FACE : Fast AES CTR mode Encryption Techniques based on the Reuse of Repetitive Data

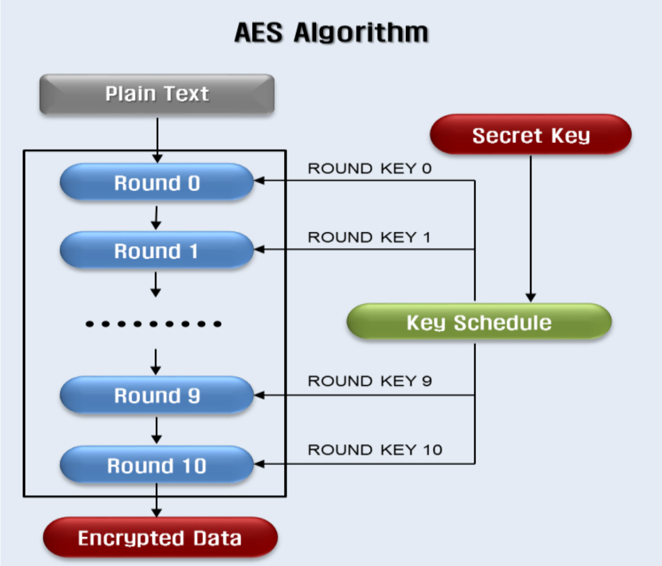
Jin Hyung Park and Dong Hoon Lee

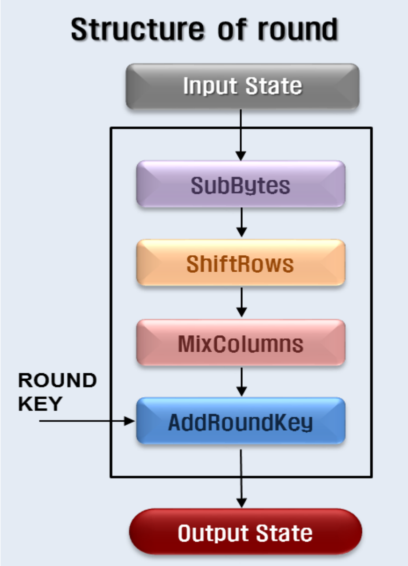
Center for Information Security Technologies,
Korea University

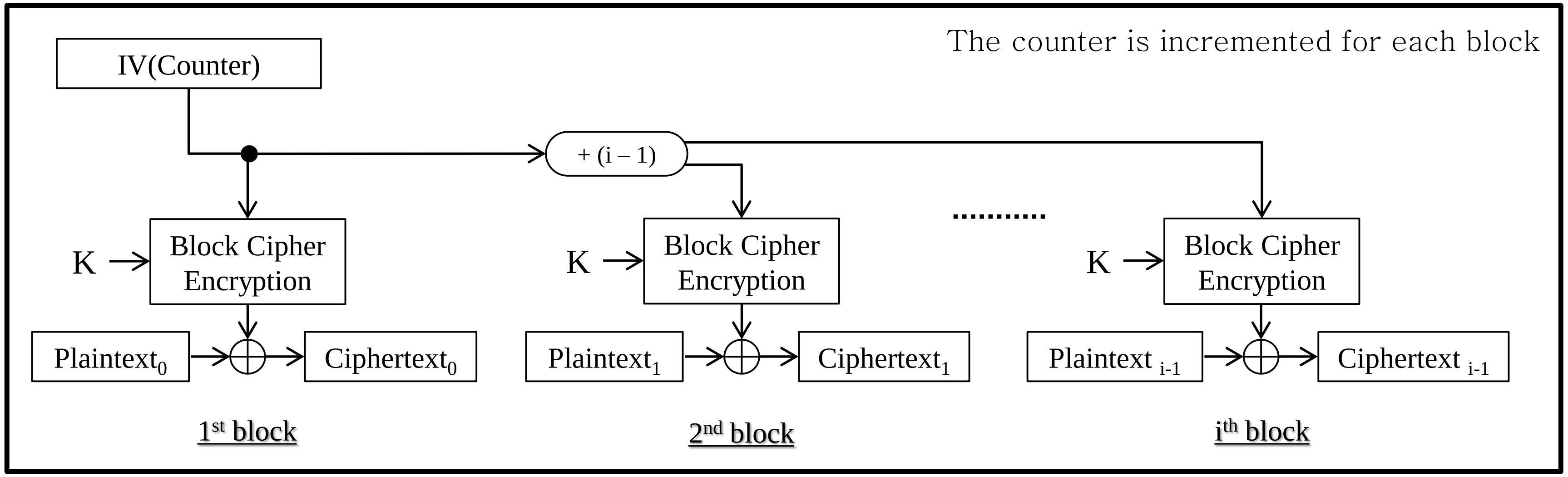
CHES 2018, Amsterdam, The Netherlands

Introduction

- ❖ **AES (FIPS 197) algorithm and Counter mode**
  - Used for numerous services as encryption technique
    - OMA DRM v2.0 : PDCF format
    - IPTV, VoIP : SecureRTP
    - SSH, SSL/TLS, and etc.
  - Parallel processing
  - Does not need to implement decryption algorithm
  - Be used in Authenticated Encryption (e.g. GCM, CCM)



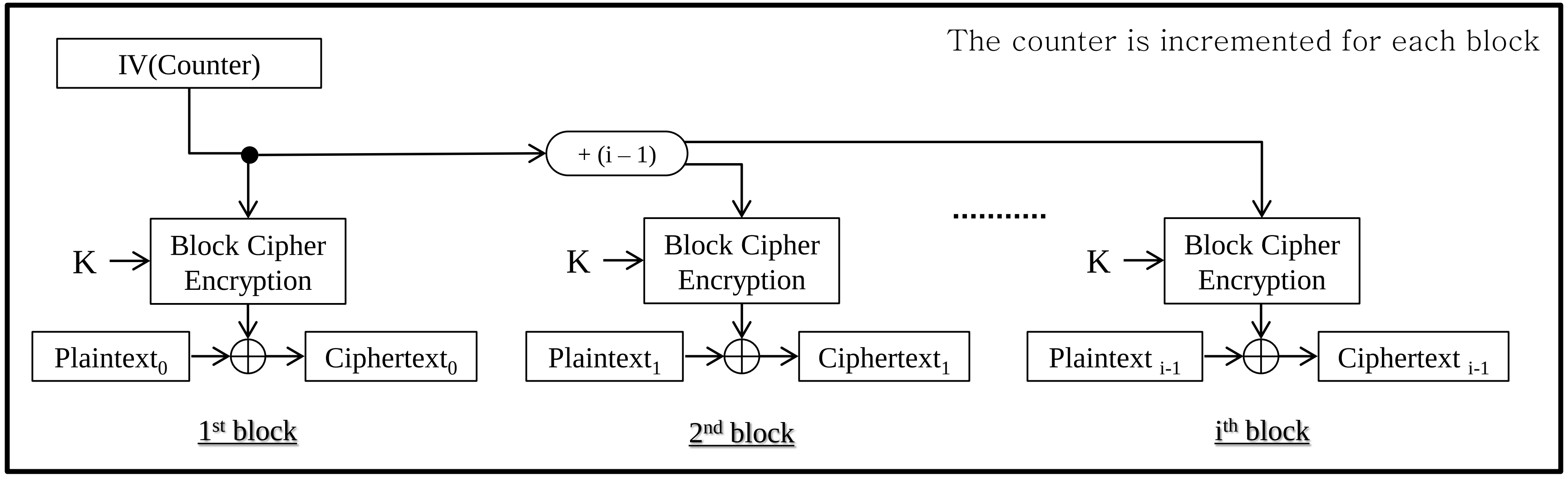




Introduction

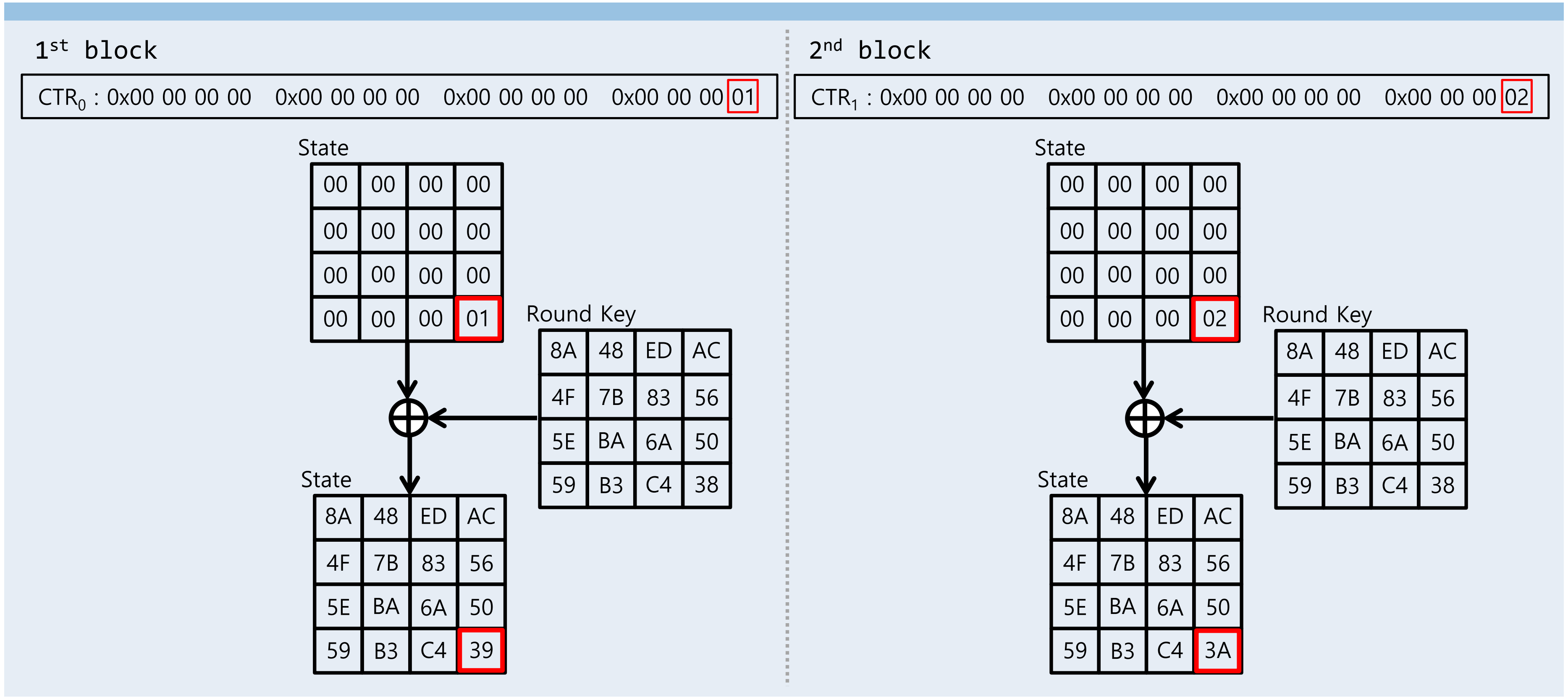
❖ **The feature of AES, which is used in Counter mode**

- **16 bytes counter is increased by 1 (more precisely, pre-defined value) for every block**
- **15 bytes of the counter remain constant for 256 blocks**



Introduction

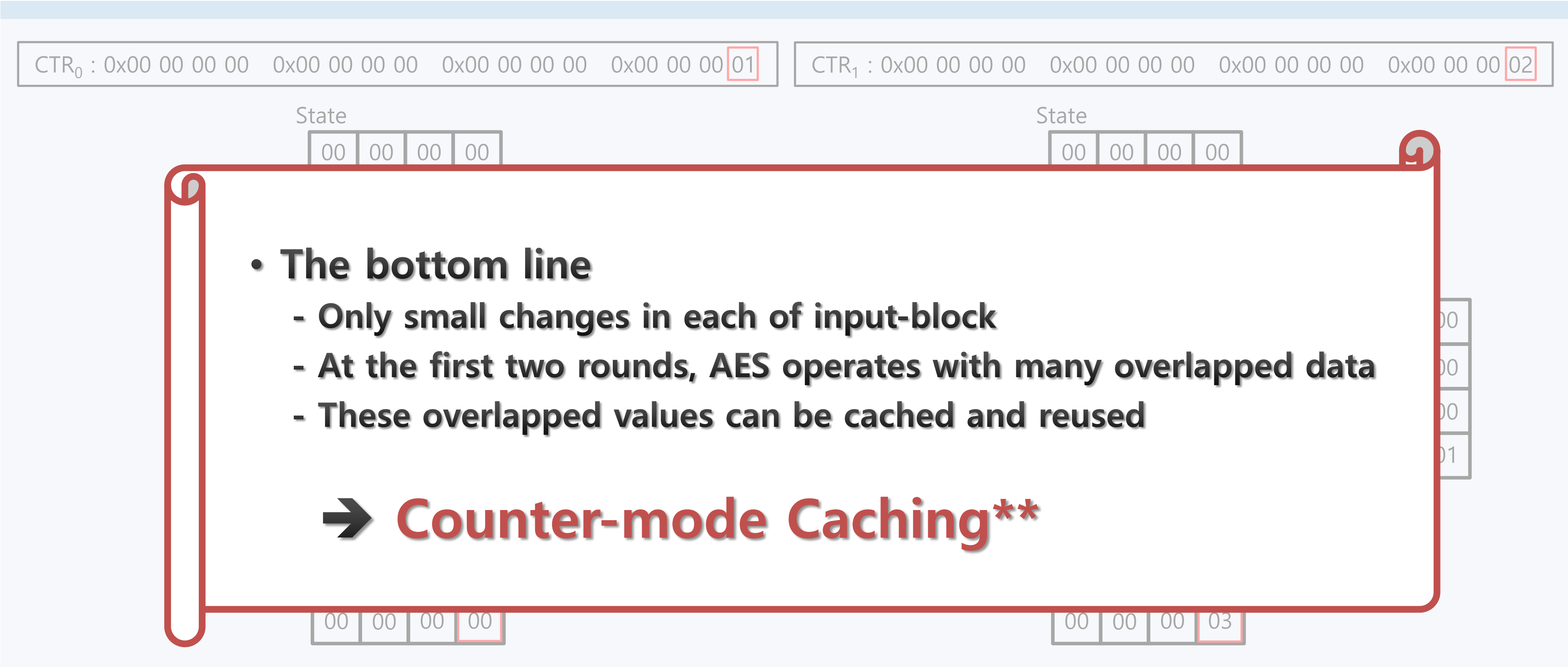
- **The feature of AES, which is used in Counter mode**
  - **16 bytes counter is increased by 1 (more precisely, pre-defined value) for every block**
  - **15 bytes of the counter remain constant for 256 blocks**



< Initial Whitening phase of AES >

Introduction

- ❖ **The feature of AES, which is used in Counter mode**
  - 16 bytes counter is increased by 1 (more precisely, pre-defined value) for every block
  - 15 bytes of the counter remain constant for 256 blocks



- **The bottom line**
  - Only small changes in each of input-block
  - At the first two rounds, AES operates with many overlapped data
  - These overlapped values can be cached and reused

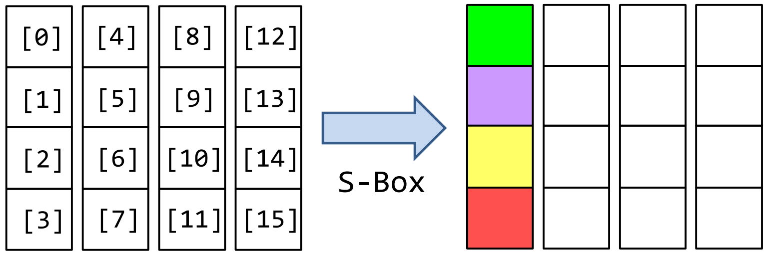
➔ **Counter-mode Caching****

* Daniel J Bernstein and Peter Schwabe, "New AES software speed records", INDOCRYPT 2008
** HongJun Wu, "Hongjun's optimized C-code for AES-128 and AES-256", eSTREAM Project, 2007

Round Function - 4 Transformations

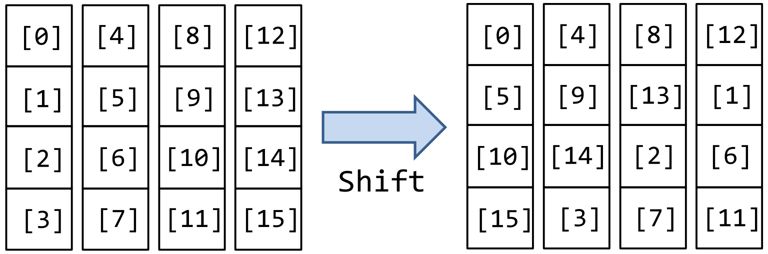
❖ SubBytes

- Substitutes one byte with another byte (S-Box)
- Each byte has no relationship with the other
- Same input always produces same output



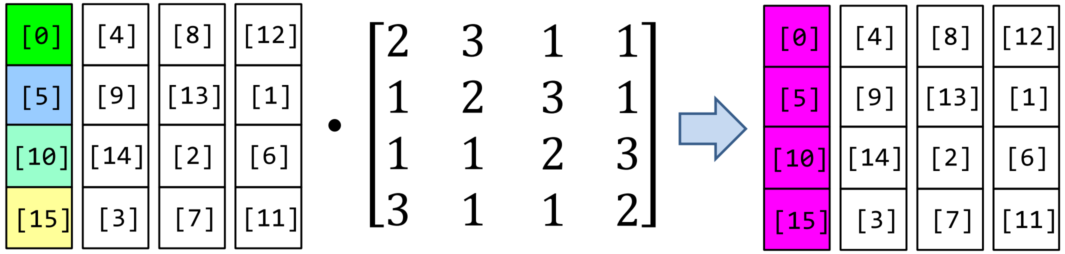
❖ ShiftRows

- Circularly transposes rows from right to left
- The amount of moving is relevant to the row pos.



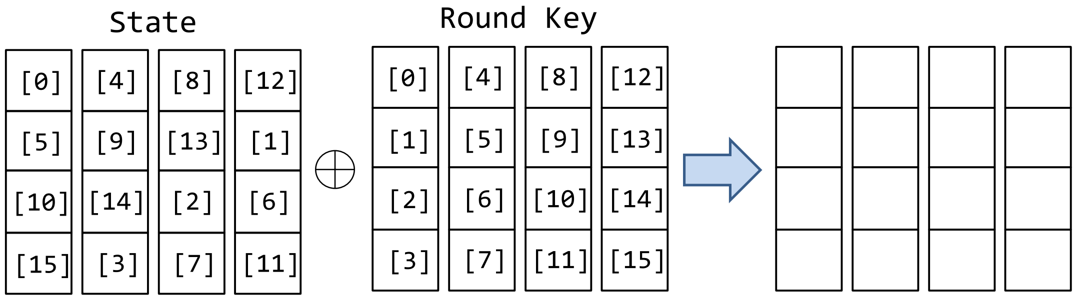
❖ Mixcolumns

- Is peformed column-by-column
- Combines the four bytes in each column



❖ AddRoundKey

- Simply XORs a given ***State*** with round keys



AES Implementation Methods

❖ Table-based Implementation

- Uses pre-computation tables

```
static const u32 Te0[256] = {
    0xc66363a5U, 0xf87c7c84U, 0xee777799U, 0xf67b7b8dU,
    0xfff2f20dU, 0xd66b6bbdU, 0xde6f6fb1U, 0x91c5c554U,
    0x60303050U, 0x02010103U, 0xce6767a9U, 0x562b2b7dU,
    0xe7fefe19U, 0xb5d7d762U, 0x4dababe6U, 0xec76769aU,
        …
    0x824141c3U, 0x299999b0U, 0x5a2d2d77U, 0x1e0f0f11U,
    0x7bb0b0cbU, 0xa85454fcU, 0x6dbbbbd6U, 0x2c16163aU,
};
```

• • •

```
static const u32 Te3[256] = {
    0x6363a5c6U, 0x7c7c84f8U, 0x777799eeU, 0x7b7b8df6U,
    0xf2f20dffU, 0x6b6bbdd6U, 0x6f6fb1deU, 0xc5c55491U,
    0x30305060U, 0x01010302U, 0x6767a9ceU, 0x2b2b7d56U,
    0xfefe19e7U, 0xd7d762b5U, 0xababe64dU, 0x76769aecU,
    …
    0x4141c382U, 0x9999b029U, 0x2d2d775aU, 0x0f0f111eU,
    0xb0b0cb7bU, 0x5454fca8U, 0xbbbbd66dU, 0x16163a2cU,
};
```

< OpenSSL >

```
s0 = GETU32(in     ) ^ rk[0];
s1 = GETU32(in +  4) ^ rk[1];
s2 = GETU32(in +  8) ^ rk[2];
s3 = GETU32(in + 12) ^ rk[3];

/* round 1: */
t0 = Te0[s0 >> 24] ^ Te1[(s1 >> 16) & 0xff] ^ Te2[(s2 >>  8) & 0xff] ^ Te3[s3 & 0xff] ^ rk[ 4];
t1 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >>  8) & 0xff] ^ Te3[s0 & 0xff] ^ rk[ 5];
t2 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >>  8) & 0xff] ^ Te3[s1 & 0xff] ^ rk[ 6];
t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >>  8) & 0xff] ^ Te3[s2 & 0xff] ^ rk[ 7];
```

AES Implementation Methods

❖ Table-based Implementation

- Uses pre-computation tables

```
static const u32 Te0[256] = {
    0xc66363a5U, 0xf87c7c84U, 0xee777799U, 0xf67b7b8dU,
    0xfff2f20dU, 0xd66b6bbdU, 0xde6f6fb1U, 0x91c5c554U,
    0x60303050U, 0x02010103U, 0xce6767a9U, 0x562b2b7dU,
    0xe7fefe19U, 0xb5d7d762U, 0x4dababe6U, 0xec76769aU,
      …
    0x824141c3U, 0x299999b0U, 0x5a2d2d77U, 0x1e0f0f11U,
    0x7bb0b0cbU, 0xa85454fcU, 0x6dbbbbd6U, 0x2c16163aU,
};
```

• • •

```
static const u32 Te3[256] = {
    0x6363a5c6U, 0x7c7c84f8U, 0x777799eeU, 0x7b7b8df6U,
    0xf2f20dffU, 0x6b6bbdd6U, 0x6f6fb1deU, 0xc5c55491U,
    0x30305060U, 0x01010302U, 0x6767a9ceU, 0x2b2b7d56U,
    0xfefe19e7U, 0xd7d762b5U, 0xababe64dU, 0x76769aecU,
    …
    0x4141c382U, 0x9999b029U, 0x2d2d775aU, 0x0f0f111eU,
    0xb0b0cb7bU, 0x5454fca8U, 0xbbbbd66dU, 0x16163a2cU,
};
```

< OpenSSL >

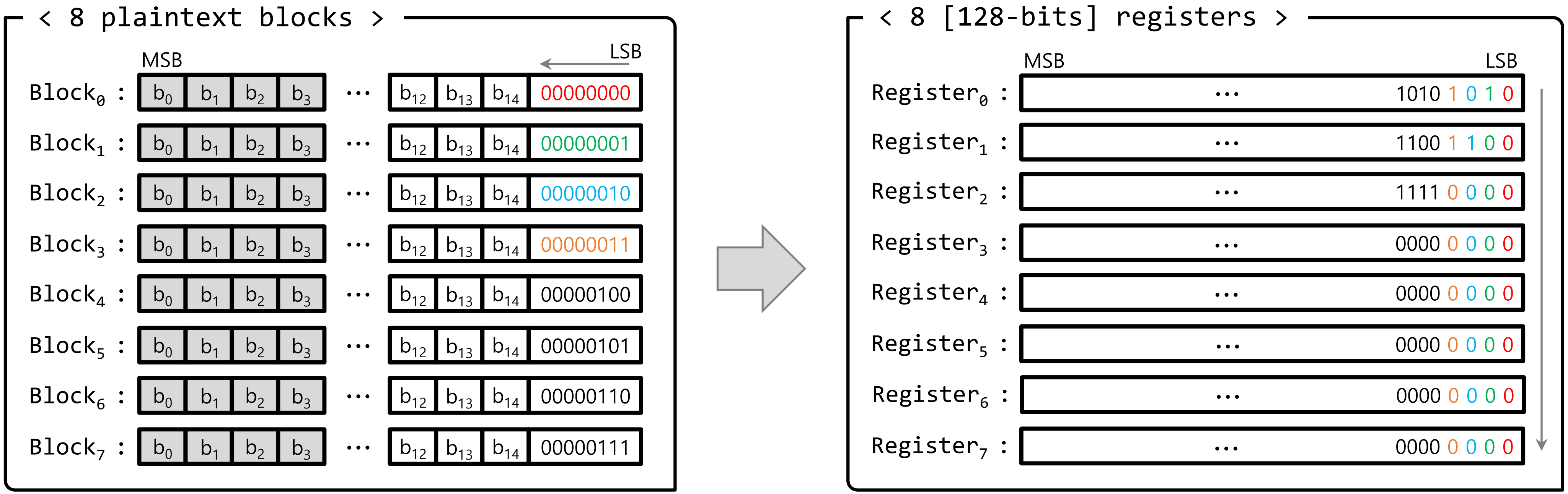
```
s0 = GETU32(in     ) ^ rk[0];
s1 = GETU32(in +  4) ^ rk[1];
s2 = GETU32(in +  8) ^ rk[2];
s3 = GETU32(in + 12) ^ rk[3];

/* round 1: */
t0 = Te0[s0 >> 24] ^ Te1[(s1 >> 16) & 0xff] ^ Te2[(s2 >>  8) & 0xff] ^ Te3[s3 & 0xff] ^ rk[ 4];
t1 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >>  8) & 0xff] ^ Te3[s0 & 0xff] ^ rk[ 5];
t2 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >>  8) & 0xff] ^ Te3[s1 & 0xff] ^ rk[ 6];
t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >>  8) & 0xff] ^ Te3[s2 & 0xff] ^ rk[ 7];
```

→ Vulnerable to Cache timing attack

AES Implementation Methods

❖ Bitsliced Implementation

- First proposed by Biham to improve the software performance of DES (1997)
- Simulates a hardware implementation in software (sequence of Boolean operations)



< bitsliced form transformation (OpenSSL implementation based on [1]) >

[1] : Emilia Käsper and Peter Schwabe, “Faster and Timing-Attack Resistant AES-GCM”, CHES 2009

AES Implementation Methods

❖ AES-NI (Hardware acceleration)

- Intel supports AES instruction set since Westmere processor (in March 2008)
- Support 7 instructions

Instruction	Description
AESENC	Perform one round of an AES encryption flow
AESENCLAST	Perform the last round of an AES encryption flow
AESDEC	Perform one round of an AES decryption flow
AESDECLAST	Perform the last round of an AES decryption flow
AESKEYGENASSIST	Assist in AES round key generation
AESIMC	Assist in AES Inverse Mix Columns
PCLMULQDQ	Carryless multiply

< Crypto++ >

```
*block = _mm_xor_si128( *block , skeys[0] ) ;

/* round 1: */
*block = _mm_aesenc_si128 ( *block , skeys[1] ) ;
```

AES Implementation Methods

❖ **Fastest throughput of each method**

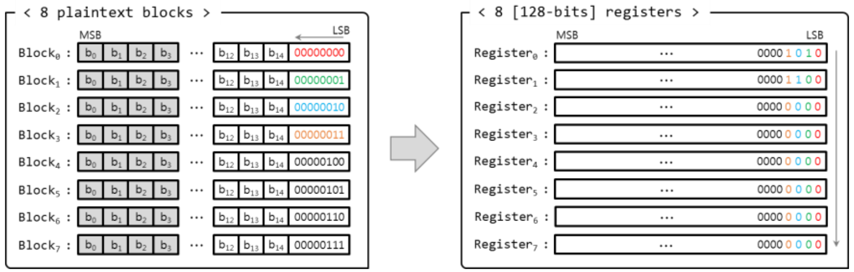
Method	Performance (Cycles per Byte)	Test Environment	Reference
Table-based	10.57 + α (not for CTR)	Core 2 Quad Q6600	INDOCRYPT 2008 [1]
Bitslicing	9.32	Core 2 Quad Q6600	CHES 2009 [2]
	7.59	Core 2 Quad Q9550	
AES-NI	1.4 - 2.0	Westmere Processor	INTEL whitepaper [3]
	0.57	Skylake Core i5	Crypto++ Benchmark [4]

[1] : Daniel J. Bernstein and Peter Schwabe, “New AES software speed records”, INDOCRYPT 2008
[2] : Emilia Käsper and Peter Schwabe, “Faster and Timing-Attack Resistant AES-GCM”, CHES 2009
[3] : Shay Gueron, “Intel Advanced Encryption Standard (AES) New Instructions Set”, May, 2010
(The first Westmere-based processors (that supports AES-NI) were launched on Jan, 2010.)
[4] : Crypto++ 6.0.0 Benchmarks, https://www.cryptopp.com/benchmarks.html, 2017. 12

Problem

- **Previous Counter-mode Caching can not work effectively on bitsliced and AES-NI-based implementations**
  - **It only covered partial data of round transformation**
    - → **The rest (which was not cached) should be calculated in every block**

Bitslice



During a format conversion, each byte of input is sliced bitwise.
And the sliced bits are spread
in the corresponding positions of each register

Necessary input bytes to calculate the rest
are spread to whole register

Almost the whole instructions of
previous implementation should be performed
with additional operations (save, load, merge)

AES-NI

aesenc xmm15, xmm1 → only 1 instruction performs round operation

Adding some operations to calculate the rest
becomes a considerable burden
even if instruction latency and
throughput differ from each instruction

Such operations (for the rest)
should be composed of
several instructions

Our Work (FACE)

- We propose an efficient implementation technique for the CTR mode of AES (FACE)
  - Extends the counter-mode caching
  - Can be employed, regardless of the platform, environment, or implementation method
- We show that FACE can be applied to existing implementation methods
  - Table-based, bitsliced, and AES-NI-based implementations
  - The first to combine counter-mode caching with bitsliced implementation
  - The first to apply counter-mode caching up to the round transformations of AES-NI implementation
- Our proposal (FACE) records the highest throughput ever achieved
  - Bitslice : 6.41 cycles/byte on an Intel Core 2 Q9550 (previous record : 7.59 cycles/byte)
  - AES-NI : 0.44 cycles/byte on an Intel Core i7 8700K (previous record : 0.55 cycles/byte)

Fast AES Counter mode Encryption

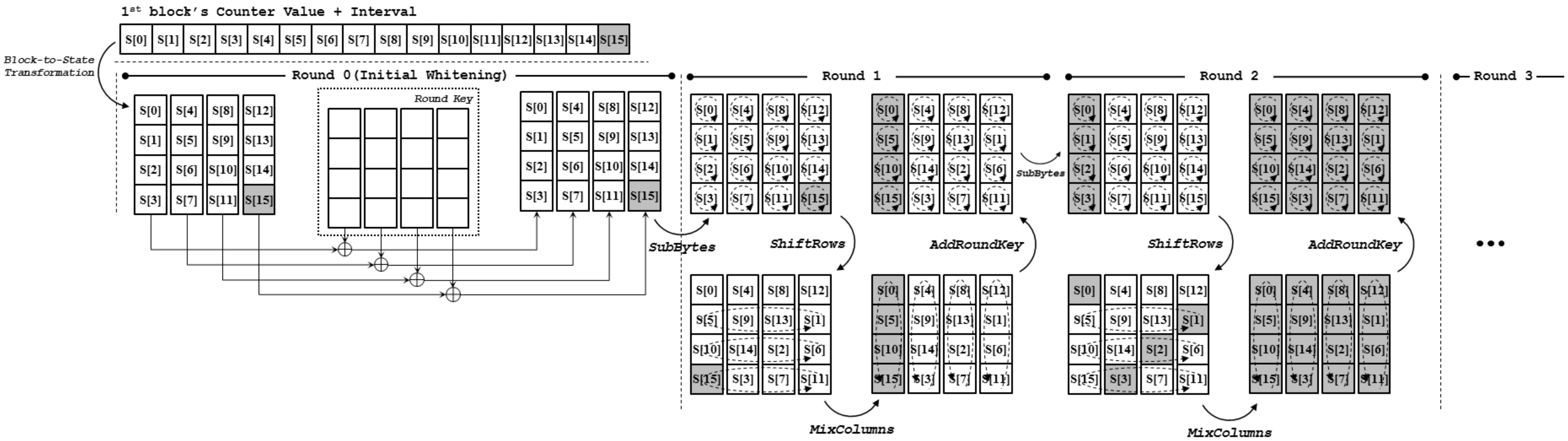
FACE (Fast AES Counter Mode Encryption)

❖ **5 types of reuse techniques**

- FACE$_{rd0}$
  - Cache 12 bytes of round 0's result
  - Reuse for 2^{32}-1 successive blocks

- FACE$_{rd1}$
  - Cache 12 bytes of round 1's result
  - Reuse for 255 successive blocks
- FACE$_{rd1+}$
  - Generate Pre-computation Table (1K)
  - Reuse for 2^{40} successive blocks

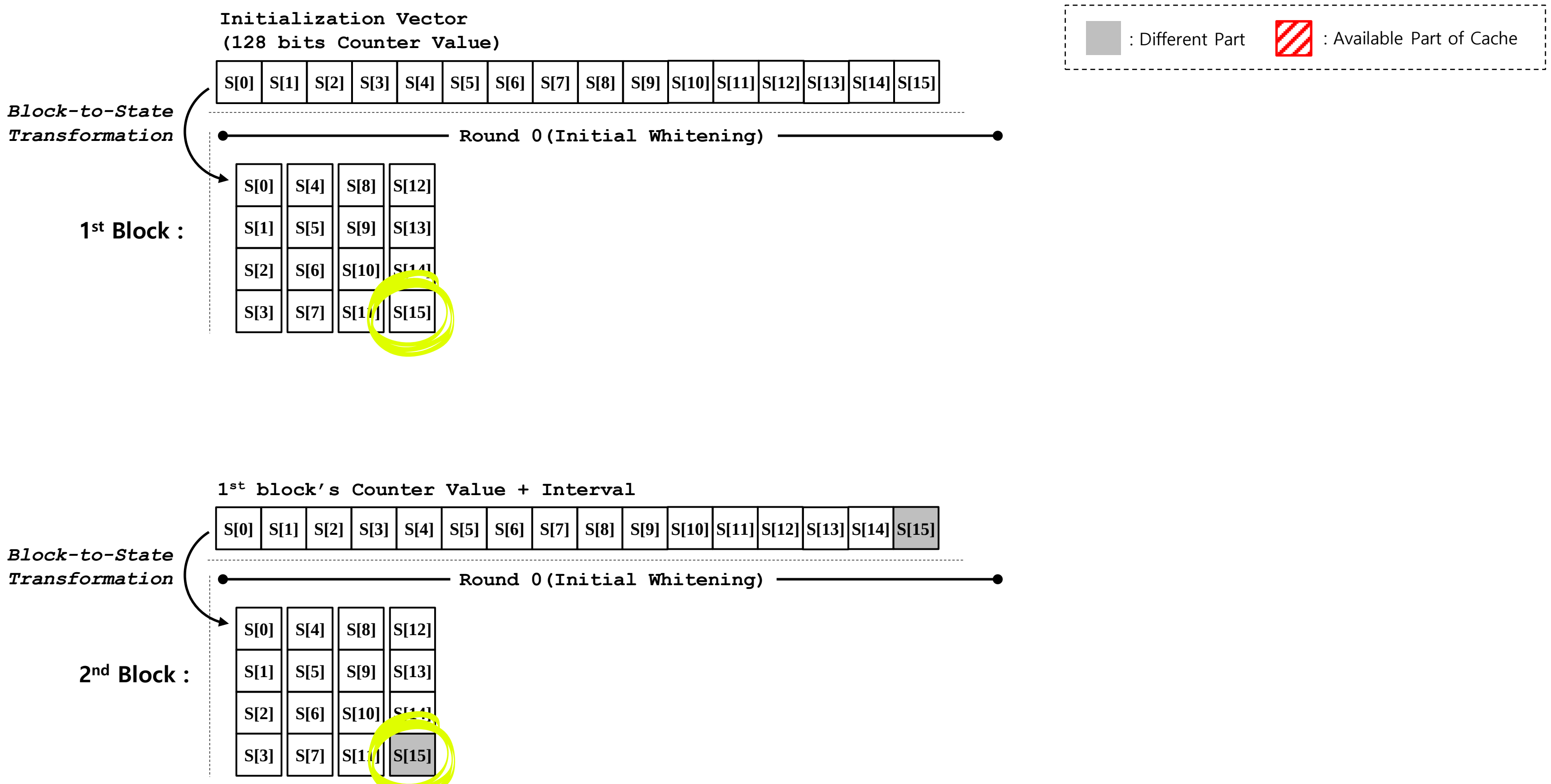
- FACE$_{rd2}$
  - Cache 16 bytes of round 2
  - Reuse for 255 successive blocks
- FACE$_{rd2+}$
  - Generate Pre-computation Table (4K)
  - Reuse for 2^{40} successive blocks



Fast AES Counter mode Encryption

$FACE_{rd0}$

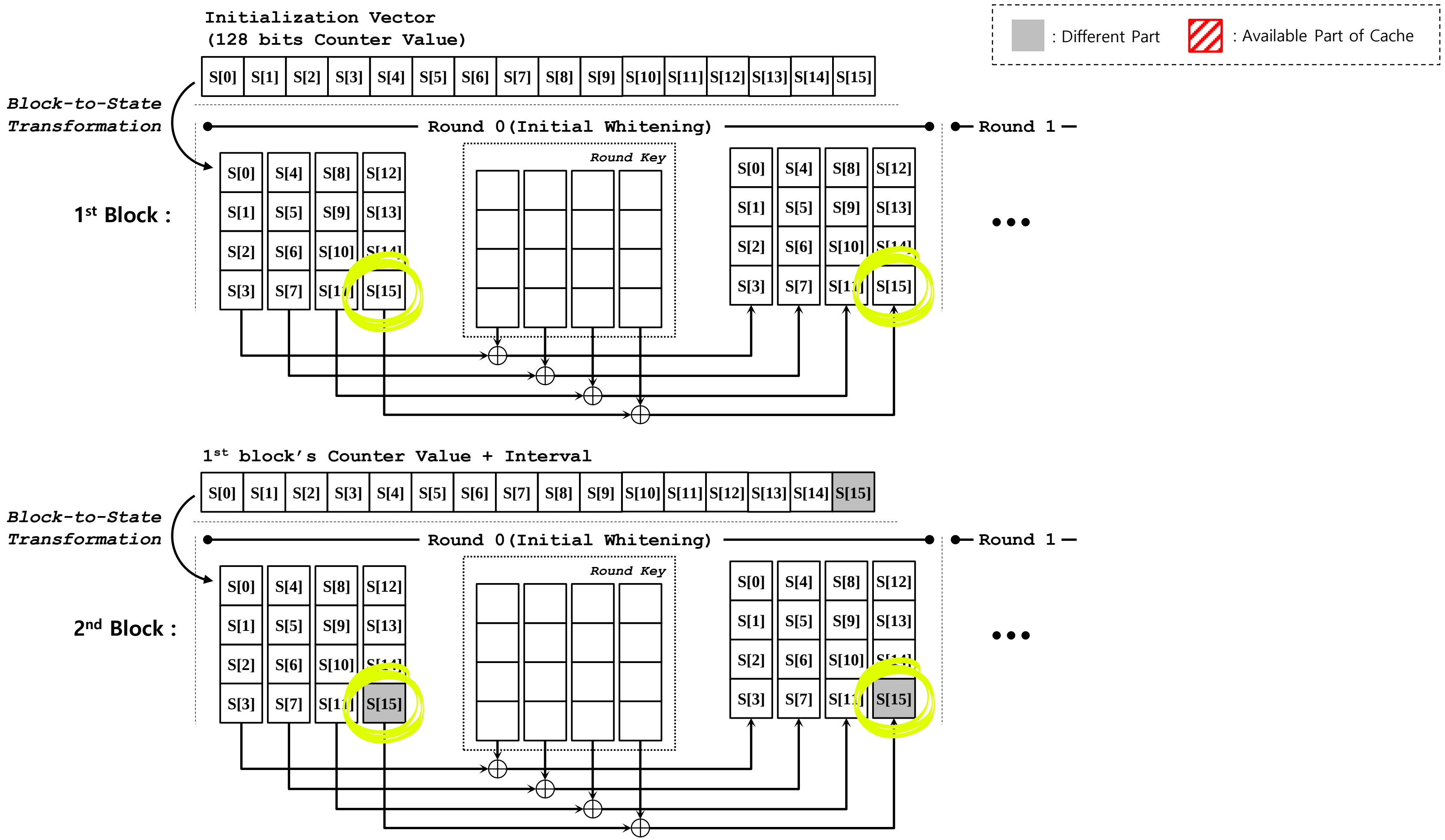
- 16 bytes counter is increased by 1 for every block
- 15 bytes of the counter remain constant for **256 block**
- **The difference** between one block and next block is just **last 1 byte**



Fast AES Counter mode Encryption

$FACE_{rd0}$

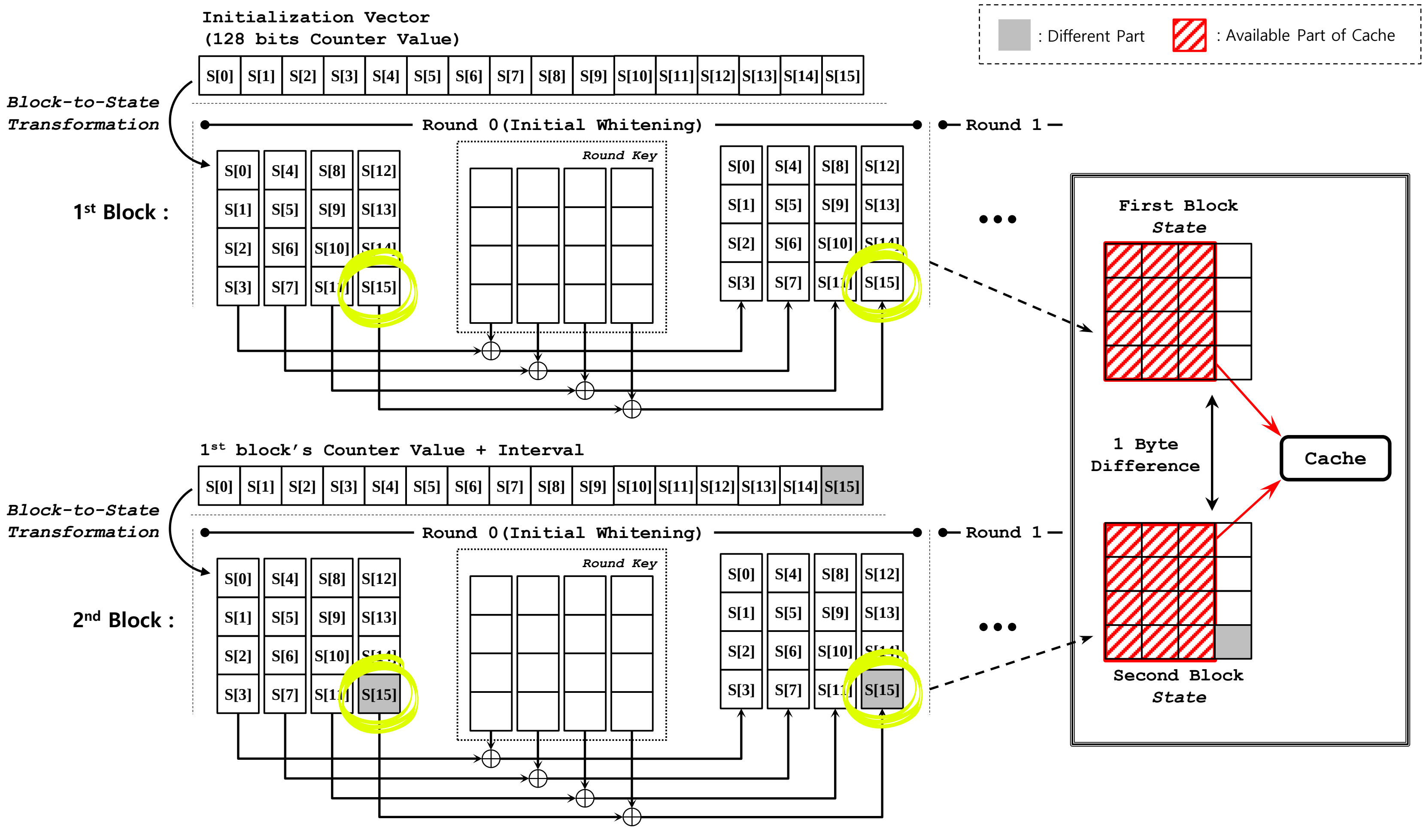
- 16 bytes counter is increased by 1 for every block
- 15 bytes of the counter remain constant for 256 block
- The difference between one block and next block is just last 1 byte



Fast AES Counter mode Encryption

$FACE_{rd0}$

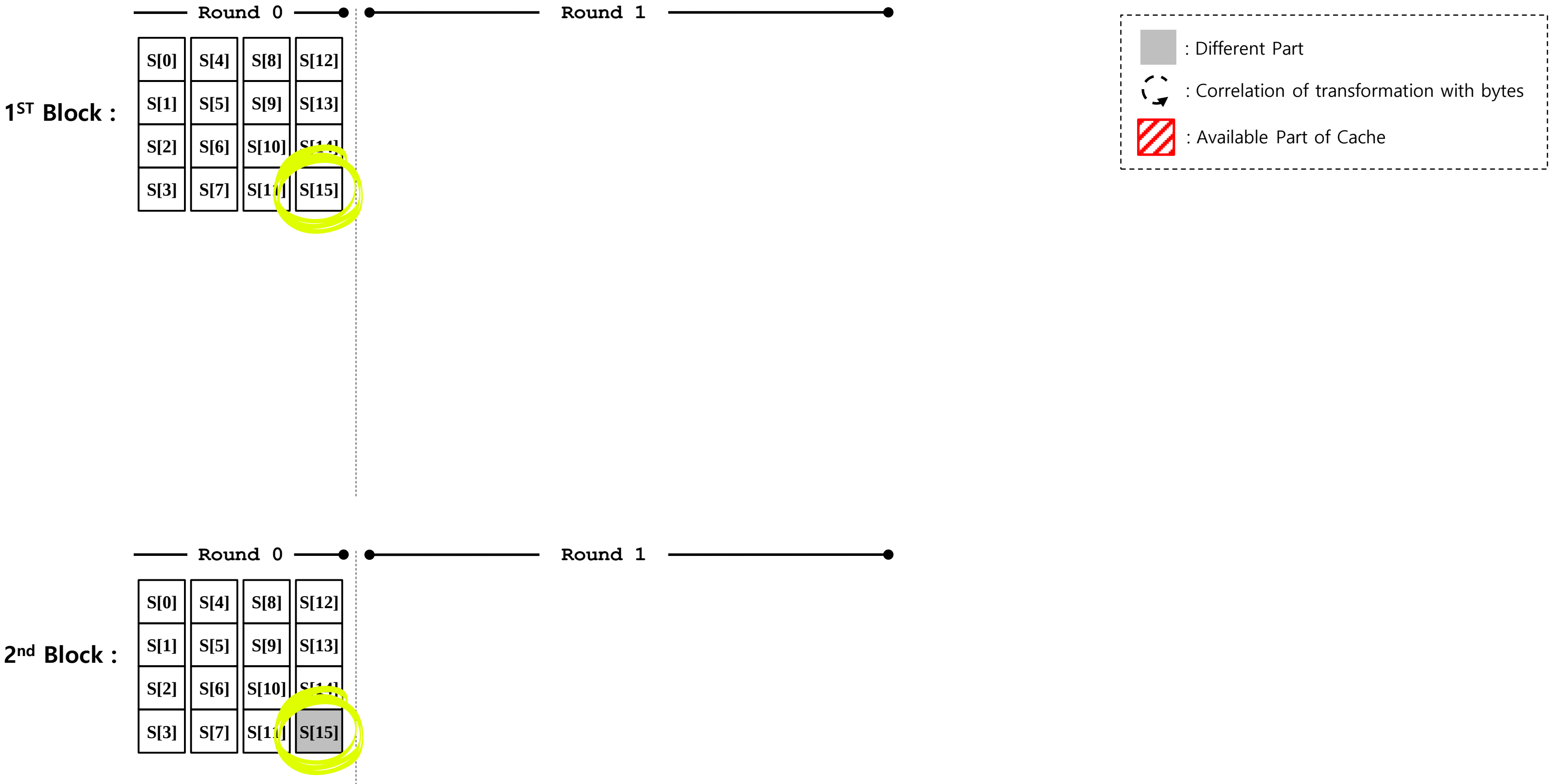
- 16 bytes counter is increased by 1 for every block
- 15 bytes of the counter remain constant for **256 block**
- **The difference** between one block and next block is just **last 1 byte**
- Cache **3 columns** of Initial whitening (Round 0)
- The cached value can be reused in $2^{32}-1$ consecutive blocks



Fast AES Counter mode Encryption

FACE$_{rd1}$

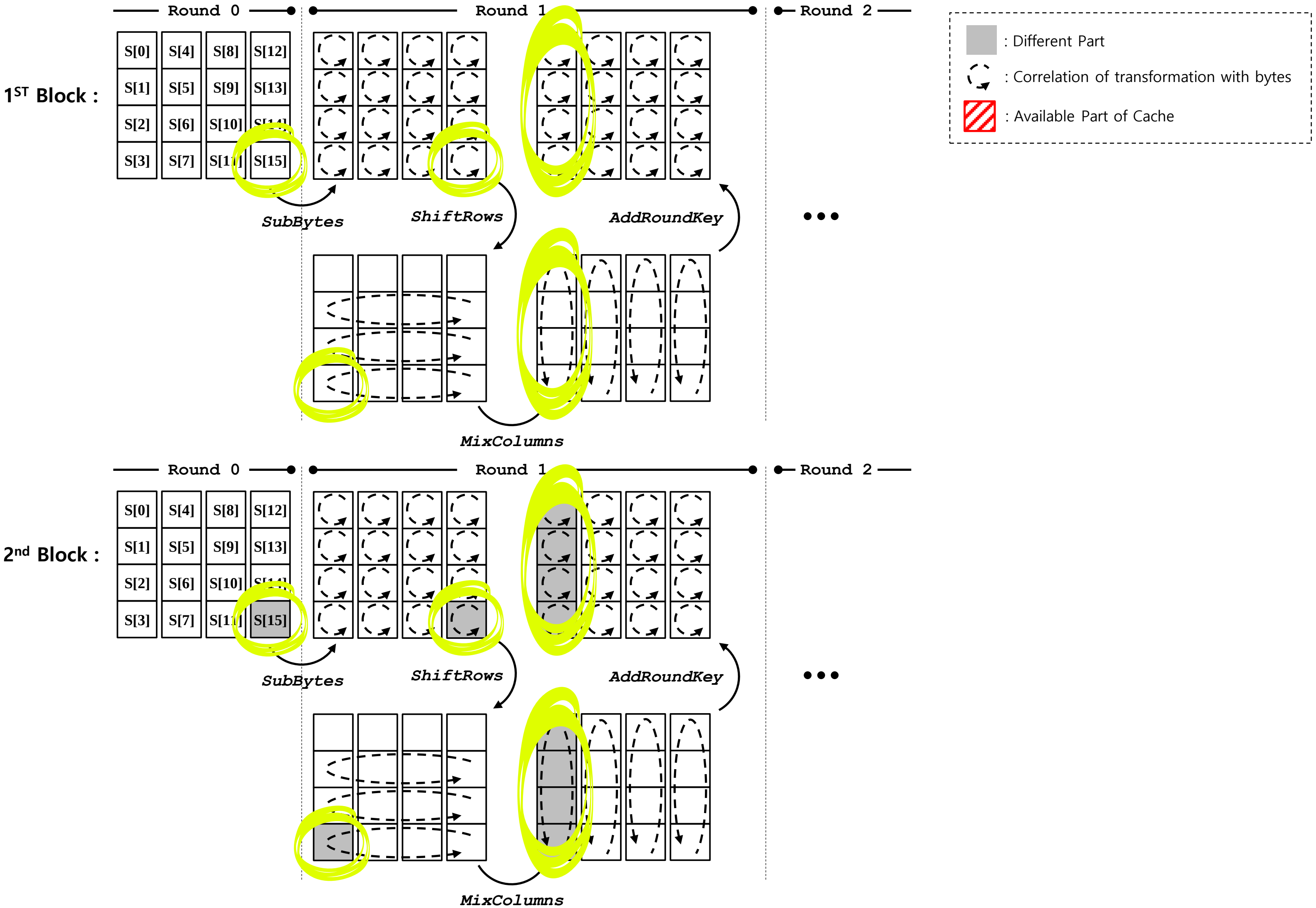
- **The difference** between two input blocks is just **last 1 byte**



Fast AES Counter mode Encryption

FACE$_{rd1}$

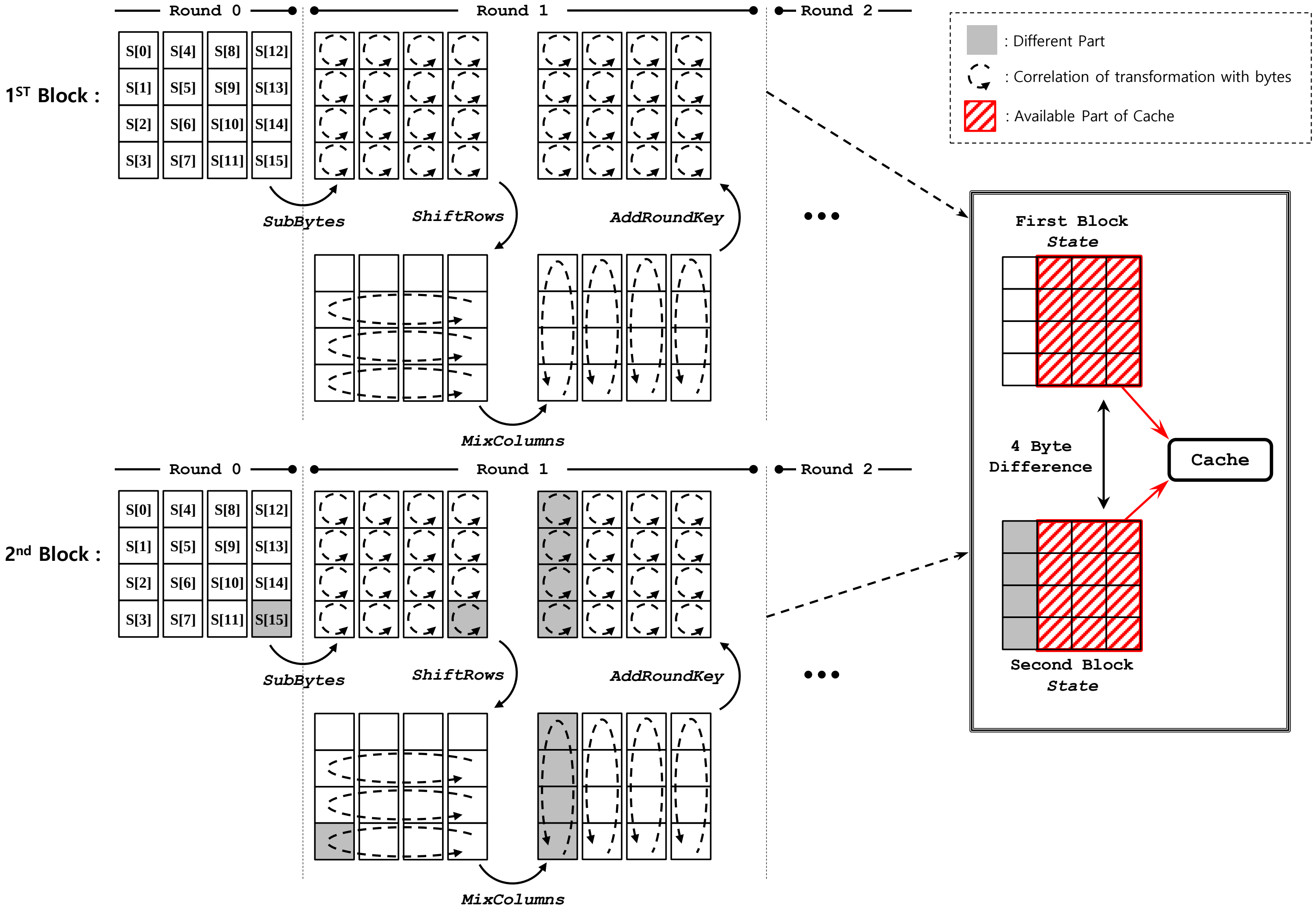
- **The difference** between two input blocks is just **last 1 byte**
- **This difference spreads** by ShiftRows() and Mixcolumns() operation



Fast AES Counter mode Encryption

$FACE_{rd1}$

- **The difference** between two input blocks is just **last 1 byte**
- **This difference spreads** by ShiftRows() and Mixcolumns() operation
- Cache **3 columns** of Round 1 result (12 bytes)
- The cached value can be reused in **255** consecutive blocks



Fast AES Counter mode Encryption

$FACE_{rd1+}$

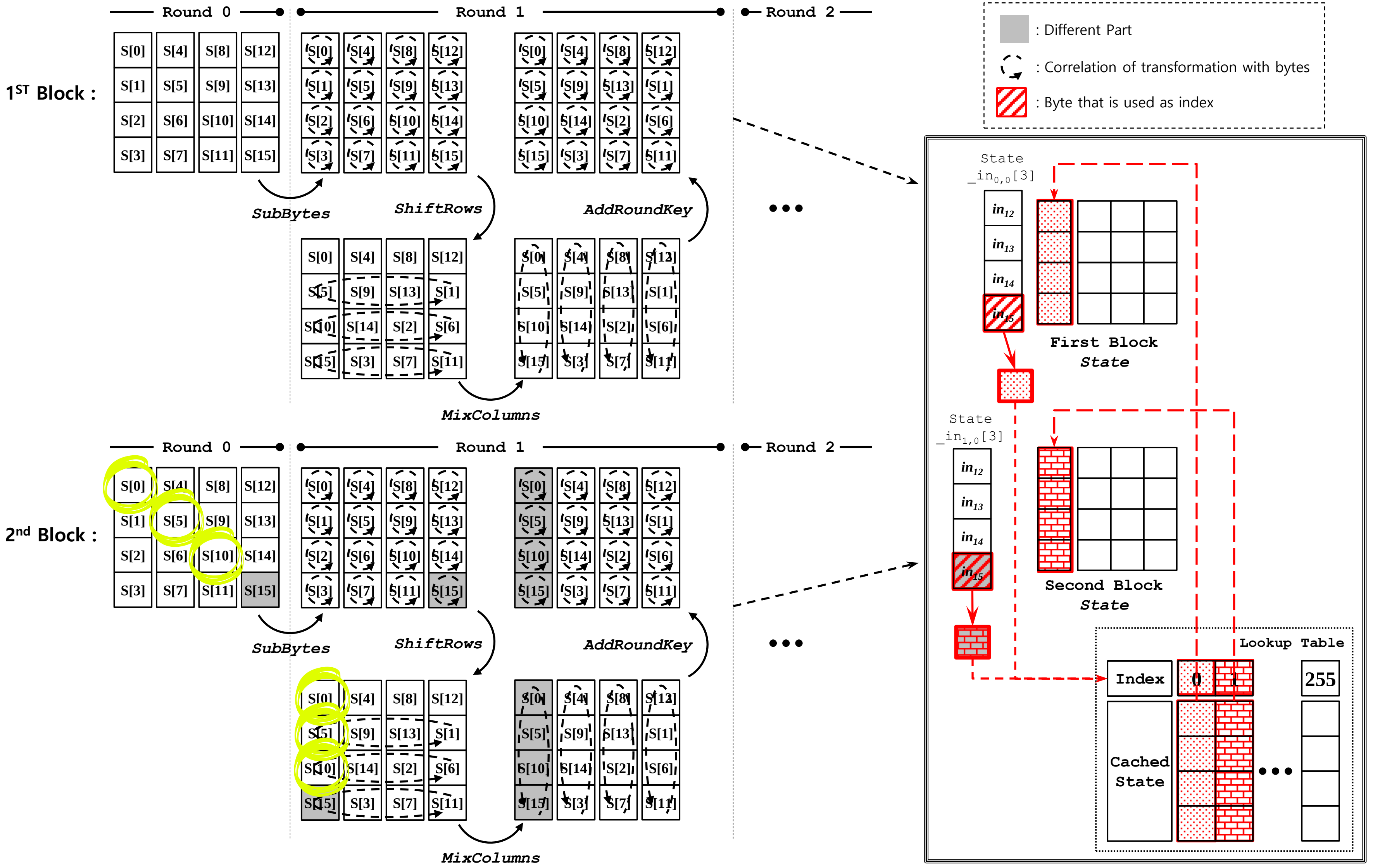
- Generate **Pre-computation lookup table** (size : 1KB)

KOREA UNIVERSITY

Fast AES Counter mode Encryption

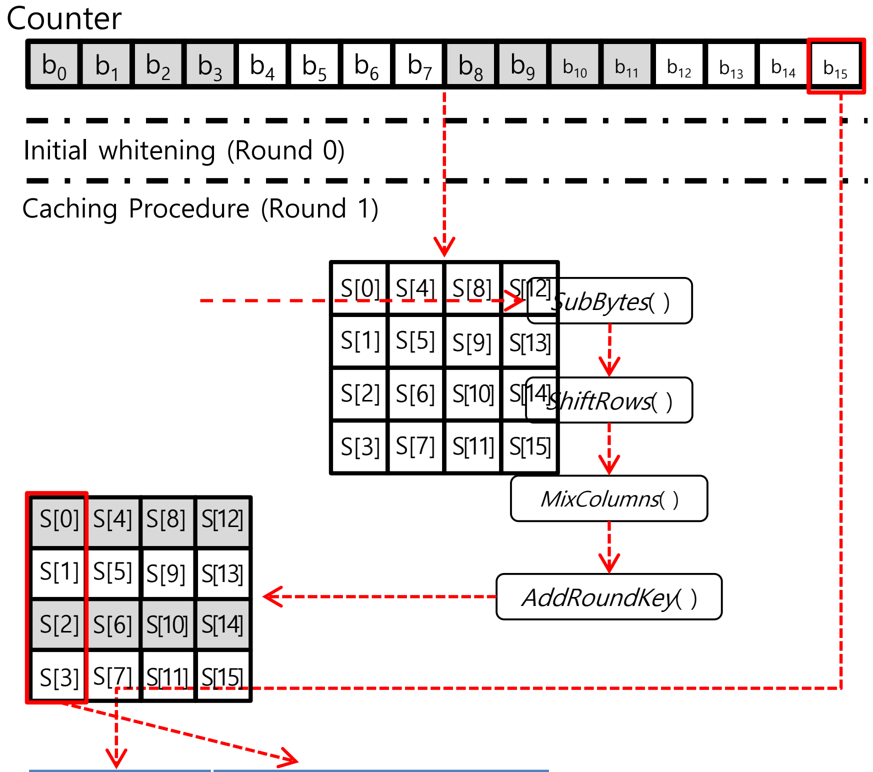
$FACE_{rd1+}$

- Generate Pre-computation lookup table (size : 1KB)
- Store and Reuse the first column of round 1
- The lookup table can be used in 2^{40} consecutive blocks (1,099,511,627,776 block = 17,592,186,044,416 bytes = 16 TB)
- The lookup index is the last byte of the counter

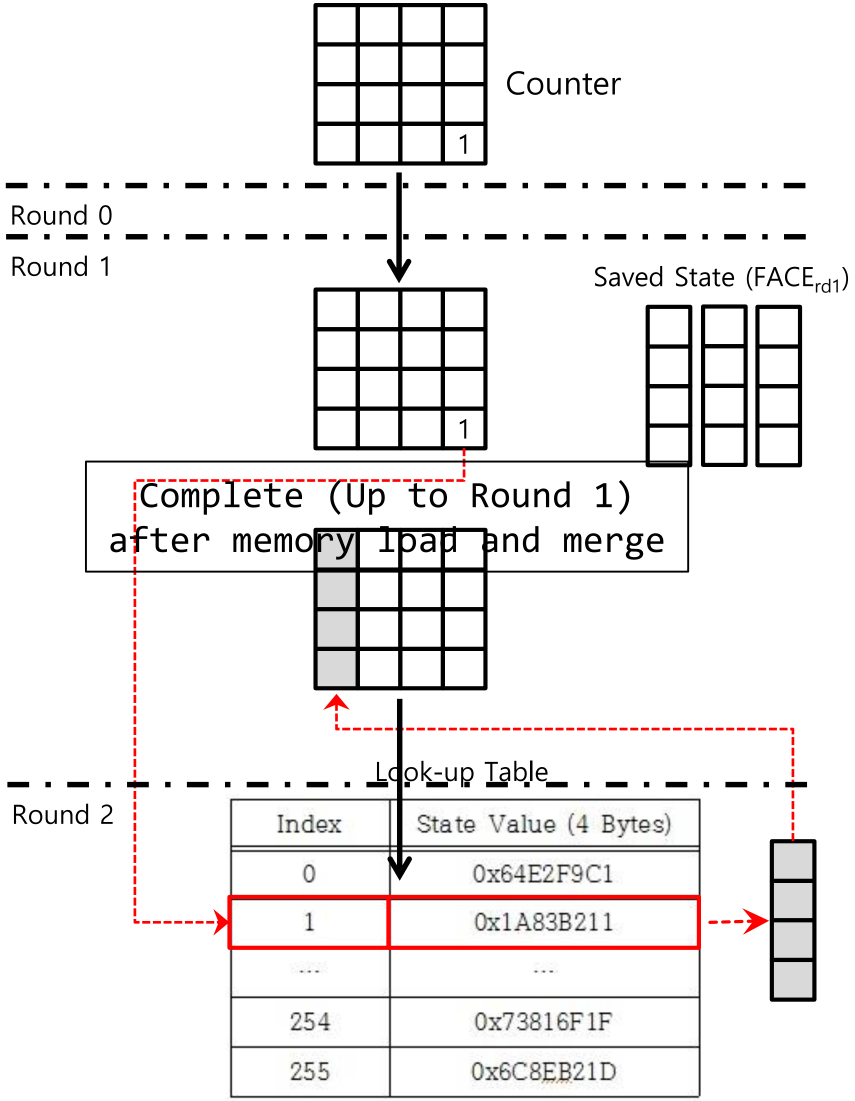


Fast AES Counter mode Encryption

Leverage $FACE_{rd1}$ & $FACE_{rd1+}$

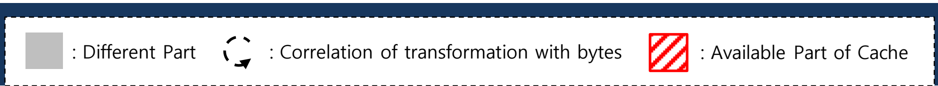


Index(b_{15})	State Value(4 Bytes)
0	0x64E2F9C1
1	0xS[0]S[1]S[2]S[3] / 0x1A83B211
…	…
254	0x73816F1F
255	0x6C8EB21D

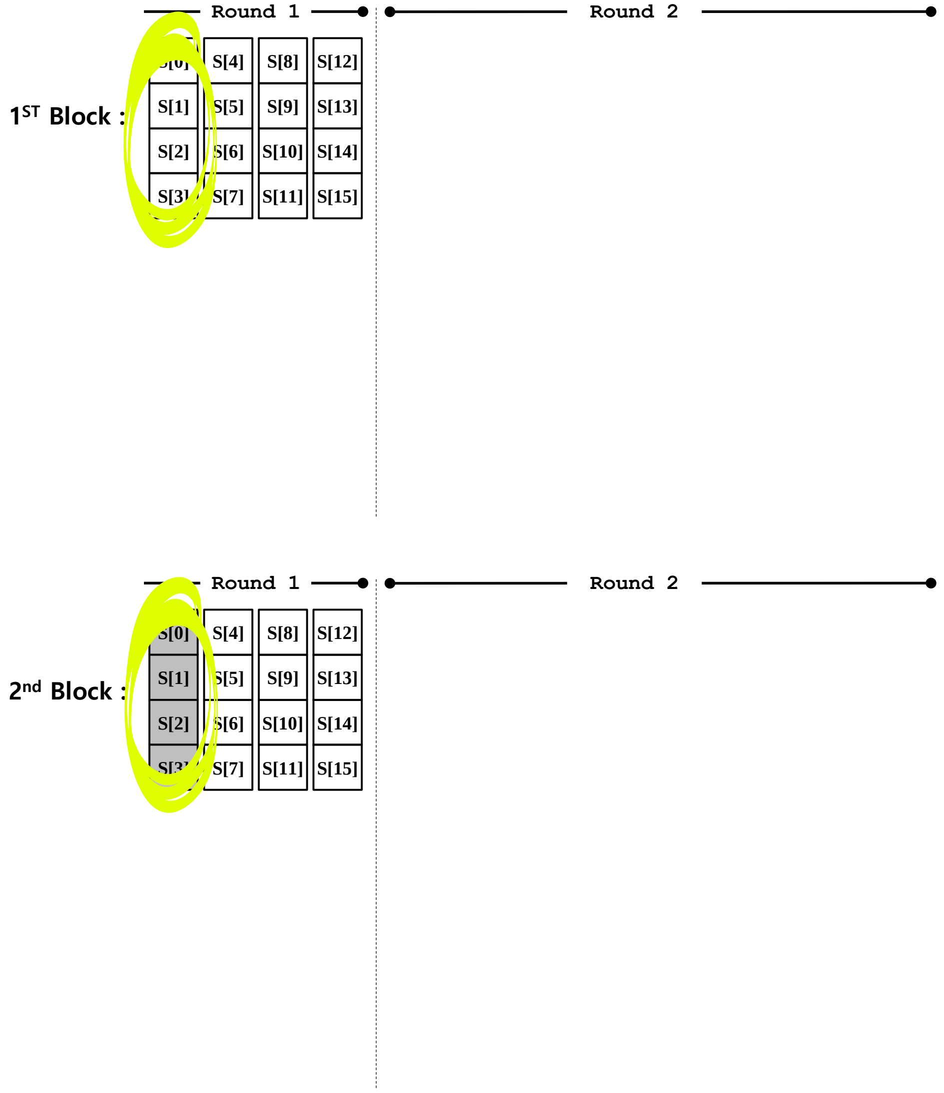


Fast AES Counter mode Encryption

$FACE_{rd2}$



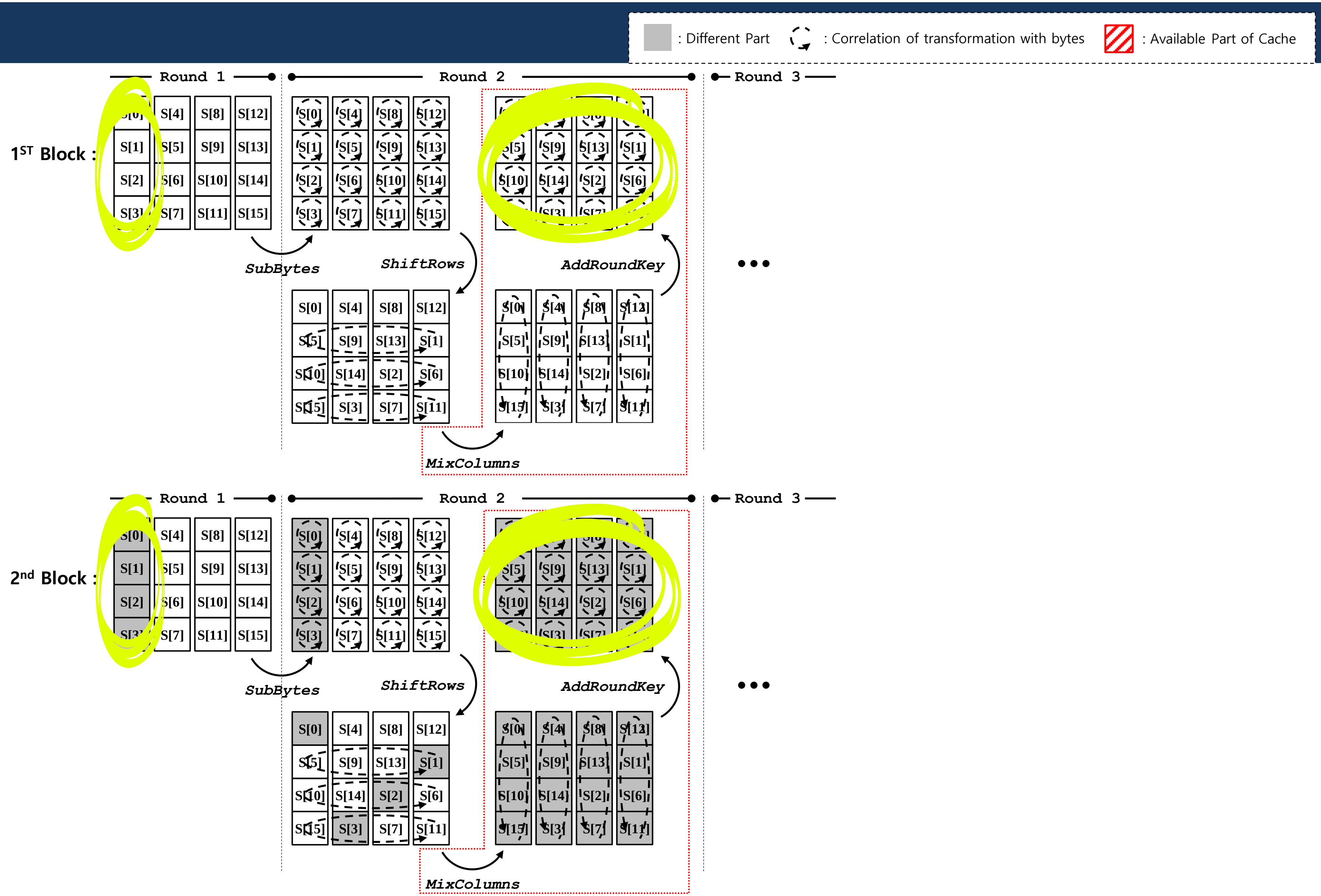
- **The difference** between two input blocks (into r2) is **the first column (4 bytes)**



Fast AES Counter mode Encryption

$FACE_{rd2}$

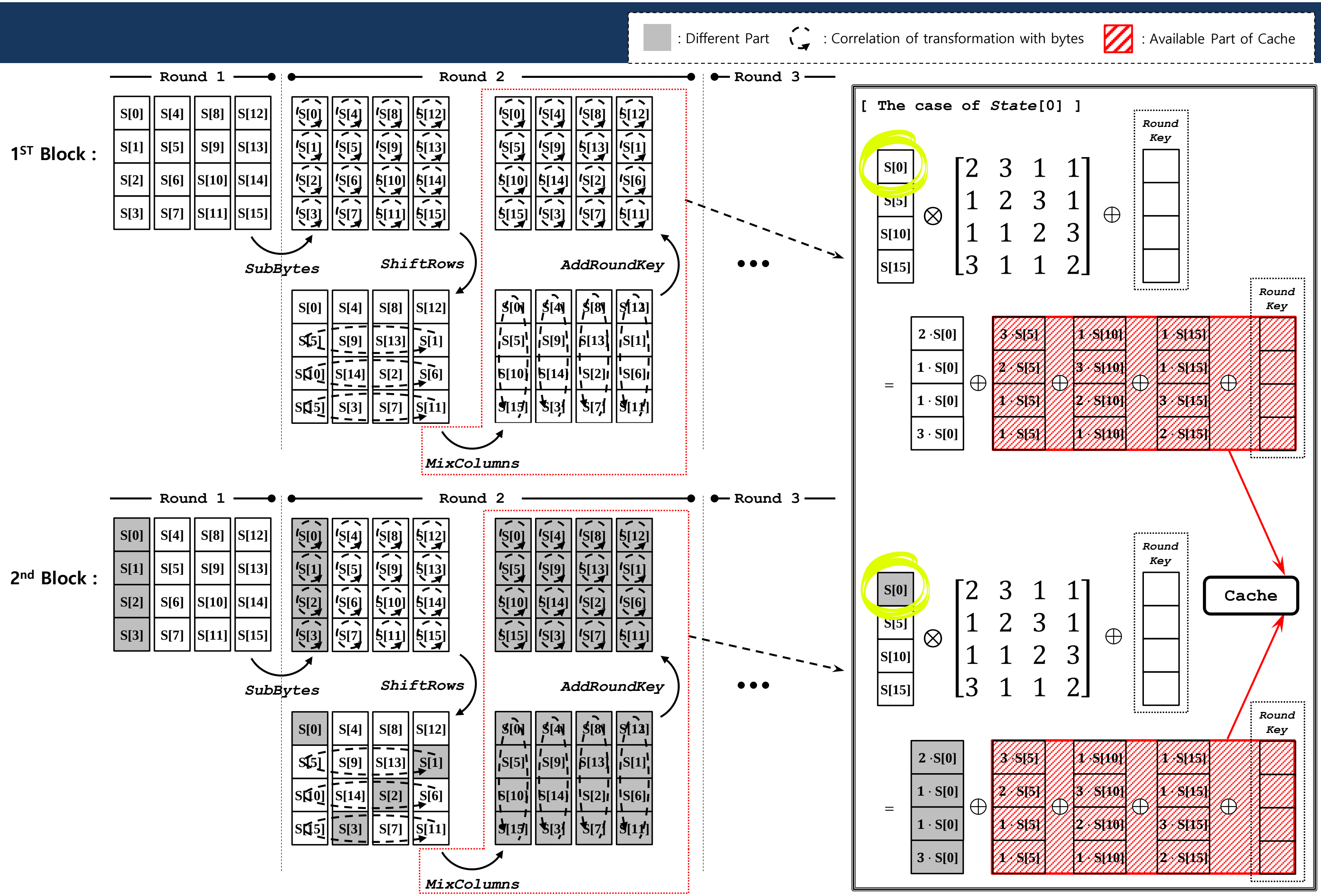
- **The difference** between two input blocks (into r2) is **the first column** (4 bytes)
- **This difference spreads** to **all States** by ShiftRows() and Mixcolumns() operation



Fast AES Counter mode Encryption

$FACE_{rd2}$

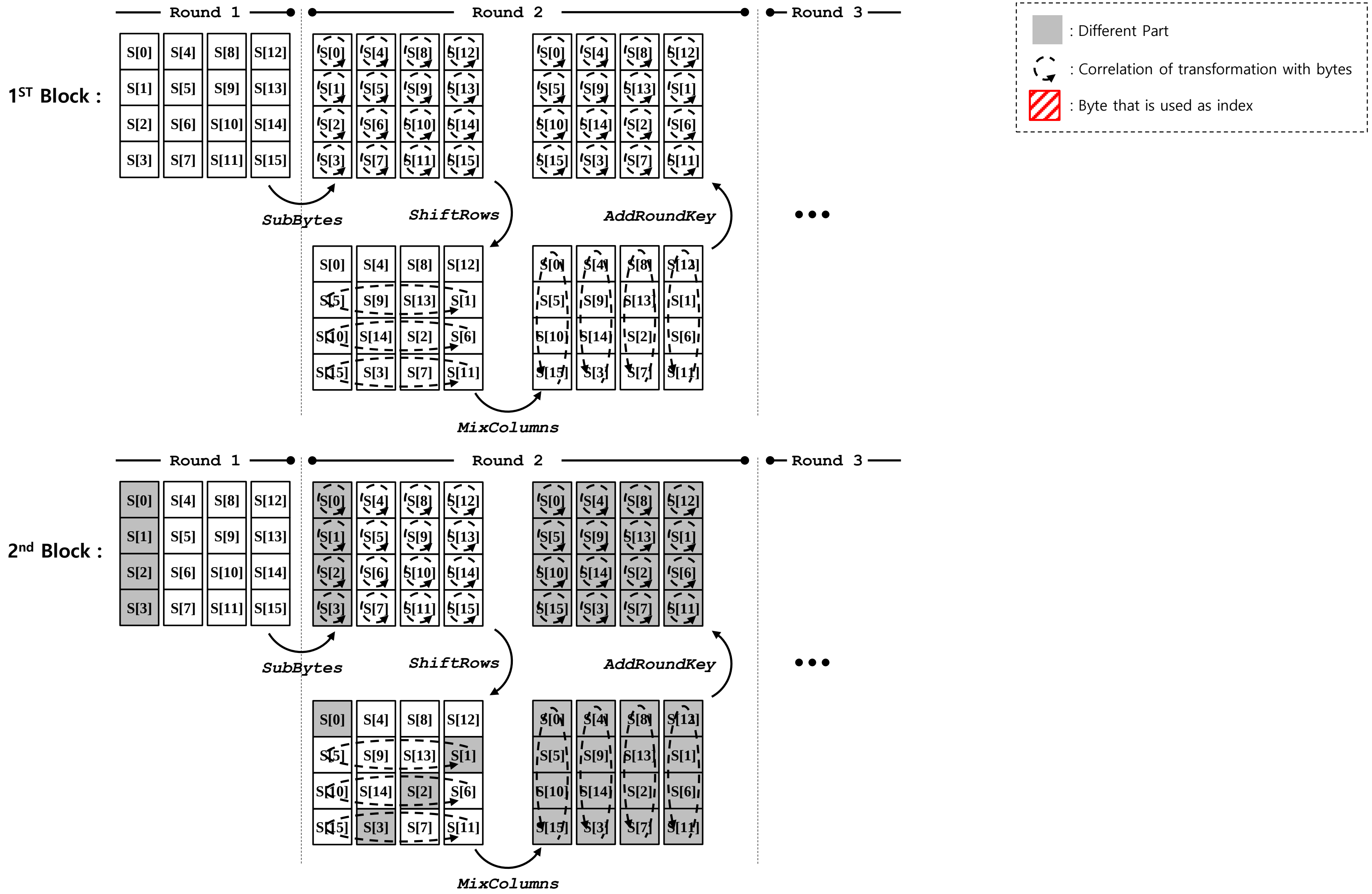
- **The difference** between two input blocks (into r2) is **the first column (4 bytes)**
- **This difference spreads** to **all States** by ShiftRows() and Mixcolumns() operation
- Cache **intermediate result** of MixColumn() operation (**16 bytes**)
- The cached value can be reused in **255** consecutive blocks



Fast AES Counter mode Encryption

$FACE_{rd2+}$

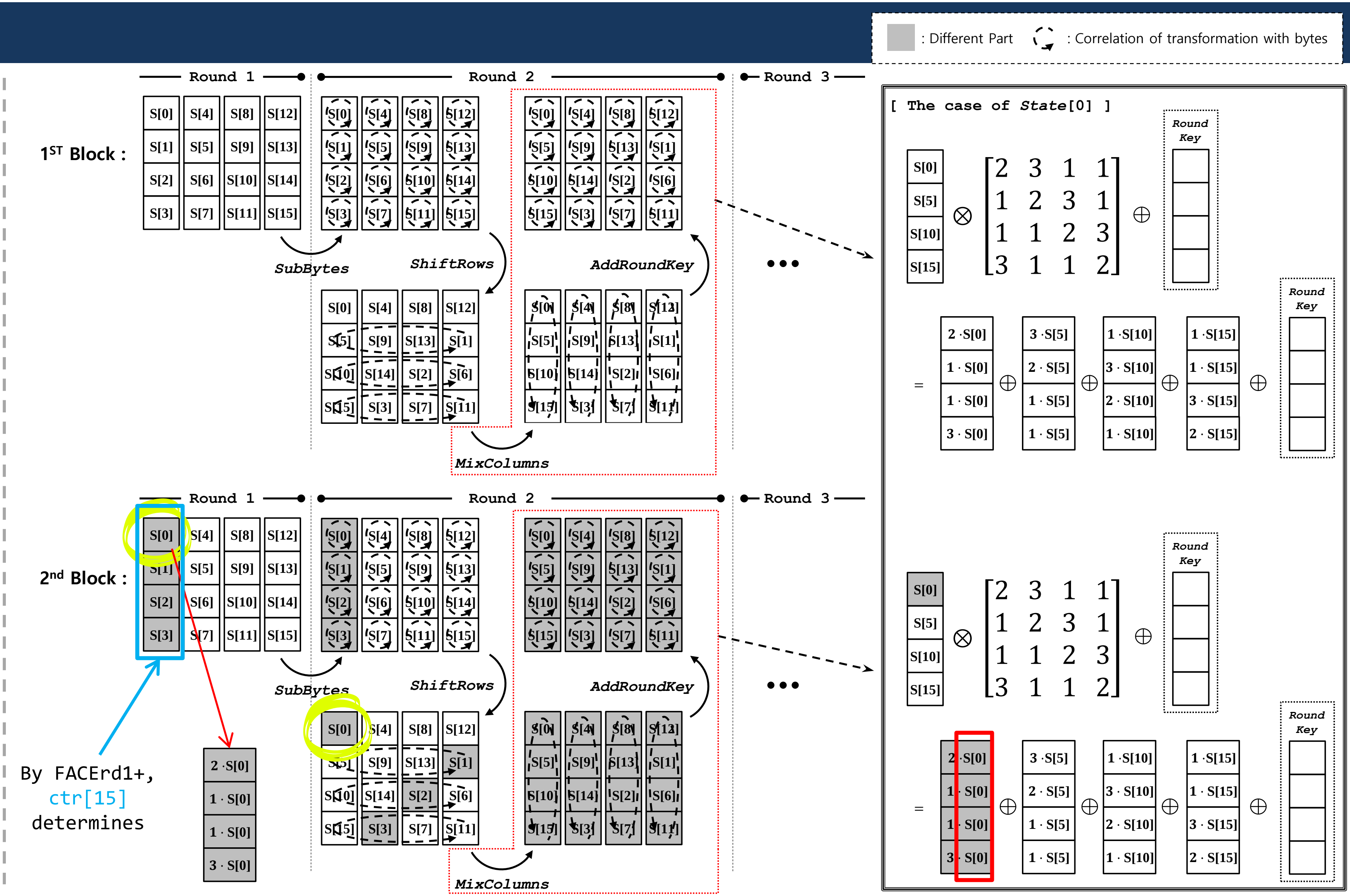
- Generate **Pre-computation lookup table** (size : 4KB)



Fast AES Counter mode Encryption

$FACE_{rd2+}$

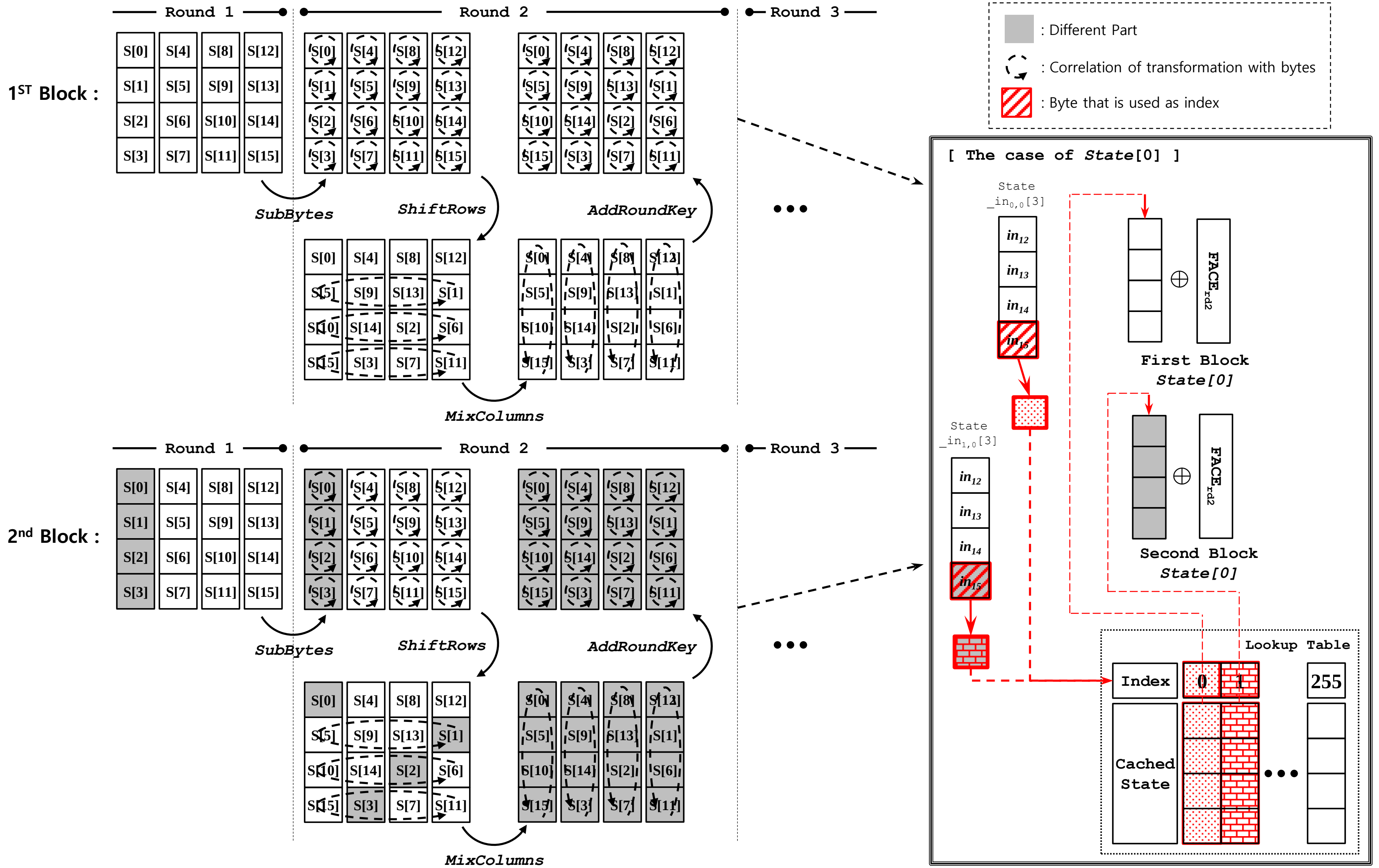
- Generate **Pre-computation lookup table** (size : 4KB)
- Store and Reuse intermediate result of MixColumns() operation



Fast AES Counter mode Encryption

$FACE_{rd2+}$

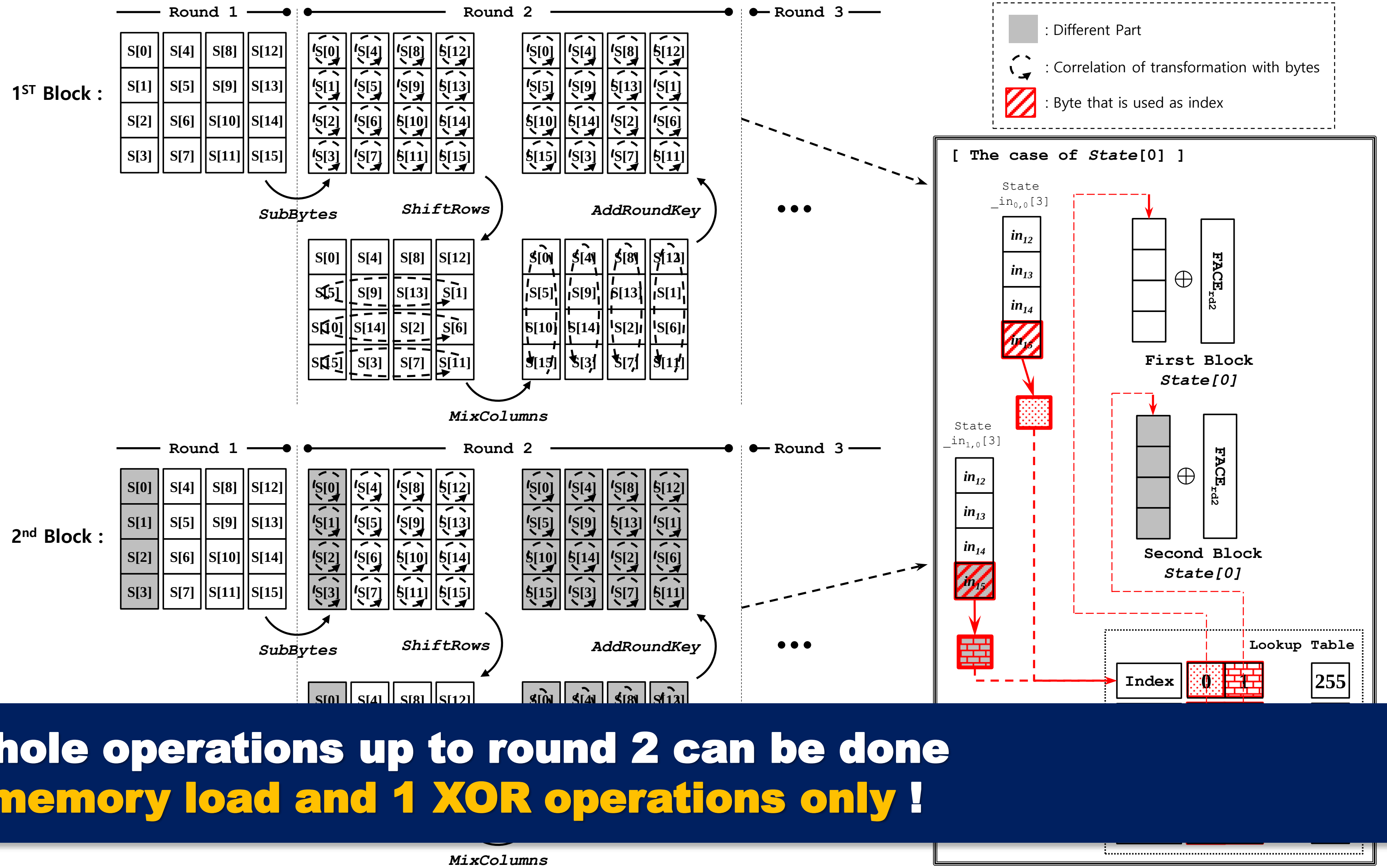
- Generate **Pre-computation lookup table** (size : 4KB)
- Store and Reuse intermediate result of MixColumns() operation
- The lookup table can be used in 2^{40} consecutive blocks
  (1,099,511,627,776 block
  = 17,592,186,044,416 bytes
  = 16 TB)
- The **lookup index** is the **last byte of the counter**



Fast AES Counter mode Encryption

$FACE_{rd2+}$

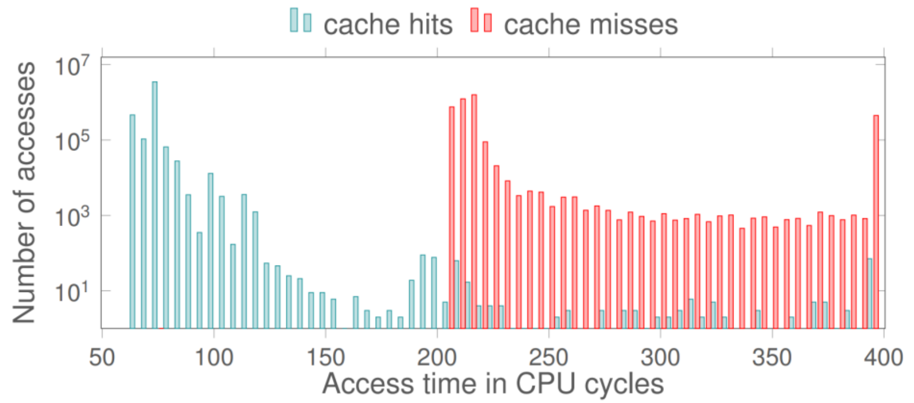
- Generate **Pre-computation lookup table** (size : 4KB)
- Store and Reuse intermediate result of MixColumns() operation
- The lookup table can be used in 2^{40} consecutive blocks
  (1,099,511,627,776 block
  = 17,592,186,044,416 bytes
  = 16 TB)



The whole operations up to round 2 can be done by 2 memory load and 1 XOR operations only !

Cache timing Attacks

** ARMageddon, USENIX 2016



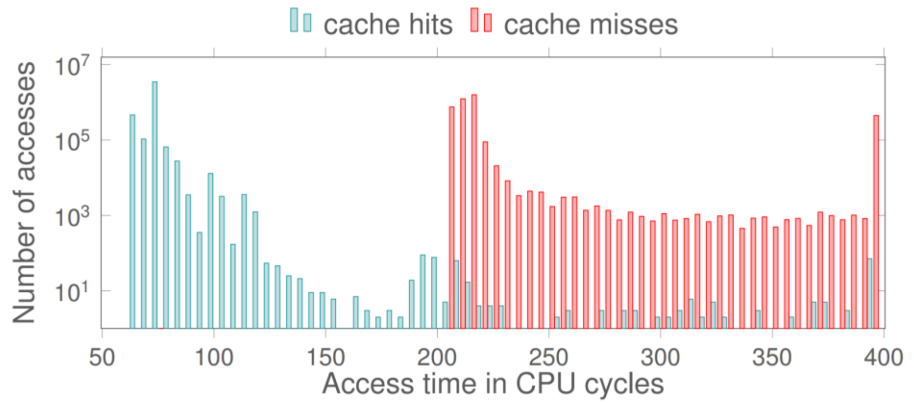
< Cache timing variation >

- **Exploits timing differences between accessing cached vs. non-cached data**
  - CacheBleed : cache-bank conflicts
    * Yuval Yarom et al. "CacheBleed: a timing attack on OpenSSL constant-time RSA", Journal of Cryptographic Engineering, June 2017.

- **Software countermeasures**
  - Constant-time implementation
    → ensures that secret information is not disclosed through the operation of the code
  - To be constant time
    - only uses fixed-time instructions with arguments that depend on secret data
    - does not use conditional branches that depend on secret data
    - does not use memory access patterns that depend on secret data

Cache timing Attacks

** ARMageddon, USENIX 2016



< Cache timing variation >

- **Exploits timing differences between accessing cached vs. non-cached data**
  - CacheBleed : cache-bank conflicts
    * Yuval Yarom et al. "CacheBleed: a timing attack on OpenSSL constant-time RSA", Journal of Cryptographic Engineering, June 2017.

- **Software countermeasures**
  - Constant-time implementation
    → ensures that secret information is not disclosed through the operation of the code
  - To be constant time
    - only uses fixed-time instructions with arguments that depend on secret data
    - does not use conditional branches that depend on secret data
    - does not use memory access patterns that depend on secret data

▣ Our method looks like vulnerable to timing attacks (the use of lookup tables)

▣ But, FACE has no operations that depend on secret data
- In case of $FACE_{rd0}$, $FACE_{rd1}$, and $FACE_{rd2}$, the size of cache is small and the indices are fixed (i.e. constant data)
- In case of $FACE_{rd1+}$ and $FACE_{rd2+}$, the index is merely a part of counter that does not need to be secret and the index increases linearly

Evaluations

❖ Implementation

- We implement FACE by modifying the AES source code contained in the open-source libraries
  - we select targets which can be considered as the fastest one
  - OpenSSL : table-based and bitsliced
    - · [BS08]* is the fastest table-based implementation. But table-based is not our main targets.
    - · Bitsliced AES is implemented based on [KS09]** (the fastest bitsliced implementation)
  - Crypto++ : AES-NI
    - · Throughput records based on eSTREAM/Crypto++ benchmark, and etc

- For a fair comparison,
  we did not re-code the existing strategy into our own implementation (the quality of code)
  → Except for our strategy, all other conditions remain the same

	Test Env_1	Test Env_2	Test Env_3
CPU	Intel Core 2 Quad Q9550	Intel Core i7 4770K	Intel Core i7 8700K
Frequency	2.8 GHz	3.5 GHz	3.7 GHz
RAM	4 GB	8 GB	16 GB
OS	Linux 3.19.0-32 x86_64	Linux 3.19.0-32 x86_64	Linux 4.13.0-36 x86_64

* [BS08] : Daniel J Bernstein and Peter Schwabe, “New AES software speed records”, INDOCRYPT 2008
** [KS09] : Emilia Käsper and Peter Schwabe, “Faster and timing-attack resistant AES-GCM”, CHES 2009

Evaluations

❖ **Experimental results (Throughput)**

Platform	Implementation Method		Target	1024 bytes 128	1024 bytes 192	1024 bytes 256	4096 bytes 128	4096 bytes 192	4096 bytes 256	20480 bytes 128	20480 bytes 192	20480 bytes 256	40960 bytes 128	40960 bytes 192	40960 bytes 256
Test Env 1	Table-based		OpenSSL	15.849	18.302	20.710	15.786	18.272	20.680	15.766	18.249	20.665	15.768	18.238	20.659
			This Paper	12.452	14.947	17.336	12.407	14.936	17.329	12.394	14.911	17.321	12.399	14.913	17.326
	Bitsliced		[KS09]	8.014	9.495	10.960	7.811(7.59)	9.251	10.686	7.763	9.195	10.624	7.764	9.192	10.618
			This Paper	6.754	8.180	9.607	**6.408(6.347)**	7.797	9.180	6.364	7.755	9.119	6.360	7.752	9.108
Test Env 2	Table-based		OpenSSL	10.562	12.309	14.036	10.553	12.348	14.067	10.529	12.276	14.023	10.528	12.276	14.023
			This Paper	8.380	10.085	11.808	8.344	10.064	11.797	8.371	10.067	11.810	8.368	10.071	11.808
	Bitsliced		[KS09]	5.687	6.745	7.803	5.530	6.554	7.573	5.514	6.491	7.511	5.500	6.482	7.495
			This Paper	4.696	5.737	6.787	4.429	5.455	6.476	4.398	5.407	6.425	4.397	5.406	6.422
	AES-NI	1 x 1	Crypto++	2.540	2.957	3.321	2.506	2.896	3.283	2.698	3.083	3.482	2.695	3.080	3.477
			This Paper (R1)	1.025	1.267	1.556	1.018	1.253	1.552	1.073	1.301	1.578	1.071	1.294	1.558
			This Paper (R2)	0.927	1.160	1.383	0.917	1.146	1.377	1.040	1.188	1.398	1.040	1.189	1.398
		4 x 1	Crypto++	0.730	0.861	0.984	0.704	0.840	0.983	0.688	0.824	0.969	0.684	0.822	0.967
			This Paper (R1)	0.634	0.781	0.923	0.623	0.769	0.920	0.621	0.765	0.911	0.620	0.765	0.910
			This Paper (R2)	0.592	0.727	0.869	0.580	0.714	0.858	0.578	0.711	0.857	0.578	0.711	0.857
Test Env 3	Table-based		OpenSSL	9.374	10.948	12.645	9.223	10.788	12.496	9.083	10.354	11.822	8.716	10.087	11.644
			This Paper	7.185	8.741	10.346	7.114	8.726	10.230	7.081	8.408	9.847	6.855	8.203	9.647
	Bitsliced		[KS09]	5.273	6.108	7.254	5.172	6.074	7.079	5.097	5.999	6.995	5.032	5.879	6.952
			This Paper	4.339	5.356	6.278	3.932	4.984	5.987	4.006	4.945	5.873	3.812	4.691	5.571
	AES-NI	1 x 1	Crypto++	1.665	1.871	2.059	1.625	1.847	2.043	1.617	1.832	2.029	1.611	1.807	2.021
			This Paper (R1)	0.778	0.867	0.986	0.739	0.827	0.959	0.737	0.822	0.956	**0.726**	0.819	0.948
			This Paper (R2)	0.703	0.786	0.880	0.662	0.775	0.867	0.659	0.732	0.874	**0.658**	0.733	0.876
		4 x 1	Crypto++	0.551	0.669	0.767	0.547	0.642	0.758	0.537	0.636	0.745	0.531	0.622	0.739
			This Paper (R1)	0.513	0.607	0.706	0.494	0.586	0.698	0.483	0.581	0.684	**0.473**	0.573	0.677
			This Paper (R2)	0.450	0.547	0.638	0.441	0.533	0.636	0.442	0.539	0.624	**0.434**	0.539	0.625

Test Env_1	Test Env_2	Test Env_3
Intel Core 2 Quad Q9550	Intel Core i7 4770K	Intel Core i7 8700K
2.8 GHz	3.5 GHz	3.7 GHz
4 GB	8 GB	16 GB
Linux 3.19.0-32 x86_64	Linux 3.19.0-32 x86_64	Linux 4.13.0-36 x86_64

Conclusion

- AES-CTR is used for numerous high throughput applications
- We propose FACE, which can improve the performance of AES CTR by using repetitive data (approximately 15-20% improved)
- FACE can be employed in any AES CTR implementation, regardless of implementation method (i.e. table-based, bitsliced, and AES-NI-based)
- And further… Verify whether caching strategy can be applied to other algorithms that have similar characteristics to the AES CTR (e.g. CAESAR finalist Deoxys)

Conclusion

- ❖ AES-CTR is used for numerous high throughput applications
- ❖ We propose FACE, which can improve the performance of AES CTR by using repetitive data (approximately 15-20% improved)
- ❖ FACE can be employed in any AES CTR implementation, regardless of implementation method (i.e. table-based, bitsliced, and AES-NI-based)
- ❖ And further… Verify whether caching strategy can be applied to other algorithms that have similar characteristics to the AES CTR (e.g. CAESAR finalist Deoxys)

Thank you for your attention!
Any Questions?