논리적 블록 이동을 적용한 Fixed-Gimli Permutation

권혁동* 엄시우* 서화정*†

*한성대학교 IT융합공학부(대학원생)

*† 한성대학교 IT융합공학부(교수)

Fixed-Gimli Permutation with logical block movement

Hyeok-Dong Kwon* Si-Woo Eum* Hwa-Jeong Seo*†

*Hansung University IT Convergence Engineering(Graduate student)

*† Hansung University IT Convergence Engineering(Professor)

요 약

Gimli는 NIST 경량 암호화 표준 공모전에 제출된 Permutation 알고리즘으로, 암호화와 해시 함수에 사용할 수 있는 알고리즘이다. Gimli는 384-bit의 평문을 입력받아 24라운드 동안 Permutation을 진행하며, 라운드 함수 도중에는 하나의 열을 대상으로 해당 열의 행 간 값을 교체하는 Swap 단계가 존재한다. 본 논문에서는 Swap 단계를 생략하여 물리적으로 값들의 위치를 고정하는 대신, 값의 호출 순서를 변경한 Fixed-Gimli를 제안한다. Fixed-Gimli는 Swap 연산을 생략하지만 호출 순서를 변경하는 것으로 논리적으로는 블록 이동이 진행한 것으로 설계하여 Swap에 소요되는 시간 자원을 줄인 알고리즘이다.

I. 서론

최근 다양한 기기들이 사물인터넷 기술로 연결되어 통신을 하며 사용되어 오고 있다. 이러한 다양한 기기들이 안전한 연결과 통신을 위해 암호기술이 사용되어야 한다. 하지만 기존에 사용하는 암호기술은 데스크톱, 서버 환경용으로 설계되어 제한된 환경(낮은 CPU 성능, 작은 용량의 메모리, 낮은 전력 등)에서 작동하는 사물인터넷 기기에서는 사용이 적합하지 않다. 이러한 제한된 환경에게 사용하기 위해 HIGHT, LEA, CHAM 등 경량암호가 개발되고 있다[1].

Gimli는 'NIST 경량 암호화 표준화 프로세스'에서 2차 후보로 올라온 암호로 제한된 환경뿐만 아니라 다양한 환경에서 높은 성능의 높은 보안을 달성하도록 설계된 384-bit 순열로, 암호화와 해시함수에 사용된다.

본 논문에서는 8-bit AVR 프로세서 상에서 Gimli Permutation을 최적 구현하기 위해 논리적 블록 이동을 사용한 Fixed-Gimli Permutation을 제시한다.

본 논문의 구성은 다음과 같다. 2장에서는 Gimli Permutation에 대해 설명한다. 3장에서는 논리적 블록 이동을 적용한 Fixed-Gimli Permutation을 제시한다. 4장에서는 기존의 Gimli Permutation과 Fixed-Gimli Permutation의 성능을 비교한다. 마지막으로 5장에서 본 논문에 대한 결론을 내린다.

II. Gimli Permutation

2.1 라운드 함수[2]

Gimli는 입력 값을 [그림 1]과 같이 32-bit의 3×4 형태로 간주하며 이를 S로 칭한다. Gimli는 24라운드가 진행되며, 각 라운드 함수는 다음 세 가지의 계층의 조합으로 구성된다.

첫 번째는 SP-box를 통과하는 비선형 계층으로 각각의 열이 혼합되는 연산을 진행한다. 두 번째는 선형 혼합 계층으로, Small Swap과 Big Swap으로 나뉜다. 세 번째는 라운드 상수

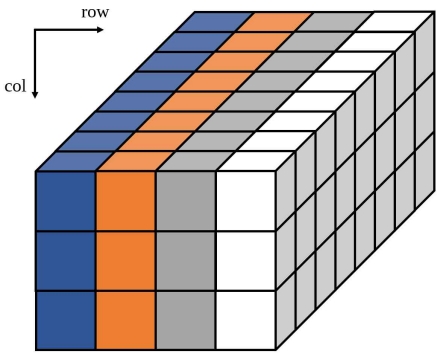

Fig. 1. State representation

추가 계층이다.

2.2 비선형 계층

비선형 계층의 SP-box는 세 가지 단계의 작업을 거친다. 첫 번째는 S_0열과 S_1열의 블록에 각각 24-bit, 9-bit rotation 연산을 취해준다. 두 번째는 S_0, S_1, S_2열 간에 혼합 연산을 취해준다. 마지막에서는 S_0와 S_2열의 값을 교체한다.

2.3 선형 계층

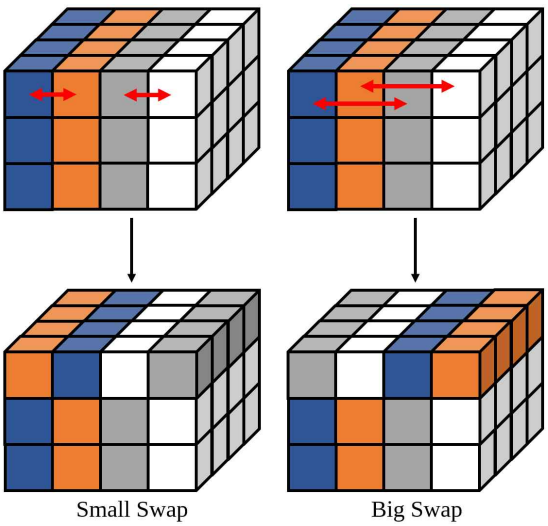

Fig. 2. Left: Small Swap operation
Right: Big Swap operation

선형 계층은 [그림 2]와 같이 행 간 값 교체가 이루어지며 두 종류로 나뉜다. Small Swap은 1라운드부터 4라운드마다, Big Swap은 2라운드부터 4라운드마다 진행한다. 값 교환은 첫 번째 열에서만 발생한다.

2.4 라운드 상수 추가 계층

4의 배수 라운드마다 현재 라운드 r과 라운드 상수 값 0x9E377900을 XOR 한다. 이렇게 획득한 값을 $S_{0,0}$과 XOR하여 라운드 상수를 반영한다.

라운드 함수의 전체 구조를 의사코드 형태로 나타내면 [그림 3]과 같다.

Algorithm
Input: state = $(state_{col,row}) \in W^{3x4}$
Output: Gimli(state) = $(state_{col,row}) \in W^{3x4}$

for round = 24 to 1
  for row = 0 to 3 // SP-box
    $X \leftarrow ROL_{24}(state_{0,row})$
    $Y \leftarrow ROL_{9}(state_{1,row})$
    $Z \leftarrow state_{2,row}$
    $state_{2,row} \leftarrow X \oplus LSL_1(Z) \oplus LSL_2(Y\&Z)$
    $state_{1,row} \leftarrow Y \oplus X \oplus LSL_1(X|Z)$
    $state_{0,row} \leftarrow Z \oplus Y \oplus LSL_3(X\&Y)$
  end for
  if round mod 4 == 0 // Small Swap
    $state_{0,0}\ state_{0,1}\ state_{0,2}\ state_{0,3} \leftarrow state_{0,1}\ state_{0,0}\ state_{0,3}\ state_{0,2}$
  else if round mod 4 == 2 // Big Swap
    $state_{0,0}\ state_{0,1}\ state_{0,2}\ state_{0,3} \leftarrow state_{0,2}\ state_{0,3}\ state_{0,0}\ state_{0,1}$
  end if
  if round mod 4 == 0 // Add Constant
    $state_{0,0} = state_{0,0} \oplus 0x9e377900 \oplus round$
  end if
end for
Return$(state_{col,row})$

Fig. 3. Pseudocode for Gimli permutation

III. 제안 기법

본 논문에서는 Small swap과 Big swap을 생략한 기법을 제안한다. 선형 계층의 Swap 연산은 첫 번째 열에 대해서만 진행된다. 따라서 Swap에 소요되는 시간만큼 연산 시간을 단축시킬 수 있다.

하지만 Swap이 생략되기 때문에 각 라운드별로 특정 행의 값을 로드해야 한다. 이 사안은 비선형 계층과 라운드 상수 추가 계층에 해당된다.

Swap을 진행한 경우, [그림 4]와 같이 $S_{0,0}$, $S_{0,1}$, $S_{0,2}$, $S_{0,3}$의 값을 $X_{0\sim4}$에 로드할 때, $S_{0,0}$, $S_{0,1}$, $S_{0,2}$, $S_{0,3}$ 순서로 값을 호출하는 것을 알 수 있다. 이는 일반적인 Gimli Permutation의 형태이다. 선형 계층을 통과하면서 첫 번째 열 값에 Swap이 적용되기 때문에 특별한 호출 구조가 필요 없다.

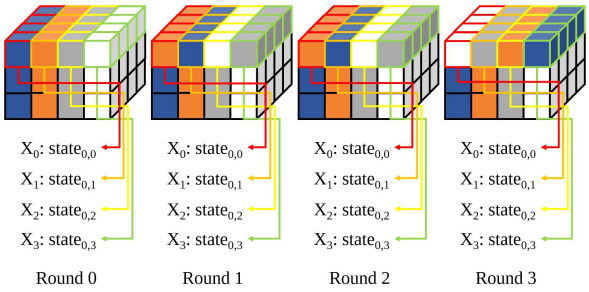

Fig. 4. Original Gimli permutation flow

하지만 Swap을 생략하게 될 경우, 첫 번째 열 호출을 다르게 구성해야 한다. Swap이 생략된다면 행 간에 값 이동이 발생하지 않는다. 하지만 각 라운드에는 값 이동이 발생한 값이 입력되어야 하므로 값을 로드하는 순서가 바뀌게 된다. [그림 5]는 이를 묘사한 그림으로, 실제로 행 간 값의 이동은 발생하지 않지만 입력되는 값은 Swap 연산이 적용된 값과 동일한 값이 입력됨을 알 수 있다.

즉, 물리적으로 Swap이 발생하지는 않기에 각 값들이 저장된 위치는 고정되지만, 논리적으로는 Swap 연산을 진행하는 것으로 취급이 가능하다.

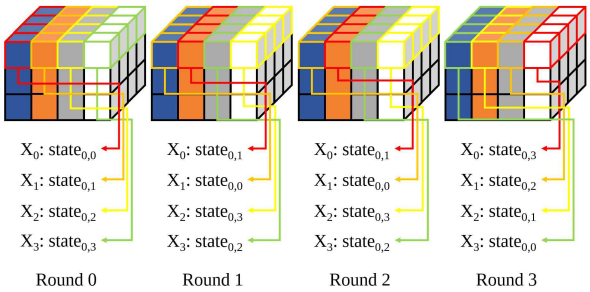

Fig. 5. Fixed-Gimli permutation flow

IV. 성능 비교

제안하는 기법인 Fixed-Gimli와 기존 Gimli의 성능 비교를 진행한다. 구현은 ATmega128 프로세서를 대상으로 구현하였으며, Microchip Studio를 사용하여 구현하였다. 구현 결과는 [표 1]에서 확인이 가능하다. 기존 구현물은 Tiny와 Fast 버전으로 나뉘며 각각 413cpb, 213cpb의 동작 속도를 지닌다. 제안 기법을 구현하기 위해서 우선 기본 구현물을 작성하고, 이를 토대로 제안 기법을 적용한 구현물을 작성한다.

기본 구현물은 266cpb로 구현되었고, 해당 구현물에 제안 기법을 적용한 Fixed-Gimli는 248cpb의 성능을 지닌다. 제안하는 기법은 기존 Tiny 기법에 비해 약 40% 향상된 속도를 지니나, Fast 구현물에 비해 느린 성능을 보인다.

이는 기존 구현물이 AVR Assembly만을 사용한 것이 아닌, Python 코드를 사용하여 Swap을 진행하기 때문에 이와 관련한 부분에서 동작 속도의 이득이 발생한다. 따라서 이와 완전한 비교는 다소 어렵다. 기존 Fast 구현물을 토대로 AVR Assembly만을 사용한 기본 구현물과 비교한다면 제안하는 기법이 성능이 우수한 것을 확인할 수 있다.

Table. 1. Comparison result table (Unit: cpb)

[2] Tiny	[2] Fast	Proposed Normal	**Proposed Fixed**
413	213	266	**248**

V. 결론

본 논문에서는 Gimli Permutation의 Swap 구조를 변형하여, 실제적인 값의 이동은 발생하지 않는 Fixed-Gimli Permutation을 제안하였다. 제안 기법은 Swap 연산이 생략된 만큼 성능 향상을 확인할 수 있었다. 이후로 AVR 환경의 특성을 더욱 활용하여 추가적인 개선을 진행하는 것을 후속 연구로 제시한다.

VI. Acknowledgment

이 성과는 2020년도 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. NRF-2020R1F1A1048478).

[참고문헌]

[1] Kwon H.D. et al. “Designing a CHAM Block Cipher on Low-End Microcontrollers for Internet of Things,” *Electronics*, Sep, 2020.

[2] Bernstein D.J. et al., “Gimli : A Cross-Platform Permutation,” *Cryptographic Hardware and Embedded Systems*, Aug, 2017.