Implementing Prøst on ARM11

Thom Wiggers

thom@thomwiggers.nl

`https://thomwiggers.nl/proest/`

Institute for Computing and Information Sciences
Radboud University Nijmegen

10th April 2015

Outline

- Introduction
  - Why optimise Prøst
- Prøst
- Optimising on ARM
- Optimising Prøst

Outline

- Introduction
  - Why optimise Prøst
- Prøst
- Optimising on ARM
- Optimising Prøst

What is..?

Authenticated Encryption

Authenticated Encryption is encryption in which you have both:

- confidentiality (nobody else can read this)
- authenticity (nobody else could have produced this message)

ARM11

ARM11 is a CPU architecture used mostly in mobile and embedded devices.

- Smartphones
- Raspberry Pi
- Nintendo 3DS

What is..?

Authenticated Encryption

Authenticated Encryption is encryption in which you have both:

- confidentiality (nobody else can read this)
- authenticity (nobody else could have produced this message)

ARM11

ARM11 is a CPU architecture used mostly in mobile and embedded devices.

- Smartphones
- Raspberry Pi
- Nintendo 3DS

Why optimise Prøst

- CAESAR[1] is an ongoing competition for Authenticated Encryption ciphers.
- "Winners" will be selected based not only on security, but also on performance in both hardware and software.
  - More implementations means judges can better compare ciphers.
- Examples of other competitions:
  - 2000, NIST announce Rijndael selected as the Advanced Encryption Standard (AES).
  - 2012, NIST announce Keccak as winner of the NIST hash function competition (SHA3).

---

[1] *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness.*

Outline

Introduction
    Why optimise Prøst

Prøst

Optimising on ARM

Optimising Prøst

Prøst permutation

PRØST combines the PRØST permutation in various ways to arrive at different modes: COPA, OTR and APE.

The round function R_i where i indicates the round number, is defined as:

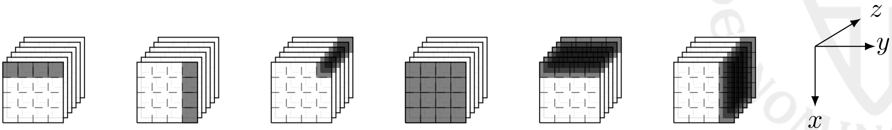
$$R_i(x) = (\texttt{AddConstants}_i \circ \texttt{ShiftPlanes}_i \circ \texttt{MixSlices} \circ \texttt{SubRows})(x).$$

Prøst state

PRØST-128 has a 256 bit state s which is considered as a $4 \times 4 \times 16$ three-dimensional block

$$\mathrm{s} = \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix}$$

where each $s_{x,y}$ is a 16-bit lane.



Row Column Lane Slice Plane Sheet Axes

Nomenclature for state parts[2]

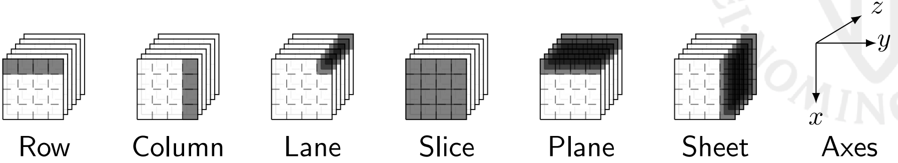
---

[2] Kavun et al. *Prø st v1.1*. 2014.

SubRows

For each row (a, b, c, d) of the state substitute (a', b', c', d') where

$$\begin{aligned} a' &= c \oplus (a \& b), \\ b' &= d \oplus (b \& c), \\ c' &= a \oplus (a' \& b'), \\ d' &= b \oplus (b' \& c'). \end{aligned}$$



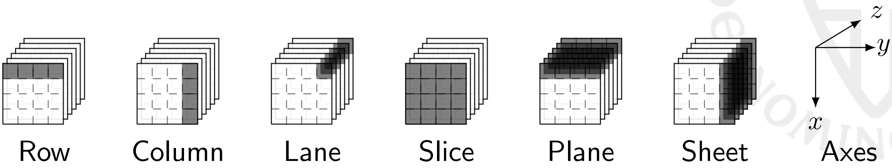
MixSlices

Mix up the slices according to this big thing:

$$
\begin{aligned}
s'_{0,0} &= s_{0,0} \oplus s_{1,0} \oplus s_{1,3} \oplus s_{2,2} \oplus s_{3,0} \oplus s_{3,2} \oplus s_{3,3} \\
s'_{0,1} &= s_{0,1} \oplus s_{1,0} \oplus s_{2,3} \oplus s_{3,0} \oplus s_{3,3} \\
s'_{0,2} &= s_{0,2} \oplus s_{1,1} \oplus s_{2,0} \oplus s_{2,1} \oplus s_{3,0} \\
s'_{0,3} &= s_{0,3} \oplus s_{1,2} \oplus s_{2,1} \oplus s_{2,2} \oplus s_{3,1} \\
s'_{1,0} &= s_{0,0} \oplus s_{0,3} \oplus s_{1,0} \oplus s_{2,0} \oplus s_{2,2} \oplus s_{2,3} \oplus s_{3,2} \\
s'_{1,1} &= s_{0,0} \oplus s_{1,1} \oplus s_{2,0} \oplus s_{2,3} \oplus s_{3,3} \\
s'_{1,2} &= s_{0,1} \oplus s_{1,2} \oplus s_{2,0} \oplus s_{3,0} \oplus s_{3,1} \\
s'_{1,3} &= s_{0,2} \oplus s_{1,3} \oplus s_{2,1} \oplus s_{3,1} \oplus s_{3,2} \\
s'_{2,0} &= s_{0,2} \oplus s_{1,0} \oplus s_{1,2} \oplus s_{1,3} \oplus s_{2,0} \oplus s_{3,0} \oplus s_{3,3} \\
s'_{2,1} &= s_{0,3} \oplus s_{1,0} \oplus s_{1,3} \oplus s_{2,1} \oplus s_{3,0} \\
s'_{2,2} &= s_{0,0} \oplus s_{0,1} \oplus s_{1,0} \oplus s_{2,2} \oplus s_{3,1} \\
s'_{2,3} &= s_{0,1} \oplus s_{0,2} \oplus s_{1,1} \oplus s_{2,3} \oplus s_{3,2} \\
s'_{3,0} &= s_{0,0} \oplus s_{0,2} \oplus s_{0,3} \oplus s_{1,2} \oplus s_{2,0} \oplus s_{2,3} \oplus s_{3,0} \\
s'_{3,1} &= s_{0,0} \oplus s_{0,3} \oplus s_{1,3} \oplus s_{2,0} \oplus s_{3,1} \\
s'_{3,2} &= s_{0,0} \oplus s_{1,0} \oplus s_{1,1} \oplus s_{2,1} \oplus s_{3,2} \\
s'_{3,3} &= s_{0,1} \oplus s_{1,1} \oplus s_{1,2} \oplus s_{2,2} \oplus s_{3,3}
\end{aligned}
$$

ShiftPlanes$_i$

- Shifts the bits in the planes over the z-direction,
- The number of bits rotated differs for odd and even rounds:
  - Even The first, second, third and forth plane are rotated 0, 1, 8 and 9 bits, respectively,
  - Odd The first, second, third and forth plane are rotated 0, 2, 4 and 6 bits, respectively.



AddConstants$_i$

Adds the constants c_1 and c_2, rotated by the round number i and the index of the lane, to the individual lanes.

$$\begin{pmatrix} s'_{0,0} \\ s'_{0,1} \\ s'_{0,2} \\ s'_{0,3} \\ s'_{1,0} \\ \vdots \\ s'_{3,3} \end{pmatrix} = \begin{pmatrix} s_{0,0} \oplus (c_1 \lll i \lll 0) \\ s_{0,1} \oplus (c_2 \lll i \lll 1) \\ s_{0,2} \oplus (c_1 \lll i \lll 2) \\ s_{0,3} \oplus (c_2 \lll i \lll 3) \\ s_{1,0} \oplus (c_1 \lll i \lll 4) \\ \vdots \\ s_{3,3} \oplus (c_2 \lll i \lll 15) \end{pmatrix}$$

Outline

ARM11

- 32-bit architecture
- 14 registers + stack pointer + program counter

Shuffling instructions in the pipeline

The same program can be much faster if it is ordered slightly differently.

```
x1 = mem16[address_a]
x = x1 + 10
x2 = mem16[address_b]
y = x2 + 10
x3 = mem16[address_c]
z = x3 + 10
```

```
x1 = mem16[address_a]
x2 = mem16[address_b]
x3 = mem16[address_c]
x = x1 + 10
y = x2 + 10
z = x3 + 10
```

Shuffling instructions in the pipeline

The same program can be much faster if it is ordered slightly differently.

Cycle: 1

```
x1 = mem16[address_a]
x = x1 + 10
x2 = mem16[address_b]
y = x2 + 10
x3 = mem16[address_c]
z = x3 + 10
```

```
x1 = mem16[address_a]
x2 = mem16[address_b]
x3 = mem16[address_c]
x = x1 + 10
y = x2 + 10
z = x3 + 10
```

Shuffling instructions in the pipeline

The same program can be much faster if it is ordered slightly differently.
Cycle: 2

```
x1 = mem16[address_a]
x = x1 + 10 # waiting...
x2 = mem16[address_b]
y = x2 + 10
x3 = mem16[address_c]
z = x3 + 10
```

```
x1 = mem16[address_a]
x2 = mem16[address_b]
x3 = mem16[address_c]
x = x1 + 10
y = x2 + 10
z = x3 + 10
```

Shuffling instructions in the pipeline

The same program can be much faster if it is ordered slightly differently.
Cycle: 3

```
x1 = mem16[address_a]
x = x1 + 10 # waiting...
x2 = mem16[address_b]
y = x2 + 10
x3 = mem16[address_c]
z = x3 + 10
```

```
x1 = mem16[address_a]
x2 = mem16[address_b]
x3 = mem16[address_c]
x = x1 + 10
y = x2 + 10
z = x3 + 10
```

Shuffling instructions in the pipeline

The same program can be much faster if it is ordered slightly differently.

Cycle: 4

```
x1 = mem16[address_a]
x = x1 + 10
x2 = mem16[address_b]
y = x2 + 10
x3 = mem16[address_c]
z = x3 + 10
```

```
x1 = mem16[address_a]
x2 = mem16[address_b]
x3 = mem16[address_c]
x = x1 + 10
y = x2 + 10
z = x3 + 10
```

Shuffling instructions in the pipeline

The same program can be much faster if it is ordered slightly differently.
Cycle: 5

```
x1 = mem16[address_a]
x = x1 + 10
x2 = mem16[address_b]
y = x2 + 10
x3 = mem16[address_c]
z = x3 + 10
```

```
x1 = mem16[address_a]
x2 = mem16[address_b]
x3 = mem16[address_c]
x = x1 + 10
y = x2 + 10
z = x3 + 10
```

Shuffling instructions in the pipeline

The same program can be much faster if it is ordered slightly differently.

Cycle: 6

```
x1 = mem16[address_a]
x = x1 + 10
x2 = mem16[address_b]
y = x2 + 10 # waiting...
x3 = mem16[address_c]
z = x3 + 10
```

```
x1 = mem16[address_a]
x2 = mem16[address_b]
x3 = mem16[address_c]
x = x1 + 10
y = x2 + 10
z = x3 + 10
```

Shuffling instructions in the pipeline

The same program can be much faster if it is ordered slightly differently.

Cycle: 7

```
x1 = mem16[address_a]
x = x1 + 10
x2 = mem16[address_b]
y = x2 + 10 # waiting...
x3 = mem16[address_c]
z = x3 + 10
```

```
x1 = mem16[address_a]
x2 = mem16[address_b]
x3 = mem16[address_c]
x = x1 + 10
y = x2 + 10
z = x3 + 10
# done after 6 cycles!
```

Shuffling instructions in the pipeline

The same program can be much faster if it is ordered slightly differently.

Cycle: 8

```
x1 = mem16[address_a]
x = x1 + 10
x2 = mem16[address_b]
y = x2 + 10
x3 = mem16[address_c]
z = x3 + 10
```

```
x1 = mem16[address_a]
x2 = mem16[address_b]
x3 = mem16[address_c]
x = x1 + 10
y = x2 + 10
z = x3 + 10
# done after 6 cycles!
```

Shuffling instructions in the pipeline

The same program can be much faster if it is ordered slightly differently.
Cycle: 9

```
x1 = mem16[address_a]
x = x1 + 10
x2 = mem16[address_b]
y = x2 + 10
x3 = mem16[address_c]
z = x3 + 10
```

```
x1 = mem16[address_a]
x2 = mem16[address_b]
x3 = mem16[address_c]
x = x1 + 10
y = x2 + 10
z = x3 + 10
# done after 6 cycles!
```

Shuffling instructions in the pipeline

The same program can be much faster if it is ordered slightly differently.

Cycle: 10

```
x1 = mem16[address_a]
x = x1 + 10
x2 = mem16[address_b]
y = x2 + 10
x3 = mem16[address_c]
z = x3 + 10 # waiting...
```

```
x1 = mem16[address_a]
x2 = mem16[address_b]
x3 = mem16[address_c]
x = x1 + 10
y = x2 + 10
z = x3 + 10
# done after 6 cycles!
```

Shuffling instructions in the pipeline

The same program can be much faster if it is ordered slightly differently.

Cycle: 11

```
x1 = mem16[address_a]
x = x1 + 10
x2 = mem16[address_b]
y = x2 + 10
x3 = mem16[address_c]
z = x3 + 10 # waiting...
```

```
x1 = mem16[address_a]
x2 = mem16[address_b]
x3 = mem16[address_c]
x = x1 + 10
y = x2 + 10
z = x3 + 10
# done after 6 cycles!
```

Shuffling instructions in the pipeline

The same program can be much faster if it is ordered slightly differently.
Cycle: 12

```
x1 = mem16[address_a]
x = x1 + 10
x2 = mem16[address_b]
y = x2 + 10
x3 = mem16[address_c]
z = x3 + 10
```

```
x1 = mem16[address_a]
x2 = mem16[address_b]
x3 = mem16[address_c]
x = x1 + 10
y = x2 + 10
z = x3 + 10
# done after 6 cycles!
```

Shuffling instructions in the pipeline

The same program can be much faster if it is ordered slightly differently.

```
x1 = mem16[address_a]
x = x1 + 10
x2 = mem16[address_b]
y = x2 + 10
x3 = mem16[address_c]
z = x3 + 10
# done after 12 cycles
```

```
x1 = mem16[address_a]
x2 = mem16[address_b]
x3 = mem16[address_c]
x = x1 + 10
y = x2 + 10
z = x3 + 10
# done after 6 cycles!
```

Free shifts and rotations

ARM support rotating and shifting one of the inputs to most arithmetic operations.

$$a \leftarrow b \odot (c \ggg n)$$

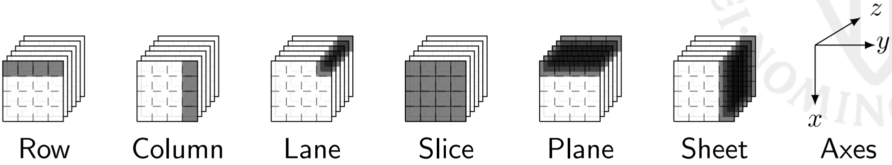
Outline

- Introduction
  - Why optimise Prøst
- Prøst
- Optimising on ARM
- Optimising Prøst

SubRows

For each row (a, b, c, d) of the state substitute (a', b', c', d') where

$$\begin{aligned} a' &= c \oplus (a \& b), \\ b' &= d \oplus (b \& c), \\ c' &= a \oplus (a' \& b'), \\ d' &= b \oplus (b' \& c'). \end{aligned}$$



SubRows

Lanes are 16 bits, but our registers are 32 bits. . .
We can load two lanes into one register in one load instruction.

```
a_and_b = mem32[address_of_s]
# b is in the upper part of a_and_b
c_and_d = mem32[address_of_s + 4]
# a' = c ^ (a & b)
newa  = a_and_b & (a_and_b >>> 16)
newa ^= c_and_d
mem16[address_of_s] = newa
```

SubRows

Lanes are 16 bits, but our registers are 32 bits. . .
We can load two lanes into one register in one load instruction.

```
a_and_b = mem32[address_of_s]
# b is in the upper part of a_and_b
c_and_d = mem32[address_of_s + 4]
# a’ = c ˆ (a & b)
newa  = a_and_b & (a_and_b >>> 16)
newa ˆ= c_and_d
mem16[address_of_s] = newa
```

SubRows

Lanes are 16 bits, but our registers are 32 bits. . .
We can load two lanes into one register in one load instruction.

```
a_and_b = mem32[address_of_s]
# b is in the upper part of a_and_b
c_and_d = mem32[address_of_s + 4]
# a' = c ^ (a & b)
newa  = a_and_b & (a_and_b >>> 16)
newa ^= c_and_d
mem16[address_of_s] = newa
```

SubRows

Lanes are 16 bits, but our registers are 32 bits. . .
We can load two lanes into one register in one load instruction.

```
a_and_b = mem32[address_of_s]
# b is in the upper part of a_and_b
c_and_d = mem32[address_of_s + 4]
# a’ = c ˆ (a & b)
newa  = a_and_b & (a_and_b >>> 16)
newa ˆ= c_and_d
mem16[address_of_s] = newa
```

SubRows

Lanes are 16 bits, but our registers are 32 bits. . .
We can load two lanes into one register in one load instruction.

```
a_and_b = mem32[address_of_s]
# b is in the upper part of a_and_b
c_and_d = mem32[address_of_s + 4]
# a' = c ^ (a & b)
newa  = a_and_b & (a_and_b >>> 16)
newa ^= c_and_d
mem16[address_of_s] = newa
```

MixSlices

Mix up the slices according to this big thing:

$$
\begin{aligned}
s'_{0,0} &= s_{0,0} \oplus s_{1,0} \oplus s_{1,3} \oplus s_{2,2} \oplus s_{3,0} \oplus s_{3,2} \oplus s_{3,3} \\
s'_{0,1} &= s_{0,1} \oplus s_{1,0} \oplus s_{2,3} \oplus s_{3,0} \oplus s_{3,3} \\
s'_{0,2} &= s_{0,2} \oplus s_{1,1} \oplus s_{2,0} \oplus s_{2,1} \oplus s_{3,0} \\
s'_{0,3} &= s_{0,3} \oplus s_{1,2} \oplus s_{2,1} \oplus s_{2,2} \oplus s_{3,1} \\
s'_{1,0} &= s_{0,0} \oplus s_{0,3} \oplus s_{1,0} \oplus s_{2,0} \oplus s_{2,2} \oplus s_{2,3} \oplus s_{3,2} \\
s'_{1,1} &= s_{0,0} \oplus s_{1,1} \oplus s_{2,0} \oplus s_{2,3} \oplus s_{3,3} \\
s'_{1,2} &= s_{0,1} \oplus s_{1,2} \oplus s_{2,0} \oplus s_{3,0} \oplus s_{3,1} \\
s'_{1,3} &= s_{0,2} \oplus s_{1,3} \oplus s_{2,1} \oplus s_{3,1} \oplus s_{3,2} \\
s'_{2,0} &= s_{0,2} \oplus s_{1,0} \oplus s_{1,2} \oplus s_{1,3} \oplus s_{2,0} \oplus s_{3,0} \oplus s_{3,3} \\
s'_{2,1} &= s_{0,3} \oplus s_{1,0} \oplus s_{1,3} \oplus s_{2,1} \oplus s_{3,0} \\
s'_{2,2} &= s_{0,0} \oplus s_{0,1} \oplus s_{1,0} \oplus s_{2,2} \oplus s_{3,1} \\
s'_{2,3} &= s_{0,1} \oplus s_{0,2} \oplus s_{1,1} \oplus s_{2,3} \oplus s_{3,2} \\
s'_{3,0} &= s_{0,0} \oplus s_{0,2} \oplus s_{0,3} \oplus s_{1,2} \oplus s_{2,0} \oplus s_{2,3} \oplus s_{3,0} \\
s'_{3,1} &= s_{0,0} \oplus s_{0,3} \oplus s_{1,3} \oplus s_{2,0} \oplus s_{3,1} \\
s'_{3,2} &= s_{0,0} \oplus s_{1,0} \oplus s_{1,1} \oplus s_{2,1} \oplus s_{3,2} \\
s'_{3,3} &= s_{0,1} \oplus s_{1,1} \oplus s_{1,2} \oplus s_{2,2} \oplus s_{3,3}
\end{aligned}
$$

MixSlices

Mix up the slices according to this big thing:

$$\begin{aligned}
s'_{0,0} &= s_{0,0} \oplus {\color{red}s_{1,0}} \oplus s_{1,3} \oplus s_{2,2} \oplus s_{3,0} \oplus s_{3,2} \oplus {\color{red}s_{3,3}} \\
s'_{0,1} &= s_{0,1} \oplus {\color{red}s_{1,0}} \oplus s_{2,3} \oplus s_{3,0} \oplus {\color{red}s_{3,3}} \\
s'_{0,2} &= s_{0,2} \oplus s_{1,1} \oplus s_{2,0} \oplus s_{2,1} \oplus s_{3,0} \\
s'_{0,3} &= s_{0,3} \oplus s_{1,2} \oplus s_{2,1} \oplus s_{2,2} \oplus s_{3,1} \\
s'_{1,0} &= s_{0,0} \oplus s_{0,3} \oplus s_{1,0} \oplus s_{2,0} \oplus s_{2,2} \oplus s_{2,3} \oplus s_{3,2} \\
s'_{1,1} &= s_{0,0} \oplus s_{1,1} \oplus s_{2,0} \oplus s_{2,3} \oplus s_{3,3} \\
s'_{1,2} &= s_{0,1} \oplus s_{1,2} \oplus s_{2,0} \oplus s_{3,0} \oplus s_{3,1} \\
s'_{1,3} &= s_{0,2} \oplus s_{1,3} \oplus s_{2,1} \oplus s_{3,1} \oplus s_{3,2} \\
s'_{2,0} &= s_{0,2} \oplus {\color{red}s_{1,0}} \oplus s_{1,2} \oplus s_{1,3} \oplus s_{2,0} \oplus s_{3,0} \oplus {\color{red}s_{3,3}} \\
s'_{2,1} &= s_{0,3} \oplus s_{1,0} \oplus s_{1,3} \oplus s_{2,1} \oplus s_{3,0} \\
s'_{2,2} &= s_{0,0} \oplus s_{0,1} \oplus s_{1,0} \oplus s_{2,2} \oplus s_{3,1} \\
s'_{2,3} &= s_{0,1} \oplus s_{0,2} \oplus s_{1,1} \oplus s_{2,3} \oplus s_{3,2} \\
s'_{3,0} &= s_{0,0} \oplus s_{0,2} \oplus s_{0,3} \oplus s_{1,2} \oplus s_{2,0} \oplus s_{2,3} \oplus s_{3,0} \\
s'_{3,1} &= s_{0,0} \oplus s_{0,3} \oplus s_{1,3} \oplus s_{2,0} \oplus s_{3,1} \\
s'_{3,2} &= s_{0,0} \oplus s_{1,0} \oplus s_{1,1} \oplus s_{2,1} \oplus s_{3,2} \\
s'_{3,3} &= s_{0,1} \oplus s_{1,1} \oplus s_{1,2} \oplus s_{2,2} \oplus s_{3,3}
\end{aligned}$$

MixSlices

Mix up the slices according to this big thing:

$$\begin{aligned}
s'_{0,0} &= s_{0,0} \oplus s_{1,0} \oplus s_{1,3} \oplus s_{2,2} \oplus s_{3,0} \oplus s_{3,2} \oplus s_{3,3} \\
s'_{0,1} &= s_{0,1} \oplus s_{1,0} \oplus s_{2,3} \oplus s_{3,0} \oplus s_{3,3} \\
s'_{0,2} &= s_{0,2} \oplus s_{1,1} \oplus s_{2,0} \oplus s_{2,1} \oplus s_{3,0} \\
s'_{0,3} &= s_{0,3} \oplus s_{1,2} \oplus s_{2,1} \oplus s_{2,2} \oplus s_{3,1} \\
s'_{1,0} &= s_{0,0} \oplus s_{0,3} \oplus s_{1,0} \oplus s_{2,0} \oplus s_{2,2} \oplus s_{2,3} \oplus s_{3,2} \\
s'_{1,1} &= s_{0,0} \oplus s_{1,1} \oplus s_{2,0} \oplus s_{2,3} \oplus s_{3,3} \\
s'_{1,2} &= s_{0,1} \oplus s_{1,2} \oplus s_{2,0} \oplus s_{3,0} \oplus s_{3,1} \\
s'_{1,3} &= s_{0,2} \oplus s_{1,3} \oplus s_{2,1} \oplus s_{3,1} \oplus s_{3,2} \\
s'_{2,0} &= s_{0,2} \oplus s_{1,0} \oplus s_{1,2} \oplus s_{1,3} \oplus s_{2,0} \oplus s_{3,0} \oplus s_{3,3} \\
s'_{2,1} &= s_{0,3} \oplus s_{1,0} \oplus s_{1,3} \oplus s_{2,1} \oplus s_{3,0} \\
s'_{2,2} &= s_{0,0} \oplus s_{0,1} \oplus s_{1,0} \oplus s_{2,2} \oplus s_{3,1} \\
s'_{2,3} &= s_{0,1} \oplus s_{0,2} \oplus s_{1,1} \oplus s_{2,3} \oplus s_{3,2} \\
s'_{3,0} &= s_{0,0} \oplus s_{0,2} \oplus s_{0,3} \oplus s_{1,2} \oplus s_{2,0} \oplus s_{2,3} \oplus s_{3,0} \\
s'_{3,1} &= s_{0,0} \oplus s_{0,3} \oplus s_{1,3} \oplus s_{2,0} \oplus s_{3,1} \\
s'_{3,2} &= s_{0,0} \oplus s_{1,0} \oplus s_{1,1} \oplus s_{2,1} \oplus s_{3,2} \\
s'_{3,3} &= s_{0,1} \oplus s_{1,1} \oplus s_{1,2} \oplus s_{2,2} \oplus s_{3,3}
\end{aligned}$$

MixSlices

Mix up the slices according to this big thing:

$$\begin{aligned}
s'_{0,0} &= s_{0,0} \oplus s_{1,0} \oplus s_{1,3} \oplus s_{2,2} \oplus s_{3,0} \oplus s_{3,2} \oplus s_{3,3} \\
s'_{0,1} &= s_{0,1} \oplus s_{1,0} \oplus s_{2,3} \oplus s_{3,0} \oplus s_{3,3} \\
s'_{0,2} &= s_{0,2} \oplus s_{1,1} \oplus s_{2,0} \oplus s_{2,1} \oplus s_{3,0} \\
s'_{0,3} &= s_{0,3} \oplus s_{1,2} \oplus s_{2,1} \oplus s_{2,2} \oplus s_{3,1} \\
s'_{1,0} &= s_{0,0} \oplus s_{0,3} \oplus s_{1,0} \oplus s_{2,0} \oplus s_{2,2} \oplus s_{2,3} \oplus s_{3,2} \\
s'_{1,1} &= s_{0,0} \oplus s_{1,1} \oplus s_{2,0} \oplus s_{2,3} \oplus s_{3,3} \\
s'_{1,2} &= s_{0,1} \oplus s_{1,2} \oplus s_{2,0} \oplus s_{3,0} \oplus s_{3,1} \\
s'_{1,3} &= s_{0,2} \oplus s_{1,3} \oplus s_{2,1} \oplus s_{3,1} \oplus s_{3,2} \\
s'_{2,0} &= s_{0,2} \oplus s_{1,0} \oplus s_{1,2} \oplus s_{1,3} \oplus s_{2,0} \oplus s_{3,0} \oplus s_{3,3} \\
s'_{2,1} &= s_{0,3} \oplus s_{1,0} \oplus s_{1,3} \oplus s_{2,1} \oplus s_{3,0} \\
s'_{2,2} &= s_{0,0} \oplus s_{0,1} \oplus s_{1,0} \oplus s_{2,2} \oplus s_{3,1} \\
s'_{2,3} &= s_{0,1} \oplus s_{0,2} \oplus s_{1,1} \oplus s_{2,3} \oplus s_{3,2} \\
s'_{3,0} &= s_{0,0} \oplus s_{0,2} \oplus s_{0,3} \oplus s_{1,2} \oplus s_{2,0} \oplus s_{2,3} \oplus s_{3,0} \\
s'_{3,1} &= s_{0,0} \oplus s_{0,3} \oplus s_{1,3} \oplus s_{2,0} \oplus s_{3,1} \\
s'_{3,2} &= s_{0,0} \oplus s_{1,0} \oplus s_{1,1} \oplus s_{2,1} \oplus s_{3,2} \\
s'_{3,3} &= s_{0,1} \oplus s_{1,1} \oplus s_{1,2} \oplus s_{2,2} \oplus s_{3,3}
\end{aligned}$$

MixSlices

Mix up the slices according to this big thing:

$$\begin{aligned}
s'_{0,0} &= s_{0,0} \oplus s_{1,0} \oplus s_{1,3} \oplus s_{2,2} \oplus s_{3,0} \oplus s_{3,2} \oplus s_{3,3} \\
s'_{0,1} &= s_{0,1} \oplus s_{1,0} \oplus s_{2,3} \oplus s_{3,0} \oplus s_{3,3} \\
s'_{0,2} &= s_{0,2} \oplus s_{1,1} \oplus s_{2,0} \oplus s_{2,1} \oplus s_{3,0} \\
s'_{0,3} &= s_{0,3} \oplus s_{1,2} \oplus s_{2,1} \oplus s_{2,2} \oplus s_{3,1} \\
s'_{1,0} &= s_{0,0} \oplus s_{0,3} \oplus s_{1,0} \oplus s_{2,0} \oplus s_{2,2} \oplus s_{2,3} \oplus s_{3,2} \\
s'_{1,1} &= s_{0,0} \oplus s_{1,1} \oplus s_{2,0} \oplus s_{2,3} \oplus s_{3,3} \\
s'_{1,2} &= s_{0,1} \oplus s_{1,2} \oplus s_{2,0} \oplus s_{3,0} \oplus s_{3,1} \\
s'_{1,3} &= s_{0,2} \oplus s_{1,3} \oplus s_{2,1} \oplus s_{3,1} \oplus s_{3,2} \\
s'_{2,0} &= s_{0,2} \oplus s_{1,0} \oplus s_{1,2} \oplus s_{1,3} \oplus s_{2,0} \oplus s_{3,0} \oplus s_{3,3} \\
s'_{2,1} &= s_{0,3} \oplus s_{1,0} \oplus s_{1,3} \oplus s_{2,1} \oplus s_{3,0} \\
s'_{2,2} &= s_{0,0} \oplus s_{0,1} \oplus s_{1,0} \oplus s_{2,2} \oplus s_{3,1} \\
s'_{2,3} &= s_{0,1} \oplus s_{0,2} \oplus s_{1,1} \oplus s_{2,3} \oplus s_{3,2} \\
s'_{3,0} &= s_{0,0} \oplus s_{0,2} \oplus s_{0,3} \oplus s_{1,2} \oplus s_{2,0} \oplus s_{2,3} \oplus s_{3,0} \\
s'_{3,1} &= s_{0,0} \oplus s_{0,3} \oplus s_{1,3} \oplus s_{2,0} \oplus s_{3,1} \\
s'_{3,2} &= s_{0,0} \oplus s_{1,0} \oplus s_{1,1} \oplus s_{2,1} \oplus s_{3,2} \\
s'_{3,3} &= s_{0,1} \oplus s_{1,1} \oplus s_{1,2} \oplus s_{2,2} \oplus s_{3,3}
\end{aligned}$$

Finding the shortest MixSlices

- We want to find a program that can do `MixSlices` in as few lines of the shape $u = v \oplus w$ as possible. (this is known as the shortest linear Straight-Line Program);
- Finding this SLP is NP-hard
- Tried to find *the* shortest program, but that wasn't feasible even on the biggest machine on campus.

Heuristic results

A new `MixSlices` in 48 instead of 72 XORs!

$$\begin{aligned}
t_1 &= x_0 \oplus x_{14} \\
t_3 &= t_1 \oplus x_{14} \\
t_5 &= x_9 \oplus x_5 \\
y_{14} &= t_3 \oplus t_5 \\
t_{12} &= x_{10} \oplus t_3 \\
t_2 &= x_{12} \oplus x_8 \\
t_4 &= t_2 \oplus x_2 \\
y_2 &= t_4 \oplus t_5 \\
t_{14} &= x_6 \oplus t_4 \\
t_{10} &= x_1 \oplus x_{11} \\
t_{19} &= x_4 \oplus t_{10} \\
t_{11} &= x_{12} \oplus x_{15} \\
y_1 &= t_{19} \oplus t_{11} \\
t_{21} &= x_3 \oplus t_{12} \\
t_{13} &= x_8 \oplus x_{11} \\
y_4 &= t_{13} \oplus t_{21}
\end{aligned}$$

$$\begin{aligned}
t_6 &= x_1 \oplus x_{13} \\
t_{22} &= x_{10} \oplus t_6 \\
y_{10} &= t_1 \oplus t_{22} \\
t_9 &= x_2 \oplus x_{14} \\
t_{23} &= x_9 \oplus t_9 \\
t_8 &= x_7 \oplus x_{13} \\
y_7 &= t_8 \oplus t_{23} \\
t_{24} &= t_{10} \oplus t_{23} \\
y_{11} &= t_5 \oplus t_{24} \\
t_{25} &= x_0 \oplus t_{13} \\
t_{15} &= x_5 \oplus x_{15} \\
y_5 &= t_{15} \oplus t_{25} \\
t_{17} &= x_3 \oplus x_9 \\
t_{26} &= x_{12} \oplus t_{26} \\
t_{18} &= x_4 \oplus x_7 \\
y_9 &= t_{18} \oplus t_{26}
\end{aligned}$$

$$\begin{aligned}
t_{27} &= t_2 \oplus t_{22} \\
t_{16} &= x_6 \oplus x_{10} \\
y_6 &= t_{16} \oplus t_{27} \\
t_{28} &= x_7 \oplus t_{11} \\
y_0 &= t_{12} \oplus t_{28} \\
t_{30} &= x_8 \oplus t_8 \\
t_7 &= x_0 \oplus x_3 \\
y_{13} &= t_7 \oplus t_{30} \\
t_{31} &= x_{13} \oplus t_{17} \\
y_3 &= t_{16} \oplus t_{31} \\
t_{32} &= x_1 \oplus t_{16} \\
y_{15} &= t_{15} \oplus t_{32} \\
t_{33} &= x_{15} \oplus t_{14} \\
y_8 &= t_{18} \oplus t_{33} \\
t_{34} &= x_{11} \oplus t_{14} \\
y_{12} &= t_7 \oplus t_{34}
\end{aligned}$$

Heuristic results

A new `MixSlices` in 48 instead of 72 XORs!

$$\begin{aligned}
t_1 &= x_0 \oplus x_{14}\\
t_3 &= t_1 \oplus x_{14}\\
t_5 &= x_9 \oplus x_5\\
y_{14} &= t_3 \oplus t_5\\
t_{12} &= x_{10} \oplus t_3\\
t_2 &= x_{12} \oplus x_8\\
t_4 &= t_2 \oplus x_2\\
y_2 &= t_4 \oplus t_5\\
t_{14} &= x_6 \oplus t_4\\
t_{10} &= x_1 \oplus x_{11}\\
t_{19} &= x_4 \oplus t_{10}\\
t_{11} &= x_{12} \oplus x_{15}\\
y_1 &= t_{19} \oplus t_{11}\\
t_{21} &= x_3 \oplus t_{12}\\
t_{13} &= x_8 \oplus x_{11}\\
y_4 &= t_{13} \oplus t_{21}
\end{aligned}$$

$$\begin{aligned}
t_6 &= x_1 \oplus x_{13}\\
t_{22} &= x_{10} \oplus t_6\\
y_{10} &= t_1 \oplus t_{22}\\
t_9 &= x_2 \oplus x_{14}\\
t_{23} &= x_9 \oplus t_9\\
t_8 &= x_7 \oplus x_{13}\\
y_7 &= t_8 \oplus t_{23}\\
t_{24} &= t_{10} \oplus t_{23}\\
y_{11} &= t_5 \oplus t_{24}\\
t_{25} &= x_0 \oplus t_{13}\\
t_{15} &= x_5 \oplus x_{15}\\
y_5 &= t_{15} \oplus t_{25}\\
t_{17} &= x_3 \oplus x_9\\
t_{26} &= x_{12} \oplus t_{26}\\
t_{18} &= x_4 \oplus x_7\\
y_9 &= t_{18} \oplus t_{26}
\end{aligned}$$

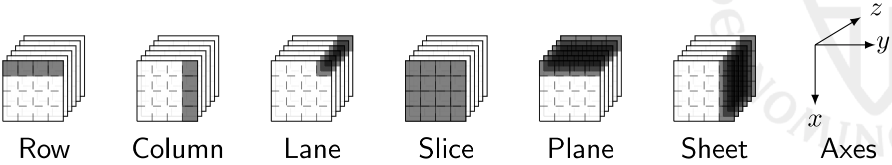
$$\begin{aligned}
t_{27} &= t_2 \oplus t_{22}\\
t_{16} &= x_6 \oplus x_{10}\\
y_6 &= t_{16} \oplus t_{27}\\
t_{28} &= x_7 \oplus t_{11}\\
y_0 &= t_{12} \oplus t_{28}\\
t_{30} &= x_8 \oplus t_8\\
t_7 &= x_0 \oplus x_3\\
y_{13} &= t_7 \oplus t_{30}\\
t_{31} &= x_{13} \oplus t_{17}\\
y_3 &= t_{16} \oplus t_{31}\\
t_{32} &= x_1 \oplus t_{16}\\
y_{15} &= t_{15} \oplus t_{32}\\
t_{33} &= x_{15} \oplus t_{14}\\
y_8 &= t_{18} \oplus t_{33}\\
t_{34} &= x_{11} \oplus t_{14}\\
y_{12} &= t_7 \oplus t_{34}
\end{aligned}$$

ShiftPlanes$_i$

- Shifts the bits in the planes over the z-direction,
- The number of bits rotated differs for odd and even rounds:

  Even The first, second, third and forth plane are rotated 0, 1, 8 and 9 bits, respectively,

  Odd The first, second, third and forth plane are rotated 0, 2, 4 and 6 bits, respectively.



ShiftPlanes

To rotate a 16 bit lane inside a 32-bit register, we need to first double the register:

```
a = mem16[addr]
a = a | (a << 16)
a >>>= 2
```

Unfortunately, that means we can't use our inline rotations any more.

AddConstants$_i$

Adds the constants c_1 and c_2, rotated by the round number i and the index of the lane, to the individual lanes.

$$\begin{pmatrix} s'_{0,0} \\ s'_{0,1} \\ s'_{0,2} \\ s'_{0,3} \\ s'_{1,0} \\ \vdots \\ s'_{3,3} \end{pmatrix} = \begin{pmatrix} s_{0,0} \oplus (c_1 \lll i \lll 0) \\ s_{0,1} \oplus (c_2 \lll i \lll 1) \\ s_{0,2} \oplus (c_1 \lll i \lll 2) \\ s_{0,3} \oplus (c_2 \lll i \lll 3) \\ s_{1,0} \oplus (c_1 \lll i \lll 4) \\ \vdots \\ s_{3,3} \oplus (c_2 \lll i \lll 15) \end{pmatrix}$$

AddConstants

Here, we can make good use of the free rotations:

```
x_0 = mem16[address]
newx0 = x_0 ^ (c1 >>> 31)
```

By reusing results still in memory from `ShiftPlanes` we don't need to shift registers loaded using the "two lanes in one register"-approach.

Benchmarks

Putting it all together, we get the following results from the SUPERCOP benchmarking suite for cryptography:

Implementation	APE	COPA	OTR
Reference (`C`)	2,975,123	2,402,577	1,569,582
Mine (ARM `asm`)	1,900,274	1,714,321	848,100
Performance improvement	36%	28%	46%

Table: Comparison of cycle counts

Conclusions

Results

- Good performance improvement,
- New implementation of `MixSlices`.

Possible further work

- Optimise Prøst-256,
- Optimise Prøst for other platforms,
- Optimise other ciphers using these techniques,
- Backport these techniques to a faster c-implementation.

Outline

Overtime

- Approximating the shortest MixSlices
- Searching the shortest MixSlices

Using a heuristic

Boyar et al. define a heuristic to approximate the shortest program.[1]

The heuristic

1. Consider your program as an input matrix M;
2. Initialise matrix S to $([1, 0, \cdots], [0, 1, 0 \cdots])$ to represent your inputs;
3. Define a Distance function $Dist[i]$ that determines the distance of S to $M[i]$ as minimum number of combinations of S that need to be made to get $M[i]$;
4. Generate all combinations of rows in S, determine the best new one by the norm of the distances until distances are 0.

Using a heuristic

Boyar et al. define a heuristic to approximate the shortest program.[1]

The heuristic

1. Consider your program as an input matrix M;
2. Initialise matrix S to $([1, 0, \cdots], [0, 1, 0 \cdots])$ to represent your inputs;
3. Define a Distance function $Dist[i]$ that determines the distance of S to $M[i]$ as minimum number of combinations of S that need to be made to get $M[i]$;
4. Generate all combinations of rows in S, determine the best new one by the norm of the distances until distances are 0.

Your program as a matrix

We can represent these programs as a matrix:

$$\begin{aligned}
y_0 &= x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \\
y_1 &= x_0 \oplus x_1 \oplus x_2 \oplus x_3 \\
y_2 &= x_0 \oplus x_1 \oplus x_2 \oplus x_4 \\
y_3 &= x_2 \oplus x_3 \oplus x_4 \\
y_4 &= x_0 \oplus x_4
\end{aligned}$$

$$M = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Using a heuristic

Boyar et al. define a heuristic to approximate the shortest program.[1]

The heuristic

1. Consider your program as an input matrix M;
2. Initialise matrix S to $([1, 0, \cdots], [0, 1, 0 \cdots])$ to represent your inputs;
3. Define a Distance function $Dist[i]$ that determines the distance of S to $M[i]$ as minimum number of combinations of S that need to be made to get $M[i]$;
4. Generate all combinations of rows in S, determine the best new one by the norm of the distances until distances are 0.

Matrix S of program lines

Each line of S is a combination of the previous lines and represents one line of our straight-line program.

$$S = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ & & \dots & & \end{pmatrix}$$

Using a heuristic

Boyar et al. define a heuristic to approximate the shortest program.[1]

The heuristic

1. Consider your program as an input matrix M;
2. Initialise matrix S to $([1, 0, \cdots], [0, 1, 0 \cdots])$ to represent your inputs;
3. Define a Distance function $Dist[i]$ that determines the distance of S to $M[i]$ as minimum number of combinations of S that need to be made to get $M[i]$;
4. Generate all combinations of rows in S, determine the best new one by the norm of the distances until distances are 0.

Using a heuristic

Boyar et al. define a heuristic to approximate the shortest program.[1]

The heuristic

1. Consider your program as an input matrix M;
2. Initialise matrix S to $([1, 0, \cdots], [0, 1, 0 \cdots])$ to represent your inputs;
3. Define a Distance function $Dist[i]$ that determines the distance of S to $M[i]$ as minimum number of combinations of S that need to be made to get $M[i]$;
4. Generate all combinations of rows in S, determine the best new one by the norm of the distances until distances are 0.

Finding the shortest MixSlices

- We want to find a program that can do `MixSlices` in as few lines of the shape $u = v \oplus w$ as possible. (this is known as the shortest linear Straight-Line Program);
- Finding this SLP is NP-hard
- Tried to find *the* shortest program, but that wasn't feasible even on the biggest machine on campus.

Trying to find the actual shortest program

Fuhs and Schneider-Kamp show in “Synthesizing Shortest Linear Straight-Line Programs over GF(2) using SAT” how to transform the SLP problem to SAT.

Transforming SLP to SAT

1. Input your program as a matrix and decide on a number of lines k;
2. Define matrices B, C and mapping f;
3. Apply constraints that only can be satisfied by valid programs;
4. If the problem is satisfiable, extract the program from B, C, and f.
5. Repeat with lower k until UNSAT.

Your program as a matrix

We can represent these programs as a matrix:

$$\begin{aligned}
y_0 &= x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \\
y_1 &= x_0 \oplus x_1 \oplus x_2 \oplus x_3 \\
y_2 &= x_0 \oplus x_1 \oplus x_2 \oplus x_4 \\
y_3 &= x_2 \oplus x_3 \oplus x_4 \\
y_4 &= x_0 \oplus x_4
\end{aligned}
\qquad
M = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Trying to find the actual shortest program

Fuhs and Schneider-Kamp show in “Synthesizing Shortest Linear Straight-Line Programs over GF(2) using SAT” how to transform the SLP problem to SAT.

Transforming SLP to SAT

1. Input your program as a matrix and decide on a number of lines k;
2. Define matrices B, C and mapping f;
3. Apply constraints that only can be satisfied by valid programs;
4. If the problem is satisfiable, extract the program from B, C, and f.
5. Repeat with lower k until UNSAT.

Defining B, C and f for $k = 6$

$$B = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad C = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad f = \begin{cases} 0 \mapsto ? \\ 1 \mapsto ? \\ 2 \mapsto ? \\ 3 \mapsto ? \\ 4 \mapsto ? \\ 5 \mapsto ? \end{cases}$$

Trying to find the actual shortest program

Fuhs and Schneider-Kamp show in “Synthesizing Shortest Linear Straight-Line Programs over GF(2) using SAT” how to transform the SLP problem to SAT.

Transforming SLP to SAT

1. Input your program as a matrix and decide on a number of lines k;
2. Define matrices B, C and mapping f;
3. Apply constraints that only can be satisfied by valid programs;
4. If the problem is satisfiable, extract the program from B, C, and f.
5. Repeat with lower k until UNSAT.

Defining constraints

One of the constraints:

Each line can exist of two incoming variables and it can only use temporary variables that we have already seen

$$\beta_1 = \bigvee_{0 \leq i < k} \mathrm{exactly}_2(b_{i,1}, \cdots, b_{i,n}, c_{i,n}, \cdots, c_{i,i-1})$$

Trying to find the actual shortest program

Fuhs and Schneider-Kamp show in “Synthesizing Shortest Linear Straight-Line Programs over GF(2) using SAT” how to transform the SLP problem to SAT.

Transforming SLP to SAT

1. Input your program as a matrix and decide on a number of lines k;
2. Define matrices B, C and mapping f;
3. Apply constraints that only can be satisfied by valid programs;
4. If the problem is satisfiable, extract the program from B, C, and f.
5. Repeat with lower k until UNSAT.

Getting our program from the valuation

$$B = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \; C = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix} f = \begin{cases} 0 \mapsto 3 \\ 1 \mapsto 4 \\ 2 \mapsto 2 \\ 3 \mapsto 5 \\ 4 \mapsto 0 \end{cases}$$

Bibliography I

[1] Joan Boyar, Philip Matthews and René Peralta. 'Logic Minimization Techniques with Applications to Cryptology'. English. In: *Journal of Cryptology* 26.2 (2013), pp. 280–312. ISSN: 0933-2790. DOI: 10.1007/s00145-012-9124-7. URL: http://dx.doi.org/10.1007/s00145-012-9124-7.

[2] *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness*. URL: http://competitions.cr.yp.to/caesar.html.

[3] Elif Bilge Kavun, Martin M. Lauridsen, Gregor Leander, Christian Rechberger, Peter Schwabe and Tolga Yalçın. *Prø st v1.1*. 21st June 2014. URL: http://competitions.cr.yp.to/round1/proestv11.pdf.