8-bit AVR 상에서 경량암호 Romulus의 Linear Feedback Shift Register 최적 구현

권혁동* 엄시우** 심민주** 서화정***

*한성대학교 정보컴퓨터공학과 (대학원생)
**한성대학교 IT융합공학부 (대학원생)
***한성대학교 IT융합공학부 (조교수)

Optimized implementation of Linear Feedback Shift Register of lightweight cipher Romulus on 8-bit AVR processors

Hyeok-Dong Kwon* Si-Woo Eum** Min-Joo Sim** Hwa-Jeong Seo***

*Dept. of Information Computer Engineering, Hansung University (Graduate student)
**Dept. of IT Convergence Engineering, Hansung University (Graduate student)
***Dept. of IT Convergence Engineering, Hansung University (Assistant professor)

요 약

본 논문에서는 경량암호공모전에 제출된 작품인 Romulus의 모듈인 Linear Feedback Shift Register를 8-bit AVR 프로세서상에서 최적 구현하였다. 구현은 배포되는 레퍼런스 C 코드를 기반으로 구현을 진행하였다. 제안하는 구현물은 2종류로, 시프트 명령어를 활용한 기본적인 구현물과 비트 스토어를 활용한 구현물이다. 각각의 구현물을 구현한 원리와 최적 구현을 위한 설계 기법을 소개한다. ATmega128 환경에서 -O3 옵션을 사용하여 성능 평가를 진행한 결과, 레퍼런스 구현물은 79 clock cycles가 소요되었으며 시프트 명령어를 사용한 구현물은 112 clock cycles가 소요되었다. 반면 비트 스토어를 사용한 구현물은 64 clock cycles가 소요되었다.

I. 서론

2018년 NIST에서는 경량암호 표준화를 위한 경량암호 공모전을 개최하였고, 현재는 Round 3를 진행하고 있다. NIST에서 경량암호 공모전을 개최하며 발표한 요구사항은 AEAD(Authenticated Encryption with Associated Data) 기능을 제공하는 것이다. 추가로 해시 기능을 제공하는 것도 있지만 이는 필수적이진 않다[1]. 최초 56종의 알고리즘이 선정되었고, Round 3에서는 10종의 알고리즘이 남게 되었다. 그중에서도 Romulus는 유일하게 Tweakable Block Cipher 기반의 알고리즘으로 해시 기능을 함께 제공한다. 본 논문에서는 경량암호 공모전에 제안된 Romulus의 Linear Feedback Shift Register 모듈을 8-bit AVR 환경에서 최적 구현한 구현물을 소개하고 그 성능 비교를 진행한다.

II. 배경

2.1 경량암호 Romulus

표 1은 NIST 경량암호 공모전 Round 3의 진출 알고리즘들을 나열한 것이다. 각각의 알고리즘 별로 분류 유형과 해시함수 지원 여부, 그리고 코어 함수를 확인할 수 있다. 제안된 알고리즘 중에서는 permutation 기반 알고리즘이 7종으로 가장 많다. 전체 알고리즘 중에서 절반인 5종의 알고리즘은 해시함수 기능을 지원한다. 코어 함수는 기존에 있던 블록암호 또는 해시함수를 사용한다.

Table. 1. Round 3 candidates of NIST Lightweight Cryptography Standardization competition.

Algorithm	Type	Hash	Core function
Grain-128AEAD	Stream cipher	X	Grain-128a
GIFT-COFB	Block cipher	X	GIFT-128
Romulus	Twekable block cipher	O	SKINNY-128-256, SKINNY-128-384
ASCON	Permutation	O	ASCON-320
ISAP	Permutation	X	Keccak-400, ASCON-320
PHOTON-Beetle	Permutation	O	PHOTON-256
Elephant	Permutation	X	Spongent-160/176, Keccak-200
SPARKLE	Permutation	O	Sparkle-256/384/512
TinyJambu	Permutation	X	JAMBU-128
Xoodyak	Permutation	O	Xoodoo-384

Romulus는 SKINNY 블록암호를 사용하는 경량암호로, Round 3의 유일한 Tweakable Block cipher이다. Romulus는 논스 기반의 AE를 제공하는 Romulus-N, Misuse 공격에 내성을 지니는 Romulus-M, leakage-resilient AE인 TEDT의 개선판인 Romulus-T, 그리고 제2 역상 저항성을 지니는 해시인 Romulus-H로 구성된다. 구현물이 여러 종류가 있지만, 중점적인 구현물은 Romulus-N과 Romulus-H이다. Romulus의 파라미터는 표 2에서 확인할 수 있다.

Romulus는 카운터로 56-bit를 사용하는데, 이때 카운터 생성에 Linear Feedback Shift Register(LFSR)을 사용한다[2]. LFSR은 시프트 레지스터에 속하는 장치로, 레지스터에 입력된 값들을 선형 연산을 통해서 다른 값을 생성한다. 이러한 특성을 활용하여 의사 난수 함수로 사용하는 경우가 많다. 비슷한 경우로, Nonlinear Feedback Shift Register(NLFSR)을 사용한 다른 후보 알고리즘인 TinyJAMBU[3]도 있다. LFSR의 장점은 간단한 구조로 의사난수를 생성할 수 있으며, 하드웨어 구현에도 유리하다. Romulus의 LFSR은 56-bit 크기의 LFSR을 사용하며, irreducible polynomial $F_{56}(x) = x^{56} + x^7 + x^4 + x^2 + x^0$을 사용하며, 최초에는 1 mod $F_{56}(x)$로 초기화된다. 이후 Romulus의 LFSR이 카운터를 증가시키는 방식은 표 3의 알고리즘과 같다.

Table. 2. Parameters of Romulus. (Unit: bit)

Member	N, M, T	H
Nonce len.	128	-
Block len.	128	256
Key len.	128	-
Counter bit len.	56	-
Tag len.	128	-
Hash len.	-	256

Table. 3. Algorithm of Linear Feedback Shift Register for Romulus.

1:	$z_i \leftarrow$ for $i \in [\![56]\!]_0 \setminus \{7,4,2,0\}$
2:	$z_7 \leftarrow z_6 \oplus z_{55}$
3:	$z_4 \leftarrow z_3 \oplus z_{55}$
4:	$z_2 \leftarrow z_1 \oplus z_{55}$
5:	$z_0 \leftarrow z_{55}$

III. 제안 기법

제안하는 구현물은 Romulus의 LFSR을 8-bit AVR 환경에서 소프트웨어 최적 구현을 제안한다. 기본적인 구현은 Romulus-N의 레퍼런스 코드를 기반으로 진행한다. 우선 Romulus의 LFSR은 표 4와 같은 코드를 통해서 구현이 된다.

전체적인 코드의 흐름은 8 크기를 가지는 8-bit 변수 배열을 매개변수로 입력받고, 그 값들을 좌측으로 1회 시프트, 우측으로 7회 시프트 한 다음 OR 연산을 통해서 하나의 값으로 합쳐주는 구조를 지닌다.

레퍼런스 코드를 기반으로 어셈블리 구현물을 작성한다. 이때 작성하는 구현물은 두 가지로, 시프트 명령어를 활용한 것과 비트 스토어 명령어를 사용한 것으로 분리한다.

Table. 4. Reference source code of LFSR.

```
fb0 = CNT[6] >> 7;
CNT[6] = (CNT[6] << 1) | (CNT[5] >> 7);
CNT[5] = (CNT[5] << 1) | (CNT[4] >> 7);
CNT[4] = (CNT[4] << 1) | (CNT[3] >> 7);
CNT[3] = (CNT[3] << 1) | (CNT[2] >> 7);
CNT[2] = (CNT[2] << 1) | (CNT[1] >> 7);
CNT[1] = (CNT[1] << 1) | (CNT[0] >> 7);
if (fb0 == 1) {
  CNT[0] = (CNT[0] << 1) ^ 0x95;
}
else {
  CNT[0] = (CNT[0] << 1);
}
```

우선 시프트 명령어를 사용한 구현물을 확인한다. 시프트 명령어를 사용한 구현은 LFSR C 코드를 어셈블리 코드로 이식한 형태가 된다. 표 5의 코드는 구현에 사용한 어셈블리 코드의 일부이며, 표 4의 두 번째 줄에 대응된다. 코드는 4개의 명령어를 사용한 것으로 보이지만, 3번째 명령어는 매크로이다. 해당 매크로는 실제로 7개의 명령어로 구성되기 때문에 실제 코드는 10개의 명령어를 사용한다..

Table. 5. Source code of Proposed shift version implementation.

```
1:  MOV   TMP,   CNT5
2:  LSL   CNT6
3:  LSR7_TMP
4:  OR    CNT6,  TMP
```

1번째 명령어 MOV는 CNT[5]의 값을 임시 레지스터로 이동시킨다. 이는 CNT[5]의 원본 값이 다음 단계에 필요하기에 소실을 방지하기 위함이다. 2번째 명령어 LSL은 CNT[6]의 값을 좌측 1회 시프트 해준다. 3번째 명령어 LSR7_TMP는 매크로로, 'LSR TMP'가 7회 반복된다. TMP에는 CNT[5]의 값이 들어있으므로, CNT[5]>>7의 값이 TMP에 저장된다. 마지막 명령어 OR은 생성한 중간값 두 개를 OR 연산으로 결과값을 계산한다.

연산 중 가장 불필요한 부분은 7회 우측 시프트로, 우측으로 7회 시프트를 진행하는데 7개의 명령어가 사용되어 비효율적이다. 따라서 비트 스토어를 활용한 새로운 구현물을 제안한다.

AVR에는 32개의 레지스터 외에도 다양한 상태 표현을 위한 플래그가 존재한다. 이때 T 플래그는 사용자가 필요할 때 값을 임시로 저장하거나 불러올 수 있는 플래그로 특정 명령어를 통해서 해당 플래그에 1-bit를 이동할 수 있다.

Romulus의 LSFR에서 가장 비효율적인 부분은 7회 우측 시프트를 진행하는 부분이다. 해당 부분의 연산 결과는 8-bit 레지스터에 저장된 최상위 비트를 최하위 비트로 이동한다. 비트 스토어를 사용하면 이를 조금 더 단순하게 구현할 수 있다. 구현 코드는 표 6과 같다. 표 5와 마찬가지로 어셈블리 코드의 일부이며, 표 4의 두 번째 줄에 대응된다.

Table. 6. Source code of Proposed bit store version implementation.

```
1:  BST   CNT5,  7
2:  LSL   CNT6
3:  BLD   CNT6,  0
```

1번째 명령어는 비트 스토어로, CNT[5]의 비트를 T 플래그로 이동시킨다. 저장할 때는 최상위 비트를 T 플래그에 저장해야 하므로 비트 번호는 7번을 지정한다. 2번째 명령어는 좌측 시프트로 CNT[6]의 값을 좌측으로 1회 시프트 한다. 그 결과 CNT[6]의 최하위 비트는 0이 저장되게 된다. 3번째 명령어는 비트 로드로, T 플래그의 값을 CNT[6]에 저장한다. 이때 CNT[6]의 최하위 비트로 저장해야 하므로 저장할 비트 번호는 0번을 지정한다.

전체적인 구현물의 동작 순서는 그림 1과 같이 간단하게 표현이 가능하다. 1단계에서 레지스터 0의 최상위 비트를 T 플래그에 저장한다. 2단계에서 레지스터 1의 값을 좌측으로 1회 시프트 한다. 3단계에서 T 플래그의 값을 레지스터 1의 최하위 비트로 가져온다. 비트 스토어 구현물은 시프트 구현물보다 7개 적은 명령어

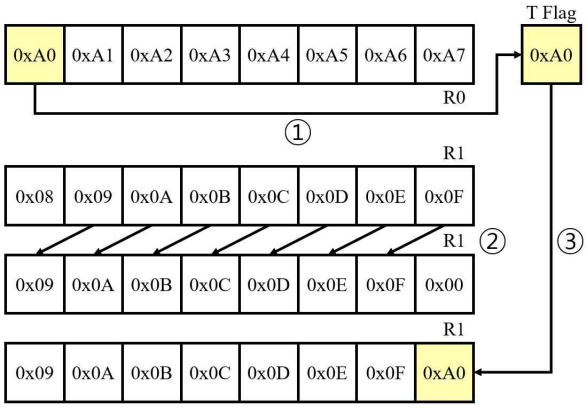

Fig. 1. LFSR working flow of bit store version implementation.

를 사용하였다. 이를 통해서 더 단순한 동작을 지니지만 결과값은 동일하게 생성할 수 있다.

IV. 성능 평가

본 장에서는 기존 레퍼런스 구현물과 제안하는 구현물 2종류와의 성능 평가를 진행한다. 구현 환경으로는 Microchip Studio 프레임워크를 사용하였고, 대상 프로세서는 ATmega128을 대상으로 한다. 컴파일 옵션은 -O3(fastest)를 사용하였다. 성능 측정 단위로는 clock cycles를 사용한다. 성능 측정 결과는 표 7에서 확인할 수 있다.

Table 7. Performance evaluation results. (Unit: clock cycles)

Reference	Proposed -shift	Proposed -bit store
79	112	**64**

성능 측정 결과, 레퍼런스 구현물은 79 clock cycles가, 시프트 명령어 구현물은 112 clock cycles가, 그리고 비트 스토어 구현물은 64 clock cycles가 소요되었다. -O3 옵션은 컴파일러가 상당한 수준의 최적화를 진행하기 때문에 C언어로 작성된 레퍼런스 구현물의 속도가 최적화 옵션을 적용하지 않았을 때보다 매우 향상되었다. 반면 시프트 구현물은 레퍼런스 구현물의 성능 대비 약 30% 떨어지는 성능을 가진다. 이는 어셈블리를 사용했지만, 구현이 효과적이지 않았음을 뜻한다. 비트 스토어를 사용한 구현물은 레퍼런스 코드 대비 23%의 성능 향상을 보였다. 성능 향상 폭이 크지는 않지만, 동일하게 어셈블리를 사용한 시프트 구현물보다 훨씬 좋은 성능을 지니고 있음을 알 수 있다.

V. 결론

본 논문에서는 경량암호 공모전의 최종라운드 후보인 Romulus의 모듈 중 LFSR을 8-bit AVR 상에서 최적 구현을 시도하였다. 제안하는 구현물은 시프트 명령어를 사용한 것과 비트 스토어 명령어를 사용한 구현물로 두 가지를 제안하였다. 구현 결과, 시프트 구현물은 레퍼런스보다 느린 성능을 지녔지만, 비트 스토어 구현물은 더 좋은 성능을 보였다.

이를 통해 어셈블리 최적 구현을 시도할 때는, C언어를 어셈블리로 이식하는 것뿐만 아니라 효과적인 구현을 위해 알고리즘을 수정할 필요가 있다는 것을 알 수 있다. 추후 연구 과제로 Romulus의 다른 부분을 최적 구현하여 Romulus 암호 자체의 최적화를 진행한다.

[참고문헌]

[1] L.Bassham, Ç.Çalık, K.McKay, and M.S.Turan, “Submission requirements and evaluation criteria for the lightweight cryptography standardization process,” *US National Institute of Standards and Technology*, 2018.

[2] L.-T.Wang, and J.McCluskey, “Linear feedback shift register design using cyclic codes”, *IEEE Transactions on Computers*, Vol. 37, No. 10, pp. 1302-1306, 1988.

[3] H.Wu, and T.Huang, “TinyJAMBU: A family of lightweight authenticated encryption algorithms (version 2)”, *Submission to the NIST Lightweight Cryptography Standardization Process*, 2021.