2022 정보보호학회
하계 학술대회 논문 리뷰 (암호 부문)

IT정보공학과 신명수

목차

1. IoT 환경을 위한 경량암호 최적화 동향분석

(최용렬, 김영범, 서석충 (국민대학교). (2022) p.34)

1. IoT 환경을 위한 경량암호 최적화 동향 분석

- 경량암호 : 가용 자원이 제한된 환경에서 사용하기 적합한 암호 알고리즘.

- NIST에서는 현재 표준 경량 암호를 위한 공모 사업을 진행중이며 현재 10개의 알고리즘이 최종 후보로 선정되었음.

	AEAD (with Hash)	AEAD (without Hash)
Block Cipher based	-	GIFT-COFB, TinyJambu
Tweakable Block Cipher based	-	Romulus
Permutation based	ASCON, PHOTON-Beetle, SPARKLE, Xoodyak	Elephant, ISAP
Stream Cipher based	-	Grain-128AEAD

[표 1] NIST 경량암호 공모사업 최종 후보 알고리즘[1]

1 IoT 환경을 위한 경량암호 최적화 동향 분석

AEAD (Authenticated Encryption with Associated Data)

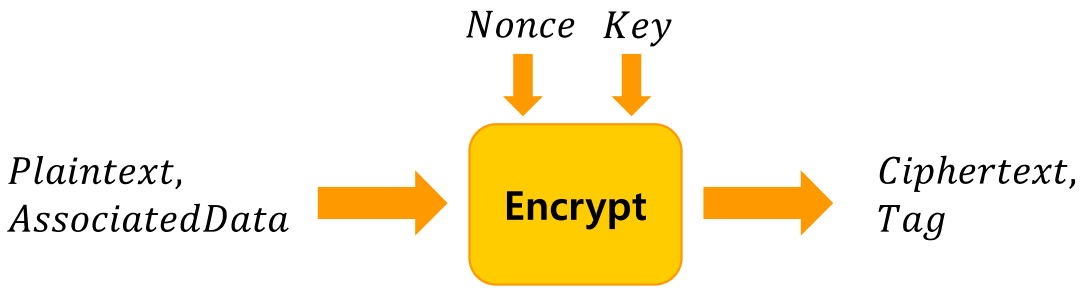
- 기밀성, 무결성, 인증을 보장하는 암호화 방식.
- Nonce와 Associated data를 사용하여 데이터의 암복호화를 진행.

1 IoT 환경을 위한 경량암호 최적화 동향 분석

AEAD (Authenticated Encryption with Associated Data)

- $Encrypt(Key, Plaintext, AssociatedData, Nonce) = (Ciphertext, Tag)$

- 송신자는 수신자에게 $(Nonce, AssociatedData, Ciphertext, Tag)$ 를 전송.
- Tag 는 $Plaintext$ 와 $AssociatedData$ 모두에 의존하며 둘 다 수정되지 않았을 때만 유효.

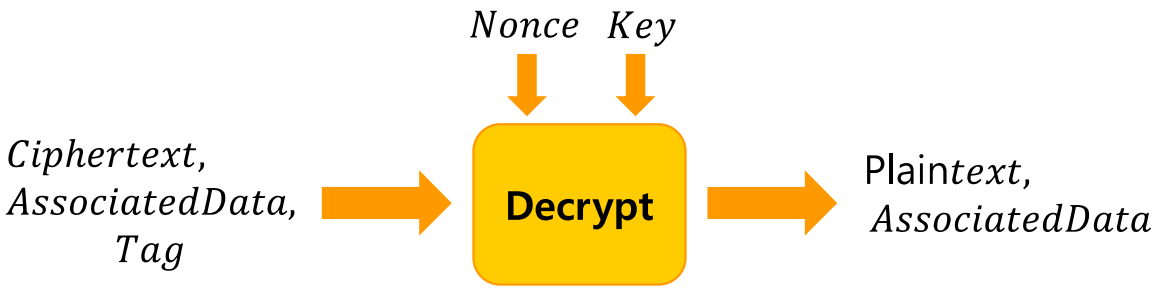


1 IoT 환경을 위한 경량암호 최적화 동향 분석

AEAD (Authenticated Encryption with Associated Data)

- $Decrypt(Key, Ciphertext, AssociatedData, Tag, Nonce) = (Plaintext, AssociatedData)$

- 인증값이 $AssociatedData$ 에 의존하므로, 수신자 쪽의 복호화에는 $AssociatedData$ 가 관여.
- 만약 $Ciphertext$ 나 $AssociatedData$ 가 손상되었으면 오류를 반환.



1 IoT 환경을 위한 경량암호 최적화 동향 분석

- 2022 암호분석경진대회 1번문제 – AE

다음은 n-bit 블록암호 E_K를 사용하는 한 인증모드에 대한 설명이다.

[블록암호 기반 인증모드 암호화 정의]

입력: nonce N, 메시지 $M = M_1 \| M_1 \| \cdots \| M_m$

출력: 암호문 $C = C_1 \| C_2 \| \cdots \| C_m$, 태그 값 T

$L = E_K(N)$

$\Sigma = 0$

$For\ i = 1 \sim m-1:$

$\quad \Sigma = \Sigma \oplus M_i$

$\quad C_i = E_K(M_i \oplus L) \oplus L$

$Pad = E_K(len(M_m) \oplus L)$

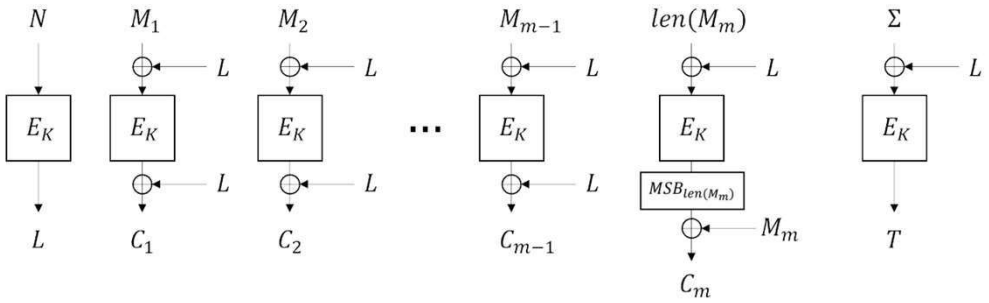
$C_m = M_m \oplus MSB_{len(M_m)}(Pad)$

$\Sigma = \Sigma \oplus Pad \oplus C_m \| 0^*$

$T = E_K(\Sigma \oplus L)$

$C = C_1 \| C_2 \| \cdots \| C_m$

return C, T



<블록암호 기반 인증모드 암호화 과정>

[문제]

1. 만약 M_m의 크기가 블록암호의 블록 크기와 같다면 Σ는 어떤 형태인가?

2. 중복된 태그값을 갖는 메시지(혹은 암호문)을 찾는 것을 태그 위조공격(forgery attack)이라고 한다. 위에 제시된 인증모드에서 서로 다른 두 메시지(혹은 암호문)이 동일한 태그값을 갖게 만드는 위조공격을 설명하시오 (단, 공격자는 nonce를 재사용 가능함.)

1 IoT 환경을 위한 경량암호 최적화 동향 분석

ARM Cortex-M

- 32비트 마이크로 프로세서 제품으로 IoT 기기와 같은 다양한 임베디드 환경에서 범용적으로 사용된다.

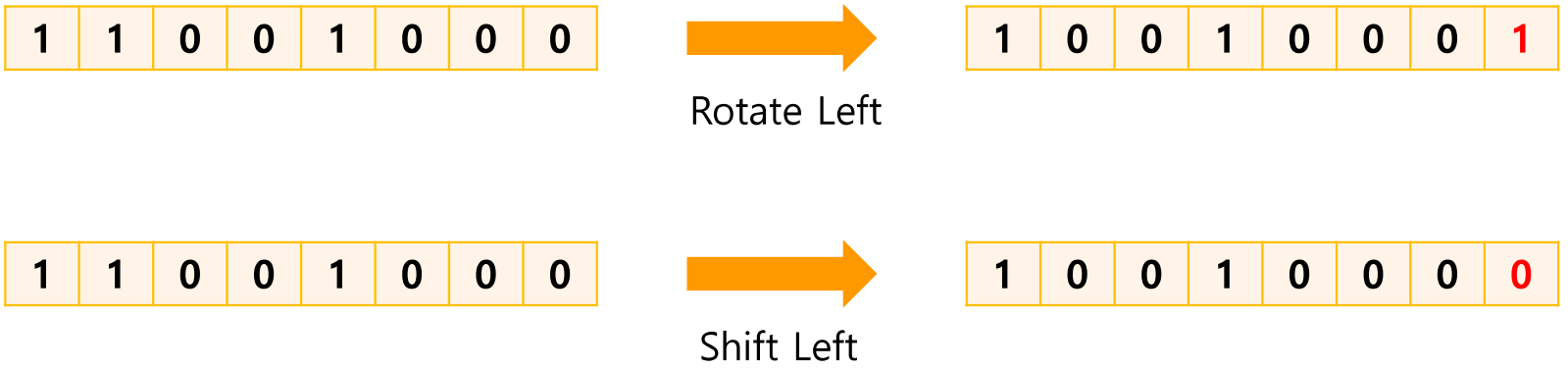
32비트 범용 레지스터 16개로 이루어져 있음.

MSP: Main Stack Pointer
PSP : Process Stack Pointer

1 IoT 환경을 위한 경량암호 최적화 동향 분석

ARM Cortex-M

- Barrel shifter
  - 비트 단위의 rotate와 shift 연산을 상수 시간으로 수행할 수 있게 하는 하드웨어 장치.



1 IoT 환경을 위한 경량암호 최적화 동향 분석

RISC-V

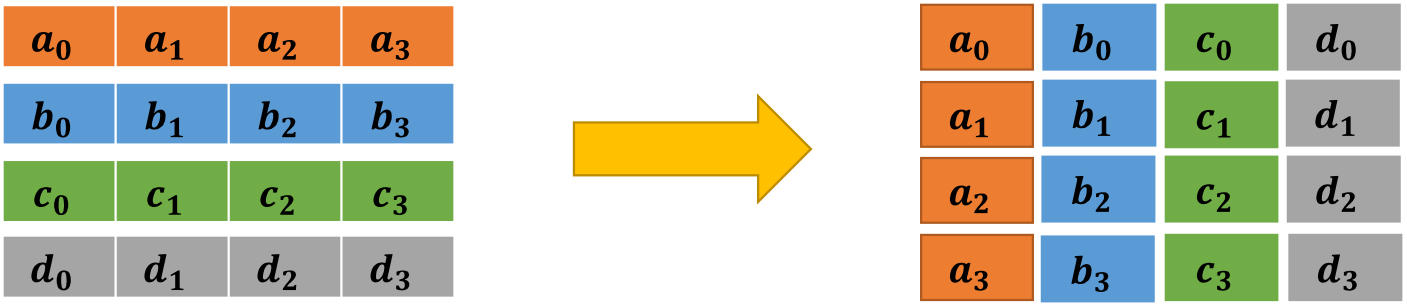
- RISC-V : (Reduced Intriction Set Computer)RISC 기반 오픈소스.
- 32개의 범용 레지스터를 가지며 한 개의 레지스터를 제외하고 모두 변수를 할당할 수 있다.
- Cortex-M 과 달리, barrel shifter를 지원하지 않음.

1 IoT 환경을 위한 경량암호 최적화 동향 분석

Fixslicing – Bitslicing 기법을 개선한 방법

- Bitslicing

1. 평문 블록을 bitslicing 포맷으로 변경한다. (Packing)

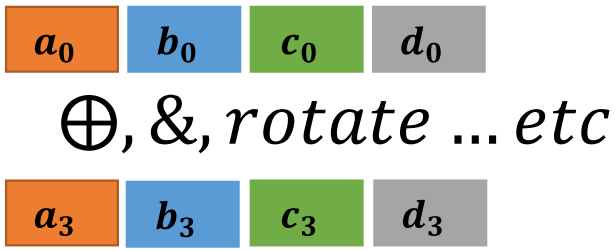


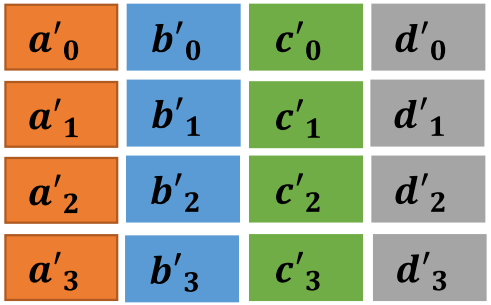
1 IoT 환경을 위한 경량암호 최적화 동향 분석

Fixslicing – Bitslicing 기법을 개선한 방법

- Bitslicing

2. 비트 연산자를 통해 데이터를 처리한다.



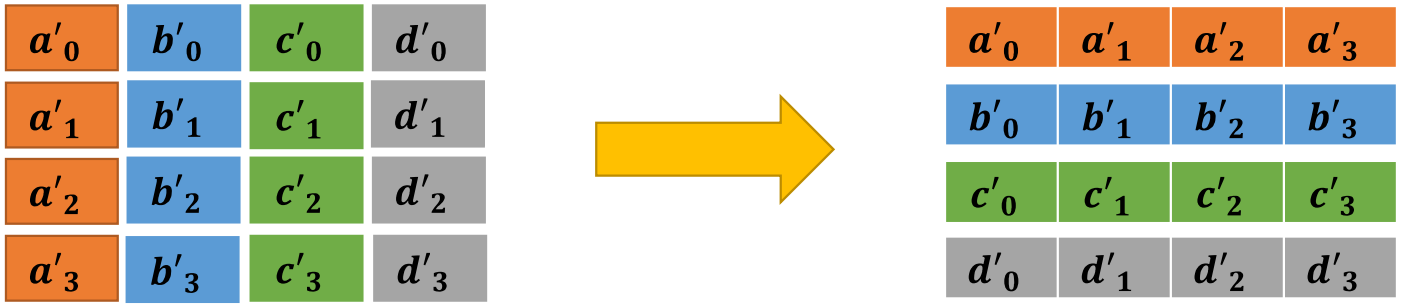


1 IoT 환경을 위한 경량암호 최적화 동향 분석

Fixslicing – Bitslicing 기법을 개선한 방법

- Bitslicing

3. bit-slice 데이터를 암호문 데이터로 변환한다. (Unpacking)



1 IoT 환경을 위한 경량암호 최적화 동향 분석

Fixslicing

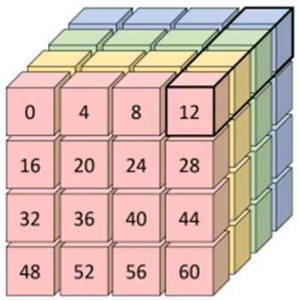


Figure 2: Cubic representation of the main state of GIFT-64. Each color refer to a slice matrix while the black cuboid is where an S-box is applied.

1 IoT 환경을 위한 경량암호 최적화 동향 분석

Fixslicing

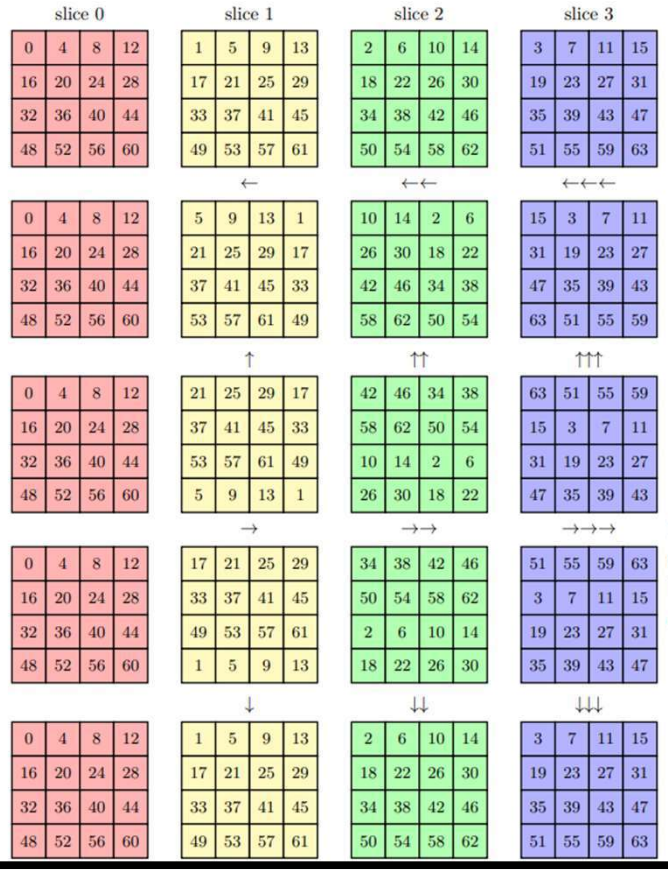
	slice 0	slice 1	slice 2	slice 3
	0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60	1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61	2 6 10 14 18 22 26 30 34 38 42 46 50 54 58 62	3 7 11 15 19 23 27 31 35 39 43 47 51 55 59 63
Round1	0 16 32 48 12 28 44 60 8 24 40 56 4 20 36 52	5 21 37 53 1 17 33 49 13 29 45 61 9 25 41 57	10 26 42 58 6 22 38 54 2 18 34 50 14 30 46 62	15 31 47 63 11 27 43 59 7 23 39 55 3 19 35 51
Round2	0 12 8 4 48 60 56 52 32 44 40 36 16 28 24 20	21 17 29 25 5 1 13 9 53 49 61 57 37 33 45 41	42 38 34 46 26 22 18 30 10 6 2 14 58 54 50 62	63 59 55 51 47 43 39 35 31 27 23 19 15 11 7 3
Round3	0 48 32 16 4 52 36 20 8 56 40 24 12 60 44 28	17 1 49 33 21 5 53 37 25 9 57 41 29 13 61 45	34 18 2 50 38 22 6 54 42 26 10 58 46 30 14 62	51 35 19 3 55 39 23 7 59 43 27 11 63 47 31 15
Round4	0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60	1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61	2 6 10 14 18 22 26 30 34 38 42 46 50 54 58 62	3 7 11 15 19 23 27 31 35 39 43 47 51 55 59 63

- Slice 0 matrix: swap row 1 with 3
- Slice 1 matrix: swap row 0 with 1, and swap row 2 with 3
- Slice 2 matrix: swap row 0 with 2
- Slice 3 matrix: swap row 0 with 3, and swap row 1 with 2

Figure 3: Classical representation of the GIFT-64 round function during 4 rounds. Each cell represents a bit, and the numbers in the cells then denote the actual index of that particular bit in the state. Slice 0 (resp. 1/2/3) depicted in red (resp. yellow/green/blue) represents all the bits at position 0 (resp. 1/2/3) of the S-boxes of the cipher state.

1 IoT 환경을 위한 경량암호 최적화 동향 분석

Fixslicing



Slice 0 을 고정하고 다른 slice를 움직여,
연산 결과가 같아지도록 최소한의 연산을 수행.

- if `i%4=0`, rotate slice j matrix by j columns to the left
- if `i%4=1`, rotate slice j matrix by j rows to the top
- if `i%4=2`, rotate slice j matrix by j columns to the right
- if `i%4=3`, rotate slice j matrix by j rows to the bottom

Figure 4: New representation of the GIFT-64 round function during 4 rounds. Each cell represents a bit, and the numbers in the cells then denote the actual index of that particular bit in the state. Slice 0 (resp. 1/2/3) depicted in red (resp. yellow/green/blue) represents all the bits at position 0 (resp. 1/2/3) of the S-boxes of the cipher state.

1 IoT 환경을 위한 경량암호 최적화 동향 분석

Fixslicing

- 기존 GIFT 구현 방식보다 성능이 약 5배 향상되었음.
- AES와 SKINNY(AES와 유사한 구조를 가짐)에 적용되어 약 26% 성능 향상.
- GIFT-COFB, Romulus 성능 또한 향상 되었음.

2. GPU 상에서 경량 블록 암호 PIPO 최적 구현

(김현준, 엄시우, 서화정 (한성대학교). (2022) p.83)

2. GPU상에서 경량 블록암호 PIPO의 최적 구현

효율적인 마스킹 구현을 위해 비선형 연산의 수를 최소화하는 데 중점을 둔 암호

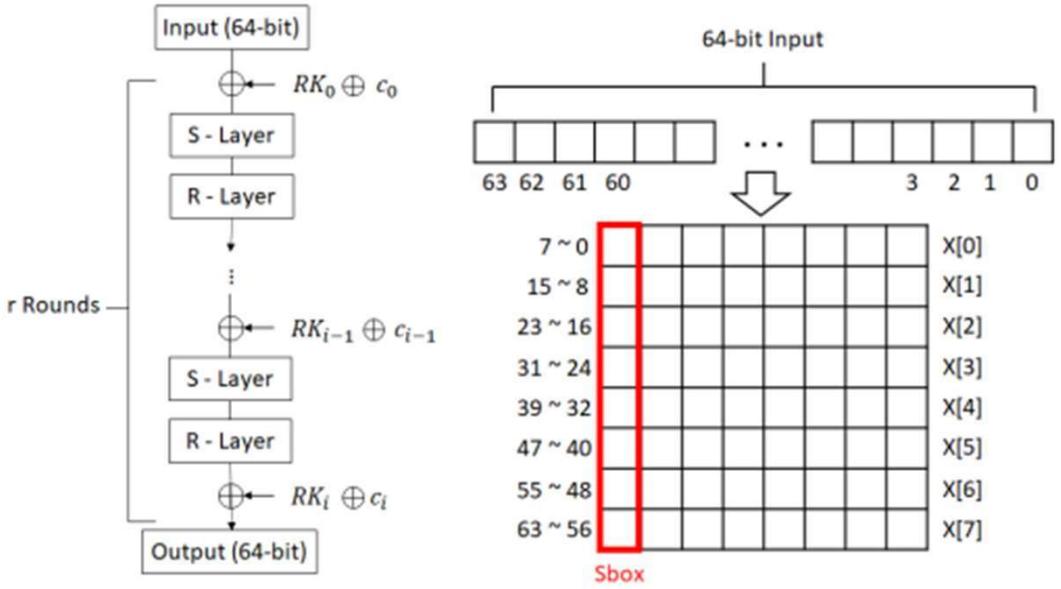
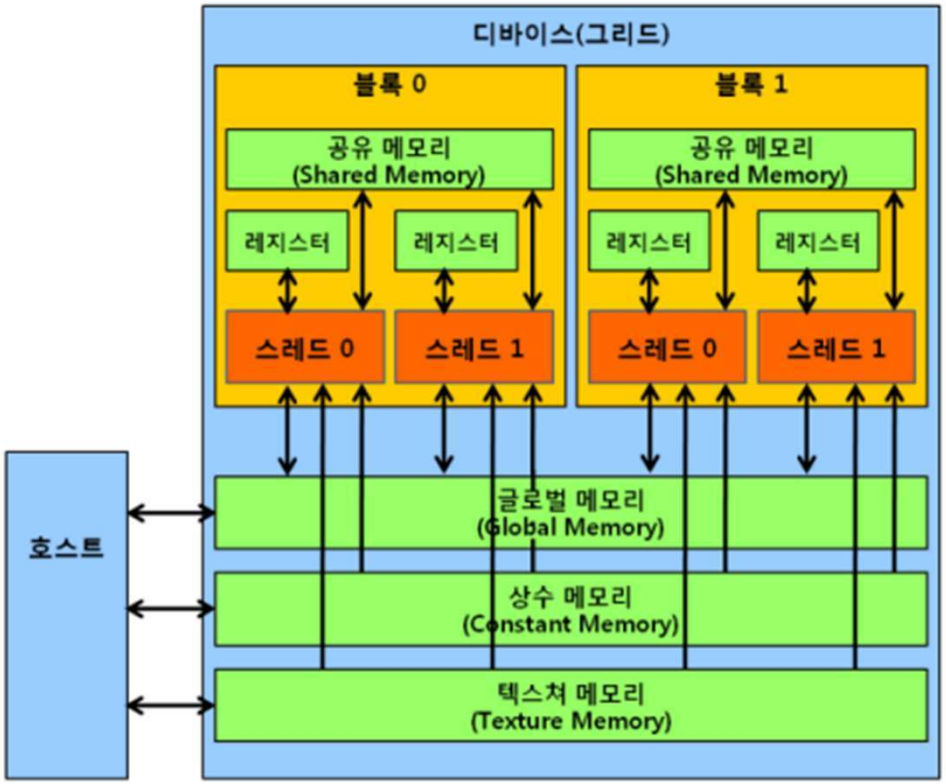


Fig. 1 Structure of PIPO block cipher[1]

2. GPU상에서 경량 블록암호 PIPO의 최적 구현

CUDA memory Architecture

글로벌 메모리 : 400~600 cycle
로컬 메모리 : 400~600 cycle
공유 메모리 : 5 cycle
레지스터 : ~1cycle



2. GPU상에서 경량 블록암호 PIPO의 최적 구현

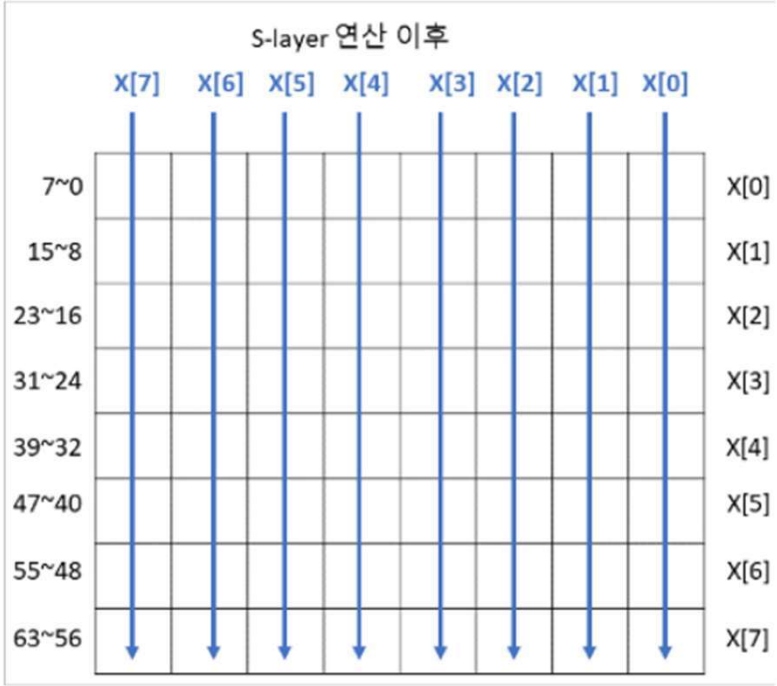


Fig. 2 Block structure after S-layer

2. GPU상에서 경량 블록암호 PIPO의 최적 구현

32블록 병렬 수행 비트슬라이싱 기법 적용

```
__device__ void pipo_sBoxLayer(uint32_t* X) {
register uint32_t T0, T1, T2;
PIPO_SBOX(X[0], X[8], X[16], X[24], X[32], X[40], X[48], X[56]);
PIPO_SBOX(X[1], X[9], X[17], X[25], X[33], X[41], X[49], X[57]);
PIPO_SBOX(X[2], X[10], X[18], X[26], X[34], X[42], X[50], X[58]);
PIPO_SBOX(X[3], X[11], X[19], X[27], X[35], X[43], X[51], X[59]);
PIPO_SBOX(X[4], X[12], X[20], X[28], X[36], X[44], X[52], X[60]);
PIPO_SBOX(X[5], X[13], X[21], X[29], X[37], X[45], X[53], X[61]);
PIPO_SBOX(X[6], X[14], X[22], X[30], X[38], X[46], X[54], X[62]);
PIPO_SBOX(X[7], X[15], X[23], X[31], X[39], X[47], X[55], X[63]);
}
```

Fig. 3 Implementation of s-Layer with bit slicing on GPU

2. GPU상에서 경량 블록암호 PIPO의 최적 구현

32블록 병렬 수행 비트슬라이싱 기법 적용

평문에 1회의 Packing만 필요하므로 병렬 연산 전에 이를 처리한다.

비트 교환은 레지스터 간에 값 교환으로 연산할 수 있다.
: R-layer의 8비트 로테이션은 레지스터 간 값 교환으로 처리된다.

S-layer에서 비트 교환을 하지 않고 R-layer에서 수행하면 연산량을 줄일 수 있다.
(S-layer 64연산 감소, R-layer 24연산 증가)

2. GPU상에서 경량 블록암호 PIPO의 최적 구현

```
__device__ void pipo_sBoxLayer(uint32_t* X) {
register uint32_t T0, T1, T2;
PIPO_SBOX(X[0], X[8], X[16], X[24], X[32], X[40], X[48], X[56]);
PIPO_SBOX(X[1], X[9], X[17], X[25], X[33], X[41], X[49], X[57]);
PIPO_SBOX(X[2], X[10], X[18], X[26], X[34], X[42], X[50], X[58]);
PIPO_SBOX(X[3], X[11], X[19], X[27], X[35], X[43], X[51], X[59]);
PIPO_SBOX(X[4], X[12], X[20], X[28], X[36], X[44], X[52], X[60]);
PIPO_SBOX(X[5], X[13], X[21], X[29], X[37], X[45], X[53], X[61]);
PIPO_SBOX(X[6], X[14], X[22], X[30], X[38], X[46], X[54], X[62]);
PIPO_SBOX(X[7], X[15], X[23], X[31], X[39], X[47], X[55], X[63]);
}
```

Fig. 3 Implementation of s-Layer with bit slicing on GPU

```
#define PIPO_SBOX(X0, X1, X2, X3, X4, X5, X6, X7)\
        X5 ^= (X7 & X6); X4 ^= (X3 & X5); \
        X7 ^= X4; X6 ^= X3; \
        X3 ^= (X4 | X5); X5 ^= X7; \
        X4 ^= (X5 & X6); X2 ^= X1 & X0; \
        X0 ^= X2 | X1; X1 ^= X2 | X0; \
        X2 = ~X2; X7 ^= X1; \
        X3 ^= X2; X4 ^= X0; \
        X6 ^= (X7 & X5); T0 = X7^X6;\
        X6 ^= (X4 | X3); T1 = X3^X5     \
        X5 ^= (X6 | X4); X2 ^= T0;      \
        T2 = X7;  X1 = X1 ^ X4 ^ (T1 & T0);\
        X0 = X0^T1; \
```

Fig. 4 S-Box implementation of the proposed method