8-비트 AVR 환경에서 블록암호 PRESENT의 최적 구현*

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

*,******고려대학교 (대학원생, 교수), **삼성전자 (연구원), ***국민대학교 (교수),
****전북대학교 (교수), *****서울시립대학교 (교수)

Optimized Implementation of PRESENT on 8-bit AVR

Myoungsu Shin*, Hanbeom Shin*, Sunyeop Kim*, Seonggyeom Kim**,
Seog Chung Seo***, Deukjo Hong****, Jaechul Sung*****, Seokhie Hong******

*,******Korea University (Graduate student, Professor),
**Samsung Electronics (Researcher), ***Kookmin University (Professor),
****Jeonbuk National University (Professor), *****University of Seoul (Professor)

요 약

SPN 경량 블록암호 PRESENT는 제한된 리소스 환경에서 암/복호화 작업이 효율적으로 수행되도록 설계되었다. 또한 다양한 환경에서 효율적으로 구현할 수 있도록 지속적으로 연구되어왔다. 본 논문에서는 8-비트 AVR 환경에서 PRESENT의 bit permutation 연산을 최적화하는 방안을 제시한다. 16-비트 단위로 구현된 bit permutation 연산을 그대로 이식하는 방법 대신 8-비트 단위로 재구성한 뒤 이를 8-비트 AVR 어셈블리 명령어를 활용하여 코드 길이와 속도 측면에서 최적화된 구현물을 제시한다. 본 논문의 제안 구현은 1바이트당 445 cycles의 성능을 가지며 이전 최적 구현물보다 코드 길이는 16.5%, 연산 속도는 13.3% 개선되었다. 이는 8-비트 AVR 상에서 구현된 PRESENT의 현재까지 구현물 중 가장 적은 cpb를 보여준다.

I. 서론

IoT 기술의 발전으로 우리의 일상생활에 많은 스마트 기기들이 도입되었다. 이에 따라 IoT 디바이스는 많은 양의 데이터를 처리 및 통신해야한다. 이러한 데이터에는 개인정보나 기밀정보를 포함할 수 있으므로 암호화가 필수적이다. 하지만 IoT 디바이스는 제한된 리소스를 가지고 있으며 저전력 환경에서 작동되기 때문에 암/복호화를 효율적으로 수행하기 위해서는 경량 블록암호의 최적 구현이 필수적이다.

경량 블록암호 PRESENT[1]는 효율적인 하드웨어 구현을 목표로 설계되었지만 다른 디바이스 또는 서버와 통신하기 위해서는 소프트웨어에서 효율적인 구현이 필수적이다. CHES 2017에서 PRESENT의 bit permutation 연산을 분해하여 소프트웨어에서 효율적인 구현 방법[2]이 제안되었다. [3]에서는 [2]에서 제안된 방법을 저사양 프로세서인 8-비트 AVR 환경으로 이식하여 1블록 암호화에서 효율적인 구현을 제시하였다.

본 논문에서는 8-비트 AVR 환경에서 기존에 제안된 구현 결과물보다 코드 길이와 속도 측면에서 최적화된 구현물을 제시한다. [3]에서는 16-비트 단위로 구현된 bit permutation 연산을 그대로 이식하였지만 본 논문에서는 8-비트 단위로 연산을 재구성한 뒤 이를 8-비트 AVR 어셈블리 명령어를 활용하여 최적화하였다.

논문의 구성은 다음과 같다. Ⅱ장에서는 경량 블록암호 PRESENT와 소프트웨어 최적화 구현에 관한 기존 연구 결과를 소개한다. Ⅲ장에서는 8-비트 AVR 환경에서의 PRESENT에 대한 새로운 최적화 구현을 제안한다. Ⅳ장에서는 제안 구현 결과의 성능을 평가하고 Ⅴ장에서 본 논문에 대한 결론을 내린다.

II. 관련 연구

2.1 경량 블록암호 PRESENT

CHES 2007에서 제안된 PRESENT[1]는 64-비트 블록 크기, 80-비트 또는 128-비트 키를 지원하는 경량 블록암호이다. SPN(Substitution Permutation Network) 구조로 구성되어 있으며 키 길이와 상관없이 31라운드를 사용하지만, 키스케줄은 키 길이에 따라 다르게 구성되어 있다. 그림 1은 PRESENT의 암호화 동작 과정을 나타낸 것이다. PRESENT는 암호화 중간 상태와 라운드키가 XOR 연산되는 addRoundKey 연산, 비선형 연산인 S(substitution 연산), 선형 연산인 P(permutation 연산)로 이루어져 있고, 키스케줄에 의해 64-비트 라운드키 32개가 생성된다. S연산은 4-비트

* 본 연구는 고려대 암호기술 특화연구센터(UD210027XD)를 통한 방위사업청과 국방과학연구소의 연구비 지원으로 수행되었습니다.

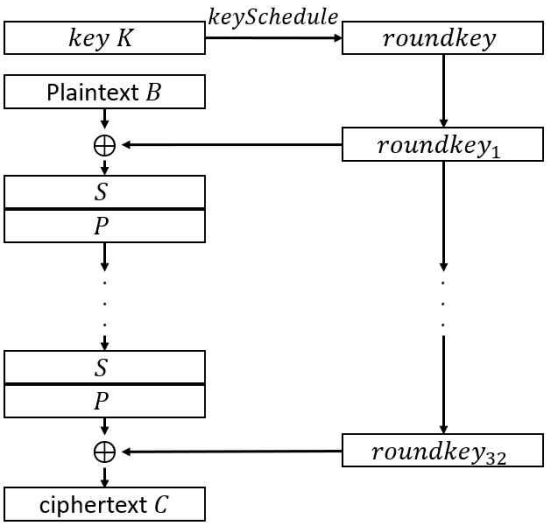

그림 1. PRESENT 암호화 동작 과정

S-box를 사용하고, LUT(Look-Up Table) 방식으로 4-비트씩 테이블을 참조하거나 입/출력 블록들을 슬라이스들로 변환하는 packing/unpacking 연산을 추가해 비트 슬라이스[4] 방법($unpack \circ S_{BS} \circ pack = S$)으로 구현할 수 있다. P연산은 비트 단위 자리 이동 연산인 bit permutation 연산으로, 수식 (1)에 표현된 $P()$를 사용하여 i번째 비트는 $P(i)$번째 자리로 이동한다.

$$P(i) = \begin{cases} 16i \bmod 63 & \text{if } i \neq 63, \\ 63 & \text{if } i = 63. \end{cases} \quad (1)$$

2.2 PRESENT 소프트웨어 최적 구현 방법

[2]는 2라운드 PRESENT에서 P연산을 새로운 분해 방법을 사용해 소프트웨어에서 효율적인 구현 방법이 제시하였다. P를 소프트웨어에서 효율적으로 구현가능한 두 개의 bit permutation 연산 P_0, P_1으로 분해하여 연속하는 두 라운드에서 사용한다. 이때 P연산이 갖는 다음 **【성질 1】**을 이용하며, P_0, P_1은 다음의 **【조건 1, 2】**를 만족해야 한다.

【성질 1】 P는 packing 구조를 가진다.
즉, $P^{-1} \circ S_{BS} \circ P = S$ 를 만족한다.

【조건 1】 P_0가 packing 구조를 가진다.
즉, $P_0^{-1} \circ S_{BS} \circ P_0 = S$ 를 만족한다.

【조건 2】 $P_1 \circ P_0 = P^2$ 를 만족한다.

그림 2는 PRESENT의 연속하는 2라운드($1 \le i \le 15$)에 대한 4가지 동등한 연산 방법을 나타낸 것이다. 위 성질 및 조건을 통해 4가지 방법의 연산 결과가 같음을 다음과 같이 증명할 수 있다. 먼저 **【성질 1】**의 양변에 P 연산을 하면 $S_{BS} \circ P = P \circ S$ 이다. 따라서 그림 2의 (a)와 (b)는 연산 결과가 같다. addRoundKey 연산은 라운드키와 XOR 연산만 하는 것이기 때문에 해당 라운드 키에 P연산을 하면 P와 연산 순서를 바꾸어도 연산 결과에 변화가 없다. 그러므로 그림 2에서 (b)와 (c)의 연산 결과는 같다. 마지막으로 **【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 양변에 P^2 연산을 수행하면 **【조건 2】**에 의해 $P \circ S_{BS} \circ P = P^2 \circ P_0^{-1} \circ S_{BS} \circ P_0 = P_1 \circ S_{BS} \circ P_0$가 성립한다. 따라서 그림 2에서 (c)와 (d)의 연산 결과가 같다.

[2]에서 최종적으로 제안한 방법인 (d)는 일부 라운드키에 추가로 수행해야 하는 P연산에 대한 부하가 발생한다. 그러나 일반적으로 한 번의 키스케줄로 생성한 라운드키들로 여러 블록들을 암/복호화 하기 때문에 처음 키스케줄할 때 P연산을 하여 라운드키를 저장하면 부하 영향이 적다.

표 1은 16-비트 워드 단위 P_1에 대한 C 코드이며 SWAPMOVE[5] 연산 4회로 구현되었다. SWAPMOVE 연산을 사용하여 bit permutation 연산을 구성하면 소프트웨어에서 효율적으로 구현할 수 있다. Algorithm 1은 SWAPMOVE 연산 알고리즘으로, M 으로 마스킹된 B의 비트와 $(M \ll N)$으로 마스킹된 A 의 비트의 자리를 서로 교환할 때 사용된다.

[Algorithm 1] SWAPMOVE [5]

Input : swap object A, B, mask M, shift size N
Output : A, B after bit swap

1. $temp = ((A \gg N) \oplus B) \wedge M$
2. $B = B \oplus temp$
3. $A = A \oplus (temp \ll N)$

표 1 PRESENT 16-비트 워드 단위 P_1 C 코드[2]

PRESENT permutation 16-bit P_1 C code[2]

-X :64-bit intermediate state (X[3],X[2],X[1],X[0])

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

2.3 8-비트 AVR 마이크로컨트롤러

다양한 IoT 기기에서 사용되고 있는 마이크로컨트롤러 AVR ATmega128은 기본 워드 크기가 8-비트이며, 기본적으로 8-비트 단위 연산을 수행한다. RISC 아키텍처로 32개의 8-비트 일반 목적 레지스터를 탑재하고 있으며 133개의 명령어 셋을 가지고 있다. 동작 주파수는 16MHz이고 4KB EEPROM, 128KB 플래시 메모리, 4KB SRAM을 탑재하고 있다.

2.4 8-비트 AVR 상에서 PRESENT의 기존 최적 구현

[3]에서는 [2]에서 16-비트 단위로 제시한 방법을 8-bit AVR 환경에서 2라운드 구현 후 이를 반복적으로 수행하는 방식으로 구현하였다. 표 2는 [3]에서 제안한 8-비트 AVR에서 P_1구현의 일부이며, 표 1의 첫 번째 줄을 AVR 어셈블리로 구현한 것이다. 16-비트 단위의 logical shift 연산은 레지스터 2개로 8-비트 AVR 명령

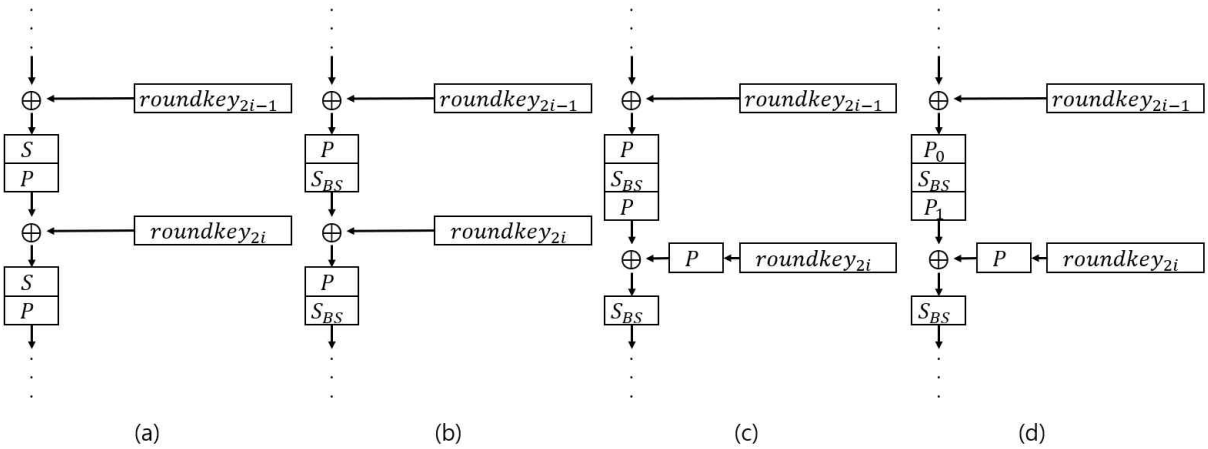

그림 2. 2라운드 PRESENT에 대한 4가지 구현 방법

어인 'LSR', 'ROR', 'LSL', 'ROL'을 사용해 구현하였고, XOR 연산에는 'EOR', 상수와 AND 연산에는 'ANDI'를 사용하였다. 여기서 사용된 명령어들의 비용은 모두 1 cycle이다.

표 2에서 shift right(left) 4 연산은 2번째 줄부터 9번째 줄(16번째 줄부터 23번째 줄)까지이므로 8개 명령어 즉, 8 cycles 씩 필요하다. 표 2와 같이 P_1을 구현하면 비용이 1 cycle인 명령어 60개, 전체 라운드에서 필요한 P_1연산은 16회이므로 총 960 cycles 가 필요하다.

표 2 16-비트 SWAPMOVE(X[0], X[1], 0x0F0F, 4) AVR 어셈블리 코드[3]

16-bit SWAPMOVE(X[0],X[1],0x0F0F,4) AVR ASM code[3]

-X0_0,X0_1 :8-bit register of 16-bit X[0](low,high)

-X1_0,X1_1 :8-bit register of 16-bit X[1](low,high)

-T0,T1 :8-bit temporary variable

```
SWAPMOVE_0x0F0F_4:
//t=((X0>>4)^X1)& 0x0F0F
 1.   MOVW T0, X0_0
 2.   LSR T1
 3.   ROR T0
 4.   LSR T1
 5.   ROR T0
 6.   LSR T1
 7.   ROR T0
 8.   LSR T1
 9.   ROR T0
10.   EOR T0, X1_0
11.   EOR T1, X1_1
12.   ANDI T0, 0X0F
13.   ANDI T1, 0X0F
// X1=X1^t;
14.   EOR X1_0, T0
15.   EOR X1_0, T1
// X0=X0^(t<<4);
16.   LSL T0
17.   ROL T1
18.   LSL T0
19.   ROL T1
20.   LSL T0
21.   ROL T1
22.   LSL T0
23.   ROL T1
24.   EOR X0_0, T0
25.   EOR X0_1, T1
```

III. 8-비트 AVR 상에서 P_1 최적 구현

본 장에서는 8-비트 AVR 어셈블리 명령어인 SWAP과 이를 활용한 logical shift 연산을 소개하고 16-비트 단위로 구현된 P_1을 8-비트 단위로 재구성한 최적 구현을 제안한다.*

* 구현코드 https://github.com/dampers/PRESENT_AVR

3.1 SWAP 명령어를 활용한 logical shift 4

'SWAP'은 8-비트 레지스터 내에서 상위 4-비트와 하위 4-비트의 위치를 서로 바꿔주는 AVR 어셈블리 명령어이다. 'SWAP' 명령어를 사용하면 1 cycle 만에 8-비트 레지스터에서 4-비트 rotation right(left) 한 것과 결과가 같다. 그림 3은 8-비트 AVR 어셈블리에서 레지스터 예시 Rx에 대한 shift right 4 연산 과정을 나타낸 것이다. 'SWAP'으로 4-비트 rotation 연산한 후, 0x0F와 AND 연산하여 상위 4-비트를 0으로 만든다. 마찬가지로 shift left 4 연산은 'SWAP' 연산한 후 0xF0과 AND 연산하여 하위 4-비트를 0으로 만든다.

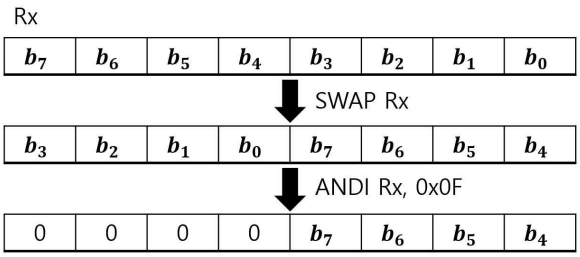

그림 3. 8-비트 AVR 레지스터에서 shift right 4 연산 과정

3.2 8-비트 단위 P_1 최적 구현

표 1의 16-비트 단위 P_1을 8-비트 AVR에 그대로 이식하면 표 2와 같이 logical shift 4 하는 과정에서 비효율이 발생한다. 본 논문에서는 16-비트 단위로 구현된 P_1을 8-비트 단위로 재구성하여 8-비트 AVR 상에서 P_1의 최적 구현을 제안한다. 16-비트 단위 P_1을 8-비트 단위로 재구성하여 C 코드로 나타내면 표 3과 같다. 표 1의 1, 2번 줄이 표 3의 1, 2, 3, 4번 줄과 대응된다. 2회의 SWAPMOVE 연산이 4회로 증가했지만 이를 8-비트 어셈블리로 최적 구현하면 연산 수가 감소한다. 표 4는 8-비트 AVR 어셈블리로 최적 구현한 코드이다. 3.1절에서 소개한 방법을 사용하면 8-비트 단위에서 shift right(left) 4 연산이 2 cycles 만에 가능하다. 그리고 shift right 4 연산할 때 0x0F와 AND 연산하는 부분이 마스킹값과 같아 소거할 수 있고, shift left 4 연산에서는

이미 하위 4-비트가 0으로 되어있기 때문에 AND연산을 소거할 수 있다. 따라서 shift right(left) 4 연산을 'SWAP' 명령어 1회만으로 가능하게 된다. 표 1의 3, 4번째 줄 부분에서는 shift 크기가 8이고 0x00FF를 마스킹한다. 이를 8-비트 P_1으로 재구성하면 shift 크기가 0이고 0xFF를 마스킹하게 된다. 이는 8-비트 단위 레지스터끼리의 교환으로 변환할 수 있다. 결과적으로 1회의 P_1연산에서 명령어 수가 60개에서 28개 줄어든 32개로 구현하였으며, 전체 라운드에서 448 cycles가 감소하였다.

표 3 PRESENT 8-비트 단위 P_1 C 코드

PRESENT permutation 8-bit P_1 C code

-X :64-bit intermediate state(X[7],X[6],X[5],X[4],X[3],X[2],X[1],X[0])

-t :8-bit temporary variable

```
    void P1(uint8_t* X)
    {
 1.     SWAPMOVE(X[5], X[7], 0x0F, 4);
 2.     SWAPMOVE(X[4], X[6], 0x0F, 4);
 3.     SWAPMOVE(X[1], X[3], 0x0F, 4);
 4.     SWAPMOVE(X[0], X[2], 0x0F, 4);

        //SWAPMOVE(X[3], X[6], 0xFF, 0);
 5.     t = X[6];
 6.     X[6] = X[3];
 7.     X[3] = t;
        //SWAPMOVE(X[1], X[4], 0xFF, 0);
 8.     t = X[4];
 9.     X[4] = X[1];
10.     X[1] = t;
    }
```

표 4 PRESENT의 8-비트 단위P_1 AVR 어셈블리 코드

PRESENT permutation 8-bit P_1 AVR ASM proposal code

-X0,X1,X2,X3,X4,X5,X6,X7: 8-bit intermediate state

-T0,T1 :8-bit temporary variable

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

//SWAPMOVE(X1,X3,0x0F,4)
 8.   SWAP T1
 9.   EOR  T1, X3
10.   ANDI T1, 0x0F
11.   EOR  X3, T1
12.   SWAP T1
13.   EOR  X1, T1

//SWAPMOVE(X4,X6,0x0F,4)
14.   MOVW T0, X4
15.   SWAP T0
16.   EOR  T0, X6
17.   ANDI T0, 0x0F
18.   EOR  X6, T0
19.   SWAP T0
20.   EOR  X4, T0

//SWAPMOVE(X5,X7,0x0F,4)
21.   SWAP T1
22.   EOR  T1, X7
23.   ANDI T1, 0x0F
24.   EOR  X7, T1
25.   SWAP T1
26.   EOR  X5, T1

//SWAPMOVE(X3,X6,0xFF,0)
27.   MOV T0, X3
28.   MOV X3, X6
29.   MOV X3, T0

//SWAPMOVE(X1,X4,0xFF,0)
30.   MOV T0, X1
31.   MOV X1, X4
32.   MOV X1, T0
.endm
```

IV. 성능 평가

본 장에서는 8-비트 AVR 환경인 ATmega128 프로세서에서 기존에 제시된 구현물과 제안하는 구현물의 성능 평가를 진행한다. Microchip Studio 프레임워크에서 구현 및 cycle을 측정하고 컴파일 옵션은 -O3(Optimize most)를 사용하였다. 키스케줄과 일부 라운드키에 수행되는 P 연산을 미리 계산한 것으로 가정하고 1블록 암호화 과정에서의 성능을 평가한다. 성능 측정 결과는 표 5에서 확인할 수 있으며, 코드 크기인 Code size와 RAM은 바이트 단위이고 cpb는 cycles per byte이다. 표 5에 제시된 방법들은 AVR 어셈블리 코드로 작성되었다. 표 5에서 본 논문의 제안 구현물인 PRESENT[this work]는 P_1을 표 4와 같이 구현한 것으로, 기존 최적 구현에 비해 코드 크기는 16.5%만큼 줄어든 820 bytes로, cpb는 13.3% 개선되어 현재까지의 8-비트 AVR 환경에서 PRESENT의 구현물 중 가장 뛰어난 성능을 보여준다.

표 5 PRESENT 성능 측정 결과

Method	Code size	RAM	cpb
PRESENT ref[3]	956	282	504.25
PRESENT[this work]	820	276	445

⊙ code size(Byte) : 암호화 함수의 코드 크기
⊙ RAM(Byte) : RAM 사용량
⊙ cpb : clock cycles per byte

V. 결론

본 논문에서는 경량 블록암호 PRESENT에 대한 8-비트 AVR 환경에서 최적 구현을 제시하고 그 성능을 기존 구현물과 비교하였다. 16-비트 단위로 구현된 P_1을 8-비트 단위 구현으로 최적화하여 기존 최적 구현물보다 16.5% 적은 코드 길이와 13.3% 빠른 연산 속도를 가진 구현물을 제시하였다. 이 구현물은 현재까지의 8-비트 AVR 환경에서 PRESENT 구현물 중 가장 뛰어난 성능을 보여준다. 추후 연구로 매 라운드마다 라운드키를 새로 생성하는 on-the-fly 구현에서 고속화하는 방법에 대한 연구를 남긴다.

[참고문헌]

[1] Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: an ultra-lightweight block cipher. In: CHES. Lecture Notes in Computer Science, vol. 4727, pp. 450 - 466. Springer (2007)

[2] Reis, T.B., Aranha, D.F., López, J. PRESENT runs fast. In Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems, Taipei, Taiwan, 25 - 28 September 2017; Springer: Berlin/Heidelberg, Germany, 2017; pp. 644 - 664.

[3] Kwon H, Kim YB, Seo SC, Seo H. High-Speed Implementation of PRESENT on AVR Microcontroller. Mathematics. 2021; 9(4):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