KOREA UNIVERSITY

Cryptographic Algorithm Laboratory

8-비트 AVR 환경에서 블록암호 PRESENT의 최적 구현

신명수, 신한범, 김선엽, 김성겸, 서석충, 홍득조, 성재철, 홍석희

고려대학교, 서울
damper99@korea.ac.kr

CISC-S'23
June, 2023

목 차

1. 서론

1. 서론

- **경량 블록암호의 최적 구현 필요성**
  - IoT 기술의 발전에 따라 데이터 암호화에 대한 중요성이 높아지고 있음.

- **경량 블록암호 PRESENT 구현 연구**
  - CHES 2017에서 PRESENT의 bit permutation 연산을 분해하여 소프트웨어에서 효율적인 구현 방법이 제안되었음.
  - 해당 방법을 8-비트 AVR 환경으로 이식하여 효율적인 구현을 제시함.

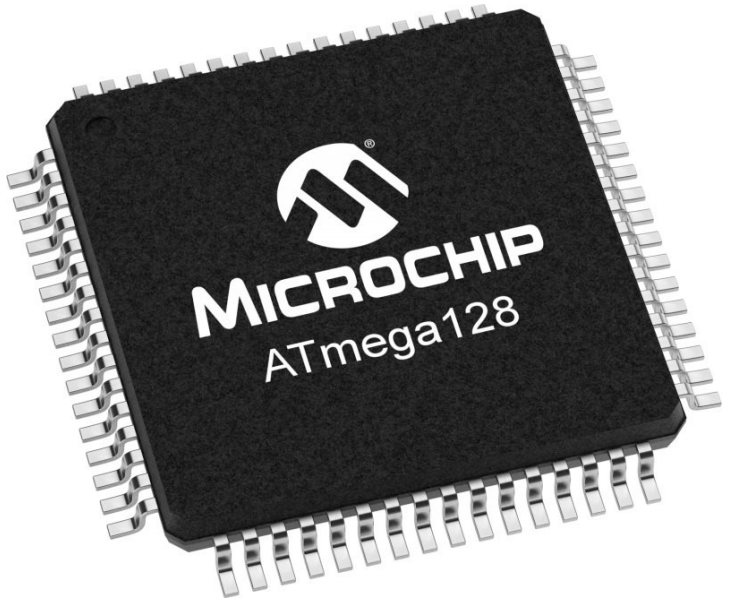
- **본 논문이 제시하는 구현 방법**
  - 16-비트 단위로 구현된 bit permutation 연산을 8-비트 단위로 재구성하여 AVR 환경에서 효율적인 구현을 제시.

1. 서론

● 구현 환경

◦ 다양한 IoT 기기에서 활용되고 있는 AVR Atmega128

- 8-bit word size
- 32개 8-bit general purpose register
- 133개 명령어 셋
- 4KB EEPROM
- 128KB flash memory
- 4KB SRAM



2. 관련 연구

2. 관련 연구

- **CHES 2007 에서 제안된 경량 블록암호 PRESENT**
  - SPN 구조
  - 64-bit 블록 / 80-bit key / 31 rounds
  - 64-bit 블록 / 128-bit key / 31 rounds

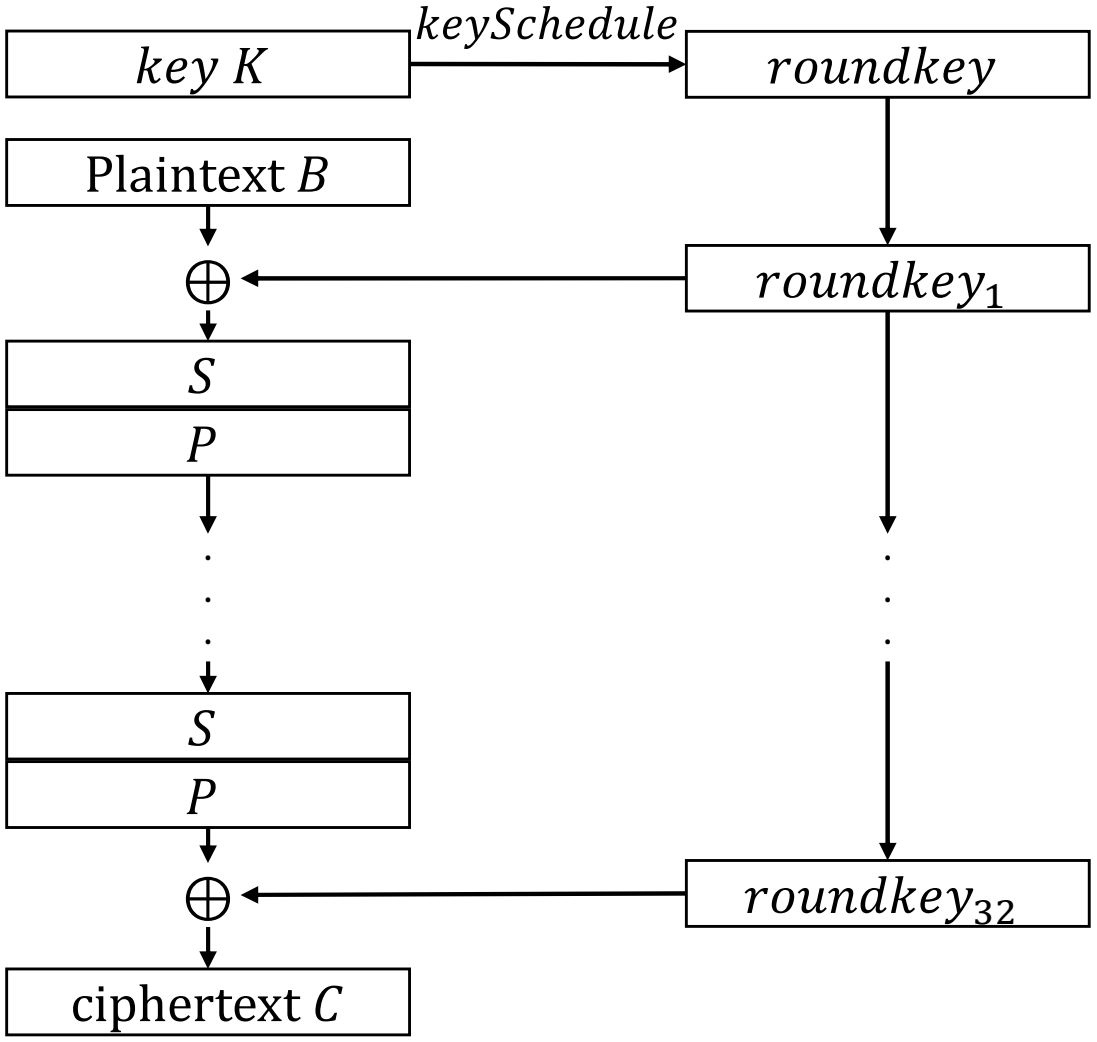


그림 1. PRESENT 암호화 동작 과정

2. 관련 연구

- **PRESENT S-box**
  - 4-bit S-box 를 사용
  - 입/출력 블록들을 슬라이스로 변환하는 packing/unpacking 연산을 추가해 비트슬라이스 방법으로 구현가능

    $(unpack \circ S_{BS} \circ pack = S)$

- **Permutation P**

$$P(i) = \begin{cases} 16i \ mod \ 63, & if \ i \neq 63, \\ 63, & if \ i = 63. \end{cases}$$

i-th bit를 $P(i)$로 이동

2. 관련 연구

- **PRESENT 의 소프트웨어 구현 최적화**
  - 2 Rounds 에서 2회의 P 연산 최적화
    - 효율적인 소프트웨어 구현이 가능한 P_0, P_1 으로 대체하여 연산

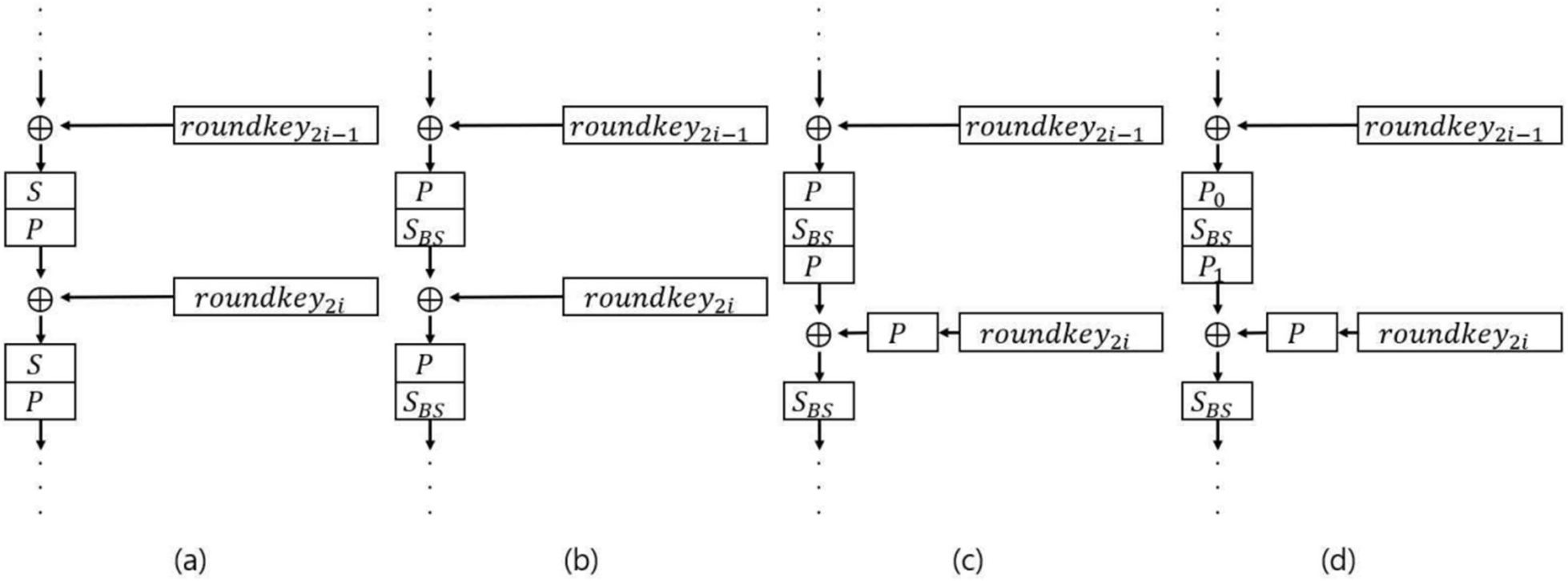


그림 2. 2라운드 PRESENT에 대한 4가지 구현 방법

2. 관련 연구

- **P_0, P_1 을 구현할 때 SWAPMOVE 를 사용해서 구현**
  - SWAPMOVE : M 으로 마스킹 된 B 와 $(M \ll N)$ 으로 마스킹 된 A의 비트 자리를 서로 교환할 때 사용

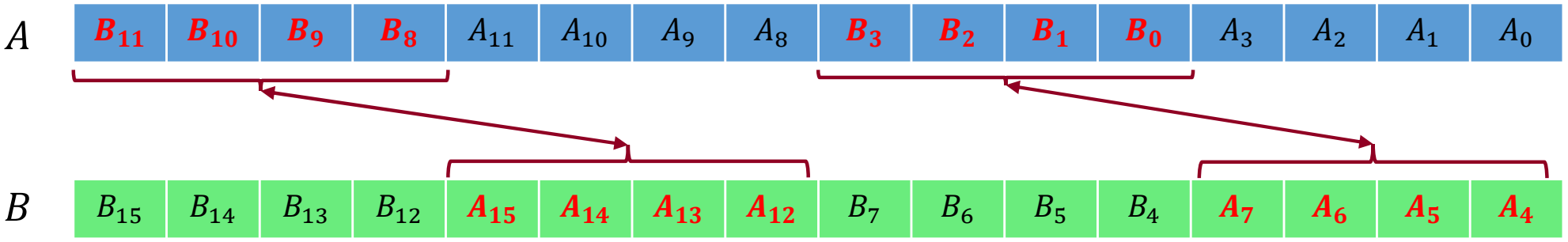
SWAPMOVE(A, B, M, n)

$1: T = \left(B \oplus (A \gg n)\right) \wedge M$

$2: B = B \oplus T$

$3: A = A \oplus (T \ll n)$

Example. **16-bit word** 에서 $n = 4, M = 0x0F0F$ 일 때



2. 관련 연구

- **PRESENT 소프트웨어 최적 구현**
  - 소프트웨어 구현에 적합한 그림 2.(d) 방법으로 SWAPMOVE를 활용하여 16-비트 단위 P_0, P_1 구현
  - X : 64-bit intermediate state (X[3], X[2], X[1], X[0])

```
void P1(uint16_t* X)
{
     SWAPMOVE(X[0], X[1], 0x0F0F, 4);
     SWAPMOVE(X[2], X[3], 0x0F0F, 4);
     SWAPMOVE(X[1], X[3], 0x00FF, 8);
     SWAPMOVE(X[0], X[2], 0x00FF, 8);
}
```

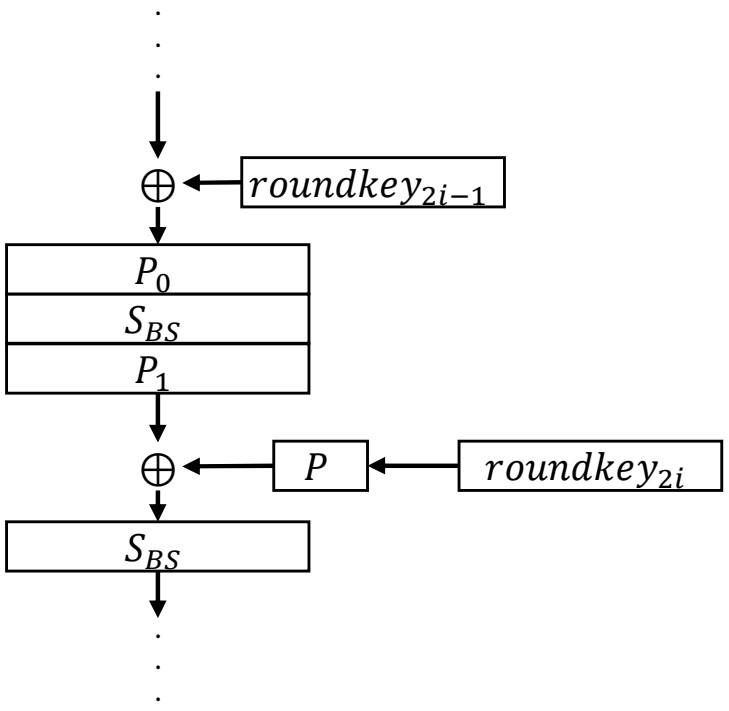


그림 2. (d)

2. 관련 연구

- P_0, P_1 를 16-비트 단위로 구성하여 2라운드 구현 후 이를 반복적으로 수행
  - X : 64-bit intermediate state

    (16-bit state X[3], X[2], X[1], X[0] = 8-bit state X3_1, X3_0, X2_1, X2_0, X1_1, X1_0, X0_1, X1_0)

```
void P1(uint16_t* X)
{
    SWAPMOVE(X[0], X[1], 0x0F0F, 4);
    SWAPMOVE(X[2], X[3], 0x0F0F, 4);
    SWAPMOVE(X[1], X[3], 0x00FF, 8);
    SWAPMOVE(X[0], X[2], 0x00FF, 8);
}
```

25 cycles 소요

```
SWAPMOVE_0x0F0F_4:
//T = ((X0>>4)^X1) & 0x0F0F;
MOVW T0, X1_0
LSR T1
ROR T0
LSR T1
ROR T0
LSR T1
ROR T0
LSR T1
ROR T0

EOR T0, X1_0
EOR T1, X1_1

ANDI T0, 0X0F
ANDI T1, 0X0F
```

```
//X1 = X1 ^ T;
EOR T0, X1_0
EOR T1, X1_1
//X0 = X0 ^ (T<<4);
LSL T0
ROL T1
LSL T0
ROL T1
LSL T0
ROL T1
LSL T0
ROL T1

EOR X0_0, T0
EOR X0_1, T1
```

3. 8-비트 AVR 상에서 P_1 최적 구현

3. 8-비트 AVR 상에서 P_1 최적 구현

- 16-비트 단위로 구현된 P_1을 8-bit 단위로 재구성

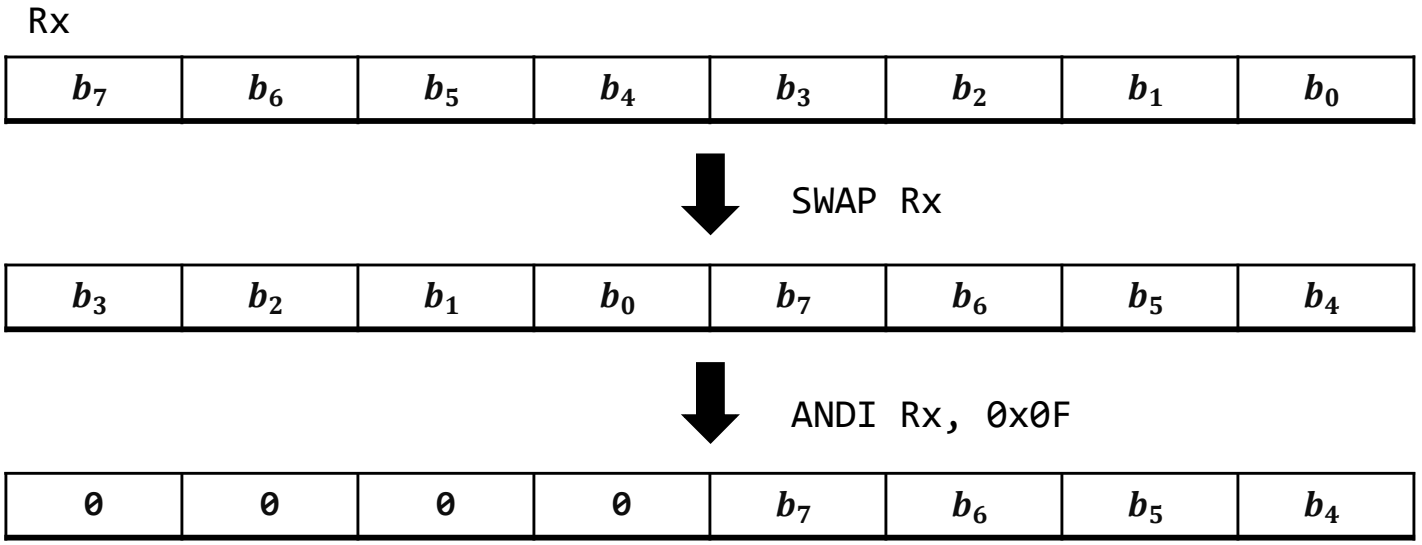
```
void P1(uint16_t* X)
{
    SWAPMOVE(X[0], X[1], 0x0F0F, 4);
    SWAPMOVE(X[2], X[3], 0x0F0F, 4);
    SWAPMOVE(X[1], X[3], 0x00FF, 8);
    SWAPMOVE(X[0], X[2], 0x00FF, 8);
}
```

```
void P1(uint8_t* X)
{
    uint8_t t;

    SWAPMOVE(X[0], X[2], 0x0F, 4);
    SWAPMOVE(X[1], X[3], 0x0F, 4);
    SWAPMOVE(X[4], X[6], 0x0F, 4);
    SWAPMOVE(X[5], X[7], 0x0F, 4);

    //SWAPMOVE(X[3], X[6], 0xFF, 8);
    t = X[6];
    X[6] = X[3];
    X[3] = t;

    //SWAPMOVE(X[1], X[4], 0xFF, 8);
    t = X[4];
    X[4] = X[1];
    X[1] = t;
}
```

3. 8-비트 AVR 상에서 P_1 최적 구현

- **8-비트 AVR 어셈블리 SWAP 명령어를 활용한 logical shift 4**
  - SWAP 명령어 : 8-비트 레지스터에서 상위 4-비트와 하위 4-비트의 자리를 서로 바꾸는 명령어

    SWAP 연산 후 0x0F 와 AND 연산하여 상위 4-비트를 0으로 만든다.



3. 8-비트 AVR 상에서 P_1 최적 구현

- **8-비트 단위 P_1 최적 구현**
  - 소요 cycle 수가 25 cycles 에서 17 cycles로 8 cycles 감소

```
void P1(uint8_t* X)
{
    uint8_t t;

    SWAPMOVE(X[0], X[2], 0x0F, 4);
    SWAPMOVE(X[1], X[3], 0x0F, 4);
    SWAPMOVE(X[4], X[6], 0x0F, 4);
    SWAPMOVE(X[5], X[7], 0x0F, 4);

    //SWAPMOVE(X[3], X[6], 0xFF, 8);
    t = X[6];
    X[6] = X[3];
    X[3] = t;

    //SWAPMOVE(X[1], X[4], 0xFF, 8);
    t = X[4];
    X[4] = X[1];
    X[1] = t;
}
```

```
.macro P1
//SWAPMOVE(X0,X2,0x0F,4) //SWAPMOVE(X1,X3,0x0F,4)
    MOVW T0, X0              SWAP T1
    SWAP T0                  ANDI T1, 0x0F
    ANDI T0, 0x0F            EOR  T1, X3
    EOR  T0, X2              ANDI T1, 0x0F
    ANDI T0, 0x0F            EOR  X3, T1
    EOR  X2, T0              SWAP T1
    SWAP T0                  ANDI T1, 0x0F
    ANDI T0, 0x0F            EOR  X1, T1
    EOR  X0, T0         . . .
```

3. 8-비트 AVR 상에서 P_1 최적 구현

- **8-비트 단위 P_1 최적 구현**

① Shift left 4 연산할 때와 마스킹 값이 같아 0x0F와 AND 연산 소거 가능

② SWAP 연산 후 이미 하위 4-비트가 0 이므로 0x0F와 AND 연산 소거 가능

```
.macro P1
//SWAPMOVE(X0,X2,0x0F,4)
    MOVW T0, X0
    SWAP T0
①   ANDI T0, 0x0F
    EOR  T0, X2
    ANDI T0, 0x0F
    EOR  X2, T0
    SWAP T0
②   ANDI T0, 0x0F
    EOR  X0, T0
```

```
//SWAPMOVE(X1,X3,0x0F,4)
    SWAP T1
①   ANDI T1, 0x0F
    EOR  T1, X3
    ANDI T1, 0x0F
    EOR  X3, T1
    SWAP T1
②   ANDI T1, 0x0F
    EOR  X1, T1
...
```

3. 8-비트 AVR 상에서 P_1 최적 구현

- **8-비트 단위 P_1 최적 구현**
  - 1회의 P_1 연산에서 명령어 수가 60개에서 28개 줄어든 32개로 구현
    전체 라운드에서 448 cycles 감소

```
.macro P1
//SWAPMOVE(X0,X2,0x0F,4)
    MOVW T0, X0
    SWAP T0
    EOR  T0, X2
    ANDI T0, 0x0F
    EOR  X2, T0
    SWAP T0
    EOR  X0, T0

//SWAPMOVE(X1,X3,0x0F,4)
    SWAP T1
    EOR  T1, X3
    ANDI T1, 0x0F
    EOR  X3, T1
    SWAP T1
    EOR  X1, T1

//SWAPMOVE(X4,X6,0x0F,4)
    MOVW T0, X4
    SWAP T0
    EOR  T0, X6
    ANDI T0, 0x0F
    EOR  X6, T0
    SWAP T0
    EOR  X4, T0

//SWAPMOVE(X5,X7,0x0F,4)
    SWAP T1
    EOR  T1, X7
    ANDI T1, 0x0F
    EOR  X7, T1
    SWAP T1
    EOR  X5, T1

//SWAPMOVE(X3,X6,0xFF,0)
    MOV T0, X3
    MOV X3, X6
    MOV X3, T0

//SWAPMOVE(X1,X4,0xFF,0)
    MOV T0, T1
    MOV X1, X4
    MOV X1, T0
```

4. 실험결과

4. 실험결과 및 결론

- **8-bit AVR 환경에서 PRESENT 64-128 1 block 암호화**
  - 키스케줄과 일부 라운드키에 P연산은 미리 계산한 것으로 가정하고 암호화 과정만 성능 평가
  - 환경 : microchip studio 7.0, ATmega128 emulator
  - 최적화 옵션 : -O3

Method	Code size (bytes)	RAM (bytes)	cycles per bytes
PRESENT ref[3]	956	282	504.25
PRESENT[this work]	820	276	445
	약 16.5% 감소		약 13.3% 성능 향상

현재까지의 8-bit AVR 환경에서 PRESENT 구현물 중 가장 뛰어난 성능을 보여준다.

5. 결론

5. 결론

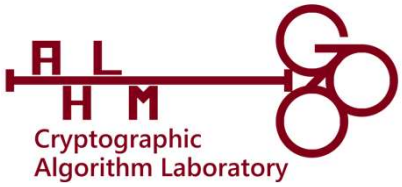
- **경량 블록암호 PRESENT에 대한 8- 비트 AVR 환경에서 최적 구현을 제시**
  - 16-비트 단위로 구현된 P_1을 8-비트 단위 구현으로 최적화
  - 기존 최적 구현물보다 16.5% 적은 코드 길이와 13.3% 빠른 연산 속도를가진 구현물을 제시

- **추후 연구**
  - PRESENT의 On-the-fly 구현에서 고속화하는 방법에 대한 연구

Q&A

Thanks

KOREA UNIVERSITY



참고자료

- [1]. Bogdanov, A. et al. (2007). PRESENT: An Ultra-Lightweight Block Cipher. In: Paillier, P., Verbauwhede, I. (eds) Cryptographic Hardware and Embedded Systems - CHES 2007
- [2]. Reis, T.B.S., Aranha, D.F., López, J. (2017). PRESENT Runs Fast. In: Fischer, W., Homma, N. (eds) Cryptographic Hardware and Embedded Systems – CHES 2017.
- [3]. Kwon, H.; Kim, Y.B.; Seo, S.C.; Seo, H. High-Speed Implementation of PRESENT on AVR Microcontroller. Mathematics 2021, 9, 374.
- [4]. Biham, Eli. "A fast new DES implementation in software." International Workshop on Fast Software Encryption. Springer, Berlin, Heidelberg, 1997.
- [5]. May, L., Penna, L., Clark, A.J.: An Implementation of Bitsliced DES on the Pentium MMXTM Processor. In Dawson, E., Clark, A.J., Boyd, C., eds.: Information Security and Privacy, 5th Australasian Conference, ACISP 2000, Brisbane, Australia, July 10-12, 2000, Proceedings. Volume 1841 of Lecture Notes in Computer Science., Springer (2000) 112–122

2.2 PRESENT 소프트웨어 최적 구현 방법

- **【성질 1】P는 packing 구조를 가진다.**
  - packing 연산과 같은 구조를 가지므로 packing/unpacking 연산을 P/P^{-1} 으로 변경하여 사용할 수 있다.
    - $unpack \circ S_{BS} \circ pack = P^{-1} \circ \mathrm{S}_{BS} \circ P = S$

$$P(B) = \begin{pmatrix} 00 & 04 & 08 & 12 & 16 & 20 & 24 & 28 & 32 & 36 & 40 & 44 & 48 & 52 & 56 & 60 \\ 01 & 05 & 09 & 13 & 17 & 21 & 25 & 29 & 33 & 37 & 41 & 45 & 49 & 53 & 57 & 61 \\ 02 & 06 & 10 & 14 & 18 & 22 & 26 & 30 & 34 & 38 & 42 & 46 & 50 & 54 & 58 & 62 \\ 03 & 07 & 11 & 15 & 19 & 23 & 27 & 31 & 35 & 39 & 43 & 47 & 51 & 55 & 59 & 63 \end{pmatrix}$$

2.2 PRESENT 소프트웨어 최적 구현 방법

- **【조건 1】 P_0는 packing 구조를 가진다.**
  - packing 연산과 같은 구조를 가지므로 packing/unpacking 연산을 P_0/P_0^{-1} 으로 변경하여 사용할 수 있다.
    - $\mathrm{u}pack \circ S_{BS} \circ pack = P_0^{-1} \circ \mathrm{S}_{BS} \circ P_0 = S$

$$P_0(B) = \begin{pmatrix} 00 & 16 & 32 & 48 & 04 & 20 & 36 & 52 & 08 & 24 & 40 & 56 & 12 & 28 & 44 & 60 \\ 01 & 17 & 33 & 49 & 05 & 21 & 37 & 53 & 09 & 25 & 41 & 57 & 13 & 29 & 45 & 61 \\ 02 & 18 & 34 & 50 & 06 & 22 & 38 & 54 & 10 & 26 & 42 & 58 & 14 & 30 & 46 & 62 \\ 03 & 19 & 35 & 51 & 07 & 23 & 39 & 55 & 11 & 27 & 43 & 59 & 15 & 31 & 47 & 63 \end{pmatrix}$$

2.2 PRESENT 소프트웨어 최적 구현 방법

- 【조건 2】 $P_1 \circ P_0 = P^2$를 만족한다.

$$P^2(B) = \begin{pmatrix} 00 & 16 & 32 & 48 & 01 & 17 & 33 & 49 & 02 & 18 & 34 & 50 & 03 & 19 & 35 & 51 \\ 04 & 20 & 36 & 52 & 05 & 21 & 37 & 53 & 06 & 22 & 38 & 54 & 07 & 23 & 39 & 55 \\ 08 & 24 & 40 & 56 & 09 & 25 & 41 & 57 & 10 & 26 & 42 & 58 & 11 & 27 & 43 & 59 \\ 12 & 28 & 44 & 60 & 13 & 29 & 45 & 61 & 14 & 30 & 46 & 62 & 62 & 31 & 47 & 63 \end{pmatrix}$$

$$P_1 \circ P_0(B) = \begin{pmatrix} 00 & 16 & 32 & 48 & 01 & 17 & 33 & 49 & 02 & 18 & 34 & 50 & 03 & 19 & 35 & 51 \\ 04 & 20 & 36 & 52 & 05 & 21 & 37 & 53 & 06 & 22 & 38 & 54 & 07 & 23 & 39 & 55 \\ 08 & 24 & 40 & 56 & 09 & 25 & 41 & 57 & 10 & 26 & 42 & 58 & 11 & 27 & 43 & 59 \\ 12 & 28 & 44 & 60 & 13 & 29 & 45 & 61 & 14 & 30 & 46 & 62 & 62 & 31 & 47 & 63 \end{pmatrix}$$

2.2 PRESENT 소프트웨어 최적 구현 방법

- **【성질 1】의 양변에 P연산을 하면 $S_{BS} \circ P = P \circ S$ 이다.**
  - $P^{-1} \circ S_{BS} \circ P = S$ 의 양변에 P 연산 → $P \circ P^{-1} \circ S_{BS} \circ P = S_{BS} \circ P = P \circ S$

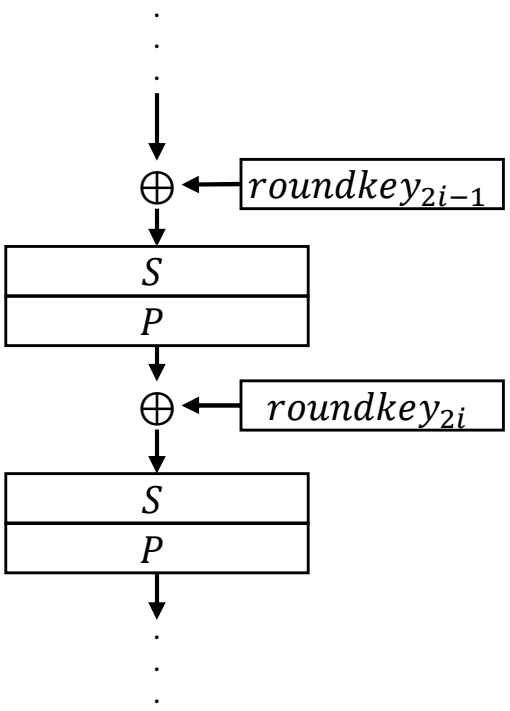


그림 2. (a)

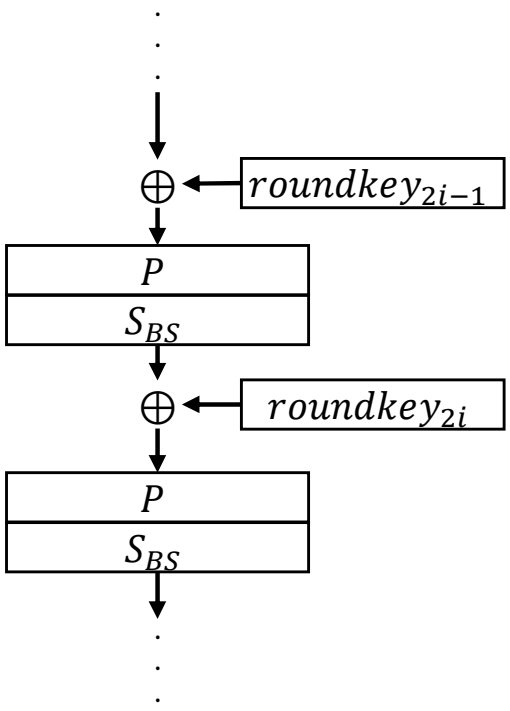


그림 2. (b)

2.2 PRESENT 소프트웨어 최적 구현 방법

- **addRoundKey 연산과 P 연산 재배치**
  - addRoundKey 연산은 라운드키를 그대로 XOR 하는 연산이기 때문에 라운드키에 P연산을 수행하면 순서를 바꾸어도 결과가 같다.

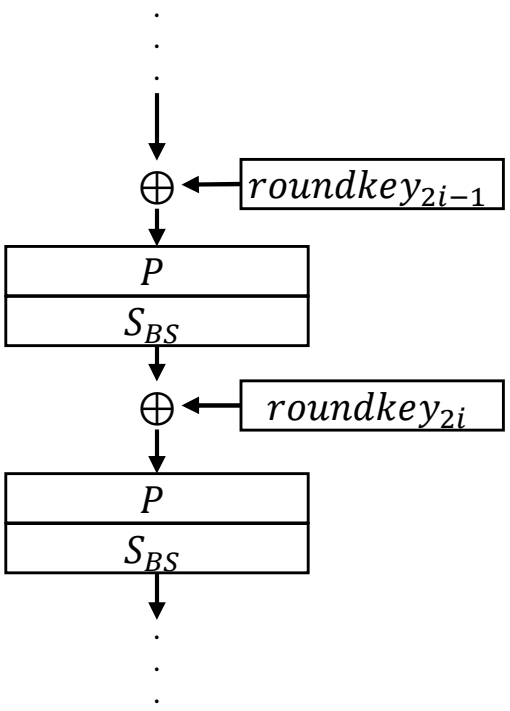


그림 2. (b)

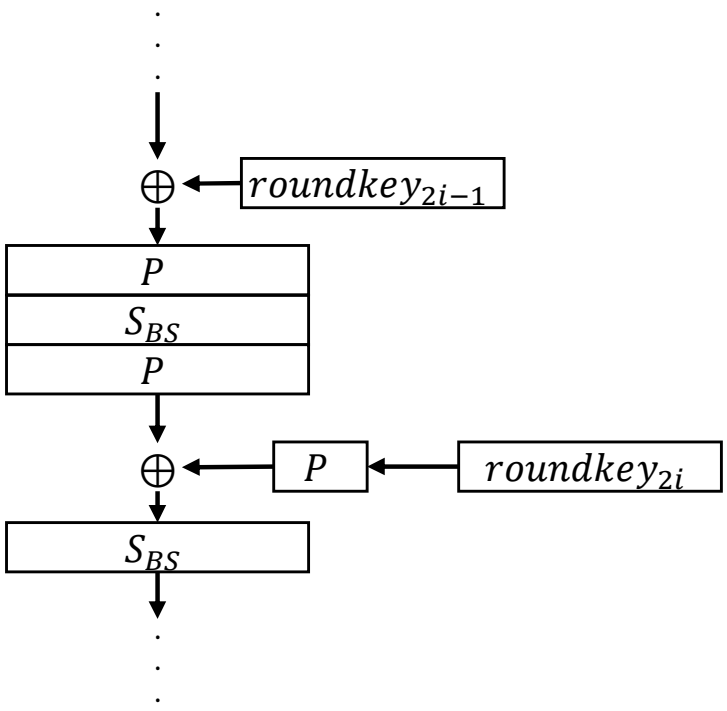


그림 2. (c)

2.2 PRESENT 소프트웨어 최적 구현 방법

- **소프트웨어 구현에 효율적인 P_0, P_1 으로 분할**
  - 【성질 1】과 【조건 1】에 의해 $P^{-1} \circ S_{BS} \circ P = P_0^{-1} \circ S_{BS} \circ P_0 = S$ 가 성립하므로, 양 변에 P^2 연산을 수행하면 【조건 2】에 의해 다음이 성립한다.
    - $P \circ S_{BS} \circ P = P^2 \circ P_0^{-1} \circ S_{BS} \circ P_0 = P_1 \circ S_{BS} \circ P_0$

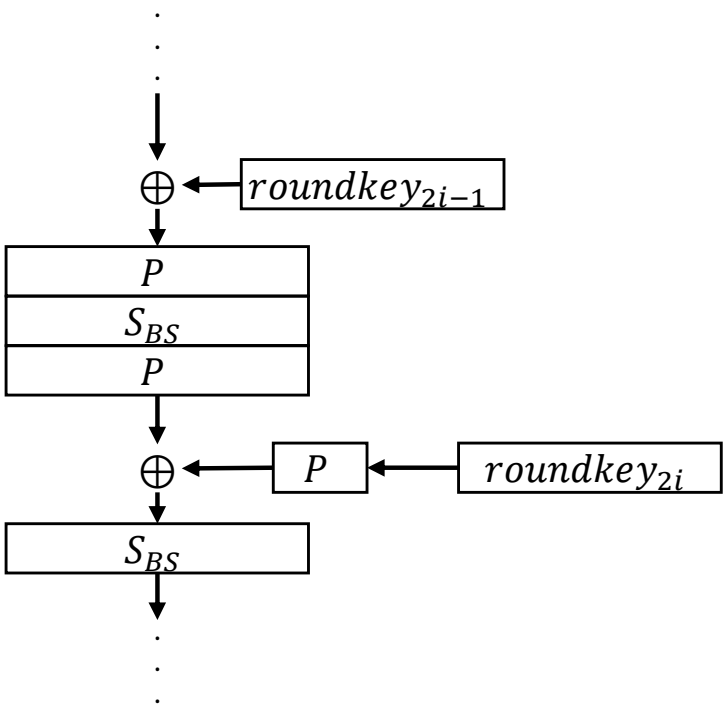


그림 2. (c)

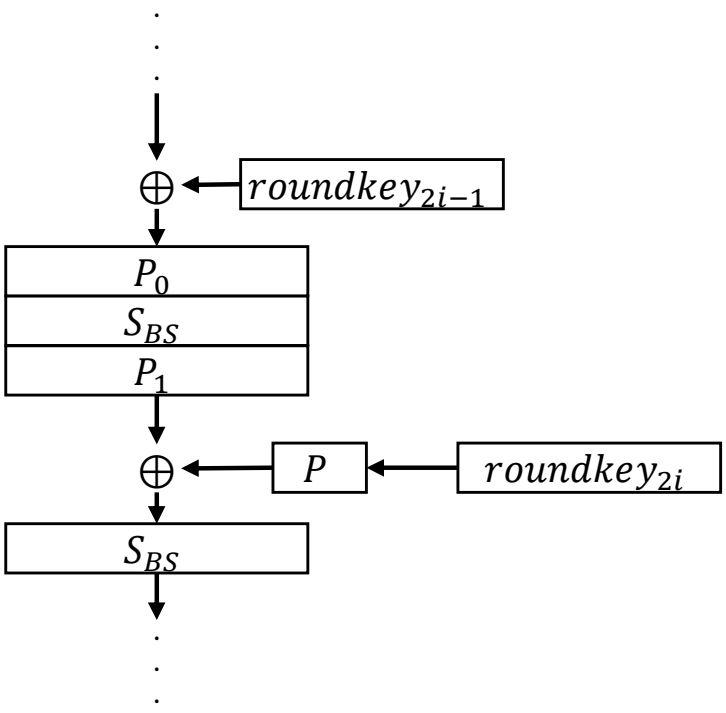


그림 2. (d)