Optimizing AES-GCM on ARM Cortex-M4

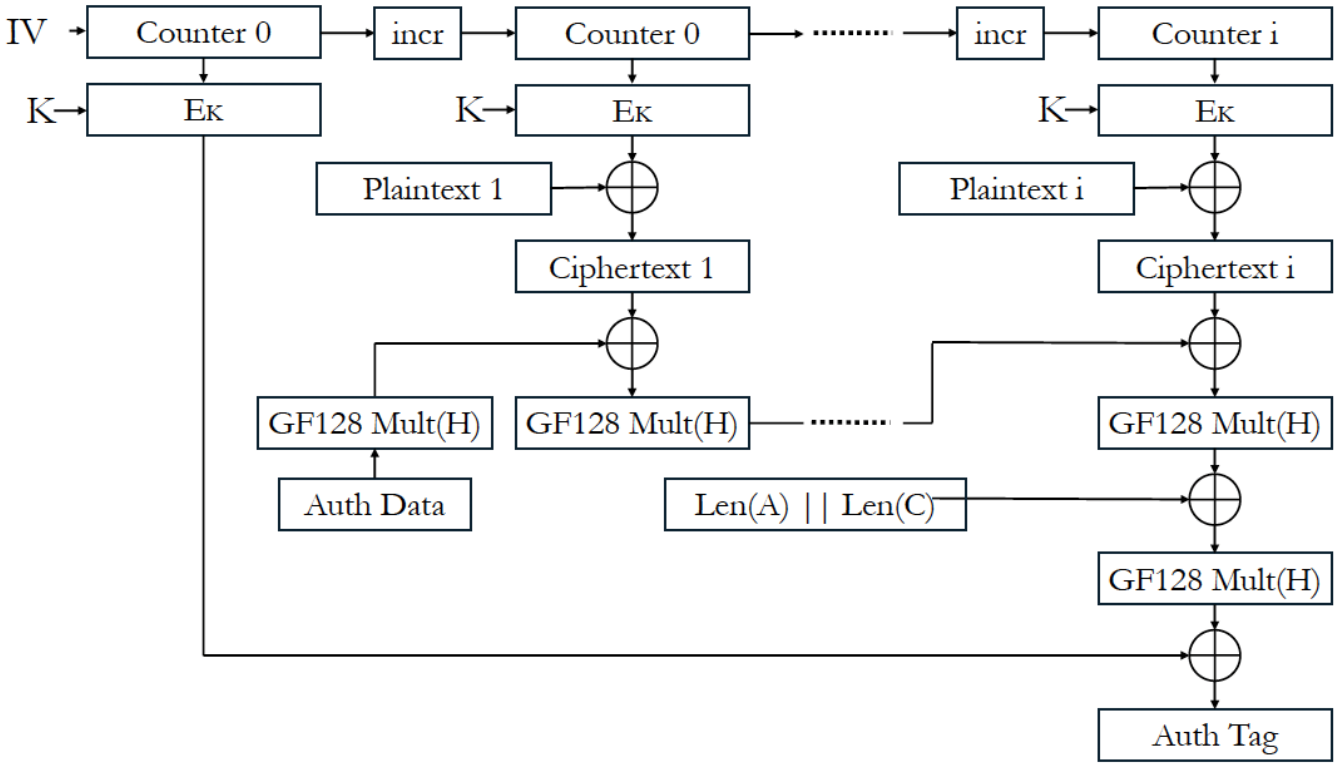
https://youtu.be/I0fRG_hE2SA

연구 목표

- 하드웨어 암호 명령어가 없는 Cortex-M4에서도 **고속 & 안전**(constant-time)한 AES-GCM 구현
- Fixslicing과 FACE 기법을 적용하여 AES-CTR 연산 최소화
- GHASH 또한 테이블 기반 vs. Karatsuba 방법 비교로 탐색

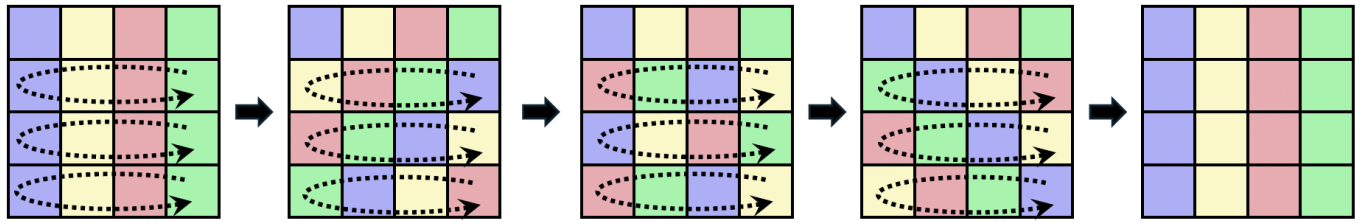
AES-GCM 개요

- **AES-CTR**
  - 블록암호 AES를 CTR(카운터) 모드로 사용해 기밀성 확보
  - 각 블록마다 카운터 값을 암호화 → Plaintext와 XOR → Ciphertext 생성
- **GHASH**
  - GF(2^128)에서의 폴리노믹 곱 연산으로 메시지 무결성·인증 태그(Tag) 생성

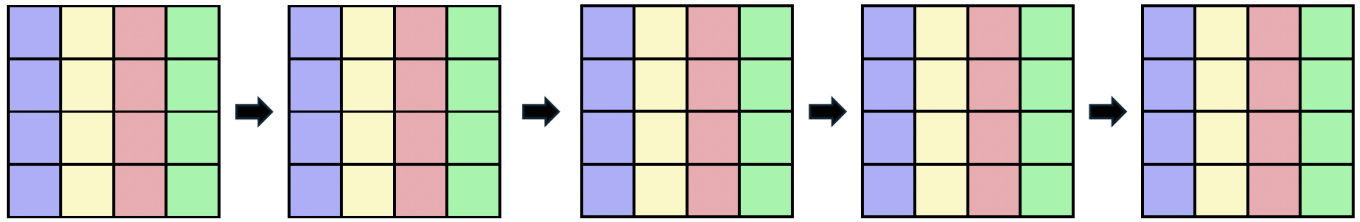


Fixslicing AES

- ARM Cortex-M4에서 **AES-128 기준 80 cycles/byte,** 상수시간 구현중 가장 빠른 성능 보고
- Fixslicing은 라운드마다 MixColumns 변형(MixColumns0, MixColumns1, MixColumns2, MixColumns3)을 적용하여, ShiftRows 효과가 나타나게 함. 비트를 옮길 필요가 없으므로 재배열 오버헤드 감소
- SubBytes + AddRoundKey를 하나로 묶어(Ark_Sub) 로직 연산 최소화



(a) Classical



(b) Fully-fixed

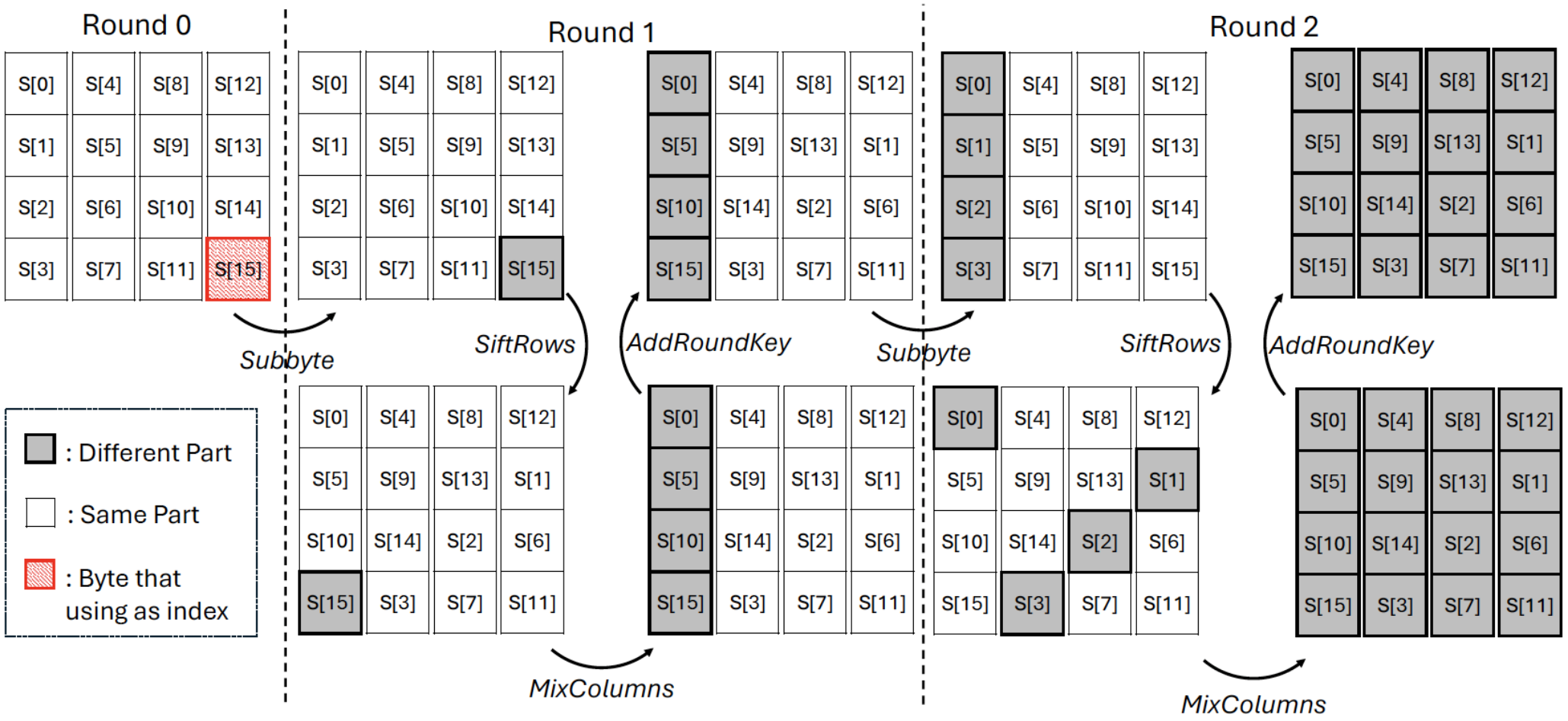
FACE(Fast AES-CTR Encryption)

- **CTR 모드에서 카운터가 1씩 증가하는 특성 활용**
  - 각 라운드에서 "변하지 않는 부분"을 캐시(재사용)함으로써 반복 연산을 줄임
- **FACE의 5가지 주요 변형**

Variant	Caching Idea	Memory Overhead	Reset Interval (Blocks)
FACE_{rd0}	Cache AddRoundKey for static bytes	12 bytes	2^8
FACE_{rd1}	Extend caching into Round 1	12 bytes	2^8
FACE_{rd1+}	256-entry lookup for Round 1	1 KB	2^{40}
FACE_{rd2}	Cache Round 2 intermediates	16 bytes	2^8
FACE_{rd2+}	Precompute Round 2 via table	4 KB	2^{40}

$$
\begin{aligned}
&3 \cdot S[5] \oplus 1 \cdot S[10] \oplus 1 \cdot S[15] \oplus roundkey_{2,0},\\
&2 \cdot S[5] \oplus 3 \cdot S[10] \oplus 1 \cdot S[15] \oplus roundkey_{2,1},\\
&1 \cdot S[5] \oplus 2 \cdot S[10] \oplus 3 \cdot S[15] \oplus roundkey_{2,2},\\
&1 \cdot S[5] \oplus 1 \cdot S[10] \oplus 2 \cdot S[15] \oplus roundkey_{2,3}.
\end{aligned}
$$

FACE(Fast AES-CTR Encryption)

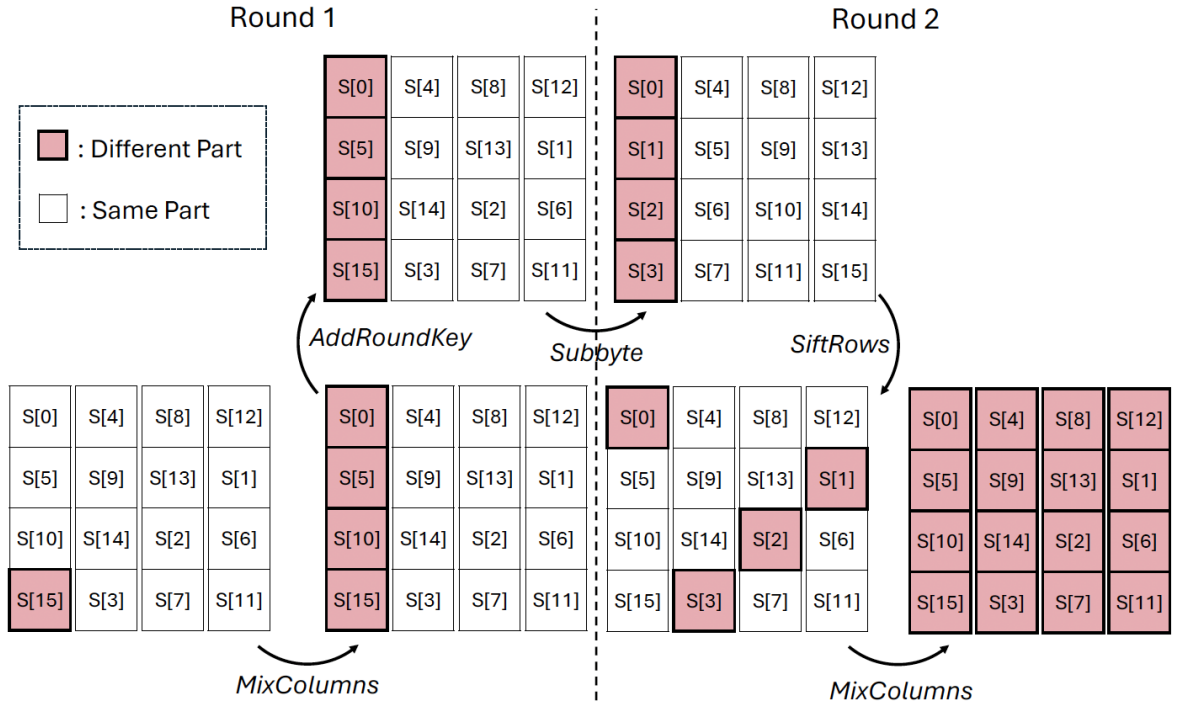


Integrating Fixslicing AES with FACE

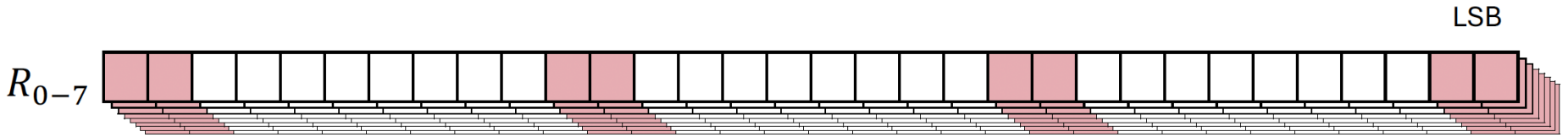
적용 포인트1. Fixslicing AES의 방식

1. Ark_Sub (AddRoundKey + SubBytes)
   - AddRoundKey + SubBytes를 하나의 루틴으로 합침
2. MixColumns(0~3)
   - 라운드마다 다른 MixColumns 변형 사용 → ShiftRows 효과를 따로 구현하지 않음
3. Packing/Unpacking
   - Bitslicing 형태로 2개 블록 이상을 병렬로 처리

Integrating Fixslicing AES with FACE



(a) Standard AES



(b) Fixslicing AES

Integrating Fixslicing AES with FACE

적용 포인트2.

- **순진한 접근 방식**:
  비트 슬라이싱 상태를 기존의 AES와 유사한 형식으로 다시 언패킹하고,
  캐싱을 적용한 다음 다시 패킹

  → 패킹과 언패킹은 사이클 오버헤드를 유발, 나머지 암호화 프로세스에 비해 상당

  → 따라서 비트 슬라이싱 상태를 보존하면서 FACE를 적용하는 전략이 필수

rd0

Fixslicing AES의 초기 과정

Packing → AddRoundKey → SubBytes

접근 방법

- (i) 패킹을 수행하고 패킹된 라운드 키와 XOR을 수행한 다음 결과 상태를 캐싱하여 Fixslicing 시퀀스를 유지

  (ii) 패킹되지 않은 라운드 키와 XOR을 수행한 다음 캐싱

Fixslicing AES의 방식과 충돌

Ark_Sub (AddRoundKey + SubBytes)

- AddRoundKey + SubBytes를 하나의 루틴으로 합침

rd1

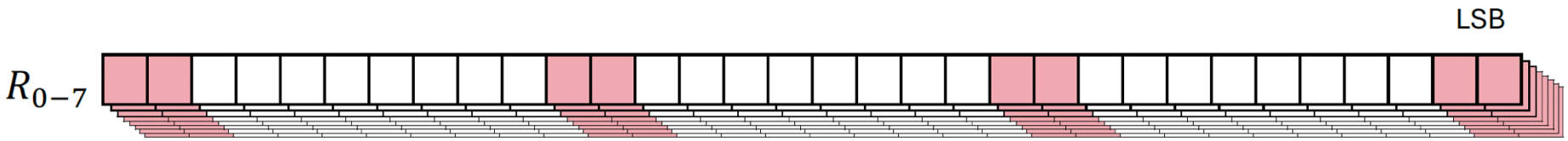
- **일반적인 바이트 지향 AES 구현**
  MixColumns는 각 열을 독립적으로 처리하므로 특정 열을 단독으로 처리가능

- **Bitslicing 또는 Fixslicing 구현**
  상태는 비트 수준에서 재정렬되어 한 열에 속하는 비트가 여러 레지스터에 분산되어
  해당 열에 대한 별도처리 비효율

  단순히 Fixslicing AES의 MixColumns0 단계를 그대로 사용하여 병렬 작업을 수행함
  MixColumns0 결과에 마스크를 적용하여 관련 비트만 남김.

rd1

- 라운드 2의 MixColumns 이전까지 추가 위치 변환을 거치지 않음
  따라서 라운드 2 Ark_Sub 단계까지 상태를 캐싱



(b) Fixslicing AES

$$\text{in} = \{S_0[0], S_0[1], \ldots, S_0[15],\ S_1[0], S_1[1], \ldots, S_1[15]\}$$

$$\xrightarrow{\text{packing}}\ \xrightarrow{\text{Ark\_Sub}}\ \xrightarrow{\text{MixColumns0}}\ \xrightarrow{\text{Ark\_Sub}}\ \xrightarrow{\wedge 0xFCF3CF3F}$$

$$\text{FACE}_{\text{rd1}} = \{b_0, b_1, b_2, b_3, b_4, b_5, 0, 0,\ b_8, b_9, b_{10}, b_{11}, 0, 0, b_{14}, b_{15},$$
$$b_{16}, b_{17}, 0, 0,\ b_{20}, b_{21}, b_{22}, b_{23}, 0, 0,\ b_{26}, b_{27}, b_{28}, b_{29}, b_{30}, b_{31}\}$$

rd1+

- **간단한 접근 방식**
  FACErd1과 동일하게 변경되는 열만 별도로 처리

$$\text{in} = \{S_i[0], S_i[1], \ldots, S_i[15],\ S_{i+1}[0], S_{i+1}[1], \ldots, S_{i+1}[15]\}$$

$$\xrightarrow{\text{packing}} \xrightarrow{\text{Ark\_Sub}} \xrightarrow{\text{MixColumns0}} \xrightarrow{\text{Ark\_Sub}} \xrightarrow{\wedge 0x030c30c0}$$

$$\text{FACE}_{\text{rd1+}} = \{0, 0, 0, 0, 0, 0, b_6, b_7,\ 0, 0, 0, 0, b_{12}, b_{13}, 0, 0,$$
$$0, 0, b_{18}, b_{19}, 0, 0, 0, 0,\ b_{24}, b_{25}, 0, 0, 0, 0, 0, 0\}$$

- 4,096바이트 테이블을 저장 해야함
- FACErd1+에서 필요했던 1,024바이트보다 4배 더 큼

rd1+ *: Parallel-Processing Method*

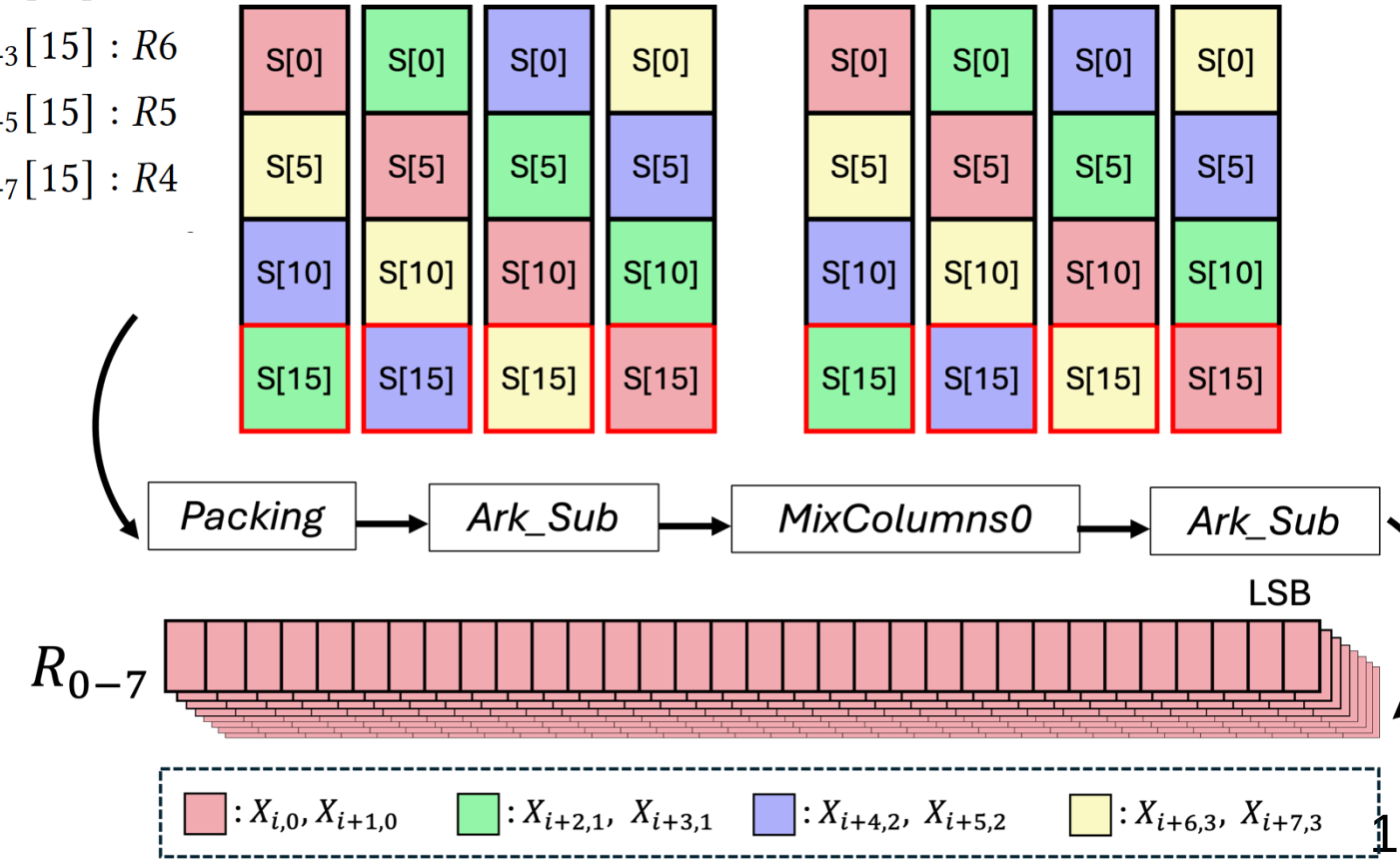
- '빈' 공간을 채워 한 번에 8개의 열을 처리함으로써 최적화
- 4배 더 많은 열을 병렬로 처리하기 때문에 전체 연산 수와 메모리 사용량을 크게 줄임

$S_0[0], S_0[5], S_0[10], S_i[15] : R3$ $S_0[0], S_0[5], S_0[10], S_{i+1}[15] : R7$

$S_0[0], S_0[5], S_0[10], S_{i+2}[15] : R2$ $S_0[0], S_0[5], S_0[10], S_{i+3}[15] : R6$

$S_0[0], S_0[5], S_0[10], S_{i+4}[15] : R1$ $S_0[0], S_0[5], S_0[10], S_{i+5}[15] : R5$

$S_0[0], S_0[5], S_0[10], S_{i+6}[15] : R0$ $S_0[0], S_0[5], S_0[10], S_{i+7}[15] : R4$



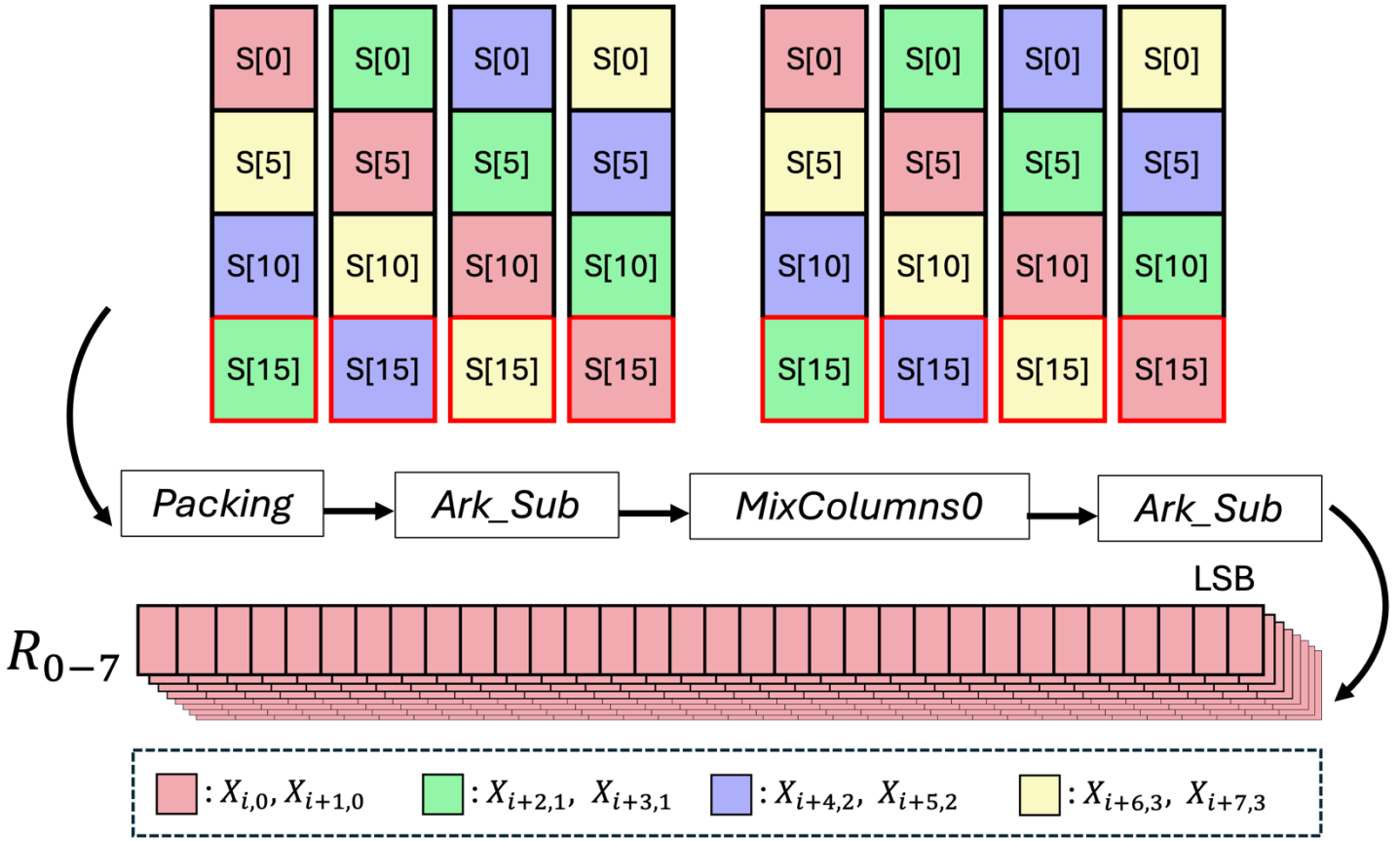
rd1+ : 상태 재배치, 라운드키 수정

```
if (block < 2) // block 1,2
    {out = x & 0x030C30C0;}
else if (block < 4) // block 3,4
    {out = (ROR32(x, 2) & 0x030C3000) ^ ROR32(x & 3, 26); }
else if (block < 6) // block 5,6
    {out = (ROR32(x, 4) & 0x030C0000) ^ (ROR32(x, 28) & 0x000030C0);}
else if (block < 8) // block 7,8
    {t = ROR32(x, 30); out = (t & 0x000C30C0) ^ ROR32(t & 3, 8);}
```

```
for(int idx = 0; idx < 16; idx++){
    temp = (original_rk[idx] & 0x030c30c0);
    rk[idx] ^= temp;
    rk[idx] ^= ((temp & 0x3000000) << 6) |((temp & 0xC30C0) >> 2);
    rk[idx] ^= ((temp & 0x3000000) << 4) | ((temp & 0xC0000) << 4)
                | ((temp & 0x3000) >> 4) | ((temp & 0xC0) >> 4);
    rk[idx] ^= ((temp & 0x3000000) << 2) | ((temp & 0xC0000) << 2)
                | ((temp & 0x3000) << 2) | ((temp & 0xC0) >> 6);
}
```

rd1+ : Table-Free Approach (FACErd1).

- 병렬 처리 버전을 " on the fly " 로 적용하여 8개 블록을 처리
- FACErd1과 결합하여 두 번째 라운드의 Ark_Sub까지 계산을 처리
- 큰 LUT의 비용을 피함.



rd2, rd2+

- rd1에서 캐싱한 결과에 MixColumns1
- Rd1+에서 캐싱한 결과에 MixColumns1
- Fixslicing에서 Ark_Sub 구조를 유지하기 위해 MixColumns1까지만

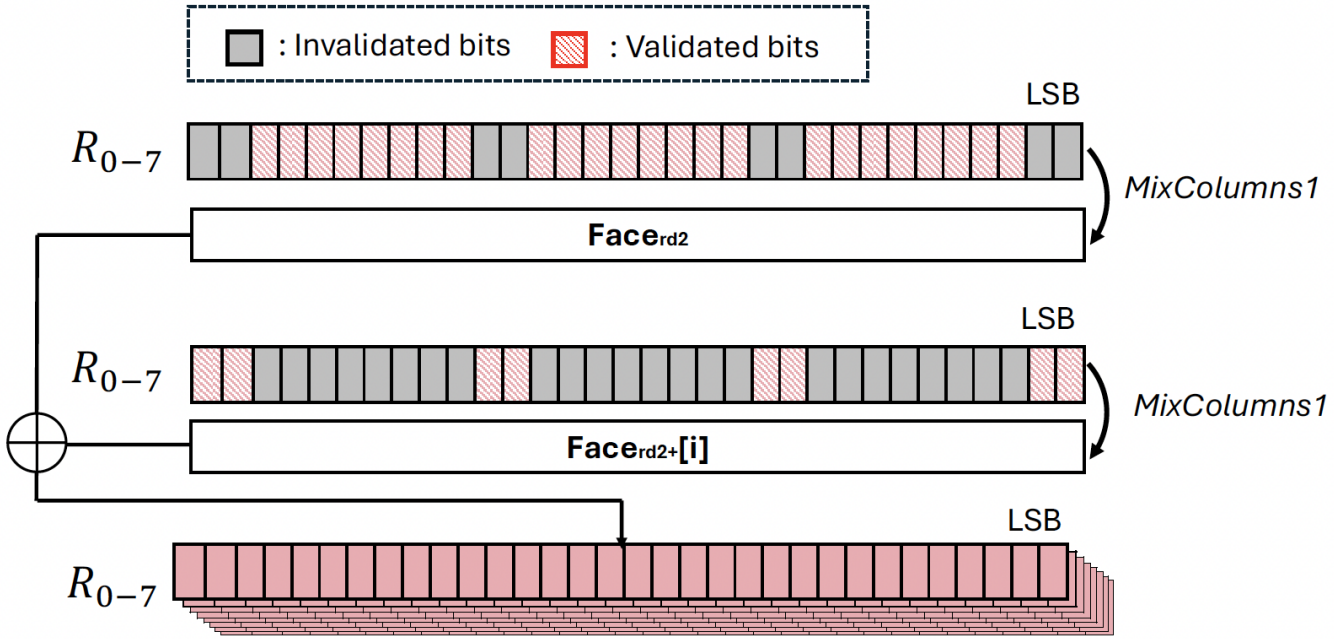


Fig. 10. Schematic of FACE$_{rd2}$ and FACE$_{rd2+}$ on Fixslicing AES.

GHASH 최적화

1.4비트 테이블 기반 방식

1. GF(2^128) 곱셈을 4비트 단위로 쪼개고, 미리 계산된 테이블(256바이트) 이용
2. 빠른 속도(실험에서 Karatsuba 대비 약 2배 빠름)
3. 하지만 완전한 constant-time 보장은 어려움(캐시 접근, 메모리 타이밍 문제)

2.Karatsuba 기반 곱셈

1. BearSSL 참고, 고정된 시퀀스로 다항식 곱셈 분할 → 캐리 없는 곱셈(⊗) 조합
2. 테이블을 거의 사용하지 않고, 데이터 종속 분기 없음 → constant-time 구현

성능 평가

실험 환경

- **하드웨어**: STM32F407G-DISC1 (ARM Cortex-M4 @ 168MHz)
- **구현 언어**: C, 어셈블리(중요 루틴 직접 최적화)
- **테스트**
  - 메시지 크기: 1KB ~ 40KB
  - 키 길이: AES-128 / AES-256
  - 측정 방식: DWT(Data Watchpoint and Trace) 카운터를 통한 Cycle 측정(평균 100회)

GCTR 비교

- AES-128: FACE_rd2+ 적용 시 최대 **약 19%** cycle 감소
- AES-256: FACE_rd2+ 적용 시 최대 **약 14%** cycle 감소
- 메시지가 클수록(precomputation이 상쇄되어) 효과 커짐

FACE Variant	1,KB		4,KB		20,KB		40,KB	
	128	256	128	256	128	256	128	256
Basic	117,456 (0%)	158,453 (0%)	469,488 (0%)	633,461 (0%)	2,345,712 (0%)	3,166,837 (0%)	4,691,312 (0%)	6,333,557 (0%)
$FACE_{rd1}$	118,666 (-1.03%)	164,420 (-3.77%)	431,971 (+8.00%)	599,738 (+5.32%)	2,102,571 (+10.36%)	2,920,616 (+7.78%)	4,190,821 (+10.67%)	5,821,796 (+8.08%)
$FACE_{rd1+}$	118,259 (-0.68%)	164,366 (-3.73%)	411,574 (+12.34%)	580,511 (+8.36%)	1,980,866 (+15.55%)	2,799,003 (+11.62%)	3,941,041 (+16.00%)	5,572,118 (+12.03%)
$FACE_{rd2+}$	119,107 (-1.41%)	164,763 (-3.98%)	400,915 (+14.60%)	568,391 (+10.28%)	1,903,287 (+18.87%)	2,718,579 (+14.15%)	3,781,252 (+19.41%)	5,406,764 (+14.63%)

GHASH 비교

- 4비트 테이블: Karatsuba 대비 약 2배 빠름
- Karatsuba: constant-time, 보안성↑ (캐시 기반 공격 취약성↓)

GHASH	1 KB	4 KB	20 KB	40 KB
Karatsuba	158,749 (0%)	611,293 (0%)	3,024,861 (0%)	6,041,821 (0%)
Table-based	79,402 (+50.0%)	302,390 (+50.5%)	1,491,670 (+50.7%)	2,978,281 (+50.7%)

전체 GCM 비교

- GHASH 부분 비중이 커서, FACE로 AES-CTR을 빨라져도 전체 향상폭은 5~10% 내외
- 큰 메시지에서는 FACE_rd2+와 4비트 테이블 GHASH 조합 시 최대 10~13% 개선

GHASH Technique	FACE Variant	Input Size (bytes)							
		1 KB		4 KB		20 KB		40 KB	
		128	256	128	256	128	256	128	256
Table-based	Basic	206,377 (0%)	250,657 (0%)	780,148 (0%)	946,962 (0%)	3,929,031 (0%)	4,659,716 (0%)	7,664,348 (0%)	9,301,295 (0%)
	$FACE_{rd1}$	207,123 -0.36%	257,089 -2.57%	740,507 (+5.07%)	915,445 (+3.32%)	3,592,621 (+8.56%)	4,425,658 (+5.02%)	7,156,129 (+6.62%)	8,813,752 (+5.23%)
	$FACE_{rd1+}$	206,894 -0.25%	256,506 -2.33%	721,076 (+7.56%)	893,638 (+5.63%)	3,470,230 (+11.69%)	4,291,684 (+7.88%)	6,904,704 (+9.90%)	8,539,152 (+8.20%)
	$FACE_{rd2+}$	207,315 -0.45%	256,511 -2.33%	708,758 (+9.16%)	880,339 (+7.03%)	3,382,451 (+13.92%)	4,206,896 (+9.72%)	6,724,602 (+12.26%)	8,365,109 (+10.05%)
Karatsuba	Basic	286,425 (0%)	330,735 (0%)	1,091,001 (0%)	1,257,903 (0%)	5,381,427 (0%)	6,201,513 (0%)	10,744,627 (0%)	12,381,353 (0%)
	$FACE_{rd1}$	287,956 -0.53%	337,339 -1.99%	1,054,620 (+3.33%)	1,226,429 (+2.50%)	5,143,006 (+4.42%)	5,967,503 (+3.77%)	10,253,496 (+4.57%)	11,893,853 (+3.94%)
	$FACE_{rd1+}$	287,320 -0.31%	336,580 -1.76%	1,033,309 (+5.28%)	1,204,553 (+4.24%)	5,011,815 (+6.87%)	5,833,415 (+5.93%)	9,984,955 (+7.06%)	11,619,500 (+6.16%)
	$FACE_{rd2+}$	288,077 -0.58%	337,182 -1.95%	1,022,429 (+6.28%)	1,193,035 (+5.16%)	4,938,363 (+8.24%)	5,756,741 (+7.16%)	9,833,288 (+8.49%)	11,461,381 (+7.45%)

Q & A