석사학위논문

RISC-V상에서의 Fixslicing AES CTR 운용 모드 최적화 구현

2023년

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

엄 시 우

석 사 학 위 논 문
지도교수 서화정

RISC-V상에서의 Fixslicing AES CTR 운용 모드 최적화 구현

Optimized Implementation of Fixslicing AES-CTR on RISC-V

2022년 12월 일

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

엄 시 우

석 사 학 위 논 문
지도교수 서화정

RISC-V상에서의 Fixslicing AES CTR 운용 모드 최적화 구현

Optimized Implementation of Fixslicing AES-CTR on RISC-V

위 논문을 공학 석사학위 논문으로 제출함

2022년 12월 일

한 성 대 학 교 대 학 원

I T 융 합 공 학 과

I T 융 합 공 학 전 공

엄 시 우

엄시우의 공학 석사학위 논문을 인준함

2022년 12월 일

심사위원장 최 원 석 (인)

심 사 위 원 구 동 영 (인)

심 사 위 원 서 화 정 (인)

국 문 초 록

RISC-V상에서의 Fixslicing AES CTR 운용 모드 최적화 구현

한 성 대 학 교 대 학 원
I T 융 합 공 학 과
I T 융 합 공 학 전 공
엄 시 우

Fixslicing 기법을 구현된 AES 블록 암호는 기존의 구현 기법인 Bitsliced 기법의 선형 계층에서의 많은 Cycle이 발생하는 문제점을 해결하기 위해 라운드 함수 중 Shiftrows 함수를 생략한 기법이다. Shiftrows 함수를 생략함으로써 Bitsliced 기법 대비 30% 높은 성능을 달성하였다. 본 논문에서는 Fixslicing 기법과 Counter 운용 모드의 Nonce 값이 고정되는 특성을 활용하여 32-bit RISC-V 프로세서 상에서의 Fixslicing AES Counter 운용 모드 최적화 구현을 진행하였다. Nonce 값이 고정됨으로 인해서 AES 블록 암호의 암호화 과정에서 2 라운드의 Shiftrows 함수까지 연산하였을 때, Count 값에 따른 고정된 값이 나오는 것을 조사를 통해 분석하였다. 이 분석을 활용하여 최적화 구현을 위해 사전 연산 기법을 적용하였으며, 사전 연산을 통해 2번의 Addroundkey, 2번의 SubBytes, 1번의 Mixcolumns 연산의 생략이 가능하였다. 코드량이 늘어나는 단점을 보완하기 위해 Shiftrows 함수 연산을 반만 생략하는 Semi-Fixsliced와 완전히 생략한 Fully-Fixsliced 두 구현 모두

에 제안 기법을 적용하여 성능 측정을 진행하였다. 결과적으로 제안하는 사전 연산 기법을 통해 구현하였을 때 하나의 블록을 암호화 하는 비용은 각각 1,345 cpb, 1,283 cpb의 성능을 보여주고 있으며 기존 구현 성능 대비 약 7%, 약 9%의 성능 향상을 확인하였다.

【주요어】 AES 블록암호, Fixslicing, 소프트웨어 구현, RISC-V, 카운터모드

목 차

표 목 차

그 림 목 차

제 1 장 서 론

현대 기술의 발전으로 컴퓨터는 점점 더 빨라지고 많은 연산이 가능해지고 있다. 이로 인해 현재 사용하고 있는 안전했던 암호는 암호화 강도가 약해지고 있다. DES(Data Encryption Standard) 블록 암호는 미국의 데이터 암호화에 표준으로 사용되던 암호화이다. 하지만 컴퓨터의 발전으로 새로운 표준 암호가 필요하게 되었으며 NIST(National Institute of Standards and Technology)에서 새로운 암호화 알고리즘을 공모하였다. 1997년부터 공모가 시작되어 최종적으로 2001년 11월 벨기에 암호학자에 의해 개발된 Rijindael 알고리즘이 AES(Advanced Encryption Standard), 즉 새로운 표준 블록 암호로 선정되어 현재까지 사용중이다.

Bitsliced AES는 함수의 구현이 논리 게이트(AND, XOR, OR 등)로 축소되어 구현되는 환경의 CPU 레지스터 폭만큼 많은 인스턴스를 병렬로 실행하여 처리량을 대폭 향상시킬 수 있는 소프트웨어 구현 기법이다. Bitsliced AES의 효율적인 구현은 Schwabe와 Stoffelen의 구현이 있다. 해당 논문에서는 124cpb(Cycle per bit: 1-bit를 암호화 하는데 필요한 Cycle)의 성능을 보여주고 있다.

CHES'21에서 발표된 새로운 구현 기법인 Fixslicing AES는 기존의 구현 기법인 Bitsliced AES에서 선형 연산 함수 구현에 필요한 연산량을 이론상으로 약 52% 감소시킨 효율적인 구현 기법이다. RISC-V 프로세서 상에서 87cpb의 성능을 보여주고 있으며, 이는 Bitsliced AES의 구현보다 약 30%의 성능 향상을 보여주고 있다.

본 논문에서는 32-bit RISC-V상에서 AES 블록 암호 Counter 운용 모드 최적화 구현을 진행한다. 이때 AES 블록 암호 구현에는 Fixslicing 구현 기법을 활용하며, Counter 운용 모드의 특성을 활용하여 사전 연산을 통한 속도 최적화 구현을 진행한다. 본 논문의 구성은 다음과 같다. 제 2장에서는 AES 블록 암호, Bitsliced AES 기법, Fixslicing AES 기법, Counter 운용 모드 그리고 RISC-V 프로세서에 대해 설명한다. 제 3장에서는 본 논문에서 제안하는 구현 기법에 대해 설명한다. 제 4장에서는 32-bit 프로세서 상에서 제안

기법을 활용한 AES CTR 운용 모드의 성능을 측정하고, 성능 평가를 진행한다. 마지막으로 제 5장에서는 본 논문의 결론을 내린다.

제 2 장 관 련 연 구

제 1 절 AES 블록 암호

NIST에서는 1977년 DES를 표준으로 지정하여 사용하였다. 하지만 기술의 발전으로 인해 DES가 안전하지 않은 암호가 되면서 NIST에서는 새로운 표준 암호를 지정하기 위한 공모를 진행하였다. 1997년부터 공모가 시작되었으며, 최종적으로 2001년 11월 Rijindael 알고리즘을 FIPS 197, AES라는 이름으로 새로운 표준 암호로 지정되었다.

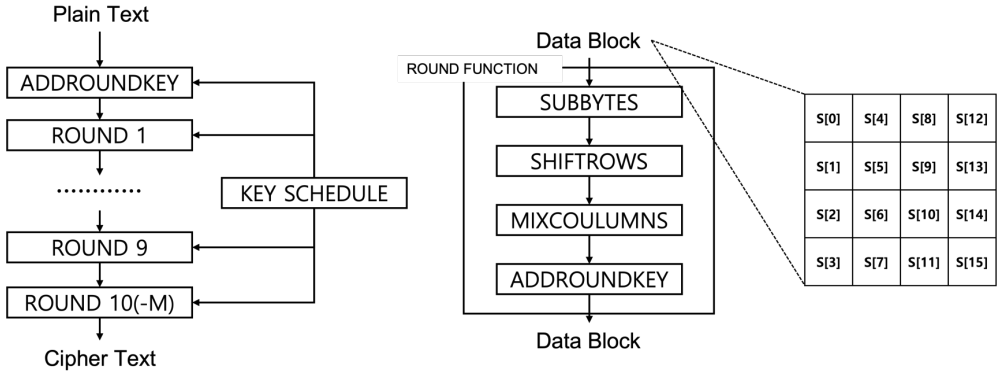
AES는 128-bit 블록 길이를 지원하며, 128, 192, 256-bit 세 개의 키 길이를 지원한다. 각 키 길이에 따라서 10, 12, 14 라운드를 실행한다. 라운드 함수로는 SubBytes, ShiftRows, MixColumns, AddRoundKey로 암호화가 진행된다. AES는 16개의 Byte로 나뉘어 4×4 행렬 상태로 라운드가 진행되며 전체적인 알고리즘 구조는 [그림 2-1]과 같다.

SubBytes 함수는 S-Box를 활용한 치환 연산을 진행한다. S-Box는 사전에 연산되어 있는 256-byte의 크기를 갖는 테이블이다. 해당 테이블을 활용하여 바이트 단위로 쪼개진 상태 값이 입력으로 사용되어 테이블에 저장된 값으로 치환된다.

ShiftRows 함수는 각 행마다 Shift 연산을 진행한다. 각 행마다 왼쪽으로 0, 1, 2, 3번 Shift를 통해 각 행의 열의 위치를 섞어주는 함수이다. 4×4 행렬 상태로 암호화가 진행되는 AES에서 ShiftRows 함수를 진행하지 않는다면 4개의 블록을 암호화 하는 것과 동일하다.

MixColumns 함수는 4×4 행렬 곱을 진행한다. 행렬 곱은 $GF(2^8)$상에서 연산되며, MixColumns 함수는 마지막 라운드에서 연산을 진행 하지 않는다.

AddRoundKey 함수는 키 확산 알고리즘을 통해 연산된 라운드 키와 연산하는 함수이다. 각 라운드마다 서로 다른 라운드 키가 사용되며, 다른 라운드 함수에와 다르게 라운드 함수 시작 전에 한 번 더 연산된다. 따라서 키 확산 알고리즘은 입력된 키를 11개의 확장된 라운드 키를 생성한다.



[그림 2-1] AES 블록 암호 알고리즘과 라운드 함수 그리고 State 상태

제 2 절 Bitsliced AES

Bitslice 구현 기법은 효율적인 소프트웨어 구현 기법이다. Bitslice 기법에는 XOR, AND, OR 및 NOT 논리 게이트를 사용하여 알고리즘을 일련의 논리 비트 연산으로 변환하는 작업이 포함된다. Bitslice 구현 기법의 장점 중 하나로는 처리량 증가가 있다. 만약 n-bit 레지스터 크기를 가지는 마이크로프로세서에서 구현될 때 레지스터의 각 bit는 서로 다른 블록의 값으로 사용되어 n개 블록의 병렬 암호화가 가능하며, 결과적으로 처리량이 증가한다.

또한 기존의 AES 블록 암호의 SubBytes 함수에서는 S-Box를 사용하여 메모리에 저장된 값을 치환하는 방법을 통해 빠른 속도로 암호화가 가능하다. 이때 메모리 접근 패턴으로 인해 암호화 분석에 취약한 단점이 있다. 하지만 Bitslice 구현 기법에서는 논리적 연산만으로 SubBytes 함수를 구현하기 때문에 테이블을 사용하지 않는다. 때문에 Constant time 구현으로 메모리 접근으로 인한 패턴이 없으며 입력 값에 상관없이 일정한 시간으로 동작 가능하다. 결과적으로 S-Box 테이블을 사용하여 구현하였을 때 취약한 캐시 타이밍 공격으로부터 안전한 장점이 있다.

b_{63}^{0}	···	b_{4}^{0}	b_{3}^{0}	b_{2}^{0}	b_{1}^{0}	b_{0}^{0}
b_{127}^{0}	···	b_{68}^{0}	b_{67}^{0}	b_{66}^{0}	b_{65}^{0}	b_{64}^{0}
b_{63}^{1}	···	b_{4}^{1}	b_{3}^{1}	b_{2}^{1}	b_{1}^{1}	b_{0}^{1}
b_{127}^{1}	···	b_{68}^{1}	b_{67}^{1}	b_{66}^{1}	b_{65}^{1}	b_{64}^{1}
b_{63}^{2}	···	b_{4}^{2}	b_{3}^{2}	b_{2}^{2}	b_{1}^{2}	b_{0}^{2}
b_{127}^{2}	···	b_{68}^{2}	b_{67}^{2}	b_{66}^{2}	b_{65}^{2}	b_{64}^{2}
b_{63}^{3}	···	b_{4}^{3}	b_{3}^{3}	b_{2}^{3}	b_{1}^{3}	b_{0}^{3}
b_{127}^{3}	···	b_{68}^{3}	b_{67}^{3}	b_{66}^{3}	b_{65}^{3}	b_{64}^{3}
.	···	.	.	.	.	.
.	···	.	.	.	.	.
.	···	.	.	.	.	.
.	···	.	.	.	.	.
.	···	.	.	.	.	.
b_{63}^{63}	···	b_{4}^{63}	b_{3}^{63}	b_{2}^{63}	b_{1}^{63}	b_{0}^{63}
b_{127}^{63}	···	b_{68}^{63}	b_{67}^{63}	b_{66}^{63}	b_{65}^{63}	b_{64}^{63}

b_{0}^{63}	···	b_{0}^{4}	b_{0}^{3}	b_{0}^{2}	b_{0}^{1}	b_{0}^{0}
b_{1}^{63}	···	b_{1}^{4}	b_{1}^{3}	b_{1}^{2}	b_{1}^{1}	b_{1}^{0}
b_{2}^{63}	···	b_{2}^{4}	b_{0}^{3}	b_{0}^{2}	b_{0}^{1}	b_{0}^{0}
b_{3}^{63}	···	b_{3}^{4}	b_{3}^{3}	b_{3}^{2}	b_{3}^{1}	b_{3}^{0}
b_{4}^{63}	···	b_{4}^{4}	b_{4}^{3}	b_{4}^{2}	b_{4}^{1}	b_{4}^{0}
b_{5}^{63}	···	b_{5}^{4}	b_{5}^{3}	b_{5}^{2}	b_{5}^{1}	b_{5}^{0}
b_{6}^{63}	···	b_{6}^{4}	b_{6}^{3}	b_{6}^{2}	b_{6}^{1}	b_{6}^{0}
b_{7}^{63}	···	b_{7}^{4}	b_{7}^{3}	b_{7}^{2}	b_{7}^{1}	b_{7}^{0}
.	···	.	.	.	.	.
.	···	.	.	.	.	.
.	···	.	.	.	.	.
.	···	.	.	.	.	.
.	···	.	.	.	.	.
b_{126}^{63}	···	b_{126}^{4}	b_{126}^{3}	b_{126}^{2}	b_{126}^{1}	b_{126}^{0}
b_{127}^{63}	···	b_{127}^{4}	b_{127}^{3}	b_{127}^{2}	b_{127}^{1}	b_{127}^{0}

[그림 2-2] Bitsliced 기법을 위한 입력된 블록의 Bit 재정렬(64-bit 프로레서 상에서의 예, b의 위첨자: 블록 번호, b의 아래첨자: bit 순서)

[그림2-2]에서 볼 수 있듯이 Bitsliced 기법을 적용하기 위해서는 입력된 블록의 재정렬이 필요하다. [그림 2-2]의 왼쪽 상태는 현재 64개의 블록이 2개의 64-bit 레지스터에 나뉘어 저장되어 있는 상태이다. 다음으로 왼쪽 상태에서 서로 같은 위치의 bit 값들을 하나의 레지스터로 합쳐주는 과정을 통해 재정렬된 상태가 [그림 2-2]의 오른쪽 상태이다. 64개 블록들의 0번째 bit가 맨 위 첫 번째 레지스터에 모여있는 것을 알 수 있다. 다음과 같이 상태를 재정렬 한 다음 암호화 과정을 진행하는 기법이며, 이를 통해 64개의 블록이 병렬 연산이 가능하다.

제 3 절 Fixclicing AES

Fixslicing 구현 기법은 CHES'21에서 소개된 AES 블록 암호의 효율적인 소프트웨어 구현 기법이다. 해당 논문에서는 Bitsliced AES 구현 기법으로 구현 시 Shiftrows 함수에서 47.5cpb가 발생하며, 이는 전체 성능에 약 35%에 해당하는 것을 분석을 통해 발견하였다. 따라서 기존의 구현 기법인 Bitsliced AES에서 선형 연산의 연산량을 줄이기 위하여 Shiftros 함수 연산을 생략하는 방법을 사용한다. [표 2-1]은 Shiftrows의 생략을 통해 감소하는 연산의 수를 보여주고 있다.

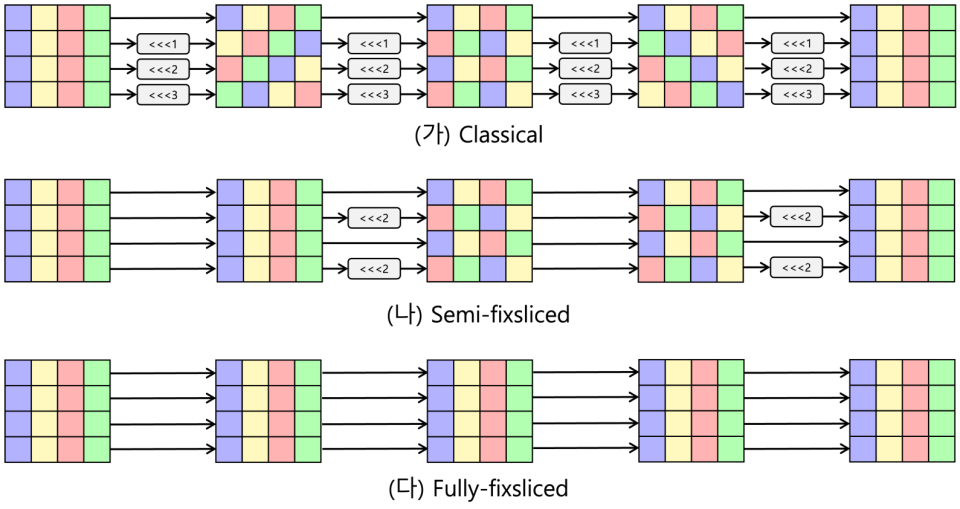
Method	Number of Operations per Linear layer												
	Round 0			Round 1			Round 2			Round 3			Sum
	LOP	LSH	ROT	LOP	LSH	ROT	LOP	LSH	ROT	LOP	LSH	ROT	
Bitsliced	131	48	16	131	48	16	131	48	16	131	48	16	780
Fixsliced	75	32	16	51	16	16	75	32	16	27	0	16	372

[표 2-1] 4 라운드 동안의 선형 연산의 수(LOP: Logical Operation, LSH: Logical Shifts, ROT: Rotations)

Shiftrows 함수의 생략으로 인해 4×4 상태로 암호화를 진행하는 상태 블록이 [그림 2-3]의 (다)와 같이 고정된 상태로 암호화가 진행된다. Shiftrows 함수가 생략되기 이전에는 [그림 2-3]의 (가)에서 볼 수 있듯이 4개의 상태가 존재하게 된다. 이때 Mixcolumns 함수의 경우 열 단위의 확산이 이루어지기 때문에 기존의 구현을 그대로 적용하게 된다면 잘못된 연산으로 잘못된 암호화 값을 생성하게 된다. 이로 인해 각 상태에 맞는 Mixcolumns 함수의 4개의 구현이 필요하며 이로 인해 코드량이 증가하여 성능과 코드량의 trade-off가 발생한다. 이를 위해 [그림 2-3]의 (나)와 (다)에서 볼 수 있듯이 Semi-Fixsliced과 Fully-Fixsliced 두 개의 방법을 제시하고 있으며, 두 구현

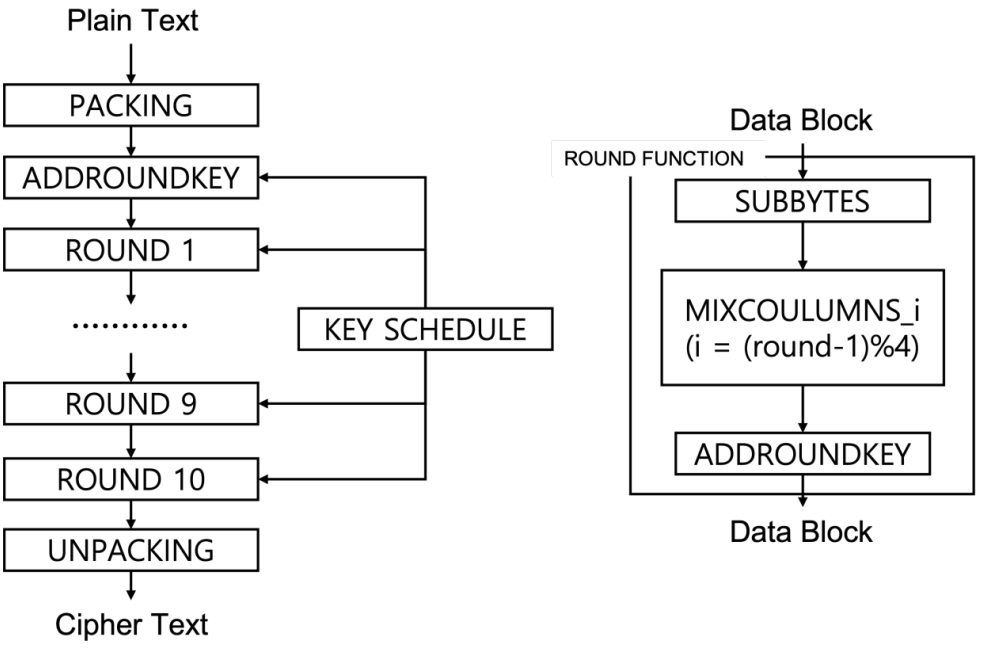
방법의 차이는 Fully 방법의 경우 Shiftrows 함수를 완전히 생략한 방법이라면 Semi 방법은 Shiftrows의 동작을 반으로 줄인 방법이다.

추가적으로 Fixslicing 기법은 Bitsliced 기법에서 개선된 방법이기 때문에 Bit 단위로 정렬하는 과정인 Packing 과정이 추가된다. 또한 한 번의 암호화를 통해 2개의 블록을 병렬 암호화가 가능하다. 전체적인 Fixslicing AES 알고리즘은 [그림 2-4]와 같다. 본 논문에서는 Fixslicing 기법을 제안한 논문에서 오픈 소스로 제공하고 있는 Fixslicing AES 구현 코드[1]를 사용하여 제안 기법을 적용하였다.



[그림 2-3] Shiftrows 함수 연산에 따른 AES 블록의 상태 변화

1) https://github.com/aadomn/aes



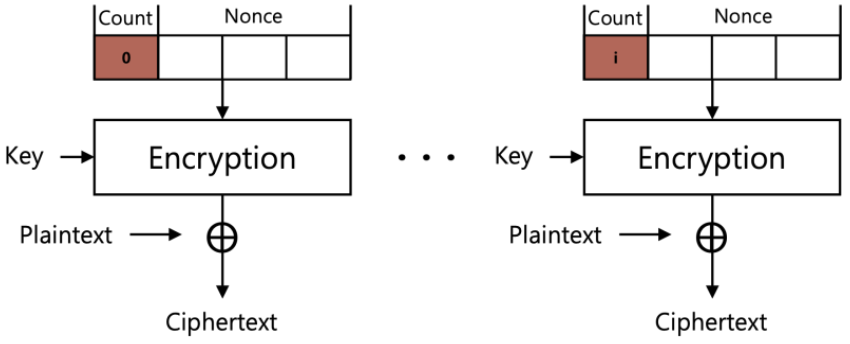
[그림 2-4] 전체적인 Fixslicing AES 알고리즘

제 4 절 블록 암호 운용 모드 : Counter 운용 모드

블록 암호 운용 모드(Block cipher modes of Operation)은 암호학에서 하나의 키를 사용할 때 블록 암호를 반복적으로 안전하게 이용하기 위한 절차를 말한다. 블록 암호의 특성상 입력된 데이터를 블록 길이에 맞추어 나눈 후 암호화가 진행된다. 이 후 나뉜 블록들을 어떤 절차로 암호화를 진행할지 정해야 하는데 이를 블록 암호 운용 방식이라 한다.

블록 암호 운용 모드에는 여러 방법이 있다. 그 중 Counter 운용 모드는 블록 암호를 스트림 암호로 바꾸는 구조를 가지고 있는 운용 모드이다. CTR 운용 모드는 1979년 Diffie와 Hellman에 의해 도입되었다. CTR 운용 모드는

입력된 데이터 블록을 암호화하지 않고 고정된 상수 Nonce 값과 값이 변하는 변수 Counter 값이 합쳐진 Initialization Vector(IV, Nonce+Counter) 값을 암호화한다. 암호화된 IV 값은 최종적으로 평문 블록과 XOR되어 최종 암호화 값이 연산된다, CTR 운용 모드의 구조는 [그림 2-5]와 같다.



[그림 2-5] Counter 운용 모드의 동작 구조

CTR 운용 모드는 하드웨어와 소프트웨어에서 효율적인 구현이 가능하다. CTR 운용 모드의 특성상 특정 위치의 Count 값을 알 수 있다면 해당 위치의 값만 암호화하거나 복호화 하는 것이 가능하다. 또한 Nonce 값이 고정되는 특서을 활용하여 사전 연산이 가능할 수 있으며, 사전 연산으로 인해 빠른 암호화나 복호화가 가능해진다.

제 5 절 32-bit RISC-V 프로세서

2010년부터 캘리포니아 대학교 버클리에서 개발중인 무료로 제공되는 오픈 소스 RISC 명령어셋 아키텍쳐이다. 현재 스마트폰이나 임베디드 장치의 CPU로 많이 사용되고 있는 ARM 아키텍쳐와 경쟁하여 이를 대체할 수 있는 고성능의 CPU를 개발하는 것을 목표로 하고 있다. RISC-V는 RV32I,

RV64I 그리고 RV128I 등 기본적인 명령어 집합이 있다. 각각은 32-bit, 64-bit, 128-bit를 지원하며 본 논문에서는 RV32I 명령어 집합을 사용하는 프로세서를 활용하여 구현을 진행한다. RV32I는 32-bit 레지스터를 사용하며 32개의 레지스터를 제공한다. [표 2-2]는 제공되는 32개의 레지스터 각각의 용도에 대해 설명한다.

먼저 x0 레지스터에 해당하는 Zero 레지스터는 항상 0의 값을 가지고 있는 레지스터를 의미한다. x1 레지스터는 ra(return address)로 반환될 주소값을 가지고 있는 레지스터이다. x2 레지스터는 sp(Stack Pointer) 레지스터로 스택 주소를 저장하고 있는 레지스터이다. 해당 레지스터는 스택 메모리에 값을 저장하거나 불러올 때 사용할 수 있다. a0~a7 레지스터는 함수 인자를 저장하거나 반환 값(a0~a1)을 저장해 놓는 레지스터로 사용된다. 예를 들어 특정 함수에서 3개의 인자가 필요하다면 순서대로 a0, a1, a2 레지스터에 저장된다. 또한 해당 함수에서 반환되어야 할 값이 있다면 반환되어야 할 값을 a0, a1에 저장하여 반환할 수 있다. s0~s11 레지스터는 Callee 레지스터이다. Callee 레지스터는 해당 레지스터 저장된 값이 변경되게 되면 프로그램 동작 중에 에러가 발생할 수 있는 레지스터이다. 만약 Callee 레지스터를 사용해야 하는 경우 해당 레지스터에 저장된 값을 다른 레지스터나 메모리에 저장하여 값을 보존하고, 사용 후 다시 원래의 값으로 되돌려 놓아야 한다. 즉, 특정 함수 구현에 Callee 레지스터를 사용하였다면 함수가 끝나기 전에 함수 구현에 사용되기 전 원래의 값으로 되돌려주어야 한다. Callee 레지스터에는 s0~s11 레지스터 외에 sp 레지스터도 포함된다. 마지막을 t0~t6 레지스터는 임시 레지스터로 함수 구현에서 편하게 사용할 수 있는 레지스터이다.

Register	Description	Saver
ZERO [x0]	Zero Register	-
ra [x1]	Return Address	Caller
sp [x2]	Stack Pointer	Callee
gp [x3]	Global Pointer	-
tp [x4]	Thread Pointer	-
t0 [x5]	Temporary / Alternate lnk Register	Caller
t1-t2 [x6-7]	Temporaries register	Caller
s0/fp [x8]	Saved Register /Frame Pointer	Callee
s1 [x9]	Saved Register	Callee
a0-a1 [x10-11]	Function Arguments / Return values	Caller
a2-a7 [x12-17]	Function Arguments	Caller
s2-s11 [x18-27]	Saved Registers	Callee
t3-t6 [x28-31]	Temporaries Registers	Caller

[표 2-2] RISC-V의 정수 레지스터 구성과 용도

RV32I에서는 기본적인 연산에 필요한 명령어를 제공하고 있으며, 제공되는 명령어 중 본 논문에서 사용되는 명령어는 [표 2-3]과 같다.

Instruction	Description
ADD rd, rs1, rs2	ADD Immediate
XOR rd, rs1, rs2	XOR Immediate
SLL rd, rs1, rs2	Shift Left Logical
SRL rd, rs1, rs2	Shift Right Logical
BNE rs1, rs2, imm13	Branch Not Equal
LB rd, imm12(rs12)	Load Byte
SB rs2, imm12(rs1)	Store Byte
JAL rd, imm21	Jump And Link

[표 2-3] RISC-V의 기본적인 명령어

[표 2-3]에서 rd는 Destination 레지스터를 의미하고, rs는 Source 레지스터를 의미한다. 예를 들어 (Add rd, rs1, rs2)는 (rs1 + rs2 = rd)를 의미하는 코드이다. 즉 rs는 값을 제공하는 레지스터를 의미하고 rd는 연산된 값이 저장될 레지스터를 의미한다.

제 3 장 구 현 기 법

본 논문에서는 32-bit RISC-V 프로세서 상에서 Fixslicing 기법을 활용한 AES 블록 암호의 CTR 운용 모드 최적화 구현을 진행한다. 최적화 구현을 위해서 CTR 운용 모드(이하 카운터 모드)의 특성을 활용한다. 카운터 모드에서는 Nonce와 Count 값을 암호화하게 된다. 상수인 Nonce 값은 고정된 값을 사용하게 되고 변수인 Count 값을 통해 블록 암호를 스트림 암호처럼 사용 가능하다. 이때 Nonce가 고정된 값을 사용하는 특성으로 인하여 특정 블록 암호 알고리즘에서는 특정 라운드까지 고정된 값으로 연산되게 된다. 이러한 특성을 활용하여 최적화 구현을 진행한다.

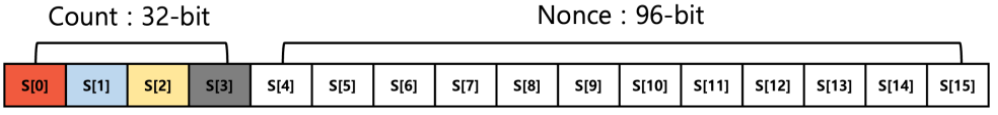
본 장에서는 Nonce의 특성을 활용한 사전 연산 기법에 대해 설명하고 이를 활용한 사전 연산 테이블을 최적화된 방법으로 구현하는 방법에 대해 설명한다. 마지막으로 생성된 사전 연산 테이블을 활용하여 최적화 구현된 Fixslicing AES-CTR 구현에 대해 설명한다.

제 1 절 사전 연산 기법

카운터 모드에서는 Nonce 값을 고정된 값으로 사용하는 특성이 있다. 이로 인하여 연산 과정에서 다른 값들도 모두 고정일 경우 연산 결과가 항상 동일하게 나오게 된다. 이때 다른 값에 속하는 값으로는 입력되는 키, Count 값이 있을 수 있다. 연산 결과가 항상 동일하게 나오게 된다면 암호화를 하기 전에 입력되는 모든 값에 대하여 연산 결과를 미리 연산하여 사용할 수 있다. 이 경우 입력되는 값에 따라서 미리 연산된 값을 불러와 다음 연산을 진행할 수 있으며 이를 통해 빠른 암호화가 가능하다. 본 논문에서는 이를 사전 연산 기법이라 칭한다.

사전 연산 기법을 활용하기 위해 먼저 AES 블록 암호의 확산되는 과정을 알아본다. AES 블록 암호의 블록 길이는 128-bit이다. [그림 3-1]과 같이 본 논문에서 IV값은 96-bit 길이의 Nonce와 32-bit 길이의 Count 값으로 정의

한다.



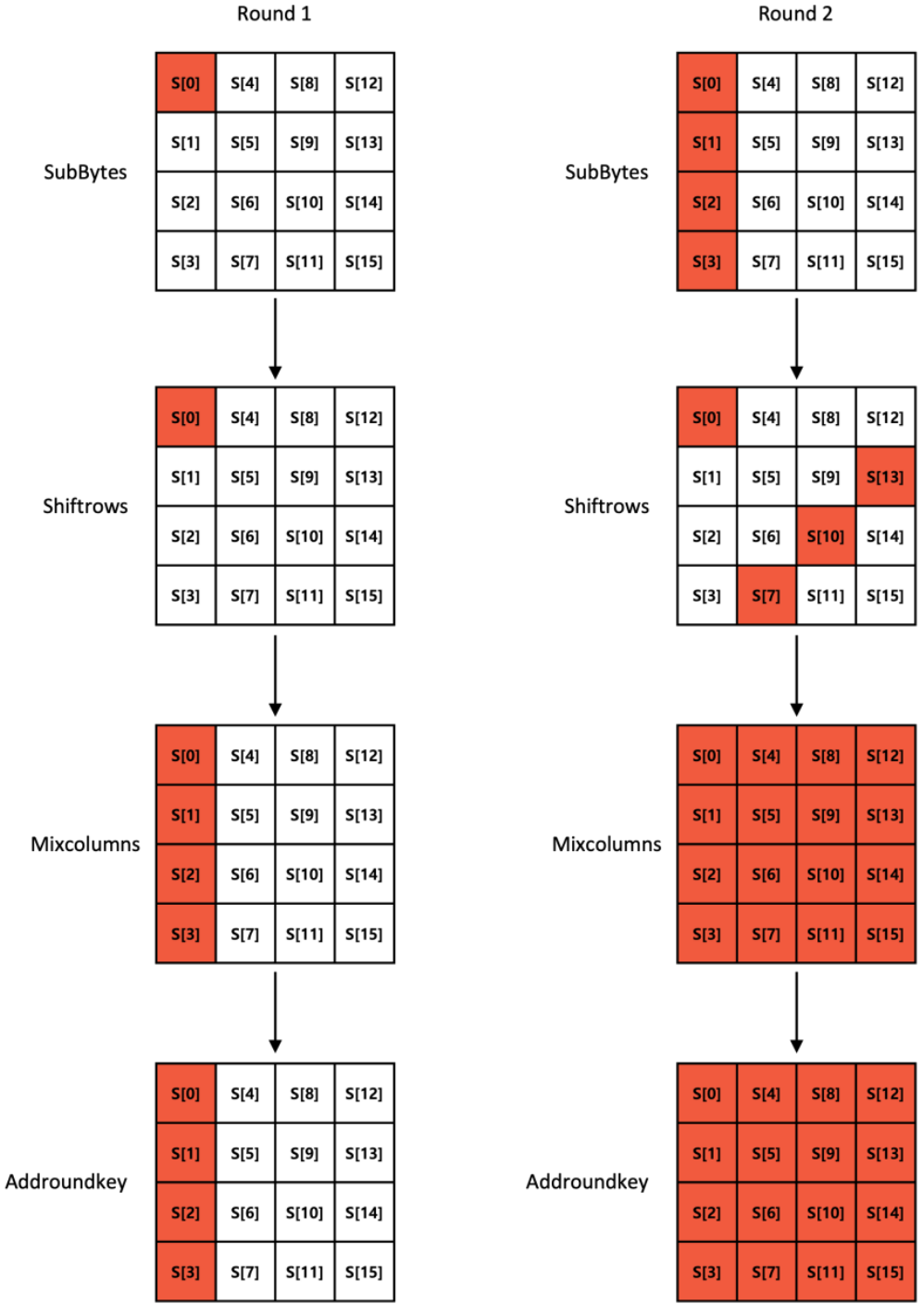
[그림 3-1] IV(Count+Nonce)의 구조

Count 값에 해당하는 S[0], S[1], S[2], S[3]은 각각 8-bit의 크기를 가지고 있다. 8-bit의 크기를 갖는 각 State는 0~255까지의 값을 가질 수 있다. 하나의 State의 값(8-bit)만 변경하고 나머지 248-bit의 값은 고정된 값으로 사용하면 해당 State로 인해 영향 받는 State를 확인할 수 있다. [그림 3-2]은 S[0]의 값만 변경하였을 때 영향 받는 State를 보여준다.

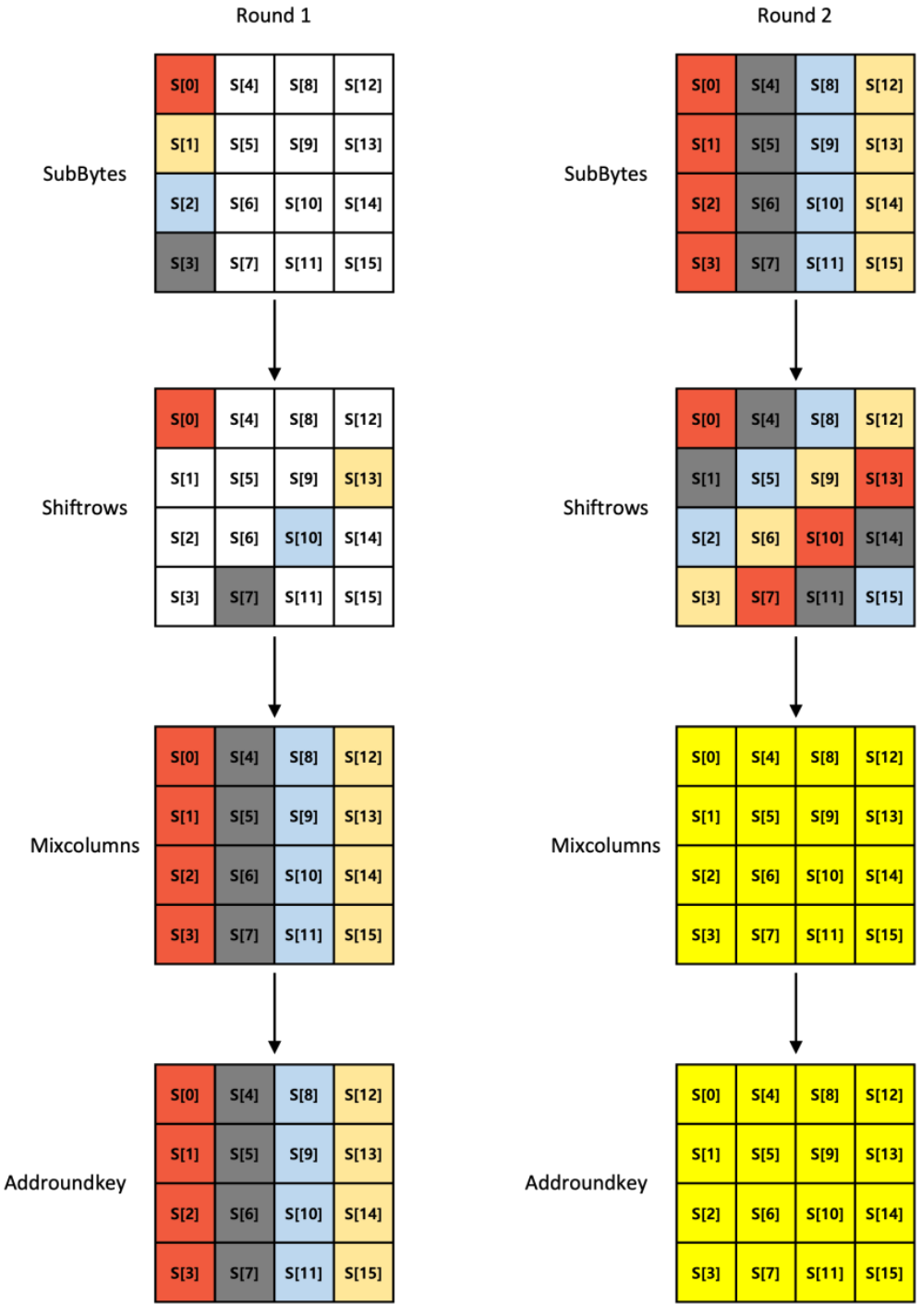
[그림 3-2]에서 색이 칠해진 State는 처음 입력되는 S[0]의 값이 변경되었을 때 값이 바뀌는 State를 의미한다. 즉, S[0]의 값이 바뀌었을 때, 색이 칠해지지 않은 State는 S[0]의 값과 상관없이 고정된 값으로 유지된다. 이를 통해서 AES 블록 암호의 경우 2 라운드의 Shiftrows 함수까지 연산하였을 때까지는 S[0]의 값이 바뀌었을 때 특정 State에만 영향을 주는 것을 알 수 있으며 2 라운드의 Mixcolumns 함수를 연산하였을 때 전체적인 확산이 이루어지는 것을 확인할 수 있다. 다음으로 [그림 3-3]은 Count값이 영향을 미치는 State를 보여준다.

[그림 3-3]에서 Count 값에 해당하는 S[0], S[1], S[2], S[3]은 각각 다른 색상으로 채워져 있으며, 각각의 State는 2 라운드의 Shiftrows 함수 연산까지는 서로 영향을 주지 않는 것을 확인할 수 있다. 전체적인 확산은 이전에 확인한 것과 동일하게 2 라운드의 Mixcolumns 함수 연산에서 이루어 졌다. 이를 통해 2 라운드의 Shiftrows 함수 연산까지 사전 연산이 가능하다.

결과적으로 Count 값에 해당하는 S[0], S[1], S[2], S[3]의 값을 0~255로 변경해가며 2 라운드의 Shiftrows 함수까지 연산한 값을 저장하고, 입력된 값에 따라서 사전에 연산된 저장된 값을 불러와 2 라운드의 Mixcolumns 함수



[그림 3-2] S[0]의 값이 AES 암호화에서 확산이 이루어지는 과정



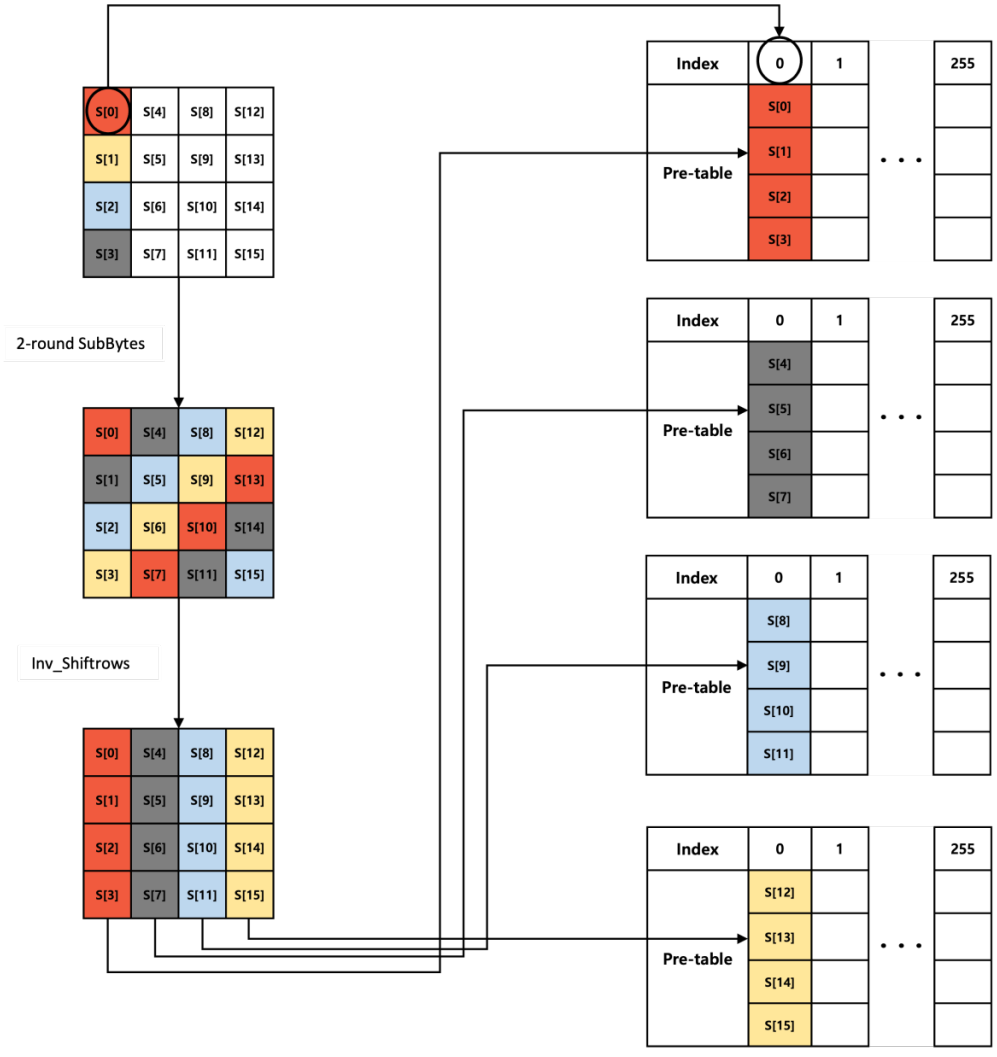
[그림 3-3] 전체 Count 값(S[0-3])이 AES 암호화에서 확산이 이루어지는 과정

부터 연산을 진행함으로써 앞의 연산이 생략된 빠른 암호화가 가능하다.

제 2 절 사전 연산 테이블 생성 최적화 구현

사전 연산 테이블을 생성하기 위해서 Count 값에 해당하는 S[0~3]에 0~255까지의 값을 넣어가며 2라운드의 Shiftrows 함수까지 연산해야 한다. [그림 3-3]에서 볼 수 있듯이 2라운드의 Shiftrows 함수까지 연산한 결과에서 Count 값 S[0]에 영향 받는 State의 인덱스는 S[0], S[7], S[10], S[13]이다. 해당 값들은 서로 다른 열에 저장되어 각각의 값을 저장하기 위해서는 Byte 단위로 메모리에 저장해주어야 한다. 이는 많은 메모리 접근을 발생하기 때문에 속도 측면에서 비효율적이다. 따라서 2 라운드의 SubBytes 연산까지 진행된 모습의 State상태가 테이블에 저장하기 효율적인 상태이다. 각 Count 값에 영향을 받는 State 값이 하나의 열에 정렬되어 있어 4번의 메모리 접근으로 사전 연산된 값을 저장할 수 있다.

이때 기존의 AES 구현의 경우 2 라운드의 SubBytes 함수까지 연산하고 저장하는 방법으로 사전 연산 테이블을 생성할 수 있다. 하지만 Fixslicing 기법으로 구현된 경우 라운드 함수 구현이 Shiftrows를 생략하고 Mixcolumns 함수에서 대체하여 구현되기 때문에 1 라운드의 Mixcolumns 함수 연산시 이미 2 라운드의 Shiftrows 함수까지 연산했을 때의 상태이다. 따라서 2 라운드 Mixcolumns 함수 연산전까지 연산된 값에 Inverse Shiftrows 함수 연산을 추가적으로 연산하면 2 라운드 SubBytes 함수까지 연산했을 때의 상태로 만들 수 있다. 이때 Packing된 상태로 Inverse Shiftrows 함수를 구현하게 되면 비효율적이기 때문에 UnPacking된 상태로 Inverse Shiftrows 함수를 연산하는 것이 효율적이다. 전체적인 사전 연산 테이블 생성 과정을 [그림 3-4]와 같다.



[그림 3-4] 사전 연산 테이블 생성 과정(S[0]: 0x00 일 때)

사전 연산된 값은 사전 연산 테이블에 저장된다. 이때 사전 연산 테이블의 크기는 Count에 해당하는 4개의 state당 32-bit×256개이다. 즉, 4×256×32 = 32,868-bit = 4,096-byte = 4KB이다. 각 state 당 1KB의 저장 공간을 사용한다. 테이블에 사전 연산된 값을 저장할 때 state의 값은 테이블에 인덱스로 사용되어 저장된다. 예를 들어 S[0]의 사전 연산된 값을 저장하는 테이블 이름이 S0_table[256] 이라고 한다면, S[0] = 4일 때의 사전 연산된 값이 저장되는 인덱스는 S0_table[4]가 되고 해당 위치에 사전 연산된 32-bit의 값이 저장된다.

추가적으로 사전 연산 테이블을 생성할 때 반복 횟수는 Count 값이 32-bit 크기이기 때문에 32-bit로 표현 가능한 부호가 없는 정수 크기인 4,294,967,295번 반복하는 것이 아니라 각 Count state가 영향을 주는 상태 State가 독립적이기 때문에 S[0~3]의 값을 같이 증가시켜 가면서 사전 연산이 가능하다. 다시 말해 count에 해당하는 S[0]와 S[1]은 사전 연산의 값에 서로 영향을 주지 않는다. 따라서 0~255의 값을 4개의 State 동시에 넣어주어 연산함으로써 반복 횟수를 줄일 수 있다. 이로 인해 반복 횟수가 256번으로 감소한다.

또한 Fixslicing 기법으로 사전 연산을 진행하기 때문에 두 개의 블록을 동시에 사전 연산할 수 있다. 결과적으로 128번의 반복을 통해 사전 연산 테이블을 생성할 수 있다. 전체적인 알고리즘은 [표 3-1]에서 확인할 수 있다.

Input : IV_1(Counter+Nonce), IV_2, RK(RoundKey), Pre_table[4][256]
Output : Pre_table[4][256]

```
1   For i = 0 to 127
2     IV_1[0] = IV_1[1] = IV_1[2] = IV_1[3] = i;
3     IV_2[0] = IV_2[1] = IV_2[2] = IV_2[3] = i+1;
4     Packing(IV_1, IV_2, State);
5     Addrounkey(State, RK);
6     SubBytes(State);
7     MixColumns_0(State);
8     Addrounkey(State, RK);
9     SubBytes(State);
10    Unpacking(State, Out_1, Out_2);
11    Inv_Shiftrow(Out_1, Out_2);
12    Store_Pretable(Out_1, Out_2, Pre_table);
13  End For
14  return Pre_table;
```

[표 3-1] 사전 연산 테이블 생성 과정의 의사코드

제 3 절 사전 연산 테이블을 활용한 최적화 구현

사전 연산 테이블을 활용한 암호화 과정은 다음과 같다.

사전 준비 단계에서는 먼저 입력 받은 키를 활용하여 키 확장 알고리즘을 통해 라운드 키를 생성한다. 생성된 라운드 키와 사전에 정의된 Nonce 값을 활용하여 사전 연산 테이블을 생성한다. 이 과정은 암호화가 진행되기 전에 사전에 준비되어 있어야 하는 과정이다. 이후 암호화 과정은 다음과 같다.

첫 번째 단계는 입력되는 데이터가 암호 알고리즘에 맞는 블록 길이에 맞춰 블록이 나뉘게 된다. AES 알고리즘의 경우 블록 길이가 128-bit 이기 때문에 128-bit 길이에 맞춰 나뉜다.

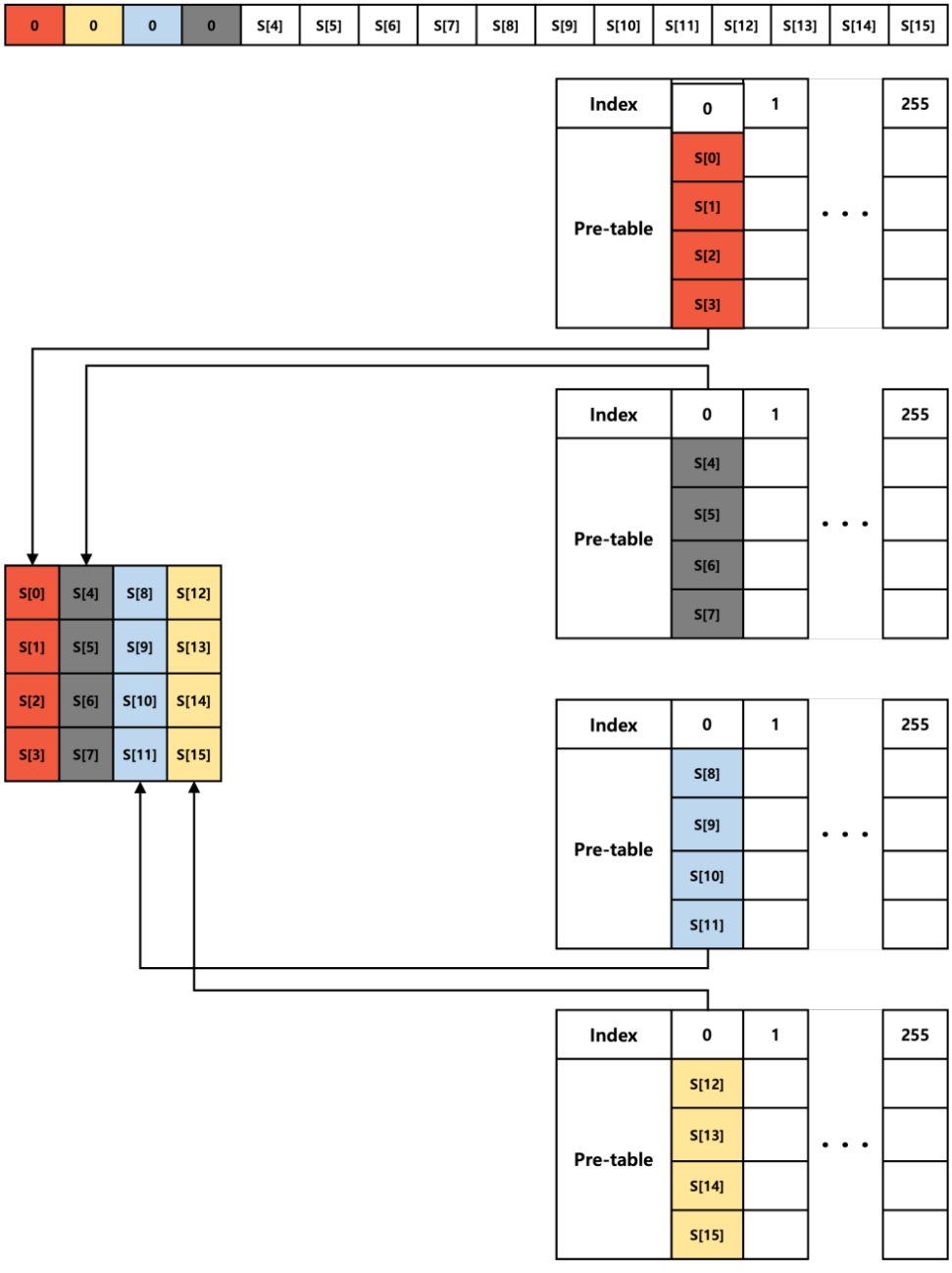
두 번째 단계는 나뉜 블록 수에 맞춰 Count 값이 정해지게 되고 각 블록은 순서에 맞게 서로 다른 Count 값이 정해지게 된다.

세 번째 단계는 평문 블록이 입력 값으로 사용되는 것이 아니라 순서에 맞게 정해진 Count 값이 암호화 함수의 입력 값으로 사용된다. 입력으로 받은 Count 값은 사전 연산된 값을 불러오는 인덱스 값으로 사용되어 해당 주소에 있는 사전 연산된 값을 불러온다. 이 과정은 [그림 3-5]와 같다.

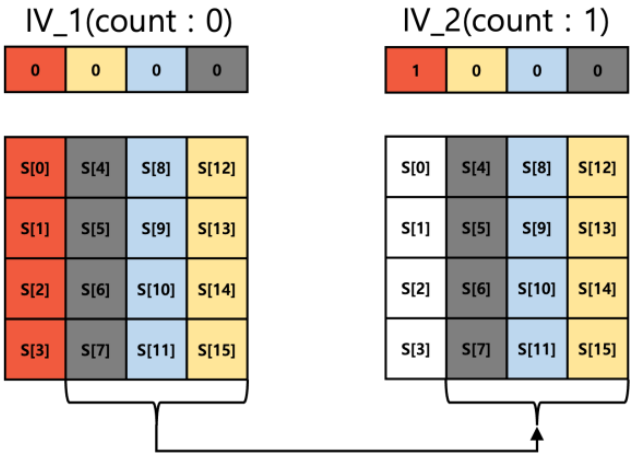
이때 Fixslicing 기법으로 구현할 경우 두 개의 블록이 병렬 구현되기 때문에 하나의 블록을 더 불러와야 한다. 하지만 두 블록의 차이는 Count 값이 1이 증가한 차이만 존재한다. 따라서 S[0~3]에서 S[0]의 값만 차이가 있다. 따라서 첫 번째 블록의 S[1~3]의 사전 연산 값으로 불러온 state를 그대로 복사하여 사용할 수 있다. 만약 복사하는 방법이 아니라 메모리에서 값을 똑같이 불러오는 경우 메모리 접근이 많아지기 때문에 성능에 영향을 줄 수 있다. 이는 [그림 3-6]과 같은 과정으로 진행된다.

네 번째 단계는 사전 연산된 값을 불러왔을 때의 State의 상태는 2 라운드의 SubBytes 함수까지 연산된 상태이다. 암호화가 진행되기 위해서는 2 라운드의 Shiftrows 함수까지 연산된 상태로 진행되어야 하기 때문에 Shiftrows 함수 연산을 한 번 진행해야 한다. 이 과정은 Packing 함수가 연산되기 전에 진행한다. 이로써 암호화 과정을 진행하기 위한 준비가 완료되고 사전 연산되지 못한 나머지 연산 과정을 진행함으로써 암호화가 완료된다. 전체적인 알고

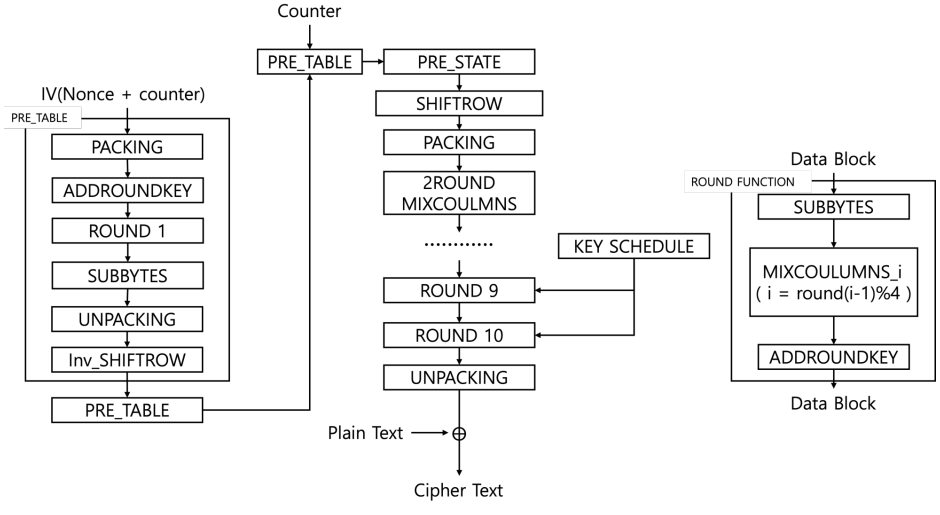
리즘 구조는 [그림 3-7]과 같다.



[그림 3-5] 사전 연산 테이블에서 값을 가져오는 과정(Count : 0x00일 때)



[그림 3-6] 사전 연산된 값을 불러올 때 첫 번째 블록에서 두 번째 블록으로 값을 복사하는 과정



[그림 3-7] 제안 기법이 적용된 Fixslicing AES-CTR의 전체적인 알고리즘

s0, s2, s4, s6 : IV_1 register			
s1, s3, s5, s7 : IV_2 register			
a1 : count value			
a4 : pretable address			
t0-1 : temp register			
	// IV_1의 값 복사		// Shiftrows 연산
1	mv s3, s2	7	srli t0, s3, 24
2	mv s5, s4	8	slli t1, s3, 8
3	mv s7, s6	9	or s3, t0, t1
	// count 값 증가	10	srli t0, s5, 16
4	add t0, t0, 0x01	11	slli t1, s5, 16
	// 사전 연산 값 불러오기	12	or s5, t0, t1
5	add a4, a4, t0	13	srli t0, s7, 8
6	lw s1, 0(a4)	14	slli t1, s7, 24
		15	or s7, t0, t1

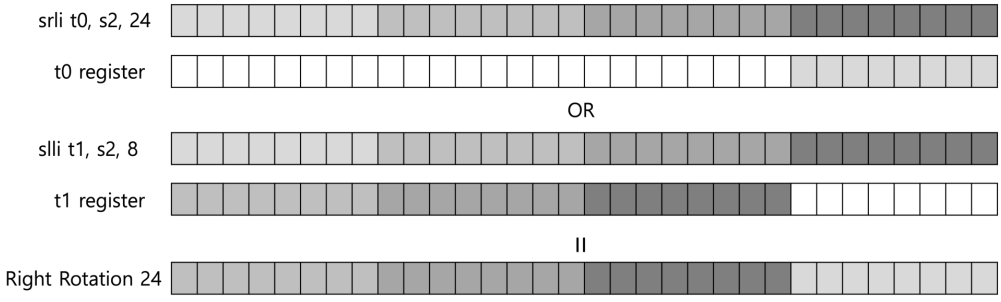
[표 3-2] IV_2 생성 과정 어셈블리 코드

[표 3-2]는 IV_2 생성 과정의 어셈블리 코드이다. 세 번째 단계와 네 번째 단계의 과정으로 IV_1이 생성된 후 효율적으로 IV_2를 초기화 하는 코드이다. 만약 IV_1처럼 IV_2의 생성을 메모리에 접근하여 값을 불러오게 되면 메모리 접근이 증가하기 때문에 비효율적이다. 메모리 접근 방법보다 레지스터 간의 복사 과정이 더 적은 Laytency로 구현할 수 있다. 1-3번 줄의 코드는 IV_1에서 IV_2에 값을 복사하는 과정이다. 이 때 IV_1의 첫 번째 열(s0 레지스터)는 Count 값이 다르기 때문에 서로 다른 값이다. 따라서 복사는 3개의 레지스터만 진행한다. 다음으로 IV_1과 IV_2는 count가 1 차이나기 때문에 Count 값을 가지고 있는 t0 레지스터의 값을 1 증가시킨다. 그 다음 5-6번 줄은 사전 연산 테이블의 주소 값을 가지고 있는 a4의 레지스터에 count 값을 활용하여 주소 값을 변경해준다. 이후 변경된 주소 값을 통해서 사전 연산된 값을 불러온다.

s0, s2, s4, s6 : IV_1 register s1, s3, s5, s7 : IV_2 register t0-1 : temp register			
	// IV_1 Shiftrows 연산		// IV_2 Shiftrows 연산
1	srli t0, s2, 24	10	srli t0, s3, 24
2	slli t1, s2, 8	11	slli t1, s3, 8
3	or s2, t0, t1	12	or s3, t0, t1
4	srli t0, s4, 16	13	srli t0, s5, 16
5	slli t1, s4, 16	14	slli t1, s5, 16
6	or s4, t0, t1	15	or s5, t0, t1
7	srli t0, s6, 8	16	srli t0, s7, 8
8	slli t1, s6, 24	17	slli t1, s7, 24
9	or s6, t0, t1	18	or s7, t0, t1

[표 3-3] 초기화된 IV 값의 Shiftrows 함수 연산 어셈블리 구현 코드

[표 3-3]은 사전 연산된 테이블에서 값을 불러온 다음 다음 연산을 진행하기 위해 Shiftorws 함수 연산을 하는 어셈블리 코드이다. 현재 state 상태는 2 라운드의 SubBytes 함수까지 연산된 상태이므로 Shiftrows 연산을 통해 다음 연산을 진행할 수 있게 State 상태를 맞춰주어야 한다. Shiftrows 함수의 연산은 Rotation 연산을 진행한다. RISC-V에서는 Rotation 명령어를 지원하지 않기 때문에 SRLI, SLLI, OR 명령어를 활용하여 Rotation 연산을 구현한다. 1-9번 줄은 IV_1의 Shiftrows 함수 연산을 의미한다. 먼저 오른쪽으로 24-bit 만큼 쉬프트 연산된 값을 Temp 레지스터에 저장하고 왼쪽으로 8-bit 만큼 쉬프트 연산된 값을 Temp 레지스터에 저장한다. 이후 두 연산된 값을 OR 연산을 해주게 되면 오른쪽을 24-bit 만큼의 Rotation 연산 구현이 완성된다. 이 과정은 [그림 3-6]과 같다. 이후 10-18번 줄은 IV_2의 Shiftrows 함수 연산으로 IV_1과 동일하기 때문에 설명은 생략한다. 이 과정 이후로 Packing 연산을 진행하고 나머지 사전 연산되지 못한 암호화 과정을 진행하게 된다.



[그림 3-8] Right Rotation 24 구현 과정

Input : PT_1(Plain Text), PT_2	
	Counter, RK(Roundkey), Pre_table
Output : Cipher_1, Cipher_2	
1	PreState_init(IV_1, IV_2, Pre_table, Counter);
2	Packing(IV_1, IV_2, State);
3	Mixcolumns_1(State);
4	AddRoundkey(State, RK);
5	**For** i = 3 **to** 10
6	SubBytes(State);
7	Mixcolumns_n(State); // n=(i%4)-1
8	AddRoundkey(State, RK);
9	**End For**
10	Double_Shiftrows(State);
11	Unpacking(State, Cipher_1, Cipher_2);
12	Cipher_1 = PT_1 XOR Cipher_1;
13	Cipher_2 = PT_2 XOR Cipher_2;
14	**return** Cipher_1, Cipher_2;

[표 3-4] Fixslicing AES-CTR 최적화 구현 의사코드

[표 3-4]는 Fixslicing AES-CTR 최적화 구현 의사코드이다. 전체적으로 정리하면 1번 줄의 PreState_init 함수에서는 사전 연산 테이블을 활용하여 IV_1, IV_2를 초기화 하는 함수이다. 이 과정에서 현재 카운터 값만을 활용하여 사전 연산 테이블에 사전 연산된 값을 불러와 IV_1, IV_2의 값을 초기화 한다. 다음으로 Fixslicing 기법을 적용하기 위해 2번 줄에서 Packing 연산을 진행하고 3~10번 줄까지는 사전 연산되지 못한 나머지 AES 연산을 진행한다. 이후 11번 줄에서 Unpacking을 통해 원래의 State 상태로 되돌려 주는 연산을 진행한다. 12~13번 줄에서는 이렇게 연산된 암호화 값과 입력으로 받은 평문을 XOR 연산을 함으로써 최종적인 암호화 값을 얻을 수 있다.

제 4 장 성 능 평 가

본 논문에서는 RV32I를 사용하는 32-bit RISC-V 프로세서 상에서 성능 측정을 진행한다. 여기에 사용된 보드는 Fixslicing 기법을 제안한 논문에서 사용된 동일한 보드인 SiFive사의 HiFive Rev B를 사용한다. HiFive1 Rev B는 4MB의 Quad SPI 플레시 메모리, E31 코어가 탑제되어 있으며 320MHz로 연산이 수행된다. 구현을 위해 SiFive사에서 제공하는 Freedom Studio 프레임워크를 사용하여 구현을 진행한다.

성능 측정은 구현된 암호화 함수가 동작할 때의 Cycle을 측정하여 비교를 진행한다. 이때 비교 단위는 Cycle이 아닌 기존 논문에서 사용된 단위 cpb(Cycle per block)를 사용한다. 이때 byte가 아닌 Block임을 주의한다.
성능 측정을 위해 암호화 함수를 10,000번 반복시켜 측정된 cpb의 평균값을 사용하여 성능을 제시한다. 성능 측정 결과는 [성능 결과 표]와 같다.

	Original Fixslicing AES	This work
Semi-Fixsliced	1,447	1,345
Fully-Fixsliced	1,398	1,283

[표 4-1] 제안된 기법과 기존 구현의 성능 측정 결과(단위 : cpb(Cycle per Block))

기존의 Fixslicing AES 구현의 경우 Semi-Fixsliced에서는 1,447cpb를 보여주고 있고, 본 논문에서 제안하는 기법을 적용하였을 때 1,345cpb의 성능을 보여주고 있으며 이는 약 7%의 성능 향상을 보여준다. 다음으로 Fully-Fixsliced의 경우 기존에 1,398cpb의 성능을 보여주고 있다. 본 논문에

서 제안하는 기법을 적용하였을 때 1,283cpb의 성능을 보여주고 약 9%의 성능 향상을 확인할 수 있다. 본 논문에서 제안하는 기법은 사전 연산 테이블을 사용하기 때문에 4KB의 메모리를 추가적으로 사용하며 사전 연산 기법을 활용하기 위한 코드량이 증가하는 단점도 존재한다. 하지만 약 9%의 성능 향상이 가능하기 때문에 빠른 속도의 암호화가 중요한 환경에서 사용한다면 더 의미있게 활용 가능하다.

제 5 장 결 론

본 논문에서는 32-bit RISC-V 프로세서 상에서 Fixslicing 기법을 활용한 AES 블록 암호의 CTR 운용 모드 속도 최적화 구현을 연구하였다. 속도 최적화 구현을 위해 사전 연산 기법을 활용하여 사전 연산 테이블을 활용한 최적화 구현을 진행하였다. CTR 운용 모드의 Nonce 값이 고정되는 특성을 통해 AES 블록 암호의 경우 2 라운드의 Shiftrows 함수까지 사전 연산이 가능한 것을 확인하였다. 이를 통해 Inverse Shiftrows 함수 1번, Shiftrows 함수 1번이 추가되지만 Mixcolumns 함수 1번, Addroundkey 2번 그리고 SubBytes 함수 2번의 연산이 생략 가능하다. 결과적으로 Semi-Fixsliced 구현과 Fully-Fixsliced 구현에서 각각 약 7%, 약 9%의 성능 향상을 달성하였다. 하지만 사전 연산된 테이블을 사용하기 위해 테이블 저장 공간으로 4KB의 추가적인 메모리 공간을 사용하고, 사전 연산 구현을 위한 코드량이 증가하는 단점도 존재한다. 하지만 더 빠른 속도로 암호화가 가능하기 때문에 빠른 암호화가 중요한 환경에서 사용하기에 적합하다. 또한 암호화에 사용되는 키 값이 자주 변하게 되면 사전 연산 테이블을 생성하는 과정을 키가 변경될 때마다 동작해야 하므로 키 값이 자주 변하지 않는 환경에서 사용되면 더 효율적으로 사용가능하다.

향후 연구로는 사전 연산 기법이 적용 가능한 다른 암호 알고리즘과 AES 블록 암호를 활용하는 암호 알고리즘에 해당 기법이 적용 가능한지 조사하고 제안 기법을 적용하여 다양한 암호 알고리즘의 최적화 구현에 대한 연구를 진행한다.

참 고 문 헌

1. 국내문헌

Eum, S. W., Kim, H. J., Sim, M. J., Song, G. J., & Seo, H. J. (2022). Implementation of Fixslicing AES-CTR Speed Optimized Using Pre-Computed on 32-Bit RISC-V. Journal of the Korea Institute of Information Security & Cryptology, 32(1), 1-9.

Kwak, Y., Kim, Y., & Seo, S. C. (2021). Benchmarking Korean block ciphers on 32-bit RISC-V processor. Journal of the Korea Institute of Information Security & Cryptology, 31(3), 331-340.

Seo, H. J., Kwon, H. D., Jang, K. B., & Kim, H. (2021). Optimized implementation of scalable multi-precision multiplication method on RISC-V processor for high-speed computation of post-quantum cryptography. Journal of the Korea Institute of Information Security & Cryptology, 31(3), 473-480.

2. 국외문헌

Adomnicai, A., & Peyrin, T. (2020). Fixslicing AES-like ciphers: New bitsliced AES speed records on ARM-Cortex M and RISC-V. Cryptology ePrint Archive.

Akkar, M. L., & Giraud, C. (2001, May). An implementation of DES and AES, secure against some attacks. In International Workshop on Cryptographic Hardware and Embedded Systems (pp. 309-318). Springer, Berlin, Heidelberg.

Biham, E. (1997, January). A fast new DES implementation in software. In International Workshop on Fast Software Encryption (pp. 260-272). Springer, Berlin, Heidelberg.

Daemen, J., & Rijmen, V. (1999). AES proposal: Rijndael.

Diffie, W., & Hellman, M. E. (1979). Privacy and authentication: An introduction to cryptography. Proceedings of the IEEE, 67(3), 397-427.

Dworkin, M. J., Barker, E. B., Nechvatal, J. R., Foti, J., Bassham, L. E., Roback, E., & Dray Jr, J. F. (2001). Advanced encryption standard (AES).

Kim, K., Choi, S., Kwon, H., Liu, Z., & Seo, H. (2020). FACE-LIGHT: Fast AES-CTR mode encryption for low-end microcontrollers. In International Conference on Information Security and Cryptology (pp. 102-114). Springer, Cham.

Käsper, E., & Schwabe, P. (2009, September). Faster and timing-attack resistant AES-GCM. In International Workshop on Cryptographic Hardware and Embedded Systems (pp. 1-17). Springer, Berlin, Heidelberg.

Lipmaa, H., Rogaway, P., & Wagner, D. (2000, October). CTR-mode encryption. In First NIST Workshop on Modes of Operation (Vol. 39).

Park, J. H., & Lee, D. H. (2018). FACE: Fast AES CTR mode encryption techniques based on the reuse of repetitive data. IACR Transactions on Cryptographic Hardware and Embedded Systems, 469-499.

Rebeiro, C., Selvakumar, D., & Devi, A. S. L. (2006, December). Bitslice implementation of AES. In International Conference on Cryptology and Network Security (pp. 203-212). Springer, Berlin, Heidelberg.

Schwabe, P., & Stoffelen, K. (2016, August). All the AES you need on Cortex-M3 and M4. In International Conference on Selected Areas in Cryptography (pp. 180-194). Springer, Cham.

Standard, D. E. (1999). Data encryption standard. Federal Information Processing Standards Publication, 112.

Waterman, A., Lee, Y., Patterson, D. A., & Asanovic, K. (2011). The risc-v instruction set manual, volume i: Base user-level isa. EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62, 116.

ABSTRACT

Optimized Implementation of Fixslicing AES-CTR on RISC-V

Eum, Si-Woo

Major in IT Convergence Engineering

Dept. of IT Convergence Engineering

The Graduate School

Hansung University

The AES block cipher implemented with the Fixslicing technique is a technique in which the Shiftrows function is omitted from the round function to solve the problem that many cycles occur in the linear layer of the Bitsliced technique, which is an existing implementation technique. Omitting the shiftrows function can bring about 30% higher performance than the Bitsliced method. In this paper, we implemented the Fixslicing AES counter operation mode optimization by utilizing the fixed nonce value of the counter operation mode and the Fixslicing technique in the 32-bit RISC-V processor. Due to the fixed nonce value, it was analyzed through investigation that a fixed value was obtained according to the count value when calculating the second shiftrows function in the encryption process of the AES block cipher. Using this analysis, a pre-calculation technique was applied for optimization implementation, and through pre-calculation, 2 AddroundKey, 2 SubBytes, and 1 Mixcolumns operations could be omitted. To compensate for the shortcomings of code volume increase, performance measurements were performed by applying the proposed technique in the Semi-Fixsliced implementation, which omits only half the Shiftrows function operation,

and the Full-Fixsliced implementation, which omits it completely. As a result, when implemented through the proposed precomputation technique, the cost of encrypting one block is 1,345 cpb and 1,283 cpb, respectively, which is about 7% and 9% performance improvement compared to the previous implementation work.

【Key words】AES Block Cipher, Fixslicing, Software Implementation, RISC-V, CTR mode