RISC-V 상에서의 Skinny Tweakable 블록암호 구현

엄시우*, 김현준*, 심민주*, 송경주*, 서화정**

*한성대학교 (대학원생)
**한성대학교 (교수)

Implementation of Skinny Tweakable Blcok Cipher on RISC-V

Si-Woo Eum*, Hyun-Jun Kim*, Min-Joo Sim*, Gyeong-Ju Song*, Hwa-Jeong Seo***

*Hansung University(Graduate student)
**Hansung University(Professor)

요 약

Skinny 블록암호는 NIST 경량 암호 공모전의 최종 라운드에 진출한 Romulus 암호에서 활용되는 블록암호로 Tweakey Framework를 사용하는 Tweakable 블록암호이다. 본 논문은 32-bit RISC-V processor 상에서의 Skinny Tweakable 블록암호의 구현 연구를 진행한다. 최적화 구현을 위해 암호화 과정에서 이루어지는 키 확장의 LFSR 연산을 32-bit 단위의 병렬 연산을 통하여 효율적인 구현을 진행한다. 또한 Addconstant 계층이 포함된 키 확장의 사전 연산을 통하여 키 길이와 상관없이 라운드수의 조정만 필요한 암호화 구현의 단일화가 가능하며 암호화 과정의 비용을 줄이는 결과를 얻을 수 있다. 결과적으로 키 확장이 포함된 구현 대비 암호화 과정만 비교하였을 때 사전 연산을 통하여 키 길이에 따라 181%, 217%, 252% 의 성능 향상을 확인 할 수 있다.

I. 서론

2015년부터 NIST(National Institute of Standards and Technology)에서는 제한된 컴퓨팅 환경에서 사용할 경량 암호 공모전을 개최하였으며 여러 경량 암호 알고리즘이 발표되고 있다. 현재 10개의 암호 알고리즘이 최종 라운드까지 올라왔으며 마지막 최종 결과를 기다리고 있다.

최종 라운드까지 올라온 10개의 암호중 Romulus[1]는 Skinny 블록 암호를 활용한 암호 알고리즘이다. Skinny 블록 암호는 Tweakey 프레임워크를 사용하는 Tweakable 블록 암호로 기존의 블록 암호와 조금 다른 특징을 가지고 있는 암호이다. 본 논문에서는 32-bit RISC-V 상에서의 Skinny 블록 암호 구현 연구를 진행한다.

본 논문의 구성은 다음과 같다. 2장에서는 Tweakable 블록암호, Skinny 블록암호, RISC-V에 관하여 설명한다. 3장에서는 제안 기법을 설명한다. 4장에서는 성능 평가를 진행한다. 마지막으로 5장에서는 본 논문의 결론을 내린다.

II. 관련 연구

2.1 Tweakable 블록암호와 Tweakey 프레임워크

일반적인 블록 암호 알고리즘에서 암호문(C)는 입력되는 평문(P)과 키(K)를 통해 얻을 수 있다(즉, C=E(P,K)). 하지만 Tweakable 블록 암호에서는 Tweak(T)이라는 값을 추가로 받아 암호문을 얻을 수 있다(즉, C=E(P,K,T))[2]. 이로 인해 Tweakable 블록암호에서는 같은 평문과 키를 사용하더라도 Tweak 값이 다르다면 다른 암호문이 나온다. 암호화 과정에서 키를 변경하는 것은 많은 비용이 필요하지만 Tweak 값을 변경하는 것은 많은 비용이 필요하지 않아 효율적일 수 있다.

Tweakable 블록암호의 응용으로 디스크 암호화를 예로 들 수 있다. Tweak 값은 공개된 값을 사용할 수 있기 때문에 블록 번호를 Tweak 값으로 사용하여 디스크 암호화를 하게 된다. 이는 동일한 키로 블록들을 암호화 할 때 서로 다른 블록이

같은 값을 가지고 있더라도 블록 번호를 Tweak 값으로 사용하게 되면 서로 다른 Tweak 값으로 인해 다른 암호값을 가지게 된다.

Tweakey 프레임워크는 Tweakable 블록암호와 Related-key 공격에 저항하는 블록암호의 설계를 통합하는 목적을 가지고 제안된 프레임워크이다[3]. Tweakey 프레임워크에서는 Key와 Tweak값이 합쳐진 값을 Tweakey로 사용하여 암호화를 진행한다. Tweakey의 구성은 t-bit의 Tweak 값과 k-bit의 Key값을 구성되어 있다. 이렇게 구성된 (t+k)는 n-bit의 내부 상태 S와 연산된다. 암호화 과정을 수식으로 보면 다음과 같다.

- $S_{i+1} = f(S_i \oplus g(tk_i))$ (1)
- $tk_{i+1} = h(tk_i)$ (2)

수식(1)은 Tweakey와 S의 XOR 값이 라운드 함수 f에 의해 다음 라운드의 S값이 되는 것을 의미한다.

수식(2)는 다음 라운드에 사용될 Tweakey가 함수 h에 의해 연산되는 것을 의미한다.

2.2 Skinny 블록 암호

CRYPTO'16에 소개된 Skinny 블록암호는 Tweakey 프레임워크를 사용하는 Tweakable 블록암호이다[4]. 블록 길이는 64, 128-bit를 지원하며, Tweakey의 길이는 블록 길이의 3배까지 지원한다. 전체적인 매개변수는 Table 1.과 같다.

Table 1. Parameter of Skinny Tweakable block cipher(rounds)

	Tweakey size t		
Block size n	n	2n	3n
64-bit	32	36	40
128-bit	40	48	56

Skinny 블록암호의 라운드함수는 전체적으로 AES 블록암호 알고리즘과 비슷하게 동작한다. 라운드함수의 구성은 Subcell, Addcounstant, Addkey, Shiftrow, Mixcolumn 으로 구성되어 있으며, Addkey 과정에서 수식(2)의 Tweakey 업데이트 과정이 포함되어 있다. 전체적인 암호화 과정은 Fig. 1.과 같다.

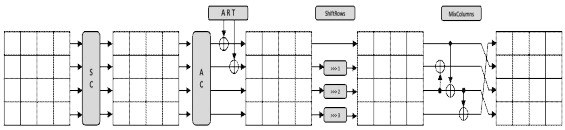

Fig. 1. Structure of Skinny Block Cipher

SubCell은 SBox 테이블의 값과 치환하는 연산을 진행한다. AddConstant는 사전에 정의된 상수값과 S[0], S[1], S[2]의 값이 XOR 연산을 진행한다. ART(Add Round Tweakey)는 Tweakey 값과 내부 상태 State의 1번째, 2번째 행이 XOR 연산을 진행한다. ShiftRows는 각 행이 정해진 로테이션 연산을 진행한다. 마지막으로 MixColumns는 행렬 곱을 통해 State의 각 열간에 확산이 이루어진다.

2.3 RISC-V processor

2010년부터 UC 버클리에서 개발중인 RISC (Reduced Instruction Set Computer) 기반의 컴퓨터 아키텍쳐이다[5]. RISC-V는 32-bit, 64-bit 레지스터를 사용하며, 이는 RV32I, RV64I의 두 가지 모델로 나뉜다. 본 논문에서 사용한 32-bit 구조의 RV32I는 32-bit 레지스터 32개를 제공한다.

III. 제안 기법

본 논문에서 제안하는 기법은 기존 Skinny 블록암호의 알고리즘은 내부에서 키 확장이 이루어진다. 따라서 키 확장을 사전연산을 통해 값을 불러와 단순 XOR 연산으로 대체한다. 또한 키 확장에 AddConstant 부분을 포함시킴으로써 암호화 과정에서 해당 함수의 연산을 생략한다. 본 논문에서는 블록 길이 128-bit의 3개의 키 길이에 따른 최적화 구현을 진행하였다.

본 장의 구성은 다음과 같다. 3.1장에서 키 확장을 포함하는 구현에서의 최적화 구현을 설명한다. 3.2장에서는 키 확장을 사전 연산한 최적화 구현을 설명한다.

3.1 키 확장 포함 최적화 구현

Skinny 블록암호의 알고리즘에서 라운드 함수 AddRoundTweakey 함수의 연산이 끝나고 다음 라운드에서 사용할 라운드 Tweakey 업데이트를 진행한다. 이때 Tweakey의 길이가 블록 길이와

같을 때는 Tweakey의 Permutation 만 진행한다. 하지만 키 길이가 블록 길이(n)의 2n, 3n일 경우 LFSR 과정이 추가된다. 예를 들어 키 길이가 3n 일 경우 128-bit의 길이를 갖는 TK1, TK2, TK3로 나뉘어 연산이 진행된다. Tweakey가 업데이트 될 때 각각의 키 블록은 Permutation을 진행하고 TK2, TK3의 경우 LSFR 과정을 추가로 진행한다. LSFR은 Fig. 2.와 같이 동작한다.

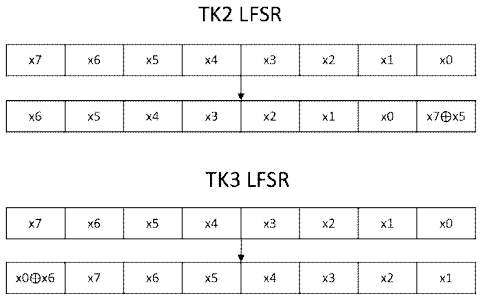

Fig. 2. Process of LFSR(unit : Bit)

Fig. 2.는 1byte State 기준 LFSR의 과정을 보여준다. LFSR 과정에서는 bit의 자리 바꿈과 XOR 연산을 진행한다. 각 바이트 단위로 LFSR 과정을 진행하게 되면 16번의 동일한 과정을 반복하기 때문에 비효율적이다. 따라서 32-bit 레지스터 상에서 4개의 State를 한번에 LFSR을 병렬 구현한다. 구현된 코드는 Table 2.와 같다.

Table 2. 에서는 TK2 1행의 LFSR 과정을 나타낸다. t1, t2에 저장된 값과 AND 명령어를 통하여 State 내부의 각 바이트가 Shift 명령어로 인해 섞인 값을 0으로 바꾸어 Bit의 자리 바꿈을 진행하게 된다. 이때 자리 바꿈을 진행하고 나면 최하위 Bit에 x7이 위치하게 된다. x5의 값을 얻기 위해 t0에 저장된 값과 AND 명령어를 사용하여 x5의 Bit를 얻는다. 그 후 현재 최하위 bit x7과 XOR 연산하여 TK2의 LFSR을 구현한다. 본 과정은 1행과 2행만 진행하며, TK3에서도 이와 비슷하게 구현한다.

3.2 키 확장의 사전 연산을 통한 최적화 구현

키 확장 과정이 암호화 과정에 포함되어 있기 때문에 기존의 다른 암호에 비하여 암호화 과정에서의 비용이 많이 든다. 따라서 다른 암호화 동일

Table 2. Process of LFSR(TK2)

Input: TK2
Output: LFSR(TK2)

```
li    t0, 0x40404040
li    t1, 0xfefefefe
li    t2, 01010101
slli   t3, a0, 1
and    t3, t3, t1
srli   a0, a0, 7
and    a0, a0, t2
or    a0, a0, t3
and    t3, a0, t0
srli   t3, t3, 6
xor    a0, a0, t3
```

하게 키 확장 과정을 암호화 과정에서 제외하고 해당 라운드의 Tweakey 값을 불러와 연산하는 방법으로 대체한다.

3n(즉, 키 길이 : 384-bit)일 때의 Tweakey의 사전 연산 의사 코드는 Table 3.과 같다. 키 확장 사전 연산 구현 기법에서는 Tweakey의 사전 연산 과정에 AddConstant 계층을 포함한다. 이로 인해 AddConstant가 연산된 Tweakey 값을 불러와 연산할 수 있으며, 암호화 과정에서 AddConstant 계층을 생략 가능하다.

사전 연산을 통하여 기존의 구현에서는 TK1, TK2, TK3의 각각의 키 블록이 State와 연산을 진행하였다. 하지만 이 과정을 사전 연산에 포함시켜 연산하였기 때문에 암호화 과정에서의 단순 라운드 Tweakey와의 XOR 연산으로 구현할 수 있다.

사전 연산을 통한 Tweakey를 사용할 경우 기

Table 3. Pseudocode of Pre-compute Tweakey Schedule(tweakey size : 384-bit)

Input: Plain text(State), TK, TK_table
Output: Cipher text

```
1   temp = Addconstant(TK1[0~2]);
2   temp[0~7] ^= TK1^TK2^TK3; //[0~7]
3   Store_TK(temp);
4   For i = 0 to 55 //(3n rounds : 56)
5     TK_permutation(TK1);
6     TK_permutation(TK2);
7     TK_permutation(TK3);
8     TK2_LFSR(TK2);
9     TK3_LFSR(TK3);
10    Addconstatn(temp[0~7]);
11    temp[0~7] ^= TK1^TK2^TK3 //[0~7]
12    Store_TK(temp[0~7]);
13  End For
```

존의 키 확장이 포함된 구현에서는 키 길이에 따라 키 확장 과정에서의 차이가 발생하게 된다. 하지만 사전 연산으로 Tweakey를 불러와 연산하게 될 경우 키 길이에 상관없이 라운드수가 증가하는 차이만 존재하기 때문에 라운드 수의 조정만 필요한 구현의 단일화가 가능하다.

IV. 성능 평가

본 논문에서는 SiFive 사의 HiFive RevB 보드를 사용하여 측정을 진행하였고, SiFive 사에서 제공하는 Freedom Studio를 사용한다.

성능 평가는 기존의 키 확장이 암호화 과정에 포함된 구현과 키 확장의 사전 연산을 통한 암호화 구현을 비교한다. 성능 비교는 Cycle 측정을 통해 비교하였으며, 사전 연산의 경우 키 확장 과정과 암호화 과정의 Cycle을 나누어 측정하였다. 측정 결과는 Table 4. 와 같다.

Table 4. Performance of Skinny-128 on 32-bit RISC-V processor (unit : Cycle)

	Cycle		
Skinny-128	**128**	**256**	**384**
Encryption(TK)	9340	13410	18175
Pre Compute	4581	7563	11323
Encryption	5167	6191	7215
Pre+Enc	9748	13754	18538

결과적으로 사전 연산 과정을 제외한 암호화 과정만 비교할 경우 키 길이에 따라서 181%, 217%, 252% 의 성능 향상을 확인 할 수 있었다. 사전 연산으로 인해 라운드 수의 차이만 존재하는 구현의 단일화가 가능하기 때문에 키 길이에 따라 사전 연산 비용의 차이는 크지만, 암호화 비용의 차이가 크지 않은 것을 확인할 수 있다. 또한 사전 연산 비용이 많이 들지만 Tweakey의 변경이 1개 블록(128-bit)을 암호화 할 때마다 이루어지지 않는다면 사전 연산을 통한 암호화의 성능이 좋은 것을 확인할 수 있다.

V. 결론

본 논문에서는 Skinny Tweakable 블록암호의 RISC-V 상에서의 구현 연구를 진행하였다. 기존의 키 확장 과정이 포함된 구현과 키 확장 과정을 사전 연산하여 암호화 과정에서는 생략한 구현을 나누어 진행하였다. 또한 사전 연산 과정에서 AddConstant 함수를 포함하여 해당 함수를 생략한 암호화 과정이 가능하였다. 결과적으로 키 확장이 포함된 구현 대비 암호화 과정의 성능은 키 길이에 따라 181%, 217%, 252%의 성능 향상을 확인 할 수 있었다. 향후 연구 과제로는 Skinny 블록암호의 다른 운용모드의 최적화 구현을 제안한다.

VI.Acknowledgment

이 논문은 부분적으로 2021년도 정부(과학기술정보통신부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임(No.2018-0-00264, IoT 융합형 블록체인 플랫폼 보안 원천 기술 연구, 50%) 그리고 이 성과는 부분적으로 2021년도 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. NRF-2020R1F1A1048478, 50%).

[참고문헌]

[1] Iwata, T., Khairallah, M., Minematsu, K., Peyrin, T.: New Results on Romulus. In: NIST LWC Workshop (2020)

[2] M Liskov, RL Rivextm D Wagner. “Tweakable block ciphers”. Advances in Cryptology-CRYPTO 2002. pp 31-46.

[3] J Jean, I Nikolić, T Peyrin. “Tweaks and Keys for Block Cipher: The TWEAKEY Framework“. Advances in Cryptology - ASIACRYPT 2014. Lecture Notes in omputer Science, vol 8874.

[4] Beierle C. et al. The SKINNY Family of Block Ciphers and Its Low-Latency Variant MANTIS. In: Robshaw M., Katz J. Advances in Cryptology - CRYPTO 2016. Lecture Notes in Computer Science, vol 9815.

[5] Waterman, Andrew, et al. "The risc-v instruction set manual, volume i: Base user-level isa." EECS Department, UCBerkeley, Tech. Rep. UCB/EECS-2011-62 116, 2011