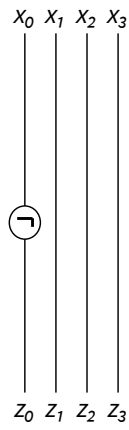


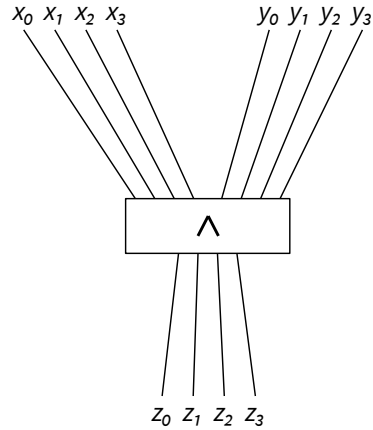


Masking a circuit

Gadgets







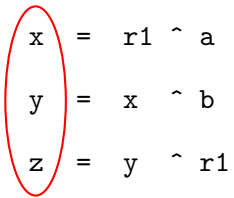
Masking a circuit

The need for refreshes

```
x = r1 ^ a

y = x  ^ b

z = y  ^ r1
```

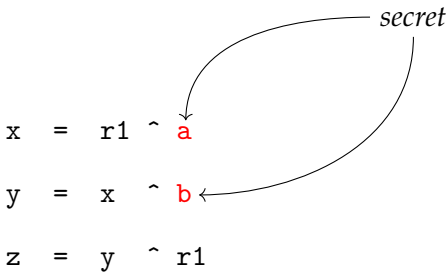
Masking a circuit

The need for refreshes



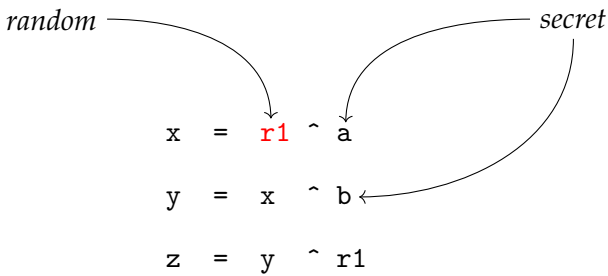
Masking a circuit

The need for refreshes



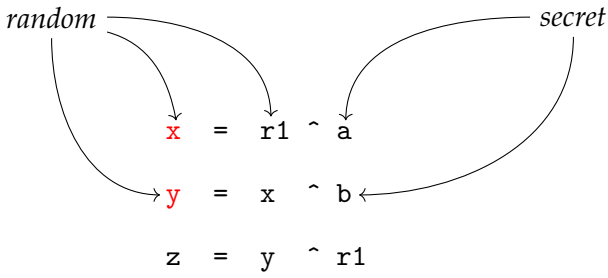
Masking a circuit

The need for refreshes



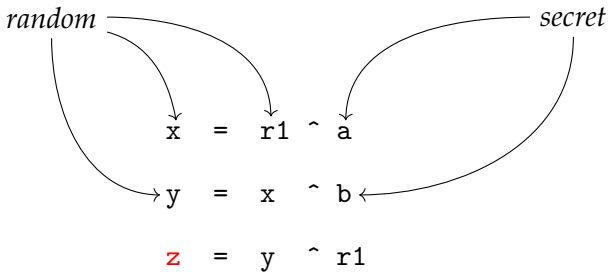
Masking a circuit

The need for refreshes



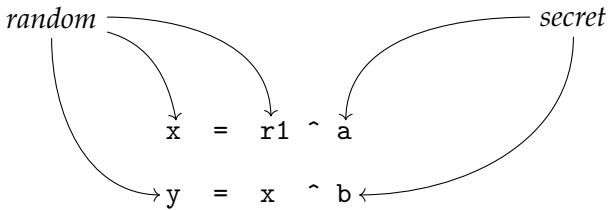
Masking a circuit

The need for refreshes



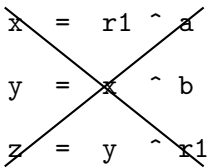
Masking a circuit

The need for refreshes



Masking a circuit

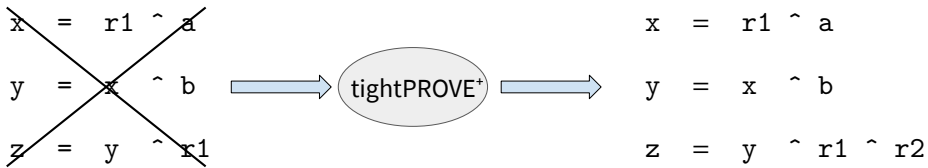
The need for refreshes



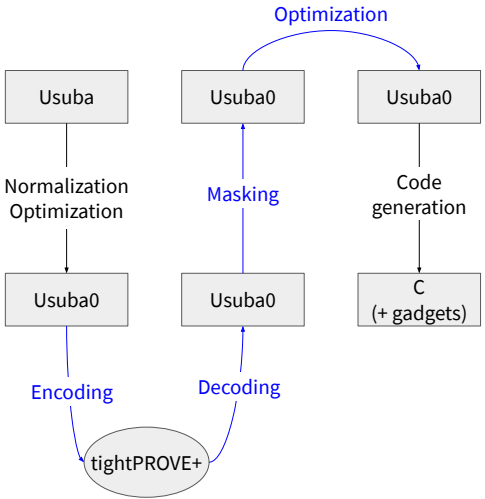

```
x = r1 ^ a
y = x ^ b
z = y ^ r1 ^ r2
```

Masking a circuit

The need for refreshes



Tornado

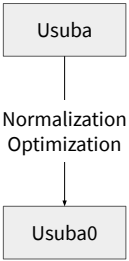

Optimization
Usuba
Usuba0
Usuba0
Normalization
Optimization
Masking
Code
generation
Usuba0
Usuba0
C
(+ gadgets)
Encoding
Decoding
tightPROVE+

Tornado

Usuba

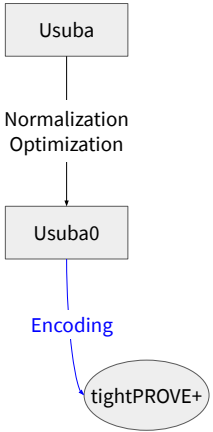
```
node f(i1, i2, i3, i4, i5 : u32)
    returns (out : u32)
let
    t1 = (i1 <<< 9) & (i2 <<< 24);
    t2 = (i3 << 1)  ^ (i4 >> 31);
    t3 = i2 ^ t1;
    t4 = t3 & t2;
    t5 = t3 ^ t2;
    t6 = (t2 <<< 3) & t5;
    out = t4 ^ t6;
tel
```

Tornado



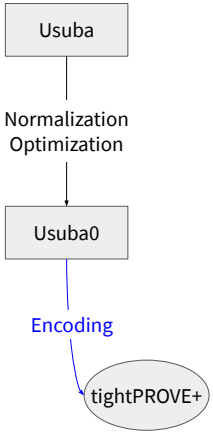

```
node f(i1, i2, i3, i4, i5 : u32)
   returns (out : u32)
vars x1, x2, x3, x4, x5, x6,
     t1, t2, t3, t4, t5 : u32
let
    t1 = i1 <<< 9;
    t2 = i2 <<< 24;
    x1 = t1 & t2;
    t3 = i1 << 1;
    t4 = i4 >> 31;
    x2 = t2 ^ t4;
    x3 = i2 ^ x1;
    x4 = x3 & x2;
    x5 = x3 ^ x2;
    t5 = x2 <<< 3;
    x6 = t5 & x5;
    out = x4 ^ x6;
tel
```

Tornado



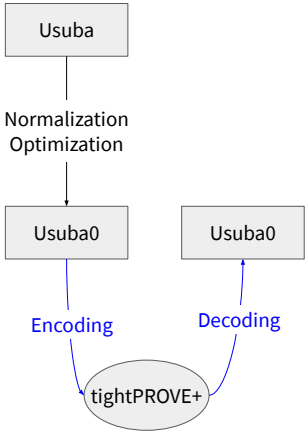

```
rs=32

in i1 i2 i3 i4 i5

t1 = i1 <<< 9
t2 = i2 <<< 24
t1 = and t1 t2
t3 = i3 << 1
t4 = i4 >> 31
t2 = xor t3 t4
t3 = xor i2 t1
t4 = and t3 t2
t5 = xor t3 t2
t5 = t2 <<< 3
t6 = and t5 t5
out = xor t4 t6
```

Tornado



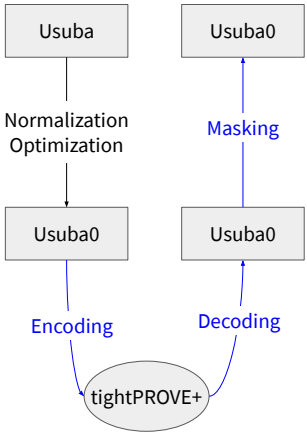

```
rs=32

in i1 i2 i3 i4 i5

t1 = i1 <<< 9
t2 = i2 <<< 24
t1 = and t1 t2
t3 = i3 << 1
t4 = i4 >> 31
t2 = xor t3 t4
t3 = xor i2 t1
t4 = and t3 t2
t5 = xor t3 t2
t5_r = ref(t5)
t5 = t2 <<< 3
t6 = and t5 t5_r
out = xor t4 t6
```

Tornado




```
node f(i1, i2, i3, i4, i5 : u32)
   returns (out : u32)
vars x1, x2, x3, x4, x5, x6
     t1, t2, t3, t4, t5, x5_r : u32
let
    t1 = i1 <<< 9;
    t2 = i2 <<< 24;
    x1 = t1 & t2;
    t3 = i1 << 1;
    t4 = i4 >> 31;
    x2 = t2 ^ t4;
    x3 = i2 ^ x1;
    x4 = x3 & x2;
    x5 = x3 ^ x2;
    x5_r = refresh(x5);
    t5 = x2 <<< 3;
    x6 = t5 & x5_r;
    out = x4 ^ x6;
tel
```

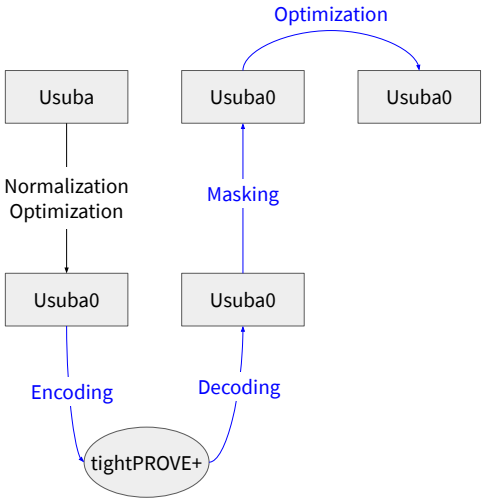
Tornado



NS = Number of Shares

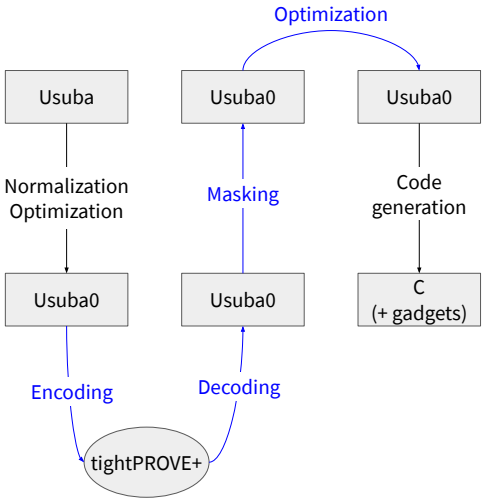
```
node f(i1, i2, i3, i4, i5 : u32[NS])
    returns (out : u32[NS])
vars x1, x2, x3, x4, x5, x6,
     t1, t2, t3, t4, t5, x5_r : u32[NS]
let
    for i in [0..NS-1]{ t1[NS] = i1[NS] <<< 9 }
    for i in [0..NS-1]{ t2[NS] = i2[NS] <<< 24 }
    x1 = MASKED_AND(t1, t2);
    for i in [0..NS-1]{ t3[NS] = i3[NS] << 1 }
    for i in [0..NS-1]{ t4[NS] = i4[NS] >> 31 }
    for i in [0..NS-1]{ x2[NS] = t3[NS] ^ t4[NS] }
    for i in [0..NS-1]{ x3[NS] = i2[NS] ^ x1[NS] }
    x4 = MASKED_AND(x3, x2);
    for i in [0..NS-1]{ x5[NS] = x3[NS] ^ x2[NS] }
    x5_r = REFRESH(x5);
    for i in [0..NS-1]{ t5[NS] = x2[NS] <<< 3 }
    x6 = MASKED_AND(t5, x5_r);
    for i in [0..NS-1]{ out[NS] = x4[NS] ^ x6[NS] }
tel
```

Tornado



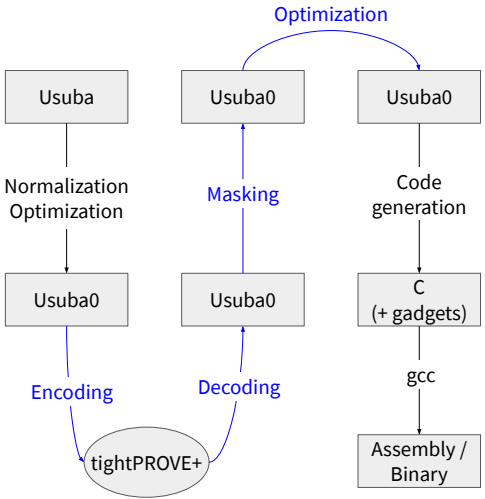

```
node f(i1, i2, i3, i4, i5 : u32[NS])
    returns (out : u32[NS])
vars x1, x2, x3, x4, x5, x6,
     t1, t2, t3, t4, t5, x5_r : u32[NS]
let
    for i in [0 .. NS-1]{
        t1[NS] = i1[NS] <<< 9;
        t2[NS] = i2[NS] <<< 24;
        t3[NS] = i3[NS] << 1;
        t4[NS] = i4[NS] >> 31;
        x2[NS] = t3[NS] ^ t4[NS];
        t5[NS] = x2[NS] <<< 3;
    }
    x1 = MASKED_AND(t1, t2);
    for i in [0 .. NS-1]{
        x3[NS] = i2[NS] ^ x1[NS];
        x5[NS] = x3[NS] ^ x2[NS];
    }
    x4 = MASKED_AND(x3, x2);
    x5_r = REFRESH(x5);
    x6 = MASKED_AND(t5, x5_r);
    for i in [0..NS-1]{ out[NS] = x4[NS] ^ x6[NS] }
tel
```

Tornado



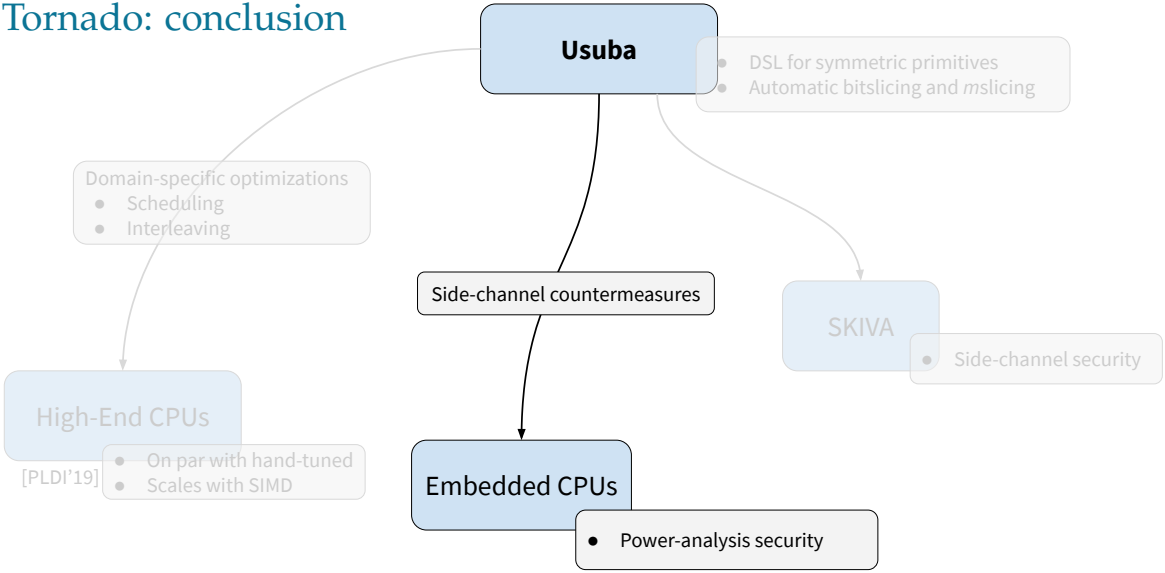

```
void f (int i1[NS], int i2[NS], int i3[NS],
        int i4[NS], int i5[NS], int out[NS]) {
  int _t1_[NS],_t2_[NS],_t3_,_t4_,_t5_[NS],x1[NS],
      x2[NS],x3[NS],x4[NS],x5[NS],x6[NS],x5_r[NS];
  for (int i = 0; i <= (NS - 1); i++) {
    _t1_[i] = L_ROTATE(i1[i],9,32);
    _t2_[i] = L_ROTATE(i2[i],24,32);
    _t3_ = L_SHIFT(i3[i],1,32);
    _t4_ = R_SHIFT(i4[i],31,32);
    x2[i] = XOR(_t3_,_t4_);
    _t5_[i] = L_ROTATE(x2[i],3,32); }
  MASKED_AND(x1,_t1_,_t2_);
  for (int i = 0; i <= (NS - 1); i++) {
    x3[i] = XOR(i2[i],x1[i]);
    x5[i] = XOR(x3[i],x2[i]); }
  MASKED_AND(x4,x3,x2);
  REFRESH(x5_r,x5);
  MASKED_AND(x6,_t5_,x5_r);
  for (int i = 0; i <= (NS - 1); i++) {
    out[i] = XOR(x4[i],x6[i]); }
}
```

Tornado

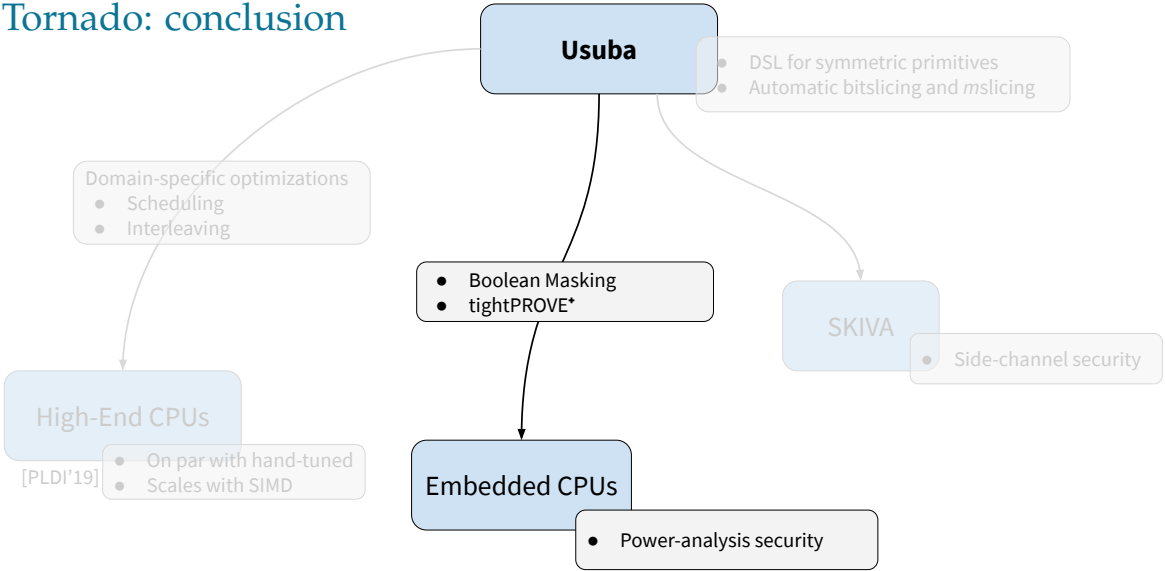



```
f:
  stmfd sp!, {r4, r5, r6, r7, r8, \
              r9, r10, fp, lr}
  sub sp, sp, #5056
  sub sp, sp, #36
  sub r5, r1, #4
  add r1, sp, #2048
  sub r6, r1, #12
  mov fp, r6
  mov r10, r5
  mov r4, #0
  sub r0, r0, #4
  sub r2, r2, #4
  sub r3, r3, #4
  add r9, sp, #4
  add r8, sp, #512
  add r7, sp, #1020
.L19:
  ldr r1, [r2, #4]!
  ldr ip, [r3, #4]!
  mov r1, r1, asl #1
  orr r1, r1, ip, lsr #31
  ...
```

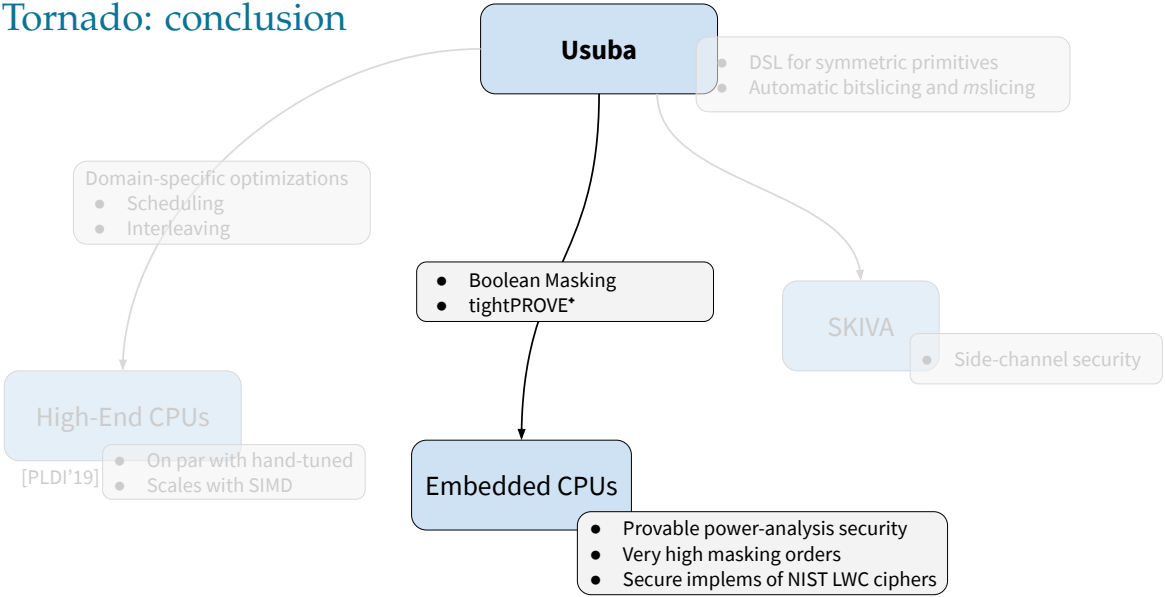
Tornado: conclusion



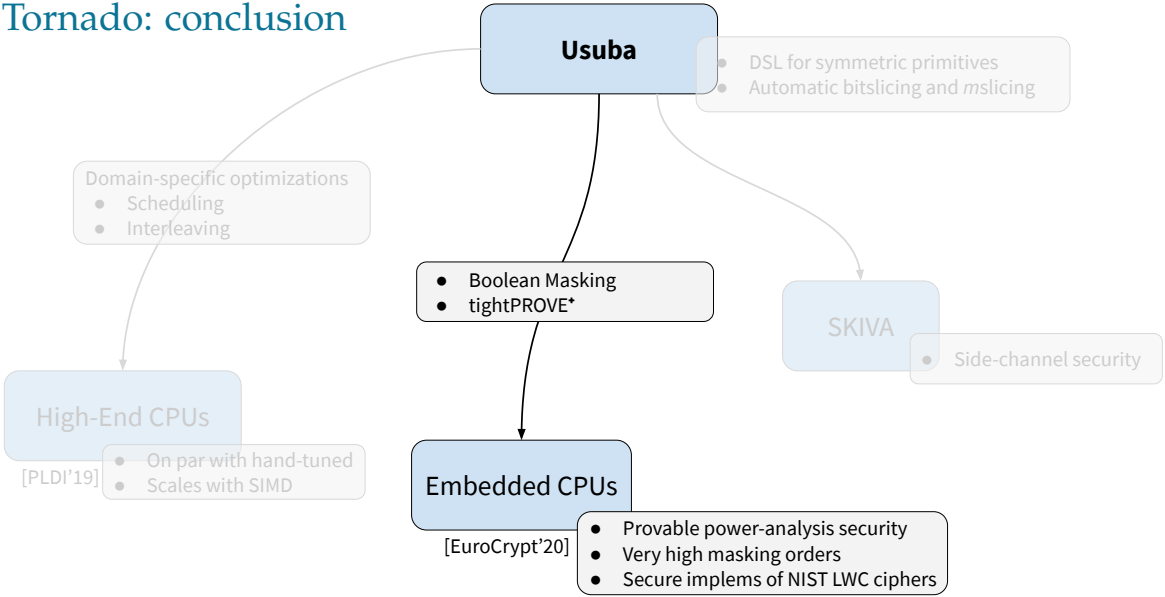
Tornado: conclusion



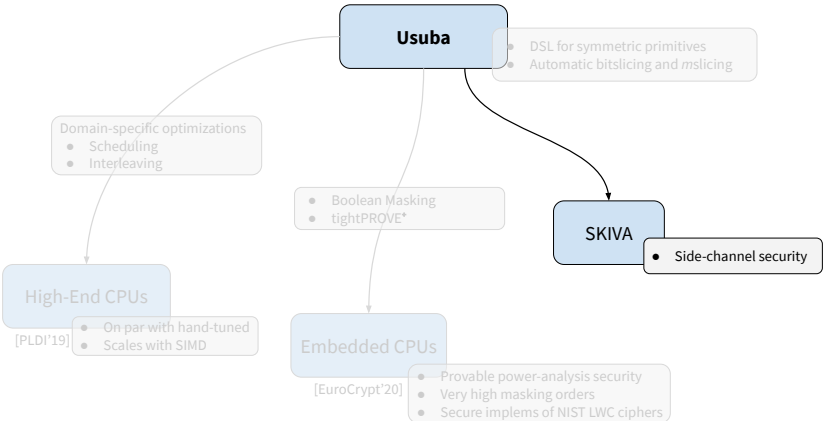
Tornado: conclusion



Tornado: conclusion

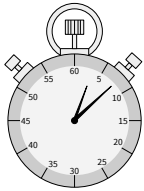


Thwarting Fault Attacks and Power-Based Side-channel Attacks



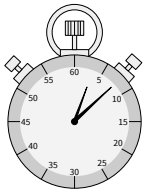
Collaboration with Pantea Kiaei, Patrick Schaumont (Worcester Polytechnic Institute) and Karine Heydemann (LIP6)

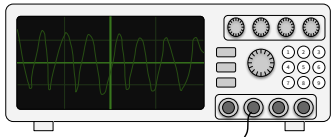
Threat model



Time-based SCA

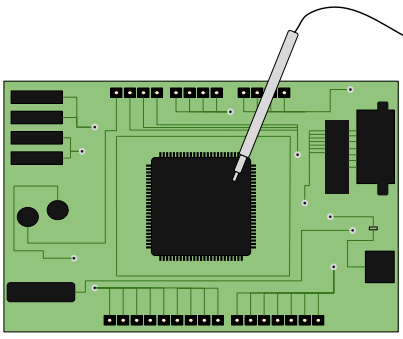
Threat model





Time-based SCA

Power-based SCA

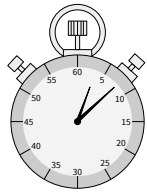


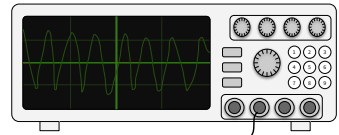
Threat model

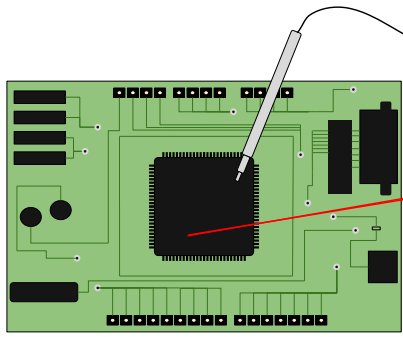
Time-based SCA

Power-based SCA

Data faults







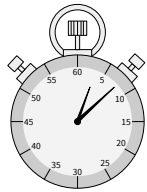
Threat model

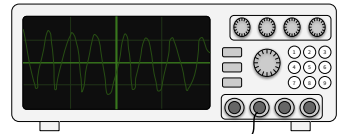
Time-based SCA

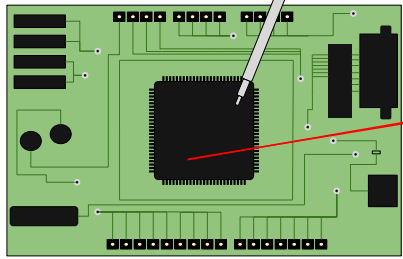
Power-based SCA

Data faults

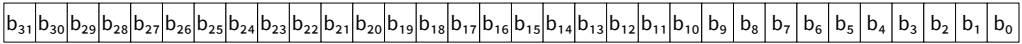
Control faults





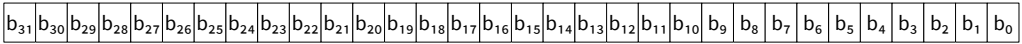


Aggregated bitslice model



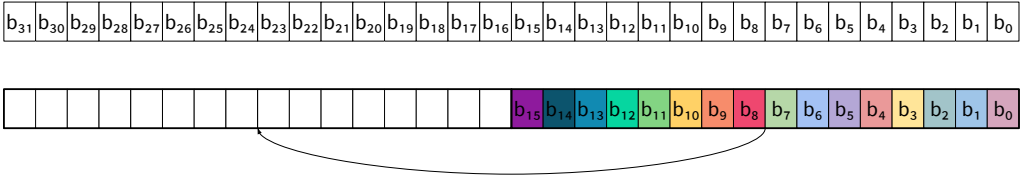
Timing attacks ✗ Data faults ✗ Power analysis ✗ Control faults ✗

Aggregated bitslice model



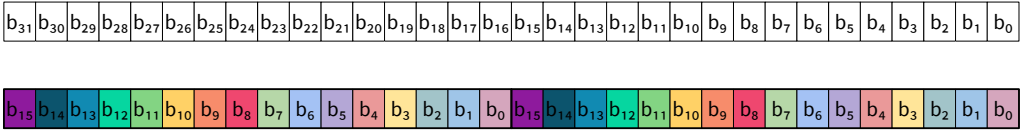
Timing attacks ✓ Data faults ✗ Power analysis ✗ Control faults ✗

Aggregated bitslice model



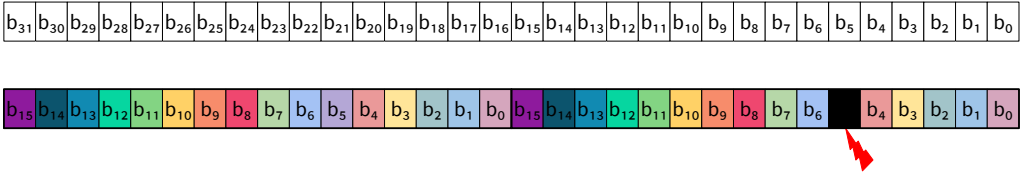
Timing attacks ✓ Data faults ✗ Power analysis ✗ Control faults ✗

Aggregated bitslice model



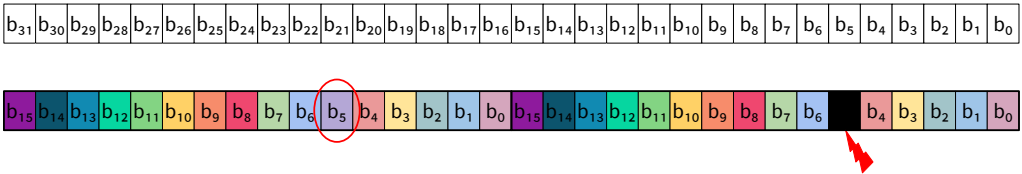
Timing attacks ✓ Data faults ✓ Power analysis ✗ Control faults ✗

Aggregated bitslice model



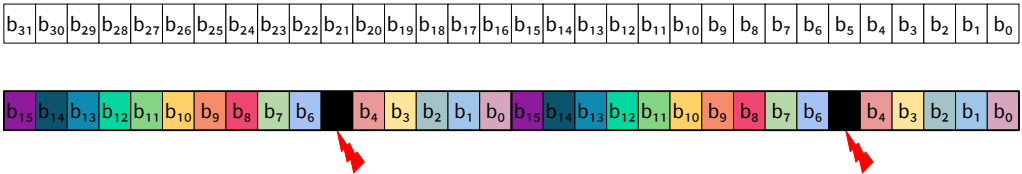
Timing attacks ✓ Data faults ✓ Power analysis ✗ Control faults ✗

Aggregated bitslice model



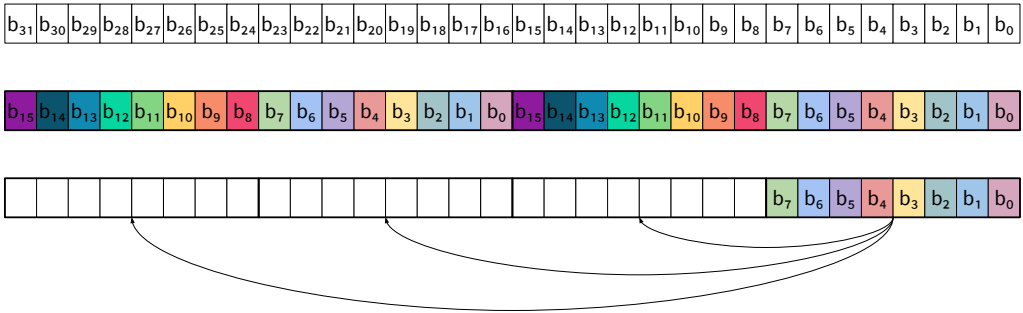
Timing attacks ✓ Data faults ✓ Power analysis ✗ Control faults ✗

Aggregated bitslice model



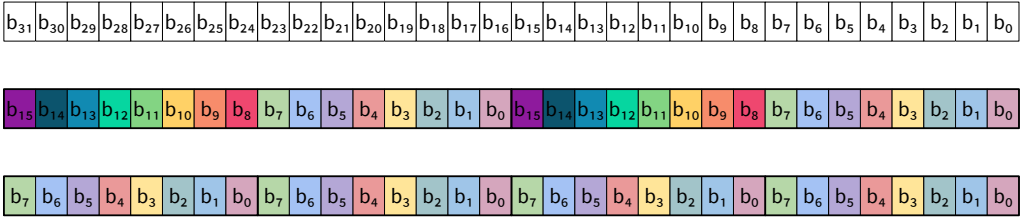
Timing attacks ✓ Data faults ✓ Power analysis ✗ Control faults ✗

Aggregated bitslice model



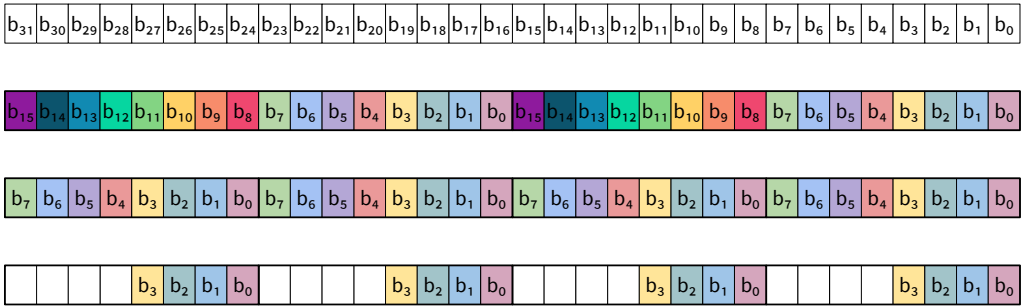
Timing attacks ✓ Data faults ✓ Power analysis ✗ Control faults ✗

Aggregated bitslice model



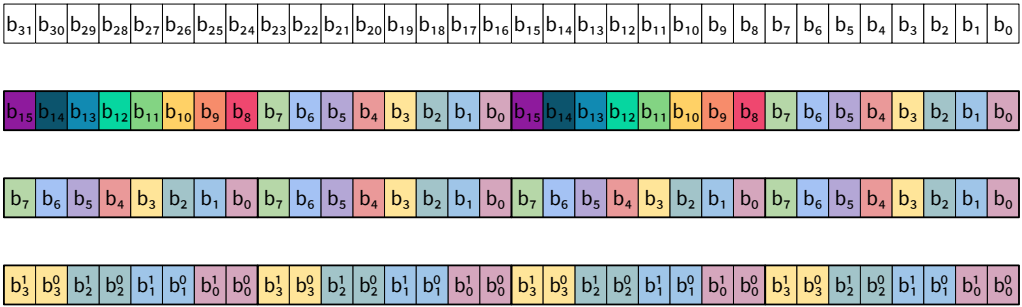
Timing attacks ✓ Data faults ✓ Power analysis ✗ Control faults ✗

Aggregated bitslice model



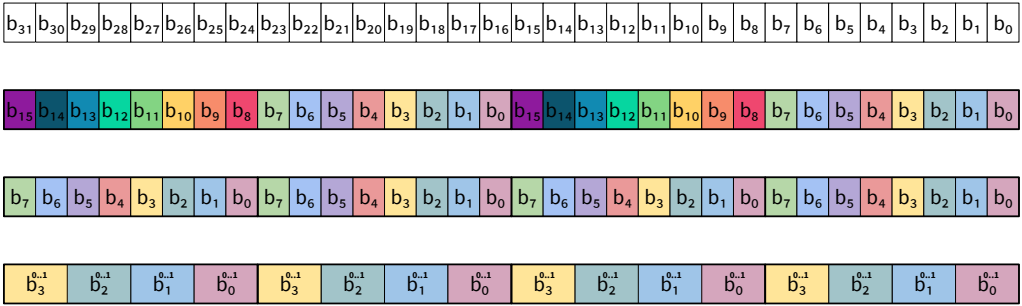
Timing attacks ✓ Data faults ✓ Power analysis ✗ Control faults ✗

Aggregated bitslice model



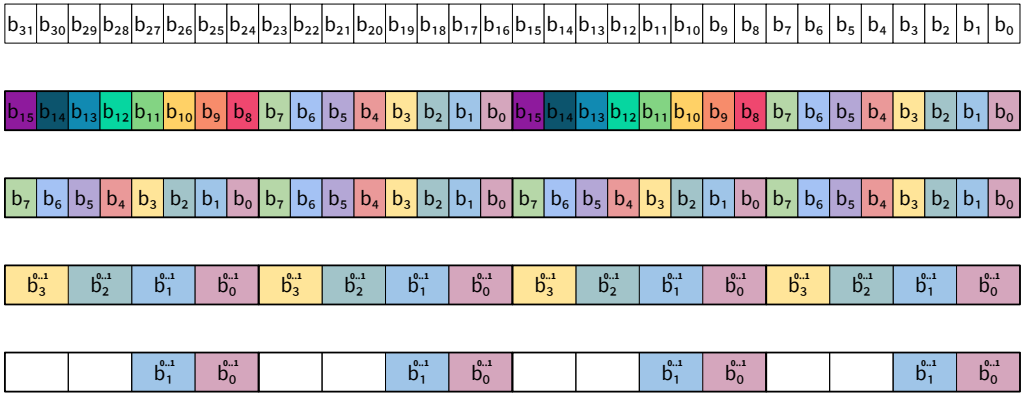
Timing attacks ✓ Data faults ✓ Power analysis ✓ Control faults ✗

Aggregated bitslice model



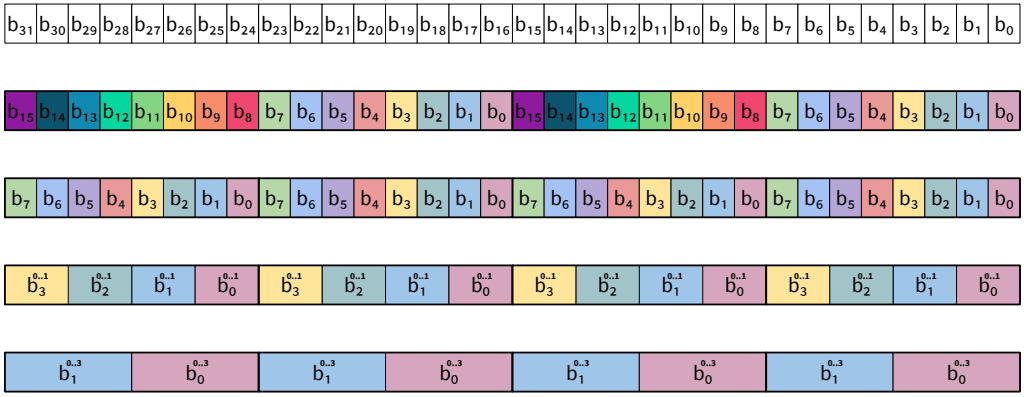
Timing attacks ✓ Data faults ✓ Power analysis ✓ Control faults ✗

Aggregated bitslice model



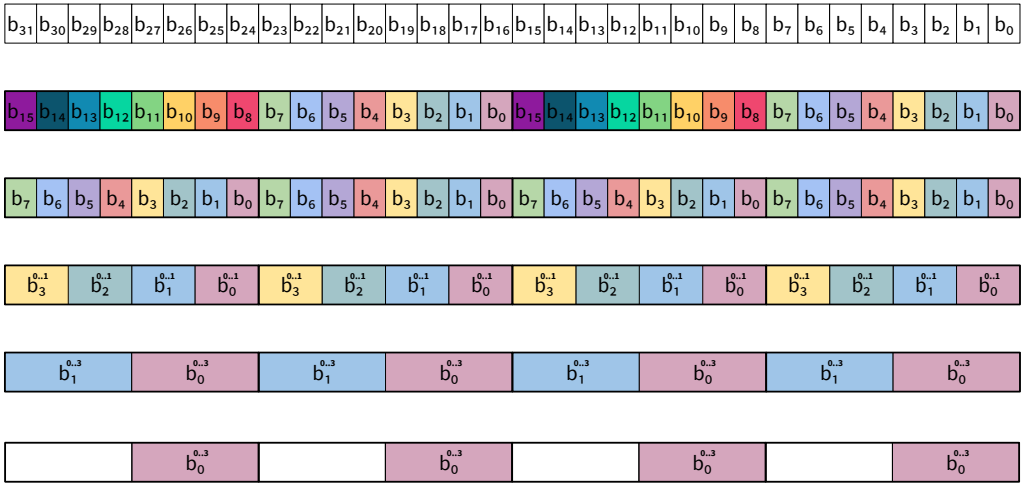
Timing attacks ✓ Data faults ✓ Power analysis ✓ Control faults ✗

Aggregated bitslice model



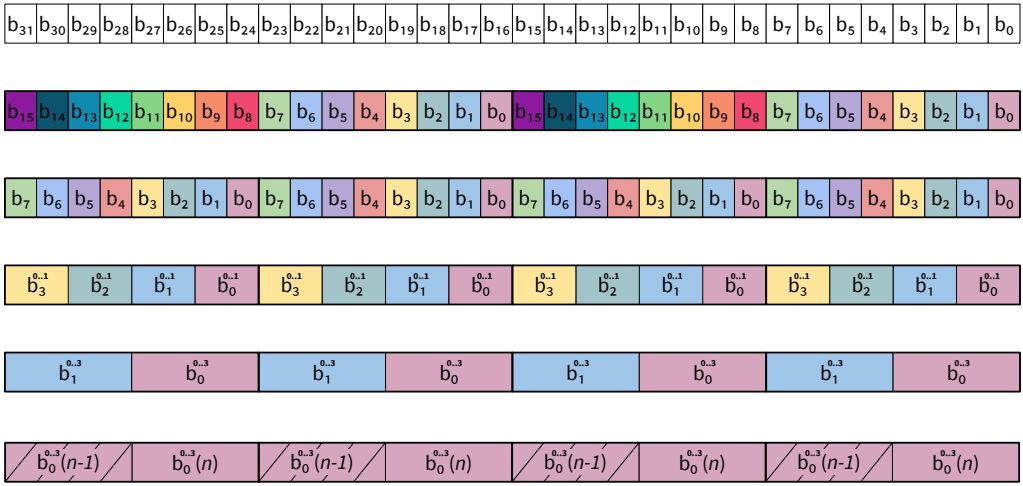
Timing attacks ✓ Data faults ✓ Power analysis ✓ Control faults ✗

Aggregated bitslice model

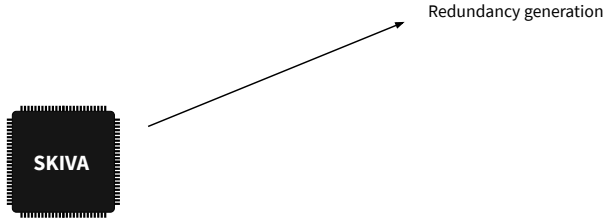


Timing attacks ✓ Data faults ✓ Power analysis ✓ Control faults ✗

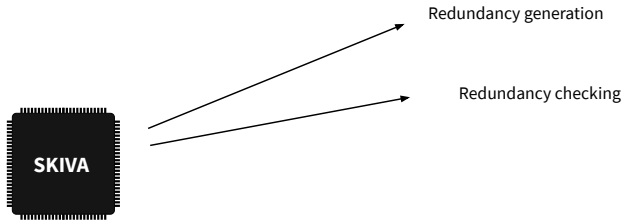
Aggregated bitslice model



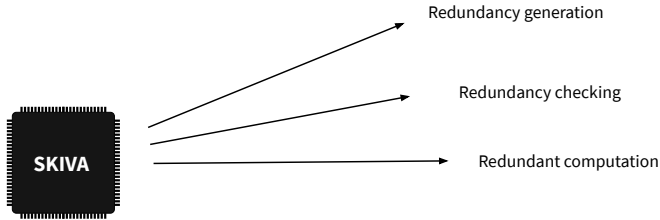
Timing attacks ✓ Data faults ✓ Power analysis ✓ Control faults ✓


Redundancy generation
SKIVA

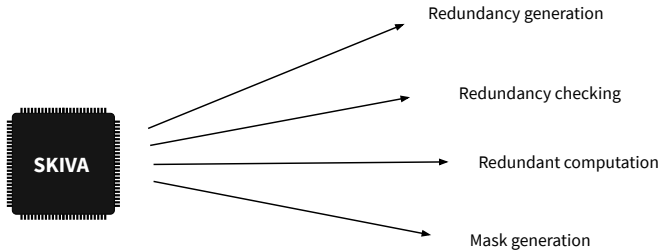
SKIVA



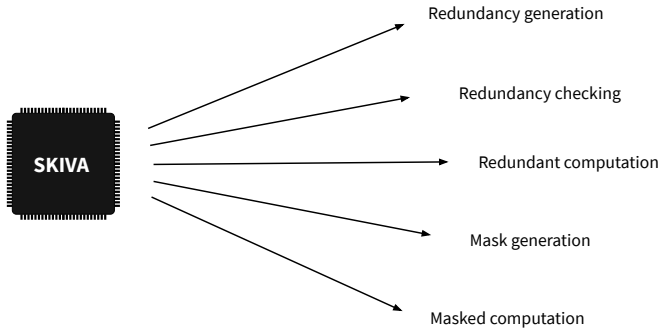
SKIVA


SKIVA
Redundancy generation
Redundancy checking
Redundant computation

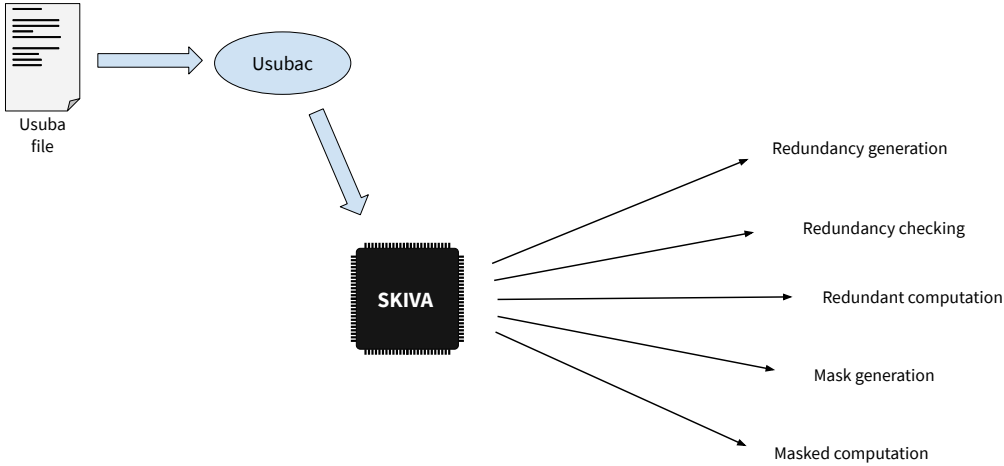
SKIVA


SKIVA
Redundancy generation
Redundancy checking
Redundant computation
Mask generation

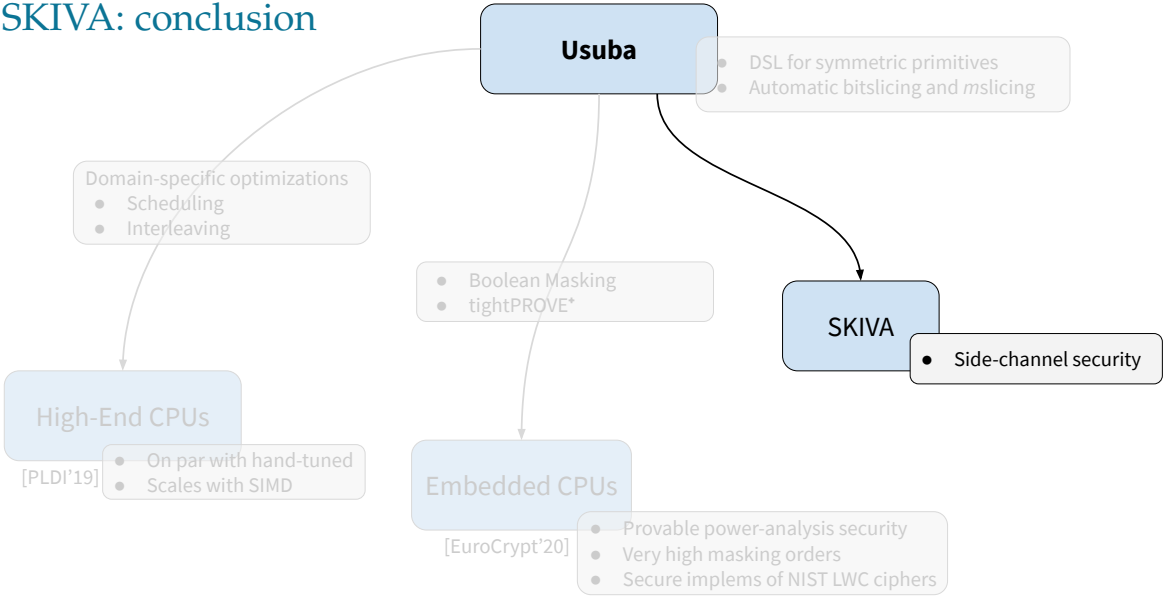
SKIVA


SKIVA
Redundancy generation
Redundancy checking
Redundant computation
Mask generation
Masked computation

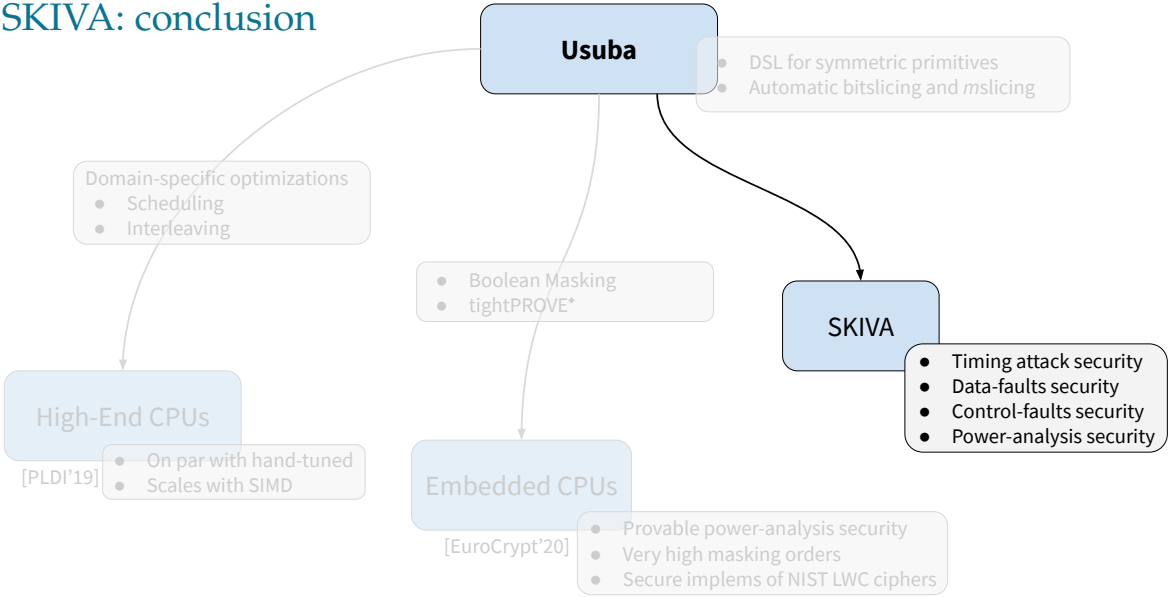
SKIVA


Usuba
file
Usubac
SKIVA
Redundancy generation
Redundancy checking
Redundant computation
Mask generation
Masked computation

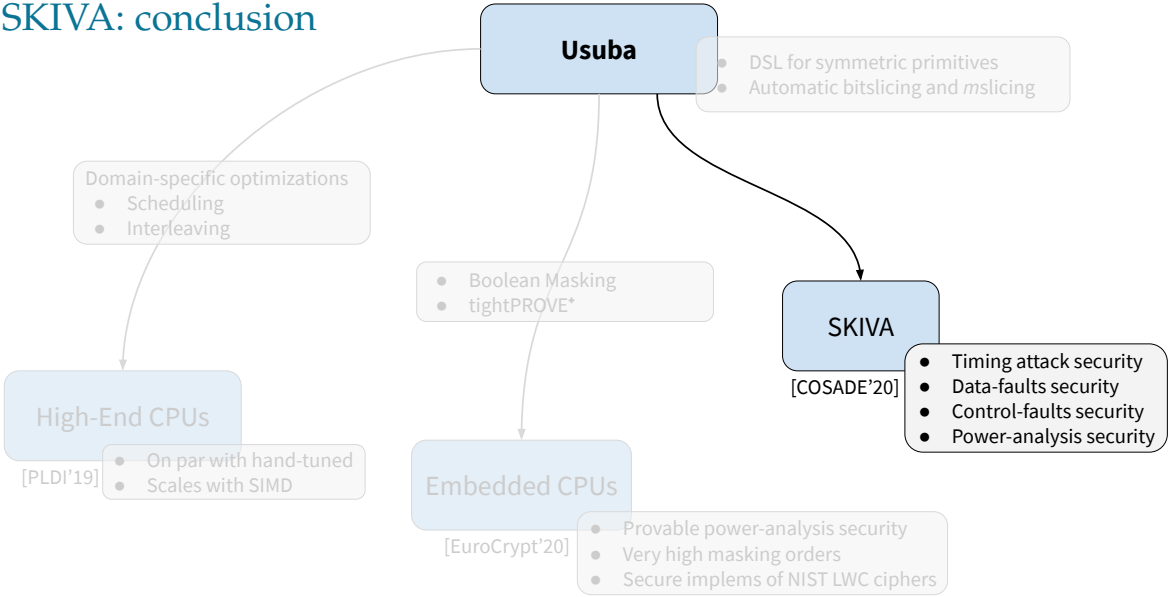
SKIVA: conclusion



SKIVA: conclusion

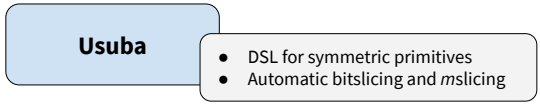


SKIVA: conclusion

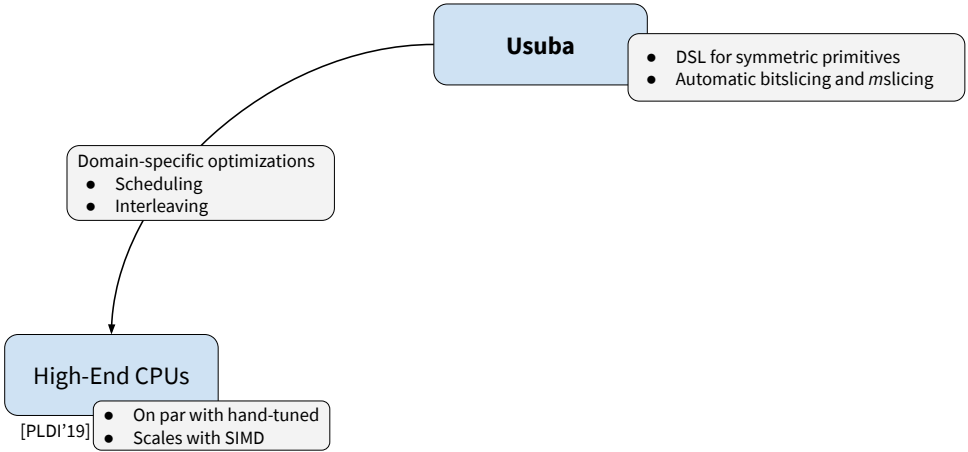


Conclusion

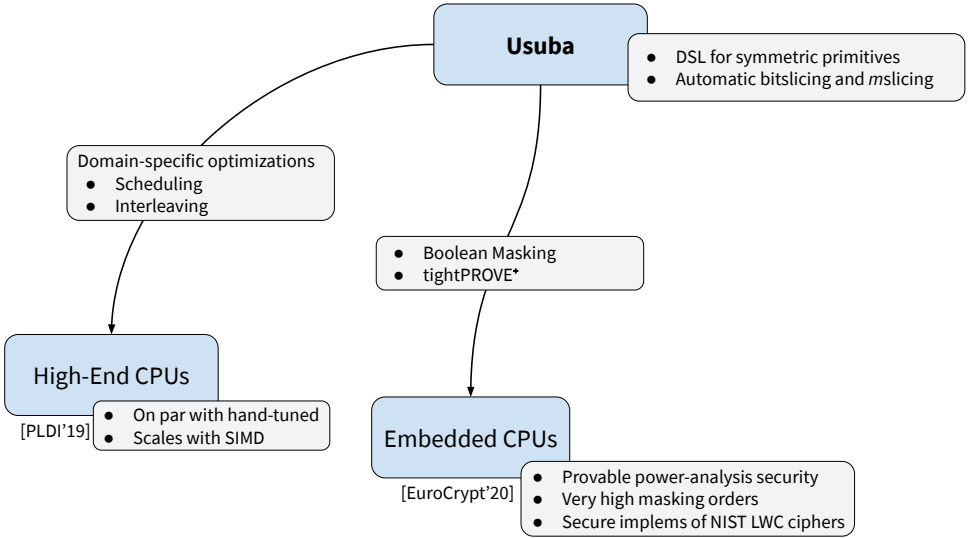
Conclusion


Usuba
DSL for symmetric primitives
Automatic bitslicing and mslicing

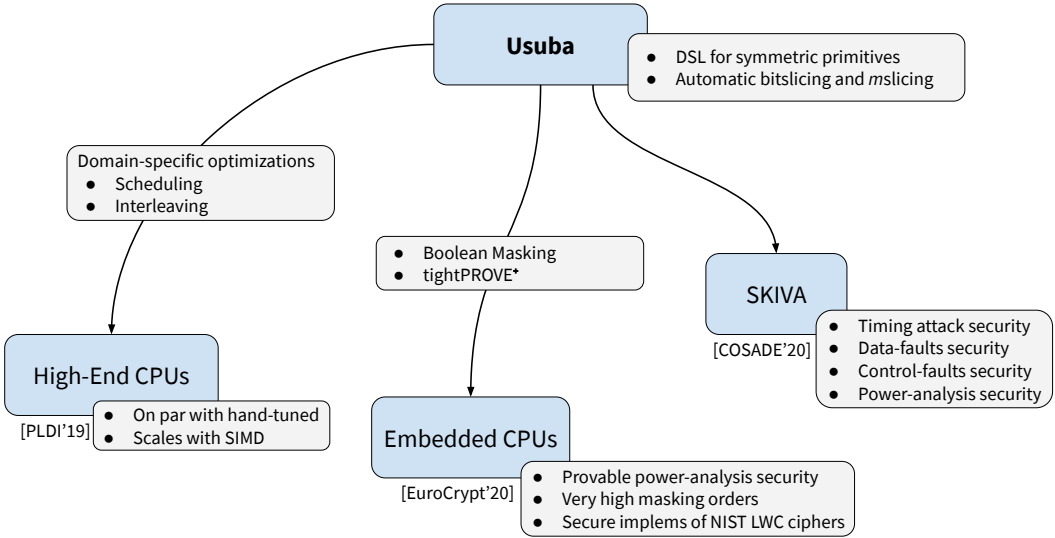
Conclusion


Usuba
DSL for symmetric primitives
Automatic bitslicing and mslicing
Domain-specific optimizations
Scheduling
Interleaving
High-End CPUs
On par with hand-tuned
Scales with SIMD
[PLDI'19]

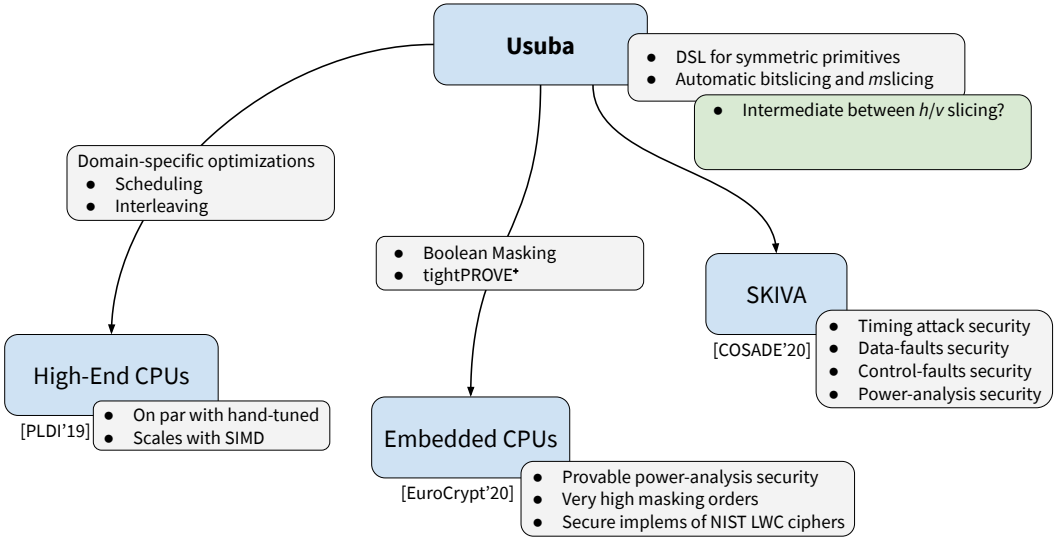
Conclusion


Usuba
DSL for symmetric primitives
Automatic bitslicing and mslicing
Domain-specific optimizations
Scheduling
Interleaving
Boolean Masking
tightPROVE+
High-End CPUs
[PLDI'19]
On par with hand-tuned
Scales with SIMD
Embedded CPUs
[EuroCrypt'20]
Provable power-analysis security
Very high masking orders
Secure implems of NIST LWC ciphers

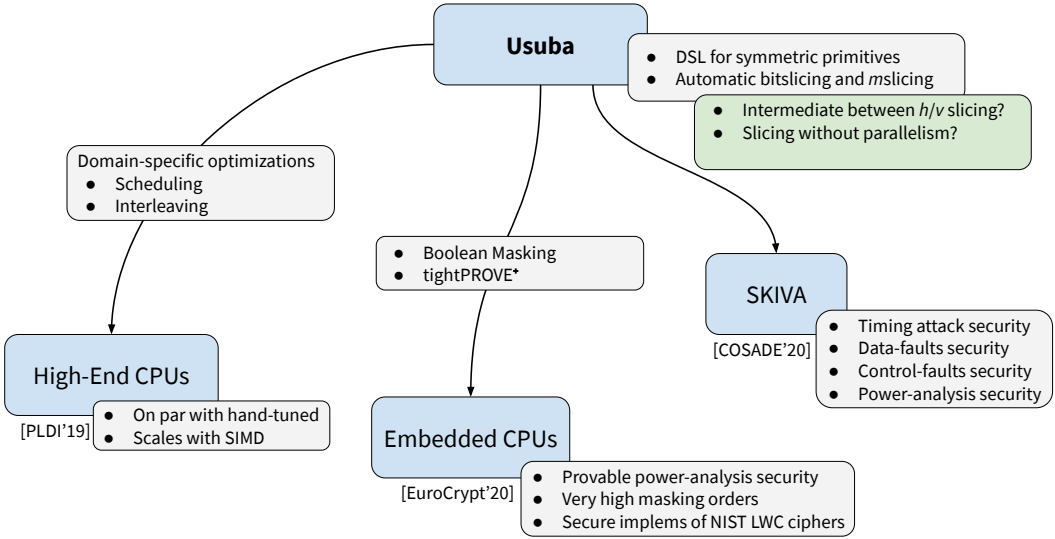
Conclusion


Usuba
DSL for symmetric primitives
Automatic bitslicing and mslicing
Domain-specific optimizations
Scheduling
Interleaving
Boolean Masking
tightPROVE+
SKIVA
[COSADE'20]
Timing attack security
Data-faults security
Control-faults security
Power-analysis security
High-End CPUs
[PLDI'19]
On par with hand-tuned
Scales with SIMD
Embedded CPUs
[EuroCrypt'20]
Provable power-analysis security
Very high masking orders
Secure implems of NIST LWC ciphers

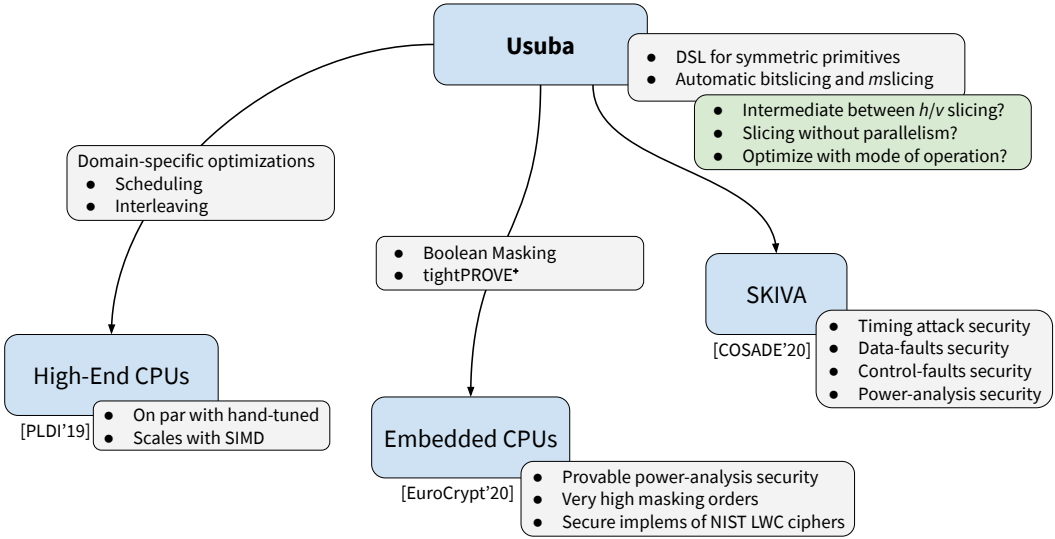
Conclusion


Usuba
DSL for symmetric primitives
Automatic bitslicing and mslicing
Intermediate between h/v slicing?
Domain-specific optimizations
Scheduling
Interleaving
Boolean Masking
tightPROVE+
SKIVA
[COSADE'20]
Timing attack security
Data-faults security
Control-faults security
Power-analysis security
High-End CPUs
[PLDI'19]
On par with hand-tuned
Scales with SIMD
Embedded CPUs
[EuroCrypt'20]
Provable power-analysis security
Very high masking orders
Secure implems of NIST LWC ciphers

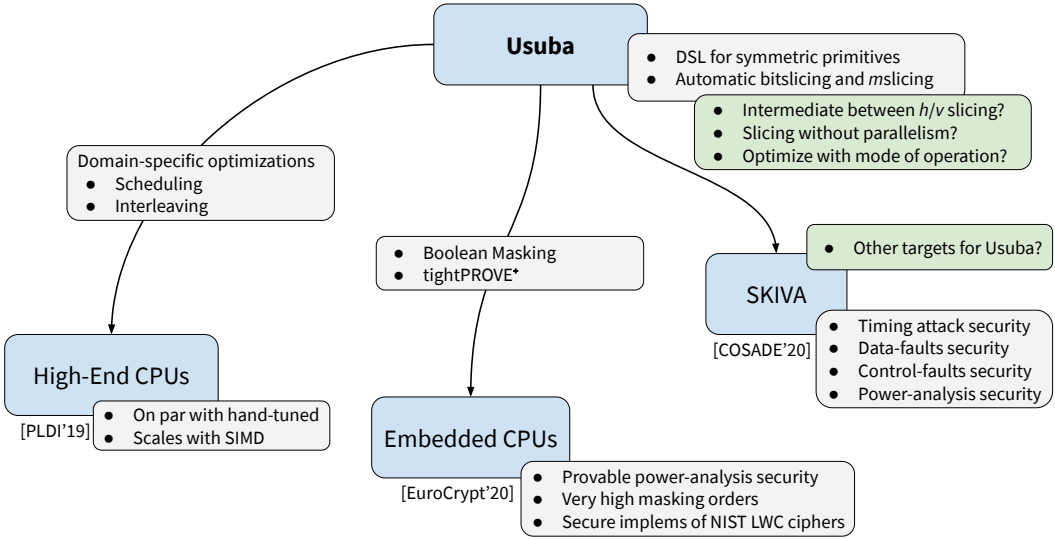
Conclusion


Usuba
DSL for symmetric primitives
Automatic bitslicing and mslicing
Intermediate between h/v slicing?
Slicing without parallelism?
Domain-specific optimizations
Scheduling
Interleaving
Boolean Masking
tightPROVE*
SKIVA
[COSADE'20]
Timing attack security
Data-faults security
Control-faults security
Power-analysis security
High-End CPUs
[PLDI'19]
On par with hand-tuned
Scales with SIMD
Embedded CPUs
[EuroCrypt'20]
Provable power-analysis security
Very high masking orders
Secure implems of NIST LWC ciphers

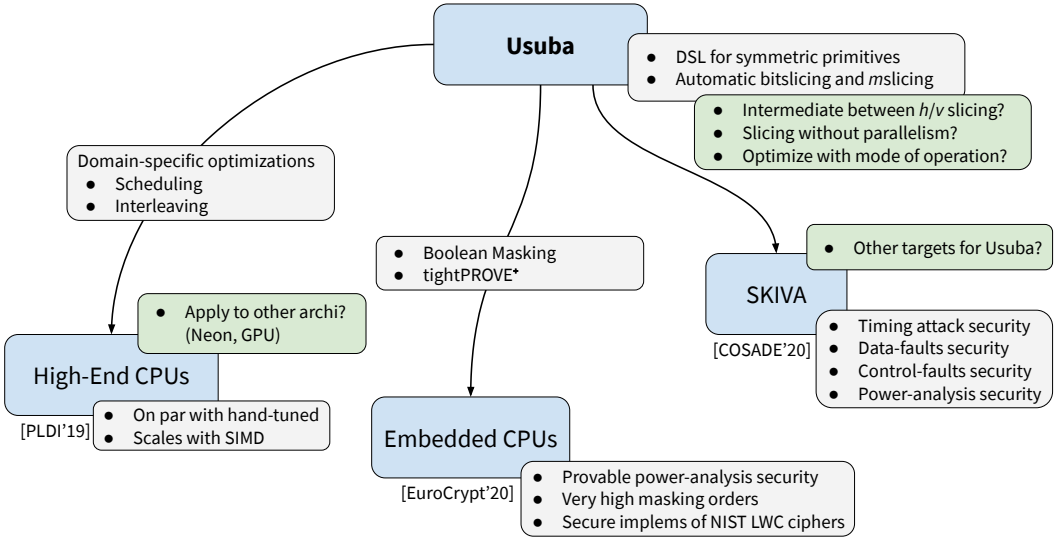
Conclusion


Usuba
DSL for symmetric primitives
Automatic bitslicing and mslicing
Intermediate between h/v slicing?
Slicing without parallelism?
Optimize with mode of operation?
Domain-specific optimizations
Scheduling
Interleaving
Boolean Masking
tightPROVE*
SKIVA
[COSADE'20]
Timing attack security
Data-faults security
Control-faults security
Power-analysis security
High-End CPUs
[PLDI'19]
On par with hand-tuned
Scales with SIMD
Embedded CPUs
[EuroCrypt'20]
Provable power-analysis security
Very high masking orders
Secure implems of NIST LWC ciphers

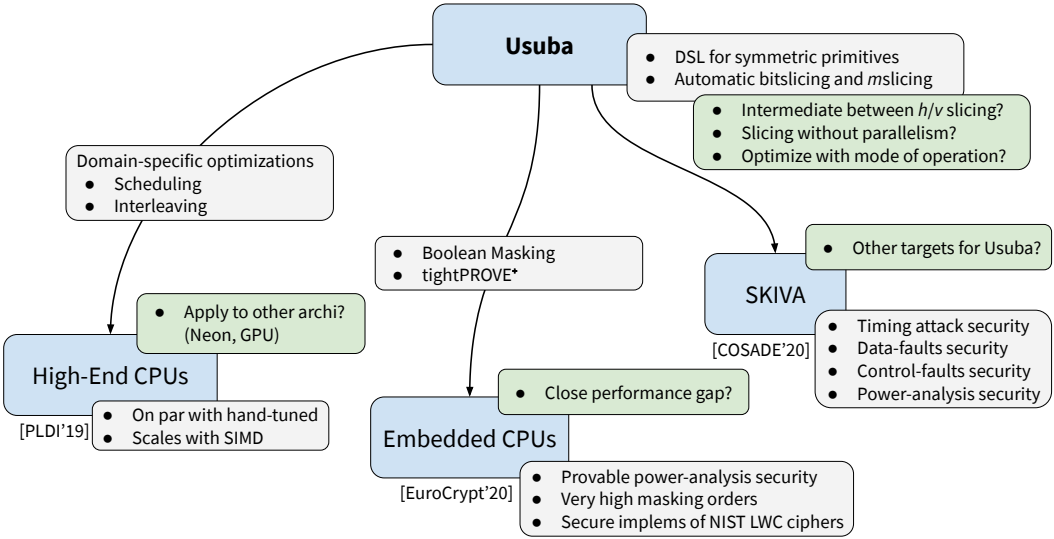
Conclusion


Usuba
DSL for symmetric primitives
Automatic bitslicing and mslicing
Intermediate between h/v slicing?
Slicing without parallelism?
Optimize with mode of operation?
Domain-specific optimizations
Scheduling
Interleaving
Boolean Masking
tightPROVE+
Other targets for Usuba?
SKIVA
[COSADE'20]
Timing attack security
Data-faults security
Control-faults security
Power-analysis security
High-End CPUs
[PLDI'19]
On par with hand-tuned
Scales with SIMD
Embedded CPUs
[EuroCrypt'20]
Provable power-analysis security
Very high masking orders
Secure implems of NIST LWC ciphers

Conclusion


Usuba
DSL for symmetric primitives
Automatic bitslicing and mslicing
Intermediate between h/v slicing?
Slicing without parallelism?
Optimize with mode of operation?
Domain-specific optimizations
Scheduling
Interleaving
Boolean Masking
tightPROVE+
SKIVA
Other targets for Usuba?
Timing attack security
Data-faults security
Control-faults security
Power-analysis security
[COSADE'20]
Apply to other archi? (Neon, GPU)
High-End CPUs
[PLDI'19]
On par with hand-tuned
Scales with SIMD
Embedded CPUs
[EuroCrypt'20]
Provable power-analysis security
Very high masking orders
Secure implems of NIST LWC ciphers

Conclusion


Usuba
DSL for symmetric primitives
Automatic bitslicing and mslicing
Intermediate between h/v slicing?
Slicing without parallelism?
Optimize with mode of operation?
Domain-specific optimizations
Scheduling
Interleaving
Boolean Masking
tightPROVE+
SKIVA
Other targets for Usuba?
Timing attack security
Data-faults security
Control-faults security
Power-analysis security
[COSADE'20]
Apply to other archi? (Neon, GPU)
High-End CPUs
On par with hand-tuned
Scales with SIMD
[PLDI'19]
Close performance gap?
Embedded CPUs
Provable power-analysis security
Very high masking orders
Secure implems of NIST LWC ciphers
[EuroCrypt'20]

References

[Pornin01] T. Pornin, Implantation et optimisation des primitives cryptographiques, Ph.D. thesis, 2001.

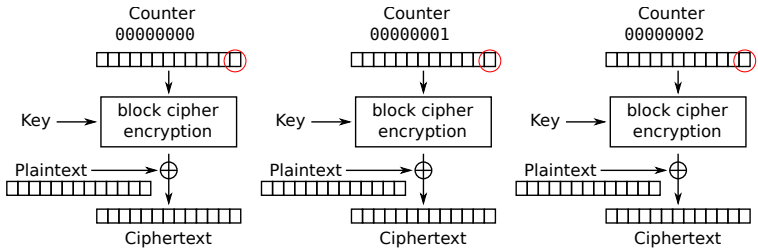
[Kasper09] E. Käsper, P. Schwabe, Faster and Timing-Attack Resistant AES-GCM, CHES, 2009

[Ishai03] Y. Ishai, A. Sahai, A. A. Wagner, Private circuits: Securing hardware against probing attacks, CRYPTO, 2003

Backup slides

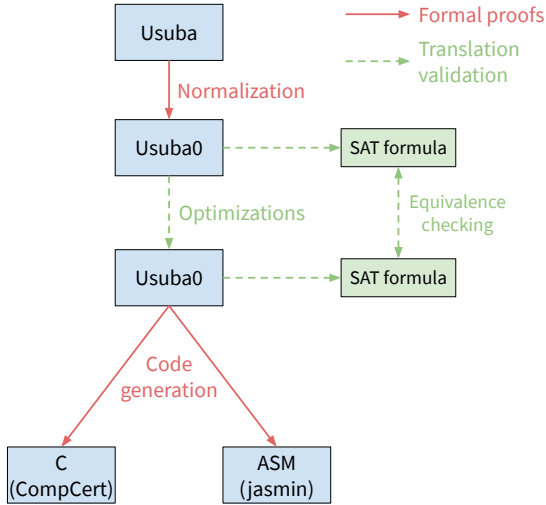
Future work

Optimization: counter caching



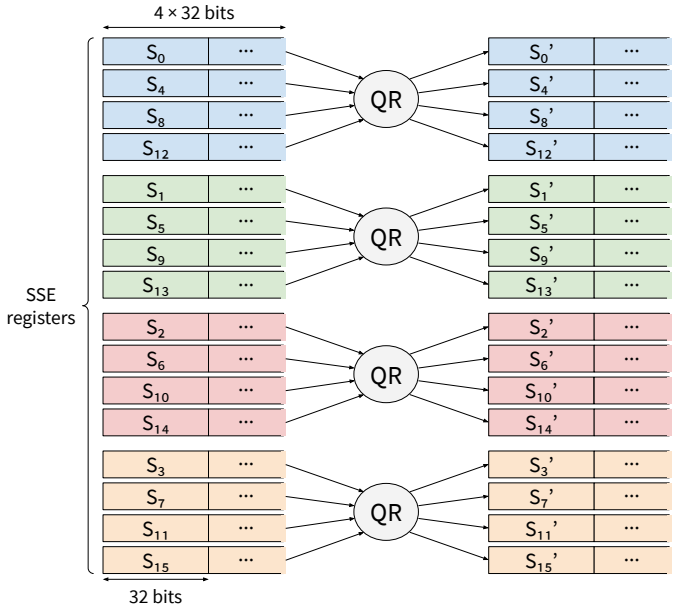
Future work

Verification



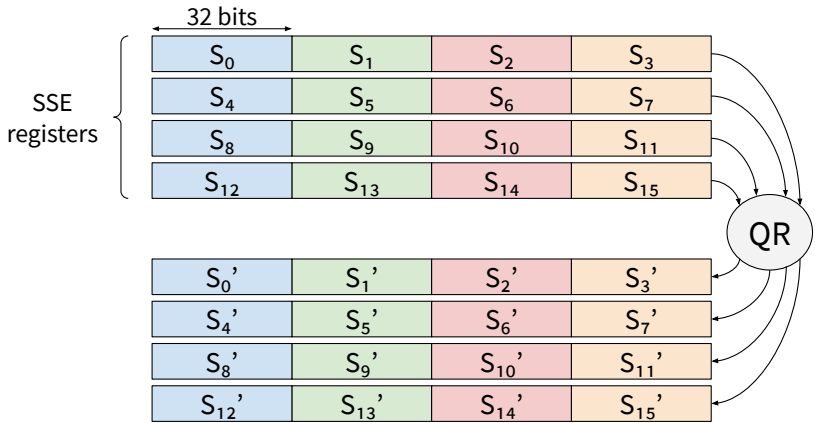
Future work

Hybrid *m*slicing



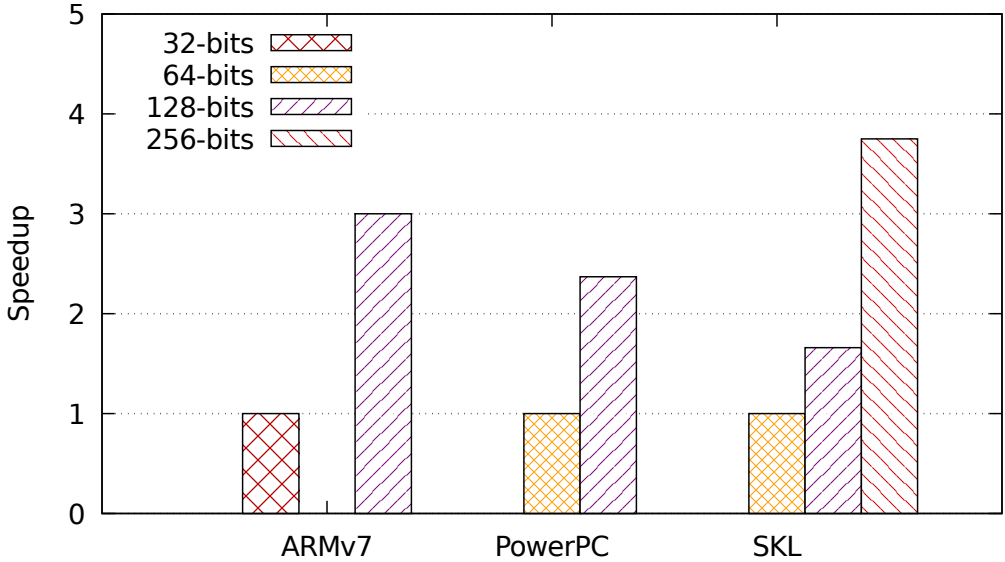
Future work

Hybrid *m*slicing

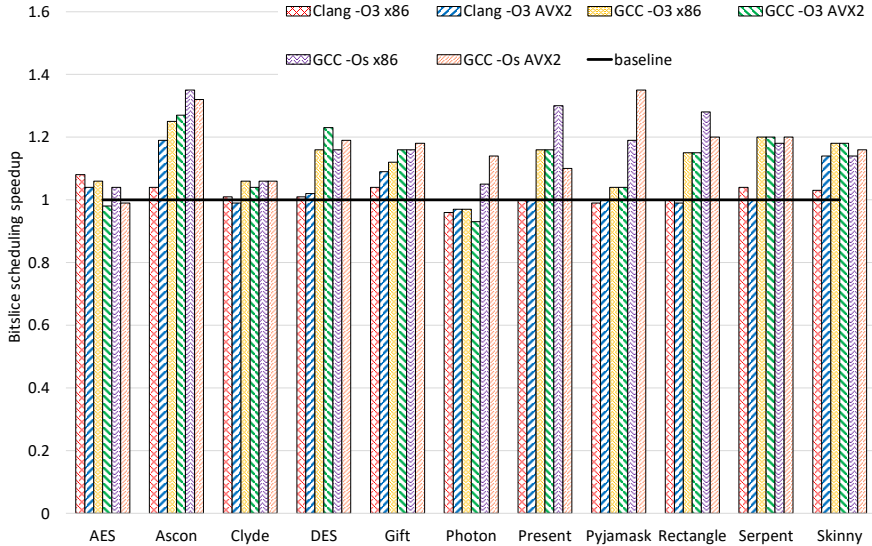


Future work

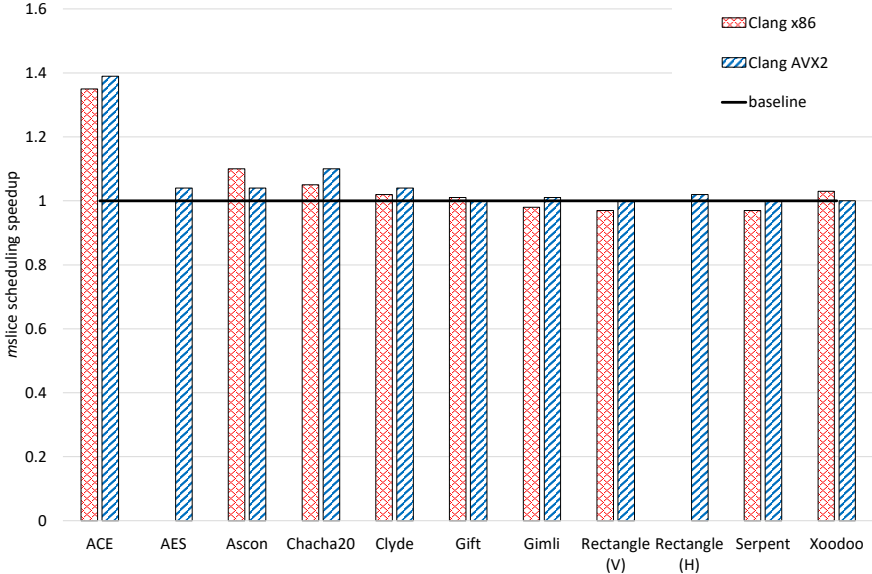
Other SIMD architectures



Bitslice scheduling

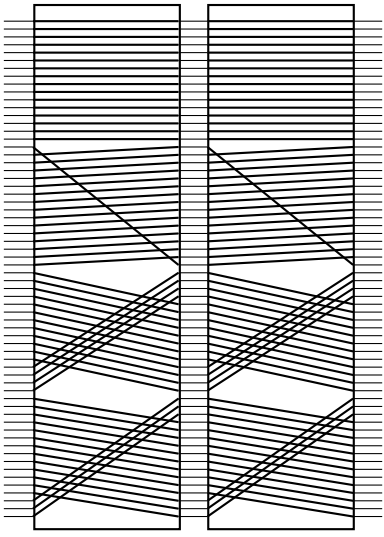


*m*slice scheduling



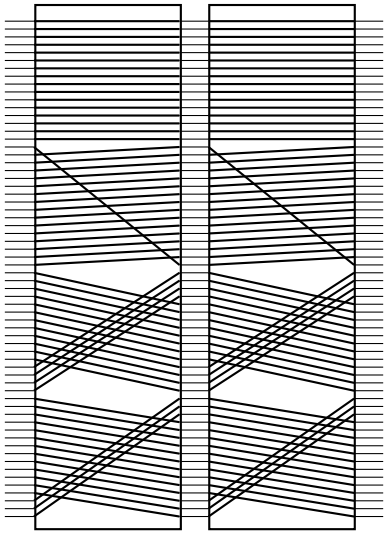
Unrolling & Inlining

```
node ShiftRows_x2 (plain:b64)
             returns (cipher:b64)
let
    forall i in [0,1] {
      plain = ShiftRows(plain)
    }
    cipher = plain
tel
```



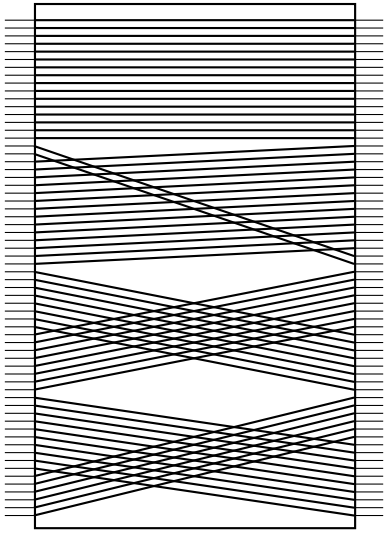
Unrolling & Inlining

```
node ShiftRows_x2 (plain:b64)
            returns (cipher:b64)
let
    forall i in [0,1] {
      tmp[0] = plain[0];
      tmp[1] = plain[1];
      ...
      tmp[16] = plain[17];
      tmp[17] = plain[18];
      ...
      tmp[63] = plain[60];
      plain = tmp;
    }
    cipher = plain
tel
```



Unrolling & Inlining

```
node ShiftRows_x2 (plain:b64)
             returns (cipher:b64)
let
    cipher[0] = plain[0];
    cipher[1] = plain[1];
    ...
    cipher[16] = plain[18];
    cipher[17] = plain[19];
    ...
    cipher[63] = plain[57];
tel
```

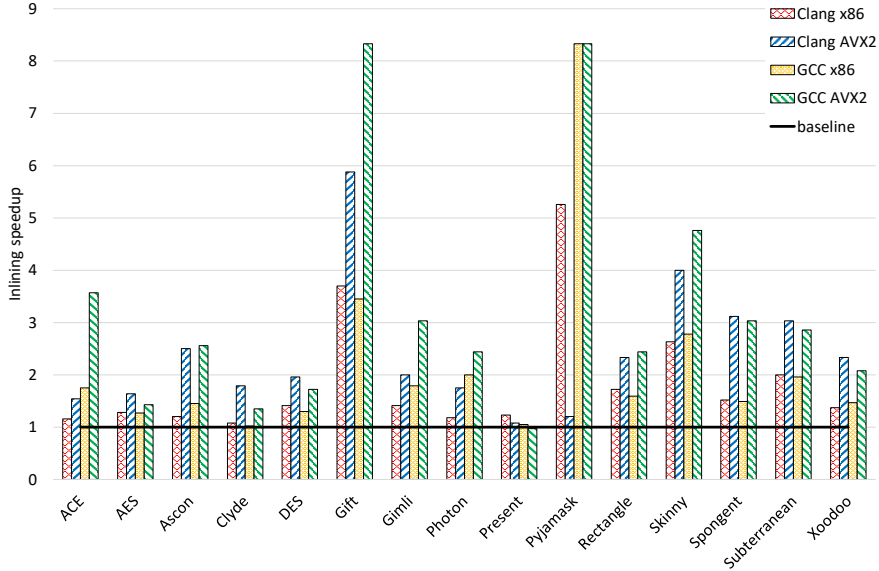


ShiftRows (x2)

Unrolling & Inlining - evaluation

Higher is better

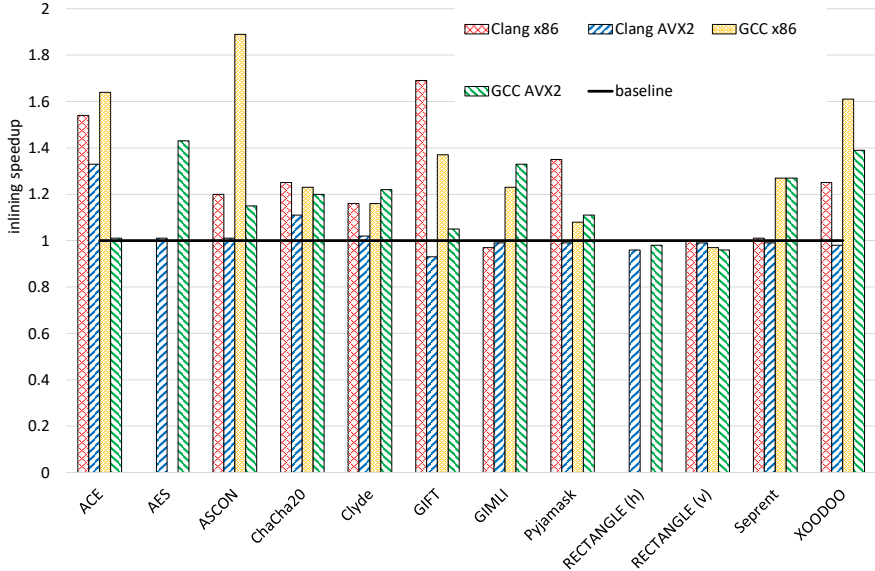
Bitslicing throughput speedup



Unrolling & Inlining - evaluation

Higher is better

*m*slicing throughput speedup



SKIVA

Custom hardware for bitslice aggregation

Countermeasure	# slices	Custom instructions	Semantics

SKIVA

Custom hardware for bitslice aggregation

Countermeasure	# slices	Custom instructions	Semantics
Bitslicing	-	`tr2`	Normal $\rightarrow$ Bitslice
		`invtr2`	Bitslice $\rightarrow$ Normal

SKIVA

Custom hardware for bitslice aggregation

Countermeasure	# slices	Custom instructions	Semantics
Bitslicing	-	`tr2`	Normal $\rightarrow$ Bitslice
		`invtr2`	Bitslice $\rightarrow$ Normal
Masking	1, 2, 4	`subrot2`	2-share rotation
		`subrot4`	4-share rotation

SKIVA

Custom hardware for bitslice aggregation

Countermeasure	# slices	Custom instructions	Semantics
Bitslicing	-	`tr2`	Normal → Bitslice
		`invtr2`	Bitslice → Normal
Masking	1, 2, 4	`subrot2`	2-share rotation
		`subrot4`	4-share rotation
Spatial Redundancy	1, 2, 4	`red`	Redundancy Generation
		`ftchk`	Redundancy Checking
		`andc16/andc8`	Redundant AND
		`xorc16/xorc8`	Redundant XOR

SKIVA

Custom hardware for bitslice aggregation

Countermeasure	# slices	Custom instructions	Semantics
Bitslicing	-	`tr2` `invtr2`	Normal $\rightarrow$ Bitslice Bitslice $\rightarrow$ Normal
Masking	1, 2, 4	`subrot2` `subrot4`	2-share rotation 4-share rotation

Spatial Redundancy	1, 2, 4	`red`
		`ftchk`
		`andc16/andc8`
		`xorc16/xorc8`
			Redundancy Generation
			Redundancy Checking
			Redundant AND
			Redundant XOR

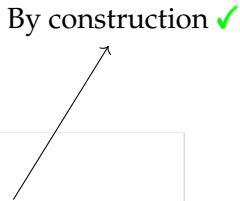
Temporal Redundancy	1, 2	-	-

SKIVA - evaluation

Threat model:

- time-based SCA $\rightarrow$ bitslicing
- electromagnetic SCA $\rightarrow$ higher-order masking
- data faults $\rightarrow$ spatial redundancy
- control faults $\rightarrow$ temporal redundancy

SKIVA - evaluation



Threat model:

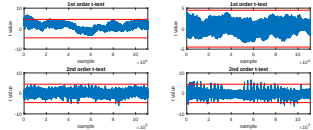
- time-based SCA → bitslicing
- electromagnetic SCA → higher-order masking
- data faults → spatial redundancy
- control faults → temporal redundancy

SKIVA - evaluation

By construction ✓

Threat model:

- time-based SCA → bitslicing
- electromagnetic SCA → higher-order masking
- data faults → spatial redundancy
- control faults → temporal redundancy



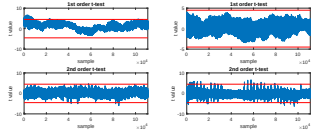
Experimental ✓

SKIVA - evaluation

By construction ✓

Threat model:

- time-based SCA → bitslicing
- electromagnetic SCA → higher-order masking
- data faults → spatial redundancy
- control faults → temporal redundancy



Experimental ✓

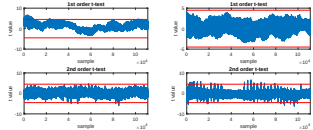
Analytical model ✓

SKIVA - evaluation

Threat model:

- time-based SCA → bitslicing
- electromagnetic SCA → higher-order masking
- data faults → spatial redundancy
- control faults → temporal redundancy

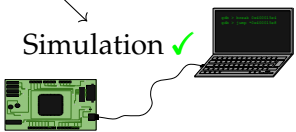
By construction ✓



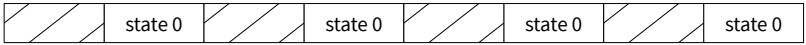
Experimental ✓

Analytical model ✓

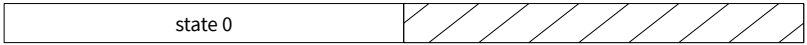
Simulation ✓



Aggregated bitslice model: temporal redundancy

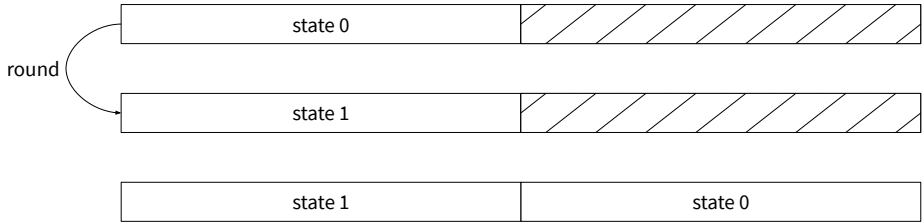


Aggregated bitslice model: temporal redundancy

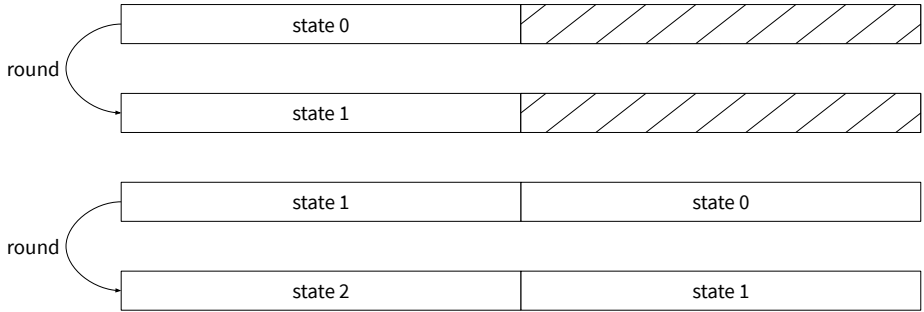


Aggregated bitslice model: temporal redundancy

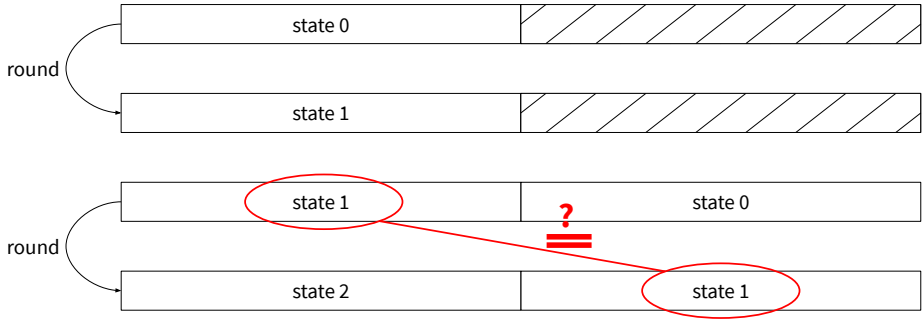
Aggregated bitslice model: temporal redundancy



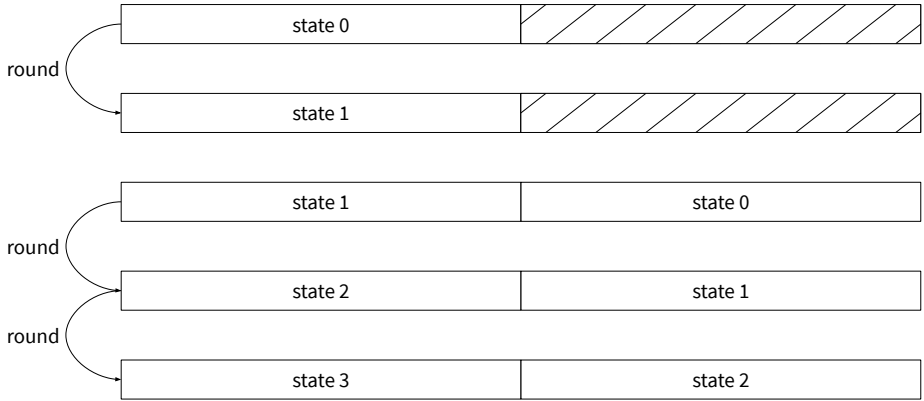
Aggregated bitslice model: temporal redundancy



Aggregated bitslice model: temporal redundancy



Aggregated bitslice model: temporal redundancy



Aggregated bitslice model: temporal redundancy