Fast AES implementation using ARMv8 ASIMD without Cryptography Extension

IT융합공학부 권혁동

HSU 한성대학교 HANSUNG UNIVERSITY

CryptoCraft LAB
https://crypto.modoo.at

Contents

CryptoCraft LAB

AES

- Advanced Encryption Standard
- 128, 192, 256 세 규격을 지님
  - **본 논문에서는 128비트를 대상으로 함**
- Rijndael 알고리즘, FIPS 197
- SubBytes, ShiftRows, MixColumns, AddRoundKey

타깃 프로세서

- ARMv8-A
  - Cortex-A53, Cortex-A57, Cortex-A72
- RISC(Reduced Instruction Set Computer) 기반
- 64-bit
- **ASIMD**(Advanced Single Instruction Multiple Data)
  - SIMD: 병렬 프로세서, 1개 명령어로 다수 값을 동시에 계산
  - ASIMD: **64/128비트 용 복합 SIMD, NEON으로도 불림**

타깃 프로세서

- ARMv8-A
- 31개 범용 레지스터 (0x ~ 0x30)
- 스택 포인터(sp), 제로 레지스터(zr)
  - 모든 레지스터와 스택 포인터, 제로 레지스터는 64비트
- ASIMD(NEON) 사용 가능
  - 스칼라/벡터 명령어, 레지스터 제공
  - 32개의 레지스터
  - 128비트

타깃 프로세서

계열	버전	코어	기능 및 특징
Cortex	ARMv7-A	Cortex-A8	VFP, NEON, 슈퍼스칼라 파이프라인, MMU+TrustZone
		Cortex-A9	VFP, NEON, DBX, MMU+TrustZone
		Cortex-A9 MPCore	Cortex-A9 + 1-4 코어 SMP
		Cortex-A12	
		Cortex-A15	
	ARMv7-R	Cortex-R4(F)	임베디드, FPU, 가변적 캐시, 선택적 MPU
	ARMv6-M	Cortex-M1	FPGA 연동, 마이크로컨트롤러
	ARMv7-M	Cortex-M3	마이크로컨트롤러
	ARMv7E-M	Cortex-M4	마이크로컨트롤러
	ARMv8-A	**Cortex-A53**	64비트 명령어 지원, MMU, TrustZone, 64비트 가상 주소
		Cortex-A57	64비트 명령어 지원, MMU, TrustZone, 64비트 가상 주소
		Cortex-A72	64비트 명령어 지원, MMU, TrustZone, 64비트 가상 주소

타깃 프로세서

- Table Lookup(tbl), Table Lookup Extended(tbx)
  - 인덱스 벡터를 기반으로 새로운 벡터를 작성
  - 최대 4개의 레지스터의 테이블 조회
- Vector Extract(ext)
  - 소스 벡터 쌍에서 바이트를 추출
  - 128비트 벡터의 rotation 연산에도 사용 가능
- Interleaved Load(ldn): n은 정수
  - ASIMD 파이프라인을 통해 n개의 인자를 레지스터로 로드

타깃 프로세서

- 라즈베리 파이 3
  - ARMv8-A가 적용된 기종
  - 3A, 3B, 3B+
- 본 논문에서는 3B를 사용
  - 쿼드코어 Cortex-A53
    - Cryptography extension 제외
  - 1.2GHz
  - aarch64-linux-gnu-gcc ver 8.3

구현: SubBytes & ShiftRows

```
sub v7.16b, v1.16b, v15.16b
tbl v1.16b, { v16.16b - v19.16b }, v1.16b
sub v6.16b, v7.16b, v15.16b
tbx v1.16b, { v20.16b - v23.16b }, v7.16b
sub v5.16b, v6.16b, v15.16b
tbx v1.16b, { v24.16b - v27.16b }, v6.16b
tbx v1.16b, { v28.16b - v31.16b }, v5.16b
```

- SubBytes & ShiftRows
  - Input block의 16바이트를 tbl명령어로 로드
  - 4개의 128비트 레지스터에 저장된 첫 1/4 AES S-Box LUT도 사용

- **0x00 ~ 0x3F 16바이트 입력에 대한 substitues 연산에 효과적**
  - 범위 초과시 0x00으로 대체

구현: SubBytes & ShiftRows

```
sub v7.16b, v1.16b, v15.16b
tbl v1.16b, { v16.16b - v19.16b }, v1.16b
sub v6.16b, v7.16b, v15.16b
tbx v1.16b, { v20.16b - v23.16b }, v7.16b
sub v5.16b, v6.16b, v15.16b
tbx v1.16b, { v24.16b - v27.16b }, v6.16b
tbx v1.16b, { v28.16b - v31.16b }, v5.16b
```

- 첫 1/4 이후 2/4
  - 모든 input byte에서 0x40(64)을 뺀다
  - tbx 명령어를 통해 인덱스 조회
  - 이를 통해 **0x40 ~ 0x7F 범위 내로 치환**
    - 범위 초과시 영향 없음
    - **tbx 명령어는 index 초과 범위를 수정 또는 삭제하지 않음**

구현: SubBytes & ShiftRows

```
sub v7.16b, v1.16b, v15.16b
tbl v1.16b, { v16.16b - v19.16b }, v1.16b
sub v6.16b, v7.16b, v15.16b
tbx v1.16b, { v20.16b - v23.16b }, v7.16b
sub v5.16b, v6.16b, v15.16b
tbx v1.16b, { v24.16b - v27.16b }, v6.16b
tbx v1.16b, { v28.16b - v31.16b }, v5.16b
```

- Substitution(SubByte) 과정을 완료하기 위해
  - Sub(뺄셈), Sub(치환) 연산이 2회 추가로 필요
  - **3/4, 4/4 부분의 S-Box까지 진행**이 되어야 하기 때문

구현: MixColumns

- MixColumns를 32-bit에서 연산: 각각의 AES 상태의 j열에 대해
  - 2회의 rotation
  - 1회의 x 곱셈
    - x는 GF(2^8)상의 정수 2 또는 3
  - 3회의 XOR
  - 바이트 연산을 고려할 시, 16회의 XOR, 8회의 곱셈이 필요

구현: MixColumns

$$A' = \begin{bmatrix} a'_{0,j} \\ a'_{1,j} \\ a'_{2,j} \\ a'_{3,j} \end{bmatrix} = \begin{bmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{bmatrix} \begin{bmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{bmatrix} = \begin{bmatrix} a_{0,j}\ a_{3,j}\ a_{2,j}\ a_{1,j} \\ a_{1,j}\ a_{0,j}\ a_{3,j}\ a_{2,j} \\ a_{2,j}\ a_{1,j}\ a_{0,j}\ a_{3,j} \\ a_{3,j}\ a_{2,j}\ a_{1,j}\ a_{0,j} \end{bmatrix} \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} =$$

$$= 2 \begin{bmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{bmatrix} + \begin{bmatrix} a_{3,j} \\ a_{0,j} \\ a_{1,j} \\ a_{2,j} \end{bmatrix} + \begin{bmatrix} a_{2,j} \\ a_{3,j} \\ a_{0,j} \\ a_{1,j} \end{bmatrix} + 3 \begin{bmatrix} a_{1,j} \\ a_{2,j} \\ a_{3,j} \\ a_{0,j} \end{bmatrix}.$$

- 32-bit MixColumns
  - 우측의 공식을 도출 가능

$$A' = (2A + \mathtt{RotLeft}^2(A)) + \mathtt{RotLeft}((2A + \mathtt{RotLeft}^2(A)) + A),$$

$$\text{where } A = \begin{bmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{bmatrix} \text{ and } \mathtt{RotLeft}(A) = \begin{bmatrix} a_{1,j} \\ a_{2,j} \\ a_{3,j} \\ a_{0,j} \end{bmatrix}.$$

구현: MixColumns

$$A = (a_0, a_1, a_2, a_3)$$

A를 기준으로

$$\texttt{RotLeft}(A) = (a_1, a_2, a_3, a_0) \quad \leftarrow$$

$$\texttt{Rev32}(A) = (a_2, a_3, a_0, a_1) \quad \leftarrow \leftarrow$$

$$\texttt{RotRight}(A) = (a_3, a_0, a_1, a_2) \quad \rightarrow$$

- AES 상태 워드 A(32bits)를 기준으로 표현식 정의
  - $\text{RotLeft}^2(X) = \text{Rev32}(X)$
  - $\text{RotRight}(X) = \text{Rev32}(\text{RotLeft}(X))$
- **Left -> Rev -> Right** 순서로 연산

구현: MixColumns

- 본 논문의 타깃 프로세서는 **ARMv8-A Cortex-A53**
- ASIMD 연산을 활용한 **128-bit MixColumns 구현** 시도
  - ASIMD 레지스터는 128-bit
- **새로운 환경에 적합한 구현**

구현: MixColumns

$$A' = \begin{bmatrix} a'_{0,j} \\ a'_{1,j} \\ a'_{2,j} \\ a'_{3,j} \end{bmatrix} = \begin{bmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{bmatrix} \begin{bmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{bmatrix} = \begin{bmatrix} a_{0,j} & a_{3,j} & a_{2,j} & a_{1,j} \\ a_{1,j} & a_{0,j} & a_{3,j} & a_{2,j} \\ a_{2,j} & a_{1,j} & a_{0,j} & a_{3,j} \\ a_{3,j} & a_{2,j} & a_{1,j} & a_{0,j} \end{bmatrix} \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} =$$

$$= 2 \begin{bmatrix} a_{0,j} \\ a_{1,j} \\ a_{2,j} \\ a_{3,j} \end{bmatrix} + \begin{bmatrix} a_{3,j} \\ a_{0,j} \\ a_{1,j} \\ a_{2,j} \end{bmatrix} + \begin{bmatrix} a_{2,j} \\ a_{3,j} \\ a_{0,j} \\ a_{1,j} \end{bmatrix} + 3 \begin{bmatrix} a_{1,j} \\ a_{2,j} \\ a_{3,j} \\ a_{0,j}. \end{bmatrix}$$

- **32bits** 워드 X가 **4개의 바이트 x_i**로 구성됨
- **GF(2^8) 상의 x(2)** 또는 **x+1(3)** 곱셈

구현: MixColumns

$$S = [A, B, C, D]$$

$$\begin{aligned}
A' &= (2A + \texttt{RotLeft}^2(A)) + \texttt{RotLeft}((2A + \texttt{RotLeft}^2(A)) + A) \\
B' &= (2B + \texttt{RotLeft}^2(B)) + \texttt{RotLeft}((2B + \texttt{RotLeft}^2(B)) + B) \\
C' &= (2C + \texttt{RotLeft}^2(C)) + \texttt{RotLeft}((2C + \texttt{RotLeft}^2(C)) + C) \\
D' &= (2D + \texttt{RotLeft}^2(D)) + \texttt{RotLeft}((2D + \texttt{RotLeft}^2(D)) + D)
\end{aligned}$$

- AES 128의 상태는 32-bit 워드 네 개에 저장
  - 상태 S로 정의, **전체 128bits**
- 각각의 상태에 대해 **상기의 식**을 도출 가능

구현: MixColumns

$$\begin{aligned}
\texttt{RotLeft_128}(S) &= [\texttt{RotLeft}(A), \texttt{RotLeft}(B), \texttt{RotLeft}(C), \texttt{RotLeft}(D)], \\
\texttt{Rev32_128}(S) &= [\texttt{Rev32}(A), \texttt{Rev32}(B), \texttt{Rev32}(C), \texttt{Rev32}(D)], \\
\texttt{RotRight_128}(S) &= [\texttt{RotRight}(A), \texttt{RotRight}(B), \texttt{RotRight}(C), \texttt{RotRight}(D)], \\
2S &= [2A, 2B, 2C, 2D].
\end{aligned}$$

- Left -> Rev -> Right 순서로 연산
- 이때 **ASIMD를 사용하여 128bits를 동시**에 연산
  - ASIMD의 128-bit 레지스터 활용

구현: MixColumns

$$\texttt{MixColumns}(S) = (2S + \texttt{RotLeft\_128}^2(S)) + \texttt{RotLeft\_128}((2S + \texttt{RotLeft\_128}^2(S)) + S)$$

- 조건: AES 상태가 **128-bit 레지스터에 저장**
- 직전 슬라이드의 공식을 정리
- 상기의 공식 획득

구현: MixColumns

$$\texttt{Rev32\_128}(S) = [\texttt{Rev32}(A), \texttt{Rev32}(B), \texttt{Rev32}(C), \texttt{Rev32}(D)]$$

$$\mathrm{S} = [\ a_{0,0}\ a_{0,1}\ a_{0,2}\ a_{0,3}\ a_{1,0}\ a_{1,1}\ a_{1,2}\ a_{1,3}\ a_{2,0}\ a_{2,1}\ a_{2,2}\ a_{2,3}\ a_{3,0}\ a_{3,1}\ a_{3,2}\ a_{3,3}\ ]$$

$$\downarrow \mathrm{S}' = \texttt{vrev32q\_u16}(\mathrm{A})$$

$$\mathrm{S}' = [\ a_{0,2}\ a_{0,3}\ a_{0,0}\ a_{0,1}\ a_{1,2}\ a_{1,3}\ a_{1,2}\ a_{1,1}\ a_{2,2}\ a_{2,3}\ a_{2,0}\ a_{2,1}\ a_{3,2}\ a_{3,3}\ a_{3,0}\ a_{3,1}\ ]$$

- Rev32_128 구현
- ASIMD 명령어 rev32(vrev342q_u16)

구현: MixColumns

$$\texttt{RotLeft\_128}(S) = [\texttt{RotLeft}(A), \texttt{RotLeft}(B), \texttt{RotLeft}(C), \texttt{RotLeft}(D)]$$

$$\mathrm{S} = [\,a_{0,0}\ a_{0,1}\ a_{0,2}\ a_{0,3}\ a_{1,0}\ a_{1,1}\ a_{1,2}\ a_{1,3}\ a_{2,0}\ a_{2,1}\ a_{2,2}\ a_{2,3}\ a_{3,0}\ a_{3,1}\ a_{3,2}\ a_{3,3}\,]$$

$$\mathrm{T} = \texttt{vrev32q\_u8}(\mathrm{S})$$

$$\mathrm{S}' = \texttt{vtrn2q\_u8}(\mathrm{S}, \mathrm{T})$$

$$\mathrm{S}' = [\,a_{0,1}\ a_{0,2}\ a_{0,3}\ a_{0,0}\ a_{1,1}\ a_{1,2}\ a_{1,3}\ a_{1,0}\ a_{2,1}\ a_{2,2}\ a_{2,3}\ a_{2,0}\ a_{3,1}\ a_{3,2}\ a_{3,3}\ a_{3,0}\,]$$

- RotLeft_128 구현
- ASIMD 명령어 vrev32q_u8, vtrn2q_u8 두 개를 사용

구현: MixColumns

```
sshr v8.16b, v0.16b, #7
 shl v4.16b, v0.16b, #1
 and v8.16b, v8.16b, v13.16b
 eor v4.16b, v4.16b,  v8.16b
```

- 16-byte 레지스터에서 GF(2^8) 상의 x 곱하기
  - 1 left-shift -> modulo P(x)
  - $P(x) = x^8 + x^4 + x^3 + x + 1$ (irreducible polynomial)
  - 16회 반복
- 마지막으로 XOR 연산

구현: MixColumns

- tbl/tbx 연산을 사용하지 않은 이유
  - Cortex-A53 32-bit 모드는 병렬 실행이 가능
  - Cortex-A53 **64-bit 모드는 연산 비용이 증가**
    - tbl/tbx 명령어의 마이크로연산량이 증가
- ASIMD를 사용한 이유
  - AES 상태 정보를 128-bit 레지스터에 저장하기 위함
  - ARMv8-A 레지스터: 64-bit
  - **ASIMD 레지스터: 128-bit**
  - **동시접근(인터리빙)이 가능**: 데이터 의존성 저하

구현: MixColumns

$$X = \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{bmatrix} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \\ b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} & b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} & b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} & b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \\ c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} & c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} & c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} & c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \\ d_{0,0} & d_{0,1} & d_{0,2} & d_{0,3} & d_{1,0} & d_{1,1} & d_{1,2} & d_{1,3} & d_{2,0} & d_{2,1} & d_{2,2} & d_{2,3} & d_{3,0} & d_{3,1} & d_{3,2} & d_{3,3} \end{bmatrix}$$

$\downarrow$ Transposition

$$X_P = \begin{bmatrix} X'_0 \\ X'_1 \\ X'_2 \\ X'_3 \end{bmatrix} = \begin{bmatrix} a_{0,0} & a_{1,0} & a_{2,0} & a_{3,0} & b_{0,0} & b_{1,0} & b_{2,0} & b_{3,0} & c_{0,0} & c_{1,0} & c_{2,0} & c_{3,0} & d_{0,0} & d_{1,0} & d_{2,0} & d_{3,0} \\ a_{0,1} & a_{1,1} & a_{2,1} & a_{3,1} & b_{0,1} & b_{1,1} & b_{2,1} & b_{3,1} & c_{0,1} & c_{1,1} & c_{2,1} & c_{3,1} & d_{0,1} & d_{1,1} & d_{2,1} & d_{3,1} \\ a_{0,2} & a_{1,2} & a_{2,2} & a_{3,2} & b_{0,2} & b_{1,2} & b_{2,2} & b_{3,2} & c_{0,2} & c_{1,2} & c_{2,2} & c_{3,2} & d_{0,2} & d_{1,2} & d_{2,2} & d_{3,2} \\ a_{0,3} & a_{1,3} & a_{2,3} & a_{3,3} & b_{0,3} & b_{1,3} & b_{2,3} & b_{3,3} & c_{0,3} & c_{1,3} & c_{2,3} & c_{3,3} & d_{0,3} & d_{1,3} & d_{2,3} & d_{3,3} \end{bmatrix}$$

- Transpose
  - 128-bit를 32-bit AES 상태에 맞도록 변환이 필요

구현: MixColumns

- Transpose 구현
  - 32-bit 상에서: 16개의 trn1/trn2 명령어로 구현
  - **64-bit 상에서: ld4 명령어로 구현**
  - ARMv8-A에서는 16회 명령어 호출보다 4회 128비트 저장 후 호출이 빠름
- 구현 방법
  - 64bytes(512bits)를 메모리에 저장
  - ld4 명령어로 호출
  - st4 명령어로 패턴 뒤집기

구현: MixColumns

$$
\begin{aligned}
X'_P &= \texttt{MixColumns}(X_P) = T_0 + T_1 \\
T_0 &= 2X_P + \texttt{RotLeft}^2_P(X_P) \\
&= [2X'_0, 2X'_1, 2X'_2, 2X'_3] + [X'_2, X'_3, X'_0, X'_1] \\
&= [2X'_0 + X'_2, 2X'_1 + X'_3, 2X'_2 + X'_0, 2X'_3 + X'_1] \\
T_1 &= \texttt{RotLeft}_P(T_0 + X_P) \\
&= \texttt{RotLeft}_P([2X'_0 + X'_2, 2X'_1 + X'_3, 2X'_2 + X'_0, 2X'_3 + X'_1] \\
&\quad + [X'_0, X'_1, X'_2, X'_3]) \\
&= \texttt{RotLeft}_P([3X'_0 + X'_2, 3X'_1 + X'_3, 3X'_2 + X'_0, 3X'_3 + X'_1]) \\
&= [3X'_1 + X'_3, 3X'_2 + X'_0, 3X'_3 + X'_1, 3X'_0 + X'_2] \\
X'_P &= [2X'_0 + X'_2, 2X'_1 + X'_3, 2X'_2 + X'_0, 2X'_3 + X'_1] \\
&\quad + [3X'_1 + X'_3, 3X'_2 + X'_0, 3X'_3 + X'_1, 3X'_0 + X'_2]
\end{aligned}
$$

- MixColumns 최종

$$
\begin{aligned}
X'_P = [&2X'_0 + 3X'_1 + X'_2 + X'_3, X'_0 + 2X'_1 + 3X'_2 + X'_3, \\
&X'_0 + X'_1 + 2X'_2 + 3X'_3, 3X'_0 + X'_1 + X'_2 + 2X'_3] \\
= [&2(X'_0 + X'_1) + X'_1 + (X'_2 + X'_3)\ ,\ 2(X'_1 + X'_2) + X'_0 + (X'_2 + X'_3), \\
&2(X'_2 + X'_3) + X'_3 + (X'_0 + X'_1)\ ,\ 2(X'_3 + X'_0) + X'_2 + (X'_0 + X'_1)]
\end{aligned}
$$

구현: MixColumns

$X_P = [X'_0, X'_1, X'_2, X'_3]$ 가 존재할 때,

1 $X'_P = \texttt{MixColumns}(X_P) = T_0 + T_1$

2 $A' = (2A + \texttt{RotLeft}^2(A)) + \texttt{RotLeft}((2A + \texttt{RotLeft}^2(A)) + A)$

3
$$T_0 = 2X_P + \texttt{RotLeft}_P^2(X_P)$$
$$T_1 = \texttt{RotLeft}_P(T_0 + X_P)$$

- 1: T_0, T_1 두개의 파트로 분할
- 2: 기존에 있던 식을 근거로
- 3: T_0, T_1 을 정의할 수 있음
  - T_1 내부에 T_0 구조가 포함됨

구현: MixColumns

$X_P = [X'_0, X'_1, X'_2, X'_3]$ 가 존재할 때,

$$
\begin{aligned}
T_0 &= 2X_P + \mathtt{RotLeft}_P^2(X_P) \\
&= [2X'_0, 2X'_1, 2X'_2, 2X'_3] + [X'_2, X'_3, X'_0, X'_1] \\
&= [2X'_0 + X'_2, 2X'_1 + X'_3, 2X'_2 + X'_0, 2X'_3 + X'_1]
\end{aligned}
$$

$$
\begin{aligned}
T_1 &= \mathtt{RotLeft}_P(T_0 + X_P) \\
&= \mathtt{RotLeft}_P([2X'_0 + X'_2, 2X'_1 + X'_3, 2X'_2 + X'_0, 2X'_3 + X'_1] \\
&\quad + [X'_0, X'_1, X'_2, X'_3]) \\
&= \mathtt{RotLeft}_P([3X'_0 + X'_2, 3X'_1 + X'_3, 3X'_2 + X'_0, 3X'_3 + X'_1]) \\
&= [3X'_1 + X'_3, 3X'_2 + X'_0, 3X'_3 + X'_1, 3X'_0 + X'_2]
\end{aligned}
$$

- X_P를 대입하여 계산

구현: MixColumns

4

$$T_0 = [2X'_0 + X'_2, 2X'_1 + X'_3, 2X'_2 + X'_0, 2X'_3 + X'_1]$$
$$T_1 = [3X'_1 + X'_3, 3X'_2 + X'_0, 3X'_3 + X'_1, 3X'_0 + X'_2]$$

5

$$\begin{aligned}
X'_P &= \texttt{MixColumns}(X_P) = T_0 + T_1 \\
&= [2X'_0 + X'_2, 2X'_1 + X'_3, 2X'_2 + X'_0, 2X'_3 + X'_1] \\
&\quad + [3X'_1 + X'_3, 3X'_2 + X'_0, 3X'_3 + X'_1, 3X'_0 + X'_2] \\
&= [2X'_0 + 3X'_1 + X'_2 + X'_3, X'_0 + 2X'_1 + 3X'_2 + X'_3, \\
&\quad X'_0 + X'_1 + 2X'_2 + 3X'_3, 3X'_0 + X'_1 + X'_2 + 2X'_3] \\
&= [2(X'_0 + X'_1) + X'_1 + (X'_2 + X'_3)\ ,\ 2(X'_1 + X'_2) + X'_0 + (X'_2 + X'_3), \\
&\quad 2(X'_2 + X'_3) + X'_3 + (X'_0 + X'_1)\ ,\ 2(X'_3 + X'_0) + X'_2 + (X'_0 + X'_1)]
\end{aligned}$$

- 4: 획득한 T_0, T_1
- 5: X'_P에 4를 대입 후 식 전개
- 4-way Transpose 완성

구현: MixColumns

4

$$T_0 = [2X'_0 + X'_2, 2X'_1 + X'_3, 2X'_2 + X'_0, 2X'_3 + X'_1]$$
$$T_1 = [3X'_1 + X'_3, 3X'_2 + X'_0, 3X'_3 + X'_1, 3X'_0 + X'_2]$$

5

$$\begin{aligned}
X'_P &= \texttt{MixColumns}(X_P) = T_0 + T_1 \\
&= [2X'_0 + X'_2, 2X'_1 + X'_3, 2X'_2 + X'_0, 2X'_3 + X'_1] \\
&\quad + [3X'_1 + X'_3, 3X'_2 + X'_0, 3X'_3 + X'_1, 3X'_0 + X'_2] \\
&= [2X'_0 + 3X'_1 + X'_2 + X'_3, X'_0 + 2X'_1 + 3X'_2 + X'_3, \\
&\quad X'_0 + X'_1 + 2X'_2 + 3X'_3, 3X'_0 + X'_1 + X'_2 + 2X'_3] \\
&= [2(X'_0 + X'_1) + X'_1 + (X'_2 + X'_3)\ ,\ 2(X'_1 + X'_2) + X'_0 + (X'_2 + X'_3), \\
&\quad 2(X'_2 + X'_3) + X'_3 + (X'_0 + X'_1)\ ,\ 2(X'_3 + X'_0) + X'_2 + (X'_0 + X'_1)]
\end{aligned}$$

- 4: 획득한 T_0, T_1
- 5: X'_P에 4를 대입 후 식 전개
- 4-way Transpose 완성

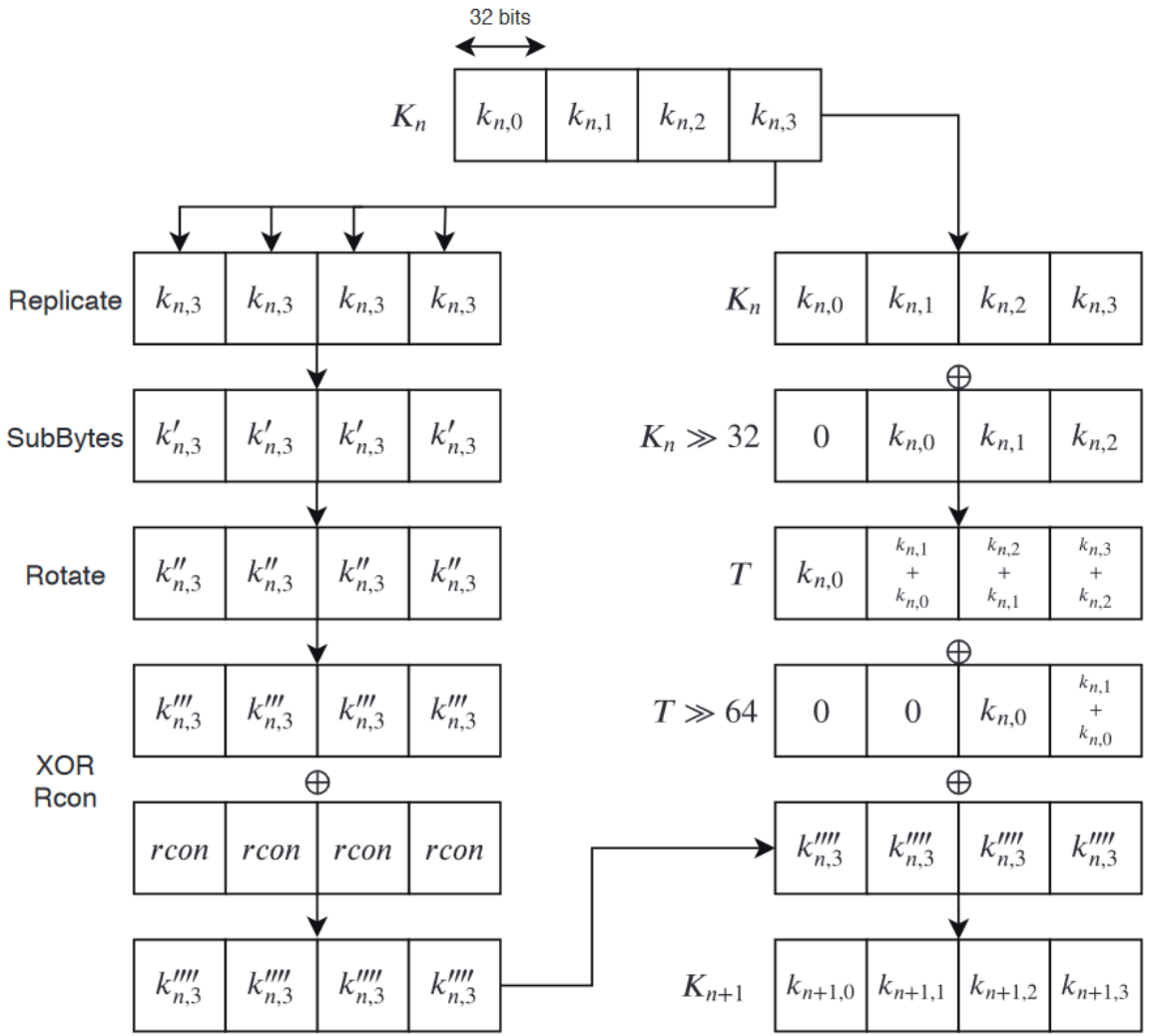
구현: MixColumns

Input: 128-bit transposed AES states $X'_i, 0 \leq i \leq 3$.
Output: 128-bit transposed AES states processed with `MixColumns` $X''_i, 0 \leq i \leq 3$.

1: $T0 \leftarrow X'_0 \oplus X'_1$
2: $T1 \leftarrow X'_2 \oplus X'_3$
3: $T2 \leftarrow X'_0 \oplus X'_2$
4: $T3 \leftarrow 2 \cdot T0$ {a multiplication by x in $GF(2^8)$}
5: $T4 \leftarrow 2 \cdot T1$
6: $T5 \leftarrow 2 \cdot T2$
7: $X''_0 \leftarrow T1 \oplus (X'_1 \oplus T3)$
8: $X''_1 \leftarrow T0 \oplus (X''_0 \oplus T5)$
9: $X''_2 \leftarrow T0 \oplus (X'_3 \oplus T4)$
10: $X''_3 \leftarrow T1 \oplus (X''_2 \oplus T5)$
11: **return** $X''_i, 0 \leq i \leq 3$

- 4-way Transpose + MixColumns의 의사 코드
  - **MixColumns**
  - **4-way Transpose**

구현: Key Schedule



- 시리얼(직렬) 알고리즘
  - 32-bit 워드 키 별로 의존성 발생
- **ASIMD 사용 시 의존성 제거**
  - 128-bit 워드 단위로 동작

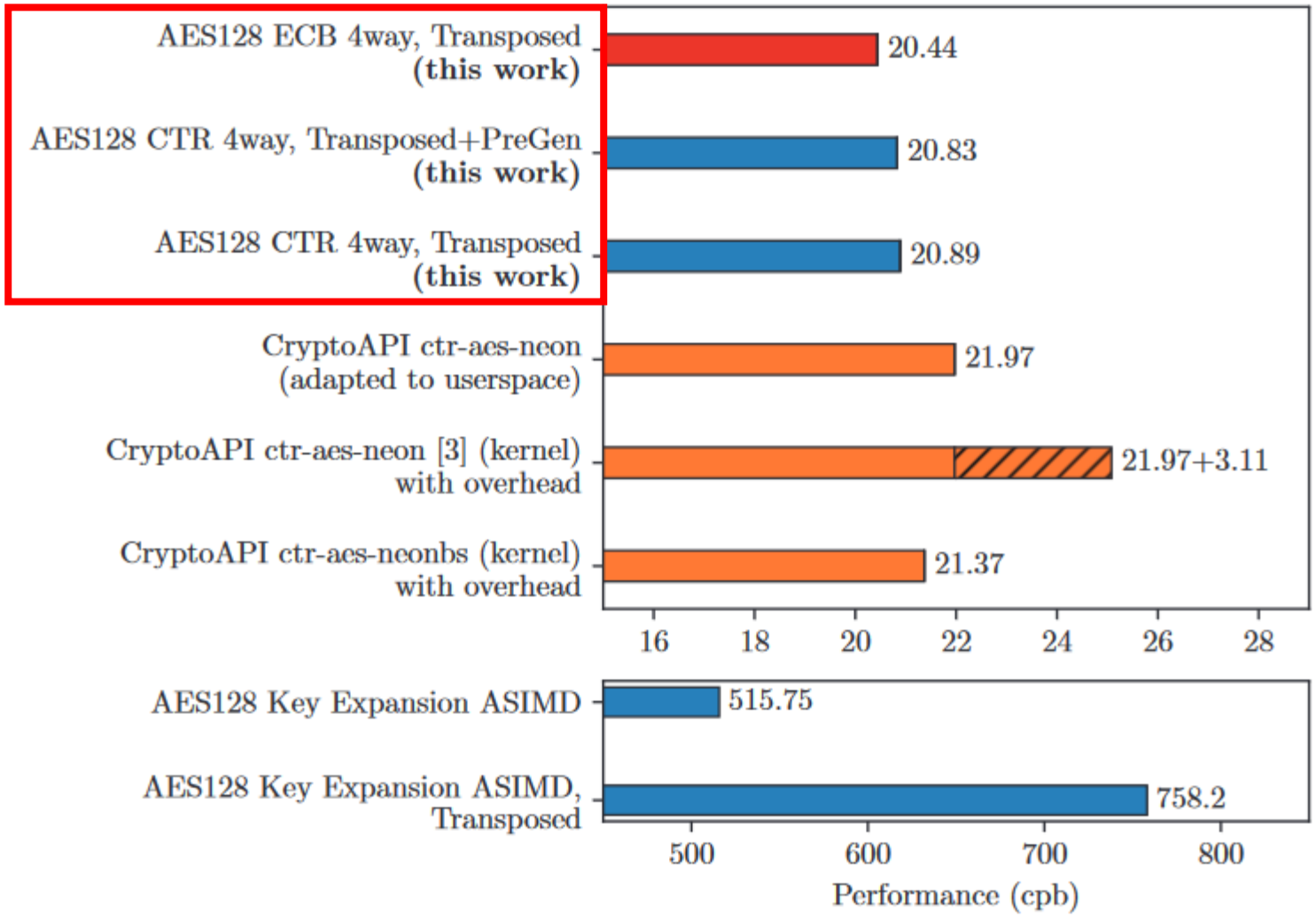
성능평가

- 시스템
  - 라즈베리 파이 3B 보드
  - 리눅스 커널 5.2
  - Broadcom BCM2837 CPU
    - ARMv8-A Cortex-A53 quad core
    - without Cryptography Extensions
  - 1.2GHz CPU Clock
- 툴체인: aarch64-linux-gnu-gcc ver 8.3

성능평가

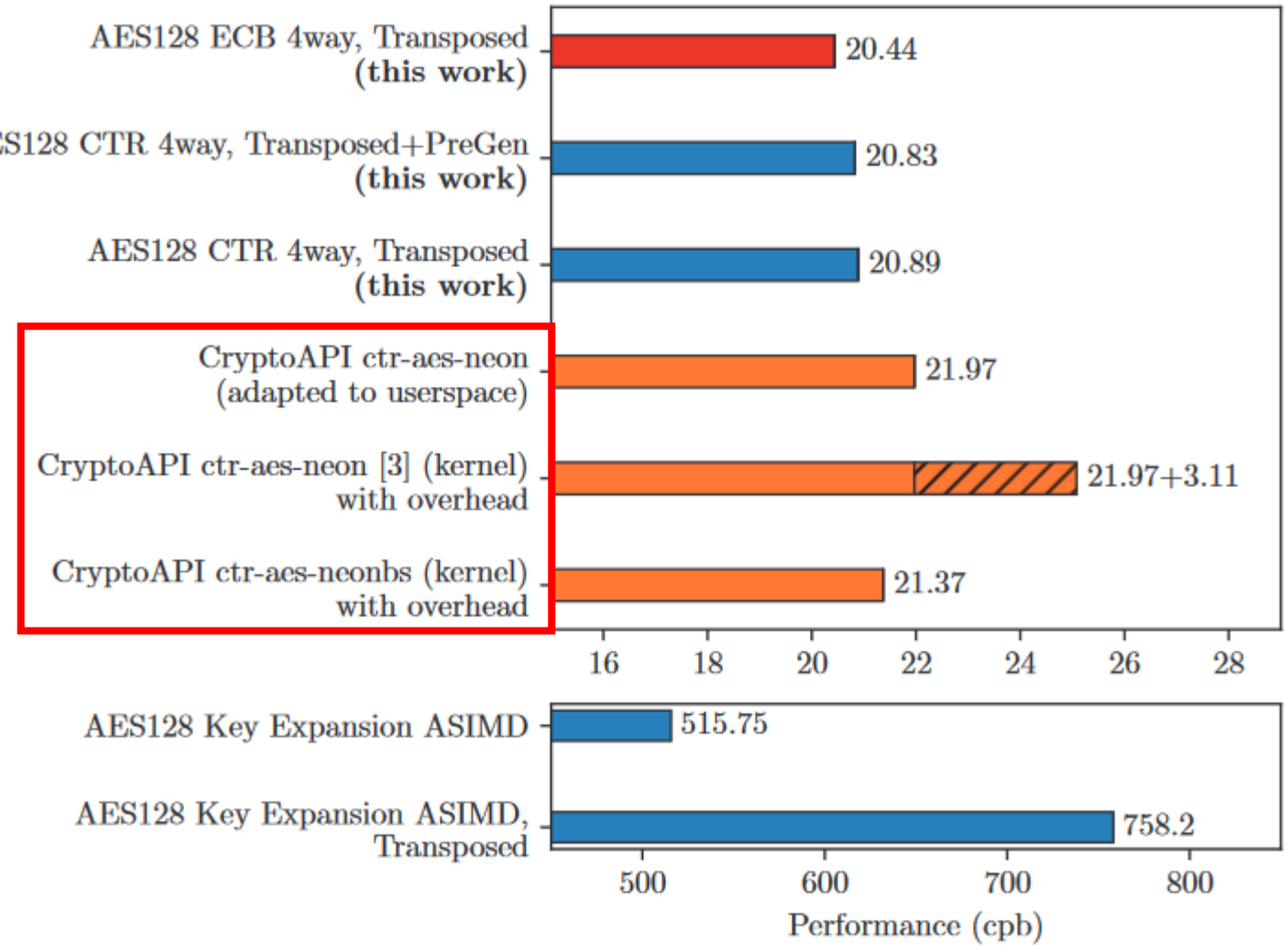
- 환경
  - AES-128 ECB, CTR 모드
  - 4KiB 메시지 사이즈
  - 2^{15}회 반복
  - 이전 출력 값이 이후 입력 값으로 적용
    - 단, 초기 입력 값은 /dev/urandom에서 획득

성능평가



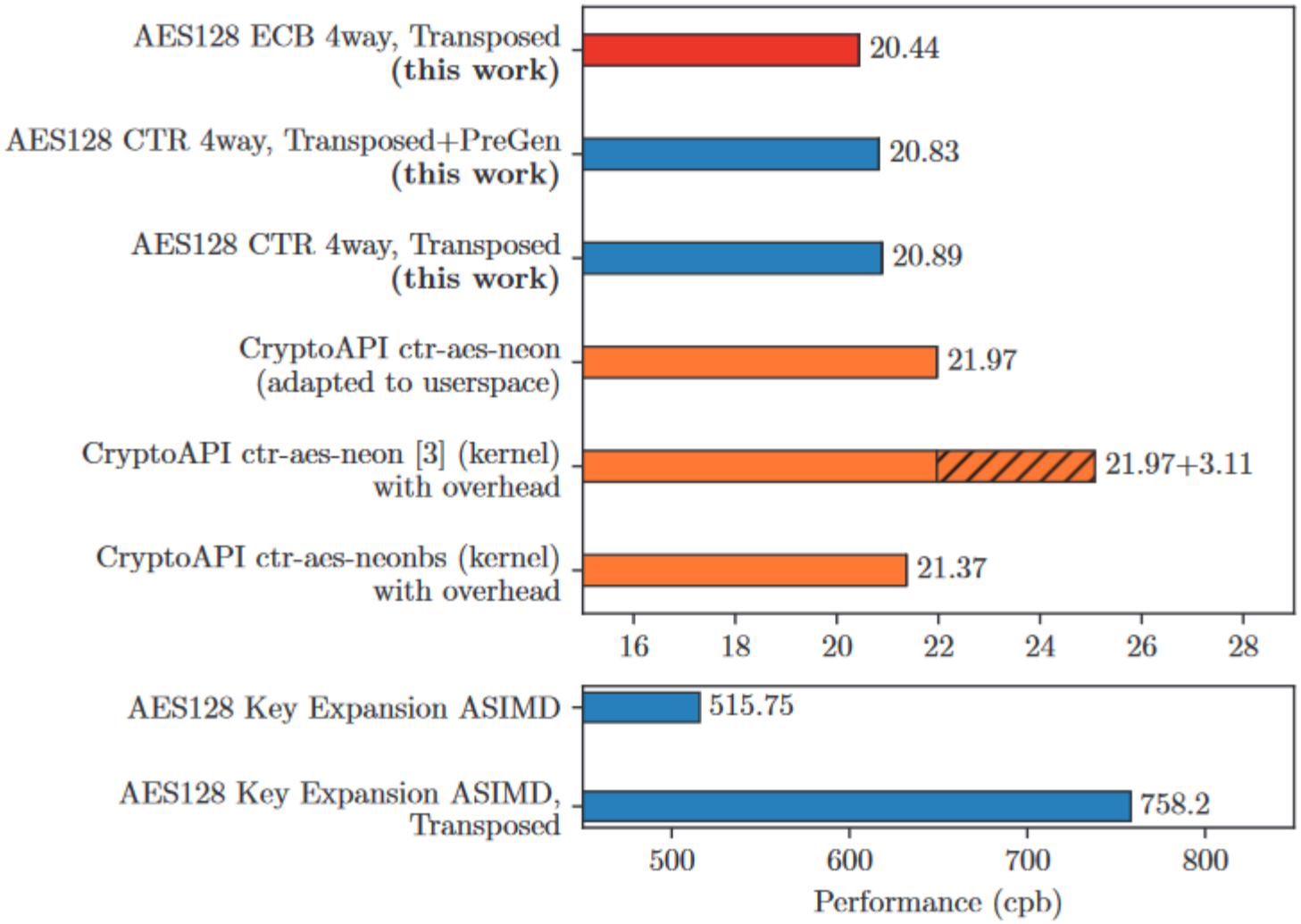
- Transposed
  - 제안 기법
- PreGen
  - 카운터 증가
  - 임시 버퍼 사용

성능평가



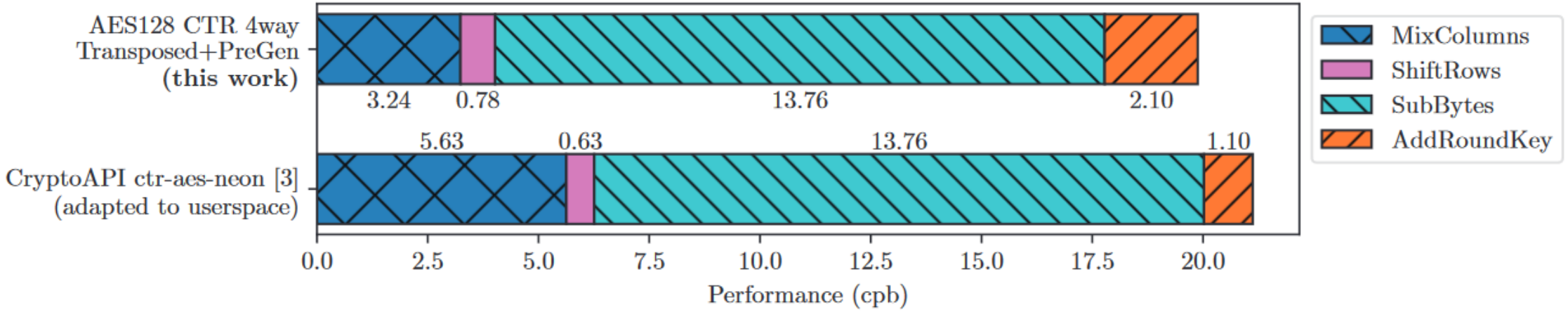
- Cryptography API
  - = CryptoAPI
  - libkcapi 라이브러리
- ctr-aes-neon
  - ASIMD 명령어 사용 버전
- ctr-aes-neonbs
  - ASIMD + Bit Slicing

성능평가



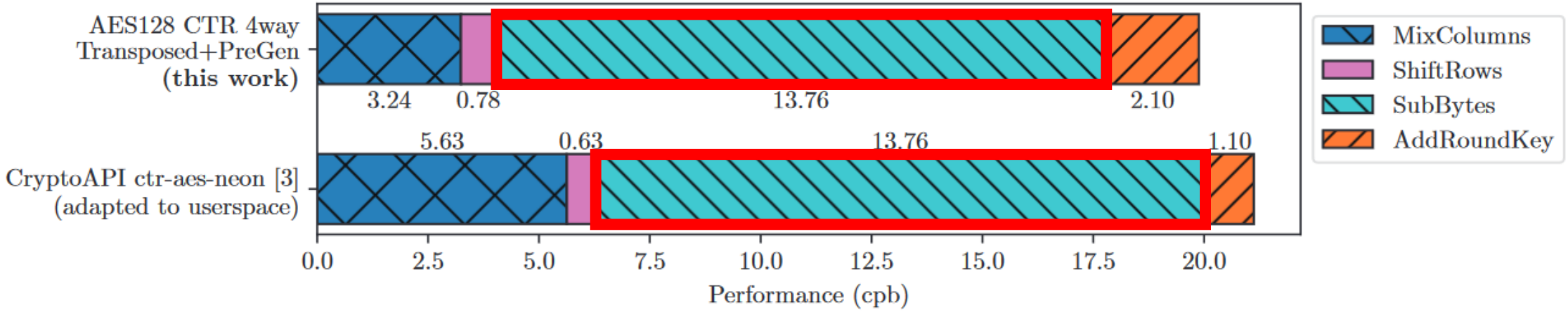
- 리눅스 커널에 포함된 AES
  - **다른 버전 또는 암호를 지원 할 필요 존재**
- 따라서 제안 방식과 직접적인 비교가 어려움
  - 비교가 **정당하지 않기** 때문

성능평가



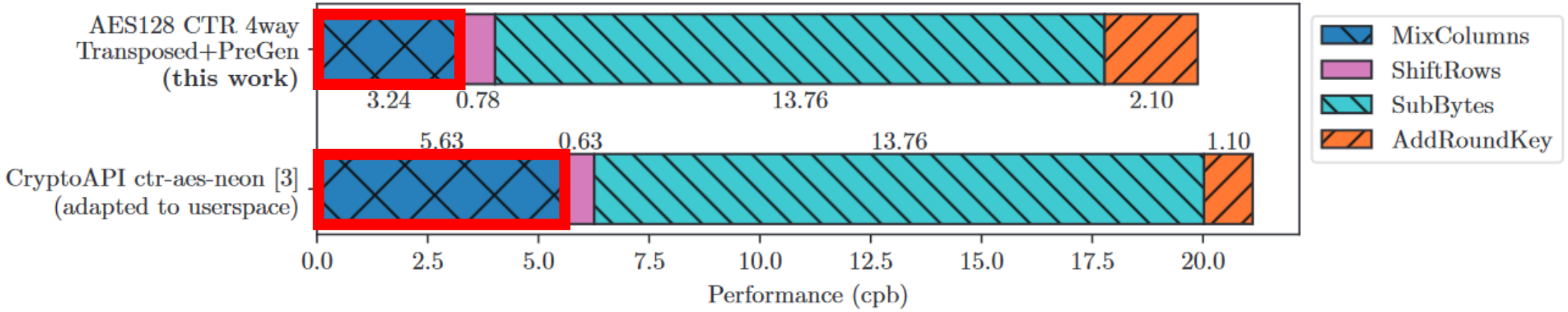
- AES의 각 단계 별로 cpb를 비교
  - 단, 본 비교에는 입력, 출력 및 암호화와 연관 없는 부분의 비용은 제외
  - 따라서 본 그래프는 **AES 암호의 전체 cpb를 완벽히 의미하지는 않음**

성능평가



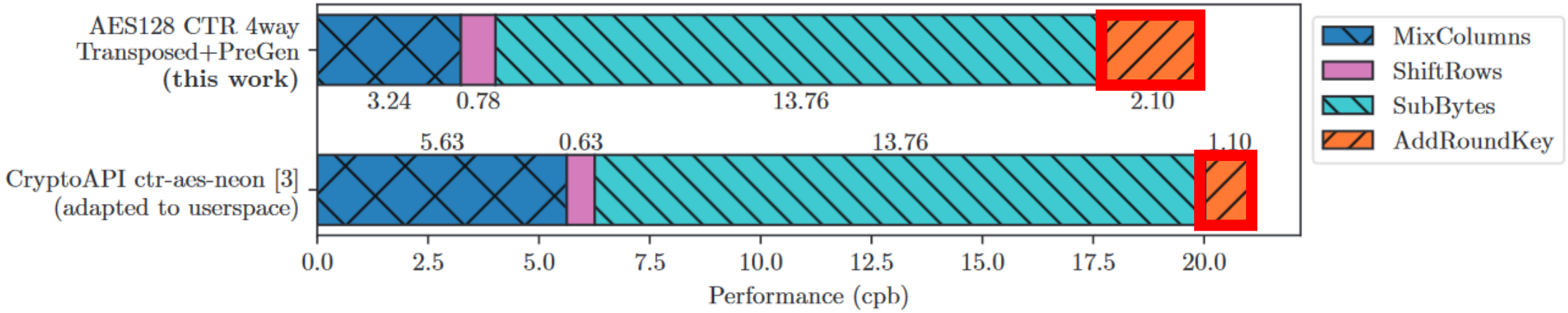
- 공통점
  - **tbl/tbx 명령어** 사용으로 인하여 **긴 SubBytes 연산 시간**이 발생
  - 전체 AES 연산의 약 66% 차지

성능평가



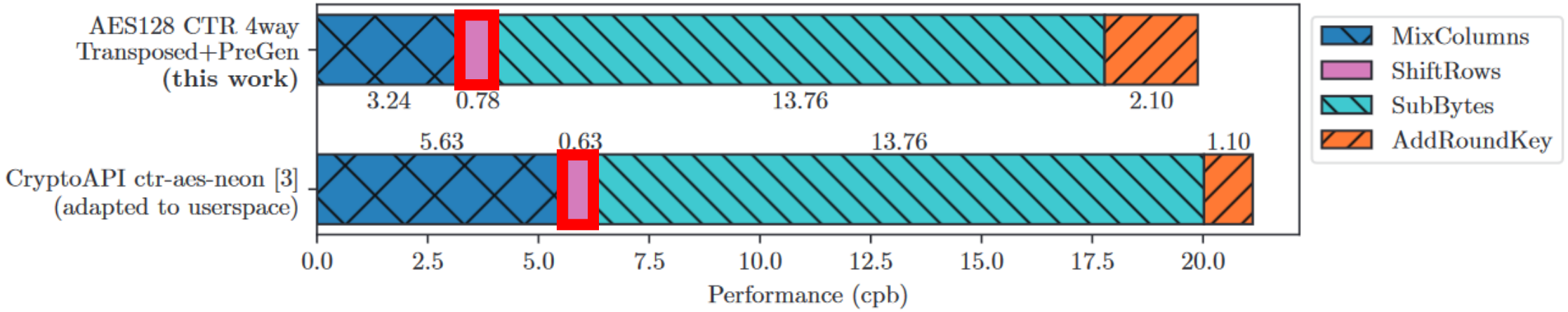
- 차이점
  - 제안 기법은 MixColumns 단계에서 permutation 과정이 생략됨
  - 비교 대상에 비해 약 **42.47% 성능 향상**

성능평가



- 차이점
  - 제안 기법은 AddRoundKey 단계에서 로딩, 복제로 인해 성능 하락
  - Transpose 과정도 성능 하락에 영향
  - 비교 대상에 비해 약 **90% 성능 하락**

성능평가



- 차이점
  - 제안 기법은 ShiftRows 단계에서 tbl 연산 대신 rev, trn 연산을 사용
  - 비교 대상에 비해 약 **25% 성능 하락**
  - 하지만 전체 연산의 5%도 차지하지 않으므로 **매우 미미한 악영향**

결론

- **64-bit** 프로세서를 대상으로 AES 구현
- **4개 블록을 동시에 처리**하는 최적 구현
  - MixColumns에 집중
- 기존 기법에 비해 **약 5%의 성능 향상**을 보임
  - 특히 MixColumns 에서 뛰어난 성능 향상 제공
- 다른 연산 부분의 추가적인 최적 구현이 필요