Fixslicing AES-like Ciphers

New bitsliced AES speed records on ARM Cortex-M and RISC-V

Alexandre Adomnicai[1] Thomas Peyrin[2]

[1]CryptoNext Security, Paris, France

[2]Nanyang Technological University, Singapore

CHES 2021: Conference on Cryptographic Hardware and Embedded Systems

Some context

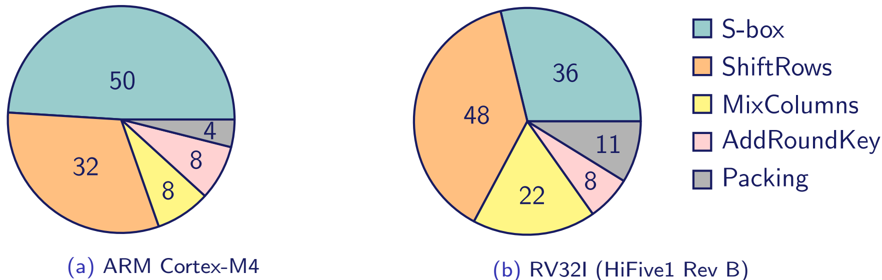
- AES is running on a **wide range of platforms**: from resource-constrained devices to high-end servers
- Many embedded devices do not enjoy hardware AES engines and rely on **software implementations** instead
- AES can be efficiently implemented in SW using **look-up tables** but the table accesses are key and data-dependent $\Rightarrow$ **cache-timing attacks**
- **Bitslicing** is a well known technique to avoid timing-based leakage
- **Our work improves bitsliced** AES (w/o the use of vector permute instructions)

Results previously reported

- Bitsliced AES-128 runs at **102 and 125 cpb on ARM Cortex-M4 and RV32I** [SS16, Sto19] with precomputed round keys

Results previously reported

- ▷ Bitsliced AES-128 runs at **102 and 125 cpb on ARM Cortex-M4 and RV32I** [SS16, Sto19] with precomputed round keys



(a) ARM Cortex-M4

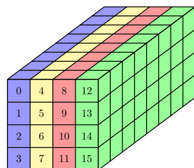
(b) RV32I (HiFive1 Rev B)

Figure: Cycles per byte (cpb) per operation (2 blocks at a time)

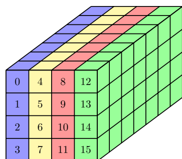
Bitslicing the AES

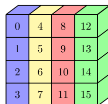
0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Bitslicing the AES

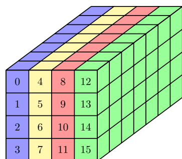


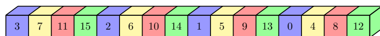
Bitslicing the AES



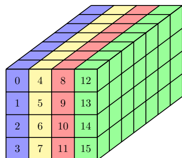


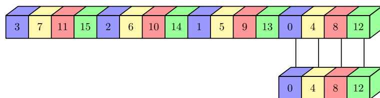
Bitslicing the AES



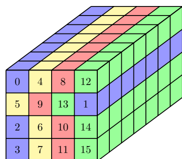


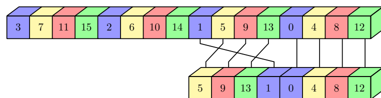
Bitslicing the AES



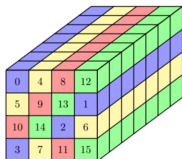


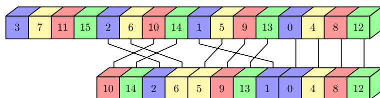
Bitslicing the AES



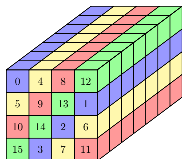


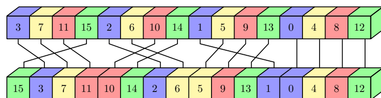
Bitslicing the AES



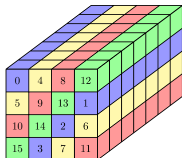


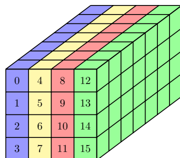
Bitslicing the AES

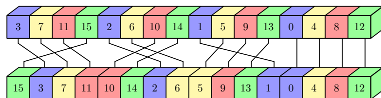




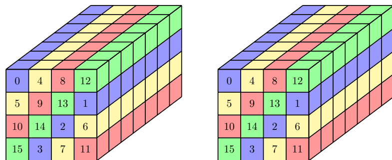
Bitslicing the AES

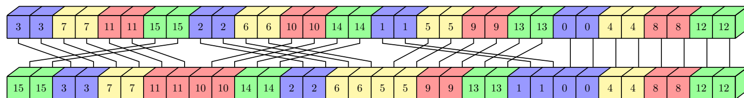




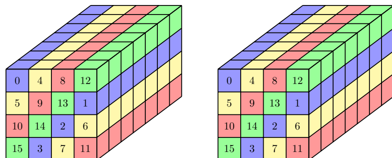


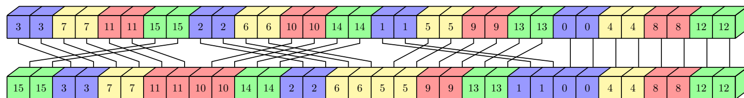
Bitslicing the AES





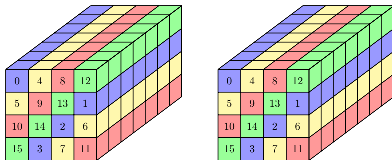
Bitslicing the AES

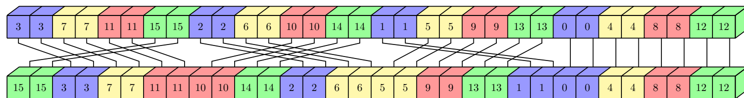





```
1  t = (r >> 6) & 0x00000300;        // shifts the second row
2  t = t | (r & 0x00003f00) << 2;    // shifts the second row
3  t = t | (r >> 4) & 0x000f0000;    // shifts the third row
4  t = t | (r & 0x000f0000) << 4;    // shifts the third row
5  t = t | (r >> 2) & 0x3f000000;    // shifts the fourth row
6  t = t | (r & 0x03000000) << 6;    // shifts the fourth row
7  r = t | (r & 0x000000ff);         // the first row isn't shifted
```

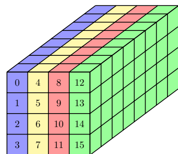
Bitslicing the AES



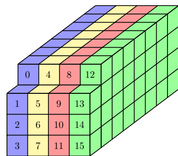


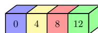

```
1  t = r ^ (r >> 4) & 0x30f0c000;       // first swapmove
2  r = r ^ t;
3  r = r ^ (t << 4);
4  t = r ^ (r >> 2) & 0xcc00cc00;       // second swapmove
5  r = r ^ t;
6  r = r ^ (t << 2);
```

The “Barrel ShiftRows” representation

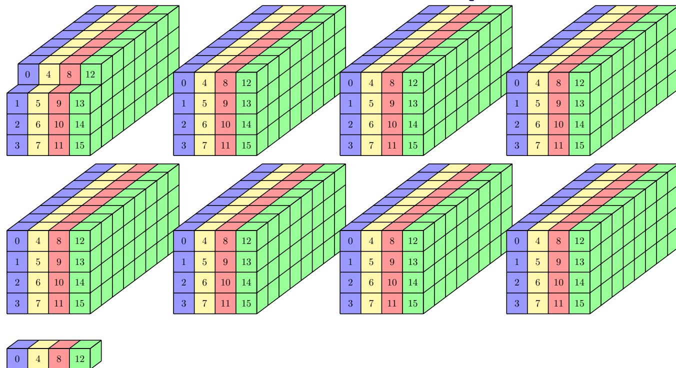


The “Barrel ShiftRows” representation

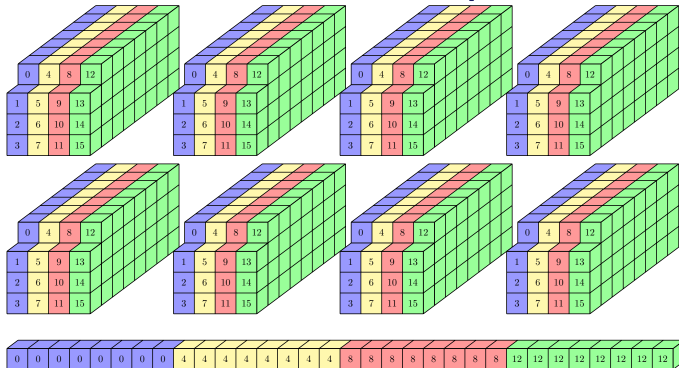




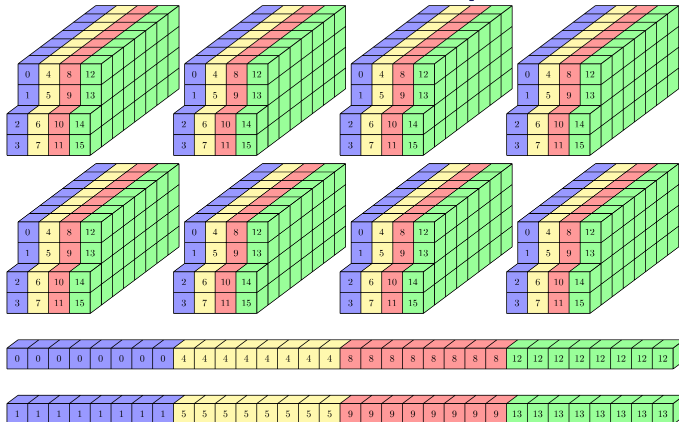
The “Barrel ShiftRows” representation



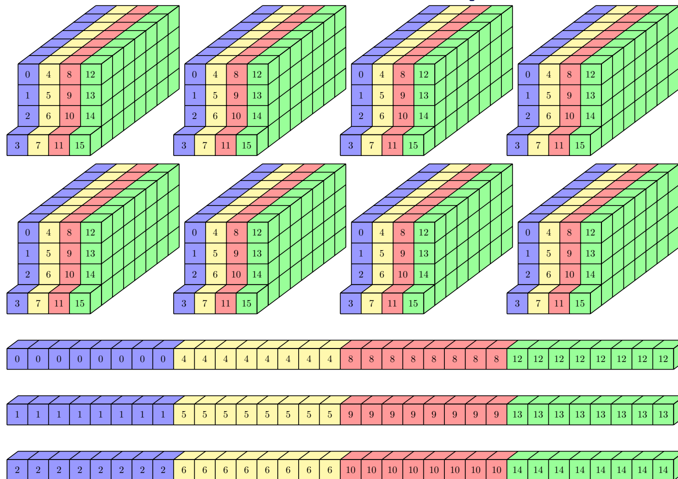
The “Barrel ShiftRows” representation



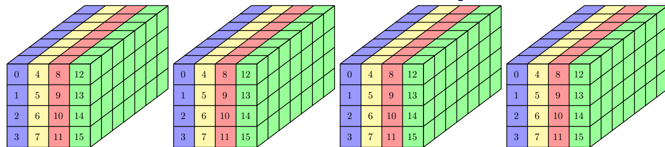
The “Barrel ShiftRows” representation

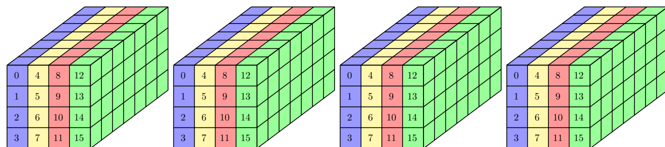


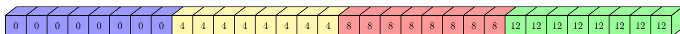
The "Barrel ShiftRows" representation

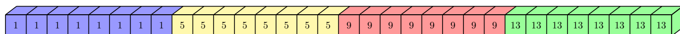


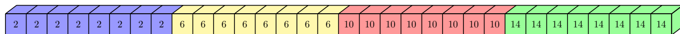
The “Barrel ShiftRows” representation

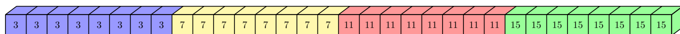












The “Barrel ShiftRows” representation

0 4 8 12
5 9 13 1
10 14 2 6
15 3 7 11

0 0 0 0 0 0 0 0 4 4 4 4 4 4 4 4 8 8 8 8 8 8 8 8 12 12 12 12 12 12 12 12

5 5 5 5 5 5 5 5 9 9 9 9 9 9 9 9 13 13 13 13 13 13 13 13 1 1 1 1 1 1 1 1

10 10 10 10 10 10 10 10 14 14 14 14 14 14 14 14 2 2 2 2 2 2 2 2 6 6 6 6 6 6 6 6

15 15 15 15 15 15 15 15 3 3 3 3 3 3 3 3 7 7 7 7 7 7 7 7 11 11 11 11 11 11 11 11

The "Barrel ShiftRows" representation

- ▷ The `ShiftRows` operation only requires $8 \times 3 = 24$ 32-bit rotations per round

The “Barrel ShiftRows” representation

- The `ShiftRows` operation only requires $8 \times 3 = 24$ 32-bit rotations per round

- 8 blocks to be processed in parallel

The “Barrel ShiftRows” representation

- The `ShiftRows` operation only requires $8 \times 3 = 24$ 32-bit rotations per round

- 8 blocks to be processed in parallel
- Requires 32 32-bit general purpose registers to store the $128 \times 8 = 1024$-bit internal state

The “Barrel ShiftRows” representation

- ▷ The `ShiftRows` operation only requires $8 \times 3 = 24$ 32-bit rotations per round

- ▷ 8 blocks to be processed in parallel
- ▷ Requires 32 32-bit general purpose registers to store the $128 \times 8 = 1024$-bit internal state
- ▷ Increases RAM consumption by a factor 4 to store the round keys

The fixslicing implementation strategy

- Initially introduced as a new representation for the GIFT block ciphers [ANP20]

The fixslicing implementation strategy

- ▷ Initially introduced as a new representation for the GIFT block ciphers [ANP20]
- ▷ Consists in **fixing a slice to never move** and adjusting the others for the S-box layer

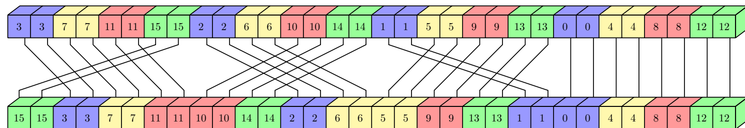
The fixslicing implementation strategy

- Initially introduced as a new representation for the GIFT block ciphers [ANP20]
- Consists in **fixing a slice to never move** and adjusting the others for the S-box layer
- Many other block ciphers require to move the bits around at some point $\Rightarrow$ fixslicing could help **define an efficient alternative representation**

The fixslicing implementation strategy

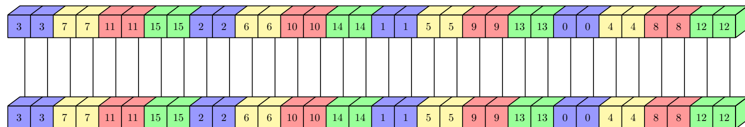
- ▷ Initially introduced as a new representation for the GIFT block ciphers [ANP20]
- ▷ Consists in **fixing a slice to never move** and adjusting the others for the S-box layer
- ▷ Many other block ciphers require to move the bits around at some point $\Rightarrow$ fixslicing could help **define an efficient alternative representation**
- ▷ What about the AES?

Fixslicing the AES: omitting the ShiftRows



- The bits are moved similarly within the 8 registers $R_0, \cdots, R_7$ during the `ShiftRows`

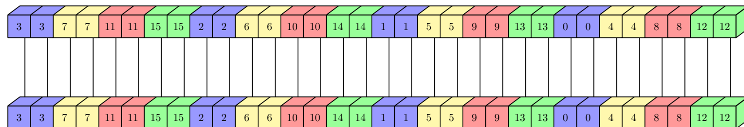
Fixslicing the AES: omitting the ShiftRows



- The bits are moved similarly within the 8 registers $R_0, \cdots, R_7$ during the `ShiftRows`

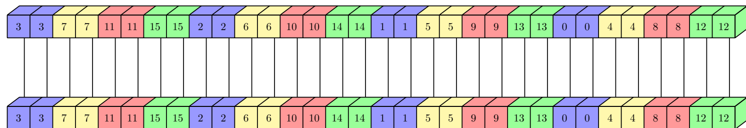
- Fixslicing the AES means that all slices remain fixed $\Rightarrow$ **simply omit the `ShiftRows`**

Fixslicing the AES: omitting the ShiftRows



- The bits are moved similarly within the 8 registers $R_0, \cdots, R_7$ during the `ShiftRows`
- Fixslicing the AES means that all slices remain fixed ⇒ **simply omit the `ShiftRows`**
- Not a problem for the S-box layer as bits within bytes remain aligned, but we **need to adapt the MixColumns operation**

Fixslicing the AES: omitting the ShiftRows



- The bits are moved similarly within the 8 registers $R_0, \cdots, R_7$ during the `ShiftRows`
- Fixslicing the AES means that all slices remain fixed $\Rightarrow$ **simply omit the `ShiftRows`**
- Not a problem for the S-box layer as bits within bytes remain aligned, but we **need to adapt the MixColumns operation**
- Synchronization with the classical representation occurs every 4 rounds

The bitsliced MixColumns by [KS09]

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} = \begin{bmatrix} a \cdot 02 + b \cdot 03 + c + d \\ b \cdot 02 + c \cdot 03 + d + a \\ c \cdot 02 + d \cdot 03 + a + b \\ d \cdot 02 + a \cdot 03 + b + c \end{bmatrix}$$

The bitsliced MixColumns by [KS09]

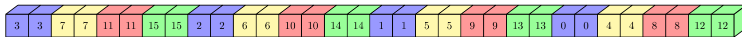
$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} = \begin{bmatrix} a \cdot 02 + b \cdot 03 + c + d \\ b \cdot 02 + c \cdot 03 + d + a \\ c \cdot 02 + d \cdot 03 + a + b \\ d \cdot 02 + a \cdot 03 + b + c \end{bmatrix}$$

with $a \cdot 02 = (a \ll 1) \oplus (a \gg 7) \wedge (00011011)_2$ and $a \cdot 03 = a \cdot 02 \oplus a$

The bitsliced MixColumns by [KS09]

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} = \begin{bmatrix} a \cdot 02 + b \cdot 03 + c + d \\ b \cdot 02 + c \cdot 03 + d + a \\ c \cdot 02 + d \cdot 03 + a + b \\ d \cdot 02 + a \cdot 03 + b + c \end{bmatrix}$$

with $a \cdot 02 = (a \ll 1) \oplus (a \gg 7) \wedge (00011011)_2$ and $a \cdot 03 = a \cdot 02 \oplus a$

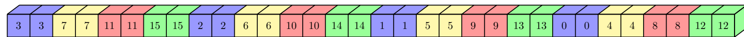


$$R'_0 = R_1$$
$$R'_1 = R_2$$
$$R'_2 = R_3$$
$$R'_3 = R_4 \oplus R_0$$
$$R'_4 = R_5 \oplus R_0$$
$$R'_5 = R_6$$
$$R'_6 = R_7 \oplus R_0$$
$$R'_7 = R_0$$

The bitsliced MixColumns by [KS09]

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} = \begin{bmatrix} a \cdot 02 + b \cdot 03 + c + d \\ b \cdot 02 + c \cdot 03 + d + a \\ c \cdot 02 + d \cdot 03 + a + b \\ d \cdot 02 + a \cdot 03 + b + c \end{bmatrix}$$

with $a \cdot 02 = (a \ll 1) \oplus (a \gg 7) \wedge (00011011)_2$ and $a \cdot 03 = a \cdot 02 \oplus a$

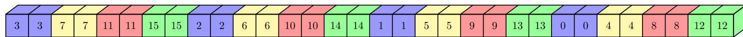


$$R'_0 = R_1 \oplus R_1^{\ggg 8} \oplus R_0^{\ggg 8}$$
$$R'_1 = R_2 \oplus R_2^{\ggg 8} \oplus R_1^{\ggg 8}$$
$$R'_2 = R_3 \oplus R_3^{\ggg 8} \oplus R_2^{\ggg 8}$$
$$R'_3 = R_4 \oplus R_0 \oplus R_4^{\ggg 8} \oplus R_3^{\ggg 8} \oplus R_0^{\ggg 8}$$
$$R'_4 = R_5 \oplus R_0 \oplus R_5^{\ggg 8} \oplus R_4^{\ggg 8} \oplus R_0^{\ggg 8}$$
$$R'_5 = R_6 \oplus R_6^{\ggg 8} \oplus R_5^{\ggg 8}$$
$$R'_6 = R_7 \oplus R_0 \oplus R_7^{\ggg 8} \oplus R_6^{\ggg 8} R_0^{\ggg 8}$$
$$R'_7 = R_0 \oplus R_0^{\ggg 8} \oplus R_7^{\ggg 8}$$

The bitsliced MixColumns by [KS09]

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} = \begin{bmatrix} a \cdot 02 + b \cdot 03 + c + d \\ b \cdot 02 + c \cdot 03 + d + a \\ c \cdot 02 + d \cdot 03 + a + b \\ d \cdot 02 + a \cdot 03 + b + c \end{bmatrix}$$

with $a \cdot 02 = (a \ll 1) \oplus (a \gg 7) \wedge (00011011)_2$ and $a \cdot 03 = a \cdot 02 \oplus a$

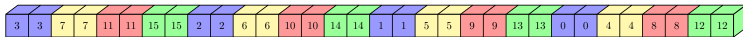


$$R_0' = R_1 \oplus R_1^{\ggg 8} \oplus R_0^{\ggg 8} \oplus R_0^{\ggg 16} \oplus R_0^{\ggg 24}$$
$$R_1' = R_2 \oplus R_2^{\ggg 8} \oplus R_1^{\ggg 8} \oplus R_1^{\ggg 16} \oplus R_1^{\ggg 24}$$
$$R_2' = R_3 \oplus R_3^{\ggg 8} \oplus R_2^{\ggg 8} \oplus R_2^{\ggg 16} \oplus R_2^{\ggg 24}$$
$$R_3' = R_4 \oplus R_0 \oplus R_4^{\ggg 8} \oplus R_3^{\ggg 8} \oplus R_0^{\ggg 8} \oplus R_3^{\ggg 16} \oplus R_3^{\ggg 24}$$
$$R_4' = R_5 \oplus R_0 \oplus R_5^{\ggg 8} \oplus R_4^{\ggg 8} \oplus R_0^{\ggg 8} \oplus R_4^{\ggg 16} \oplus R_4^{\ggg 24}$$
$$R_5' = R_6 \oplus R_6^{\ggg 8} \oplus R_5^{\ggg 8} \oplus R_5^{\ggg 16} \oplus R_5^{\ggg 24}$$
$$R_6' = R_7 \oplus R_0 \oplus R_7^{\ggg 8} \oplus R_6^{\ggg 8} R_0^{\ggg 8} \oplus R_6^{\ggg 16} \oplus R_6^{\ggg 24}$$
$$R_7' = R_0 \oplus R_0^{\ggg 8} \oplus R_7^{\ggg 8} \oplus R_7^{\ggg 16} \oplus R_7^{\ggg 24}$$

The bitsliced MixColumns by [KS09]

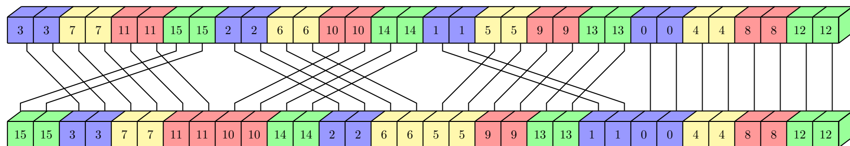
$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} = \begin{bmatrix} a \cdot 02 + b \cdot 03 + c + d \\ b \cdot 02 + c \cdot 03 + d + a \\ c \cdot 02 + d \cdot 03 + a + b \\ d \cdot 02 + a \cdot 03 + b + c \end{bmatrix}$$

with $a \cdot 02 = (a \ll 1) \oplus (a \gg 7) \wedge (00011011)_2$ and $a \cdot 03 = a \cdot 02 \oplus a$



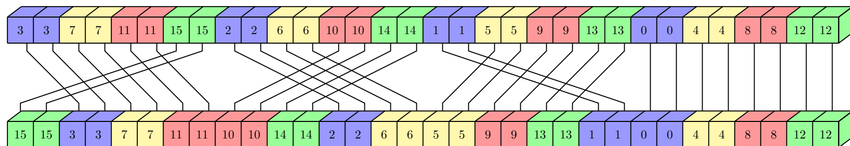
$$R_0' = (R_1 \oplus R_1^{\ggg 8}) \oplus R_0^{\ggg 8} \oplus (R_0 \oplus R_0^{\ggg 8})^{\ggg 16}$$
$$R_1' = (R_2 \oplus R_2^{\ggg 8}) \oplus R_1^{\ggg 8} \oplus (R_1 \oplus R_1^{\ggg 8})^{\ggg 16}$$
$$R_2' = (R_3 \oplus R_3^{\ggg 8}) \oplus R_2^{\ggg 8} \oplus (R_2 \oplus R_2^{\ggg 8})^{\ggg 16}$$
$$R_3' = (R_4 \oplus R_4^{\ggg 8}) \oplus R_3^{\ggg 8} \oplus (R_3 \oplus R_3^{\ggg 8})^{\ggg 16} \oplus (R_0 \oplus R_0^{\ggg 8})$$
$$R_4' = (R_5 \oplus R_5^{\ggg 8}) \oplus R_4^{\ggg 8} \oplus (R_4 \oplus R_4^{\ggg 8})^{\ggg 16} \oplus (R_0 \oplus R_0^{\ggg 8})$$
$$R_5' = (R_6 \oplus R_6^{\ggg 8}) \oplus R_5^{\ggg 8} \oplus (R_5 \oplus R_5^{\ggg 8})^{\ggg 16}$$
$$R_6' = (R_7 \oplus R_7^{\ggg 8}) \oplus R_6^{\ggg 8} \oplus (R_6 \oplus R_6^{\ggg 8})^{\ggg 16} \oplus (R_0 \oplus R_0^{\ggg 8})$$
$$R_7' = (R_0 \oplus R_0^{\ggg 8}) \oplus R_7^{\ggg 8} \oplus (R_7 \oplus R_7^{\ggg 8})^{\ggg 16}$$

Adjusting the MixColumns (round 0)



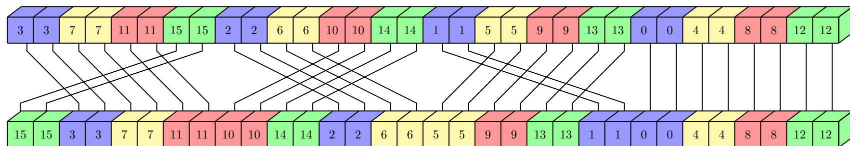
$$R_0' = \left(R_1 \oplus R_1^{\ggg 8}\right) \oplus R_0^{\ggg 8} \oplus \left(R_0 \oplus R_0^{\ggg 8}\right)^{\ggg 16}$$

Adjusting the MixColumns (round 0)



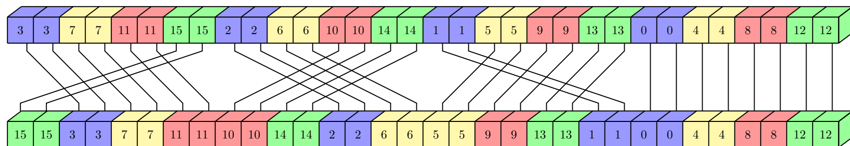
$$R_0' = \left(R_1 \oplus (R_1^{\ggg 8} \underset{8}{\ggg} 6)\right) \oplus (R_0^{\ggg 8} \underset{8}{\ggg} 6) \oplus (R_0^{\ggg 16} \underset{8}{\ggg} 4) \oplus (R_0^{\ggg 24} \underset{8}{\ggg} 2)$$

Adjusting the MixColumns (round 0)



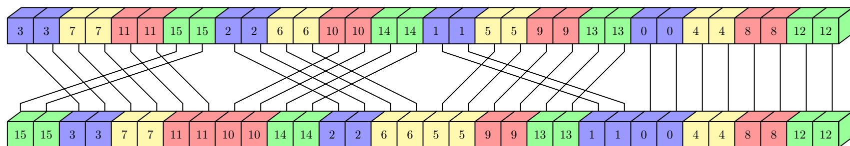
$$R_0' = \left(R_1 \oplus \left(R_1^{\ggg 8} \underset{8}{\ggg} 6\right)\right) \oplus \left(R_0^{\ggg 8} \underset{8}{\ggg} 6\right) \oplus \left(R_0 \oplus \left(R_0^{\ggg 8} \underset{8}{\ggg} 6\right)\right)^{\ggg 16} \underset{8}{\ggg} 4$$

Adjusting the MixColumns (round 0)



$$R_0' = \left(R_1 \oplus \left(R_1^{\ggg 8} \underset{8}{\ggg} 6\right)\right) \oplus \left(R_0^{\ggg 8} \underset{8}{\ggg} 6\right) \oplus \left(R_0 \oplus \left(R_0^{\ggg 8} \underset{8}{\ggg} 6\right)\right)^{\ggg 16} \underset{8}{\ggg} 4$$

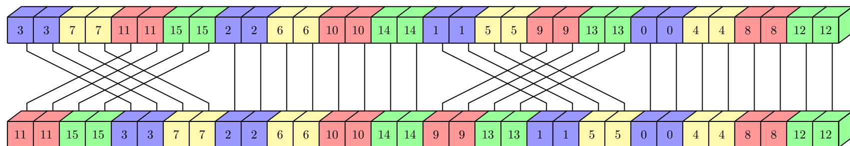
Adjusting the MixColumns (round 0)



$$R_0' = \left(R_1 \oplus (R_1^{\ggg 8} \underset{8}{\ggg} 6)\right) \oplus (R_0^{\ggg 8} \underset{8}{\ggg} 6) \oplus \left(R_0 \oplus (R_0^{\ggg 8} \underset{8}{\ggg} 6)\right)^{\ggg 16} \underset{8}{\ggg} 4$$

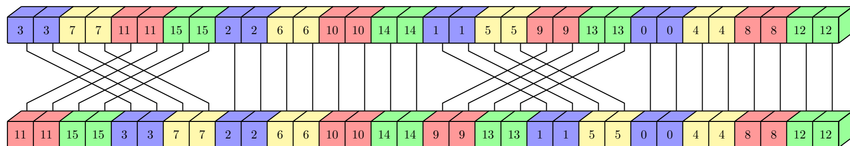
▷ It saves 56 logical operations and 16 logical shifts compared to the classical representation

Adjusting the MixColumns (round 1)



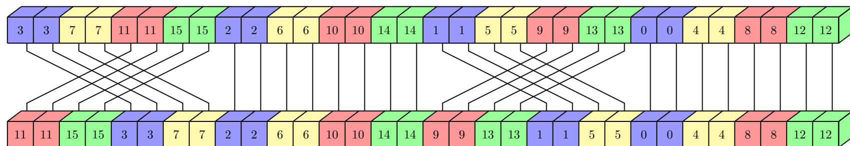
$$R'_0 = \left(R_1 \oplus R_1^{\ggg 8}\right) \oplus R_0^{\ggg 8} \oplus \left(R_0 \oplus R_0^{\ggg 8}\right)^{\ggg 16}$$

Adjusting the MixColumns (round 1)



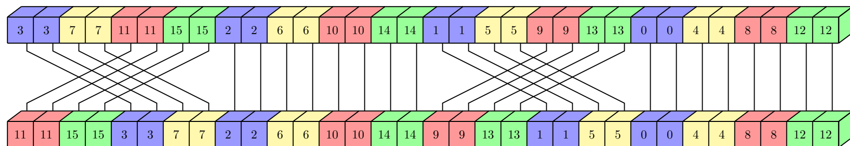
$$R_0' = \left(R_1 \oplus \left(R_1^{\ggg 8} \underset{8}{\ggg} 4\right)\right) \oplus \left(R_0^{\ggg 8} \underset{8}{\ggg} 4\right) \oplus \left(R_0 \oplus \left(R_0^{\ggg 8} \underset{8}{\ggg} 4\right)\right)^{\ggg 16}$$

Adjusting the MixColumns (round 1)



$$R'_0 = \left(R_1 \oplus (R_1^{\ggg 8} \underset{8}{\ggg} 4)\right) \oplus (R_0^{\ggg 8} \underset{8}{\ggg} 4) \oplus \left(R_0 \oplus (R_0^{\ggg 8} \underset{8}{\ggg} 4)\right)^{\ggg 16}$$

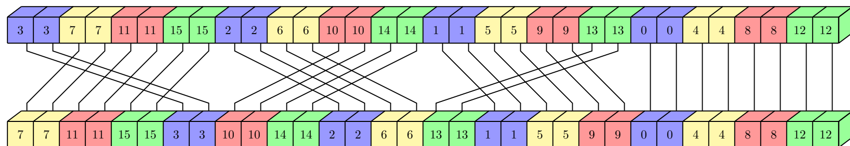
Adjusting the MixColumns (round 1)



$$R_0' = \left(R_1 \oplus (R_1^{\ggg 8} \underset{8}{\ggg} 4)\right) \oplus (R_0^{\ggg 8} \underset{8}{\ggg} 4) \oplus \left(R_0 \oplus (R_0^{\ggg 8} \underset{8}{\ggg} 4)\right)^{\ggg 16}$$

▷ It saves 80 logical operations and 32 logical shifts compared to the classical representation

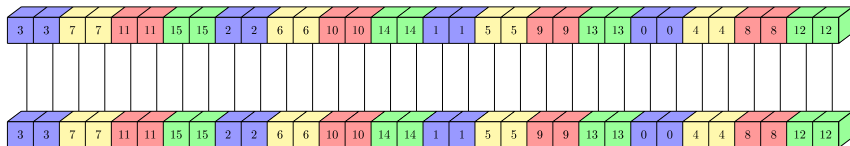
Adjusting the MixColumns (round 2)



$$R_0' = \left(R_1 \oplus \left(R_1^{\ggg 8} \underset{8}{\ggg} 2\right)\right) \oplus \left(R_0^{\ggg 8} \underset{8}{\ggg} 2\right) \oplus \left(R_0 \oplus \left(R_0^{\ggg 8} \underset{8}{\ggg} 2\right)\right)^{\ggg 16} \underset{8}{\ggg} 4$$

▷ It saves 56 logical operations and 16 logical shifts compared to the classical representation

Adjusting the MixColumns (round 3)



$$R_0' = \left(R_1 \oplus R_1^{\ggg 8}\right) \oplus R_0^{\ggg 8} \oplus \left(R_0 \oplus R_0^{\ggg 8}\right)^{\ggg 16}$$

▷ It saves 104 logical operations and 48 logical shifts compared to the classical representation

Fixslicing the AES: pros and cons

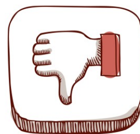
- Omitting the ShiftRows allows to speed up the linear layer when bitslicing on 32-bit platforms

Fixslicing the AES: pros and cons

- ▷ Omitting the ShiftRows allows to speed up the linear layer when bitslicing on 32-bit platforms
- ▷ Processing only 2 blocks at a time allows to take full advantage of 32-bit registers

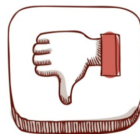
Fixslicing the AES: pros and cons

- Omitting the ShiftRows allows to speed up the linear layer when bitslicing on 32-bit platforms
- Processing only 2 blocks at a time allows to take full advantage of 32-bit registers



- 4 different implementations of the `MixColumns` are required $\Rightarrow$ impact on code size

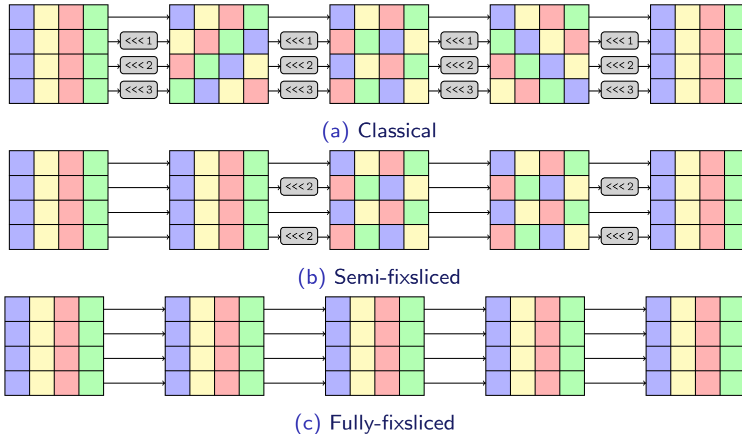
Fixslicing the AES: pros and cons



- ▷ Omitting the ShiftRows allows to speed up the linear layer when bitslicing on 32-bit platforms
- ▷ Processing only 2 blocks at a time allows to take full advantage of 32-bit registers

- ▷ 4 different implementations of the `MixColumns` are required $\Rightarrow$ impact on code size
- ▷ The round keys have to be adapted accordingly

Fixslicing the AES: trade-offs



(a) Classical

(b) Semi-fixsliced

(c) Fully-fixsliced

Figure: Overview of the AES internal state over 4 rounds.

Linear layer implementation improvements

Representation	Ref	Number of operations for the linear layer (over 4 rounds)			
		LOP	LSH	ROT	LOP + LSH + ROT
Classical bitsliced	[SS16]	524	192	64	780
Fully-fixsliced	Ours	276	112	64	452
Semi-fixsliced	Ours	332	128	64	524

Table: Number of operations to compute the AES linear layer over 4 rounds when processing 2 blocks in parallel. LOP, LSH and ROT refer to logical operations, logical shifts and rotations, respectively.

Implementation results on ARM Cortex-M4

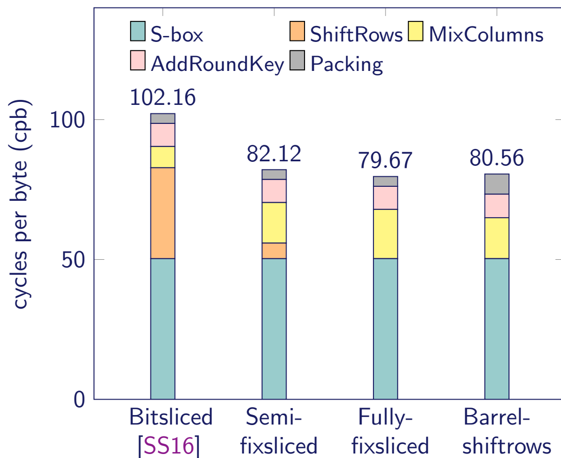


Figure: Benchmark results for AES-128 encryption

Implementation results on RV32I (HiFive1)

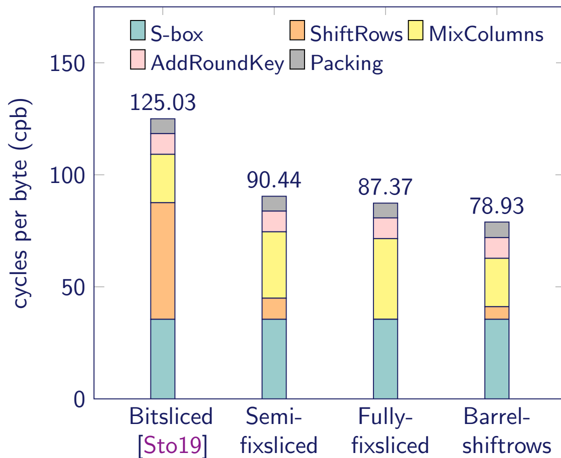


Figure: Benchmark results for AES-128 encryption

Summary results

- Fixslicing outperforms previous bitsliced results by **21% and 30% on ARM Cortex-M4 and RV32I**
- The barrel-shiftrows representation fits well the RV32I architecture
  - Can still be significantly enhanced thanks to the **BitManip extension** [Wol21]
- Directly improves masked AES implementations based on bitslicing
- **Applies to all AES-like ciphers!** An application to Skinny-128 led to improvements up to a factor of 4 compared to previous results
- Our code is available at: `https://github.com/aadomn/aes`

Fixsliced AES already in different projects

Rust Crypto
@RustCryptoOrg

We're working on optimizing the `aes-soft` crate using a state-of-the-art technique called "fixslicing".

We've PoC'd a 32-bit zero-unsafe pure Rust implementation, and preliminary benchmarks show it's 2.5x faster than the previous implementation 🎉

github.com/RustCrypto/blo…

Traduire le Tweet

```
test aes128_encrypt           ... bench:        668 ns/iter (+/- 54) = 23 MB/s
test aes128_encrypt8          ... bench:      3,114 ns/iter (+/- 530) = 41 MB/s
test aes128_encrypt2_fixsliced ... bench:        290 ns/iter (+/- 38) = 110 MB/s


test aes256_encrypt           ... bench:        949 ns/iter (+/- 105) = 16 MB/s
test aes256_encrypt8          ... bench:      4,332 ns/iter (+/- 412) = 29 MB/s
test aes256_encrypt2_fixsliced ... bench:        409 ns/iter (+/- 167) = 78 MB/s
```

6:39 AM · 25 oct. 2020 · Twitter Web App

Fixsliced AES already in different projects

Rust Crypto
@RustCryptoOrg

We're working on optimizing the `aes-soft` crate using a state-of-the-art technique called "fixslicing".

We've PoC'd a 32-bit zero-unsafe pure Rust implementation, and preliminary benchmarks show it's 2.5x faster than the previous implementation 🎉

github.com/RustCrypto/blo…

Traduire le Tweet

```
test aes128_encrypt           ... bench:       668 ns/iter (+/- 54) = 23 MB/s
test aes128_encrypt8          ... bench:     3,114 ns/iter (+/- 530) = 41 MB/s
test aes128_encrypt2_fixsliced ... bench:      290 ns/iter (+/- 38) = 110 MB/s


test aes256_encrypt           ... bench:       949 ns/iter (+/- 105) = 16 MB/s
test aes256_encrypt8          ... bench:     4,332 ns/iter (+/- 412) = 29 MB/s
test aes256_encrypt2_fixsliced ... bench:      409 ns/iter (+/- 167) = 78 MB/s
```

6:39 AM · 25 oct. 2020 · Twitter Web App

mupq / pqm4

Code | Issues 4 | Pull requests 2 | Actions | Projects

Constant-time AES (https://eprint.iacr.org/2020/1123) (#173)

```
* switch to fixsliced AES

* tweak kyber-90s to use t-table AES for public inputs

* update kyber-90s benchmarks with fixsliced AES

* use t-tabe AES in Frodo for public matrix A

* make ntrulpr work with fixsliced AES

* update fixsliced AES from upstream

* update performance of kyber-90s, ntrulpr, and hqc with new fixsliced AES

* update AES information in README

* rename _leaktime to _publicinputs

* switch to mupq master; simply change include order
```

master (#173)

mkannwischer committed 19 days ago

1 parent 157e271   commit 6841a6bc3cc5bc0b0e01e5ee33567882e9bca8d3

References

Alexandre Adomnicai, Zakaria Najm, and Thomas Peyrin.
Fixslicing: A New GIFT Representation: Fast Constant-Time Implementations of GIFT and GIFT-COFB on ARM Cortex-M.
IACR Transactions on Cryptographic Hardware and Embedded Systems, 2020(3):402–427, Jun. 2020.

Emilia Käsper and Peter Schwabe.
Faster and Timing-Attack Resistant AES-GCM.
In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, pages 1–17, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

Peter Schwabe and Ko Stoffelen.
All the AES You Need on Cortex-M3 and M4.
In *Selected Areas in Cryptography - SAC 2016*, pages 180–194, 2016.

Ko Stoffelen.
Efficient Cryptography on the RISC-V Architecture.
In Peter Schwabe and Nicolas Thériault, editors, *Progress in Cryptology – LATINCRYPT 2019*, pages 323–340, Cham, 2019. Springer International Publishing.

Claire Wolf.
RISC-V Bitmanip Extension (Document Version 0.94-draft), 2021.

Thanks for your attention!

Feel free to contact us for any questions/remarks

alex.adomnicai@gmail.com
thomas.peyrin@ntu.edu.sg