8-bit AVR 상에서의 Reverse shift를 사용한 경량암호 TinyJAMBU 최적 구현

권혁동, 엄시우, 심민주, 양유진, 서화정
한성대학교

한성대학교 HANSUNG UNIVERSITY

CryptoCraft LAB

서론

TinyJAMBU

제안 기법

결론

서론

- IoT 기술이 발전함에 따라 무선 보안이 중요해짐
  - 대부분의 암호는 연산량이 많음
  - IoT 기기는 제한적인 환경으로 **가용 자원이 부족**함

- IoT 환경에서 효과적으로 동작하는 경량암호의 등장
  - NIST에서는 **경량암호 공모전**을 개최

- 공모전 출품 작품 중 TinyJAMBU의 최적 구현을 제시
  - AVR 프로세서를 대상

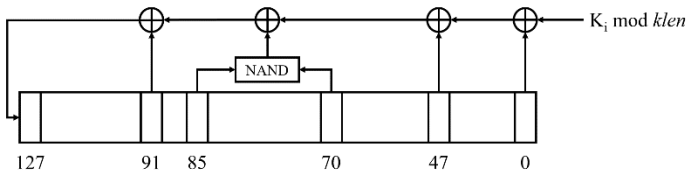
TinyJAMBU

- NIST 경량암호 공모전은 2018년 개최되어 현재 최종 라운드 진행 중
  - 10개의 후보 암호가 존재

기반	알고리즘	해시 지원	코어 함수
Stream cipher	Grain-128 AEAD	X	Grain-128a
Block cipher	GIFT-COFB	X	GIFT-128
Tweakable block cipher	Romulus	O	SKINNY-128-256 SKINNY-128-384
Permutation	ASCON	O	ASCON-320
	ISAP	X	Keccak-400 ASCON-320
	PHOTON-Beetle	X	Spongent-160/176 Keccak-200
	SPARKLE	O	Sparkle-256/384/512
	TinyJAMBU	**X**	**JAMBU-128**
	Xoodyak	O	Xoodoo-384

TinyJAMBU

- TinyJAMBU는 CAESAR 경진대회에 출품되었던 JAMBU에 기반함
- **Keyed permutation**을 반복하는 순열 구조의 암호
  - Round2에서는 블록암호로 분류되었음
  - 반복 횟수는 P_n으로 표현하며,
    상황에 따라 반복 횟수가 다름
- **Keyed permutation의 횟수가 너무 많음**
  - 이를 중점적으로 최적 구현




```
StateUpdate(State, Key, Round):
    for i = 0 to Round
        t1 = (State1>>15)|(State2<<17)
        t2 = (State2>>6)|(State3<<26)
        t3 = (State2>>21)|(State3<<11)
        t4 = (State2>>27)|(State3<<5)
        fb = State0^t1^(~(t2&t3))^t4^key
        State0 = State1
        State1 = State2
        State2 = State3
        State3 = fb
    end
```

키 길이	128	192	256
Initialization: Key	1024	1152	1280
Initialization: Nonce	640	640	640
Processing AD	640	640	640
Enc/Dec	1024	1152	1280
Finalization	1024, 640	1152, 640	1280, 640

TinyJAMBU

- TinyJAMBU의 소스코드는 2종류로 배포됨
  - 일반: 연산 마지막 부분에서 32비트 회전 연산 진행
  - 최적: 32비트 회전 연산 진행을 제거한 대신 코드 길이가 약 4배

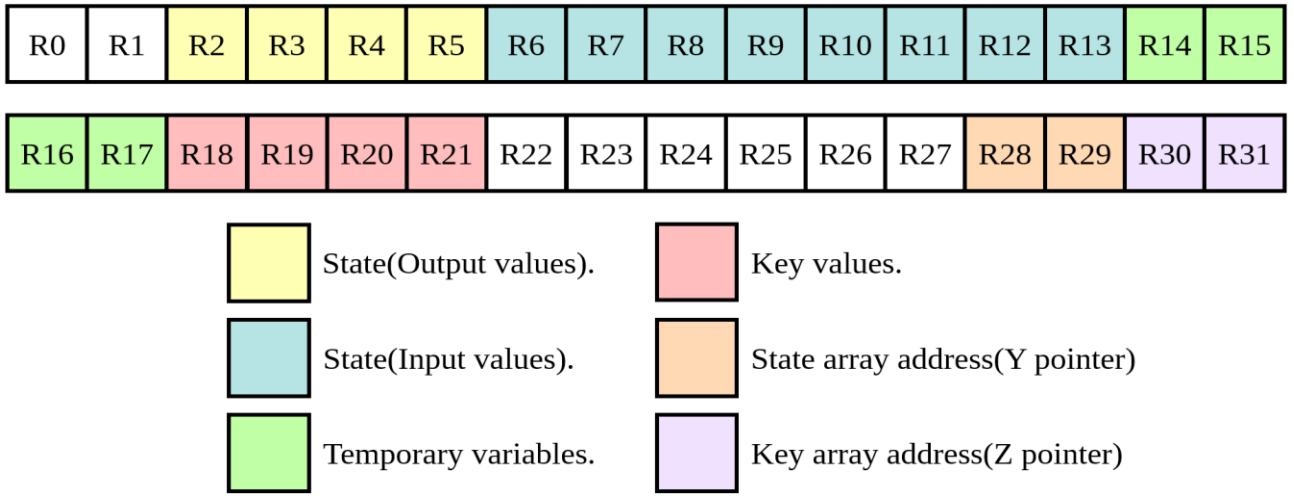
```c
void state_update(unsigned int *state, const unsigned char *key, unsigned int number_of_steps)
{
    unsigned int i;
    unsigned int t1, t2, t3, t4, feedback;
    for (i = 0; i < (number_of_steps >> 5); i++)
    {
        t1 = (state[1] >> 15) | (state[2] << 17);  // 47 = 1*32+15
        t2 = (state[2] >> 6)  | (state[3] << 26);  // 47 + 23 = 70 = 2*32 + 6
        t3 = (state[2] >> 21) | (state[3] << 11);  // 47 + 23 + 15 = 85 = 2*32 + 21
        t4 = (state[2] >> 27) | (state[3] << 5);   // 47 + 23 + 15 + 6 = 91 = 2*32 + 27
        feedback = state[0] ^ t1 ^ (~(t2 & t3)) ^ t4 ^ ((unsigned int*)key)[i & 3];
        // shift 32 bit positions
        state[0] = state[1]; state[1] = state[2]; state[2] = state[3];
        state[3] = feedback ;
    }
}
```

```c
void state_update(unsigned int *state, const unsigned char *key, unsigned int number_of_steps)
{
    unsigned int i;
    unsigned int t1, t2, t3, t4;

    //in each iteration, we compute 128 rounds of the state update function.
    for (i = 0; i < number_of_steps; i = i + 128)
    {
        t1 = (state[1] >> 15) | (state[2] << 17);  // 47 = 1*32+15
        t2 = (state[2] >> 6)  | (state[3] << 26);  // 47 + 23 = 70 = 2*32 + 6
        t3 = (state[2] >> 21) | (state[3] << 11);  // 47 + 23 + 15 = 85 = 2*32 + 21
        t4 = (state[2] >> 27) | (state[3] << 5);   // 47 + 23 + 15 + 6 = 91 = 2*32 + 27
        state[0] ^= t1 ^ (~(t2 & t3)) ^ t4 ^ ((unsigned int*)key)[0];

        t1 = (state[2] >> 15) | (state[3] << 17);
        t2 = (state[3] >> 6)  | (state[0] << 26);
        t3 = (state[3] >> 21) | (state[0] << 11);
        t4 = (state[3] >> 27) | (state[0] << 5);
        state[1] ^= t1 ^ (~(t2 & t3)) ^ t4 ^ ((unsigned int*)key)[1];

        t1 = (state[3] >> 15) | (state[0] << 17);
        t2 = (state[0] >> 6)  | (state[1] << 26);
        t3 = (state[0] >> 21) | (state[1] << 11);
        t4 = (state[0] >> 27) | (state[1] << 5);
        state[2] ^= t1 ^ (~(t2 & t3)) ^ t4 ^ ((unsigned int*)key)[2];

        t1 = (state[0] >> 15) | (state[1] << 17);
        t2 = (state[1] >> 6)  | (state[2] << 26);
        t3 = (state[1] >> 21) | (state[2] << 11);
        t4 = (state[1] >> 27) | (state[2] << 5);
        state[3] ^= t1 ^ (~(t2 & t3)) ^ t4 ^ ((unsigned int*)key)[3];
    }
}
```

제안 기법

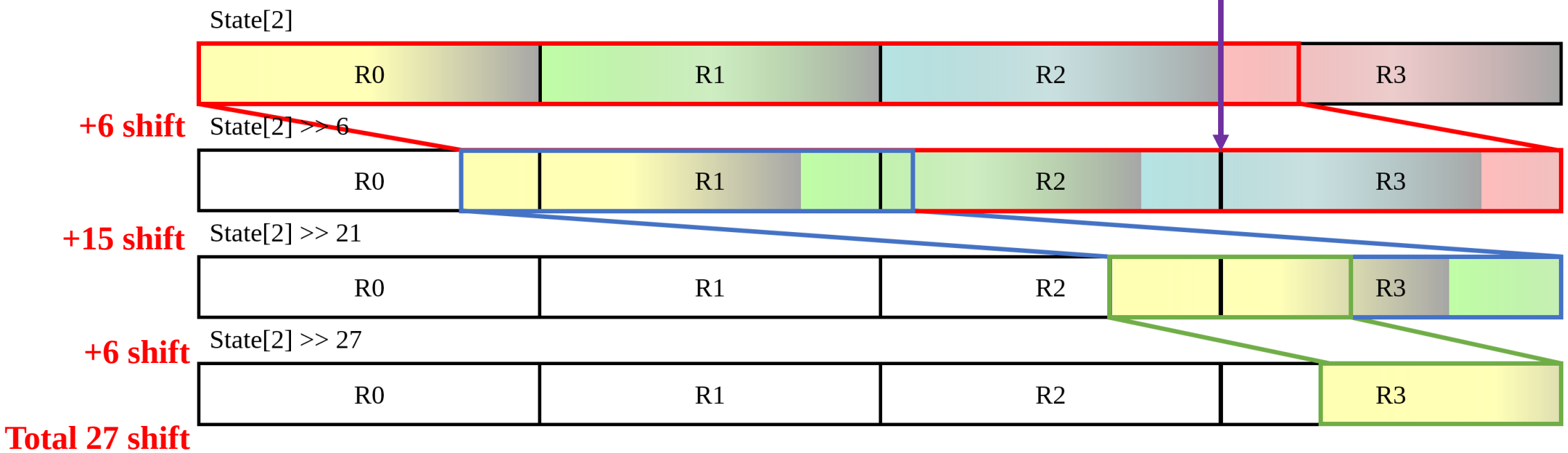
- 제안하는 기법은 Keyed permutation의 내부 연산을 최적화
- **Reverse shift**를 통해 회전 연산 횟수를 줄임
  - 원래의 시프트 방향과 반대로 시프트
  - **연산 횟수는 1회, 2회 정도**로 크게 줄어들지만, 연산 결과는 동일
  - AVR 어셈블리 활용

- AVR의 레지스터가 32개 밖에 없으므로 할당 계획을 수립



제안 기법

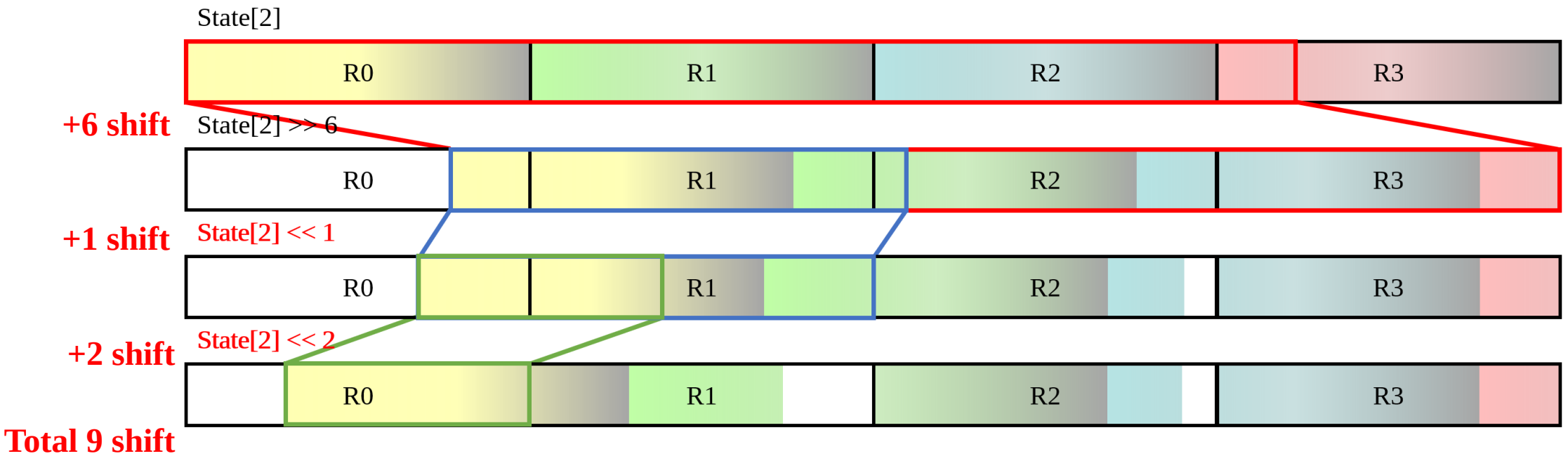
- AVR의 레지스터는 8비트까지 저장할 수 있음
  - State 하나의 크기는 32비트이므로, 4개의 레지스터를 사용하여 저장
- 각 단계 별로 시프트를 진행
  - 1, 2, 3단계로 진행 (예시는 우측 t2 생성)
  - 이전 단계의 **색 있는 테두리 부분이 다음 단계의 결과 값**이 됨
  - **결과 값과 상관 없는 부분은 연산에 필요가 없음**



제안 기법

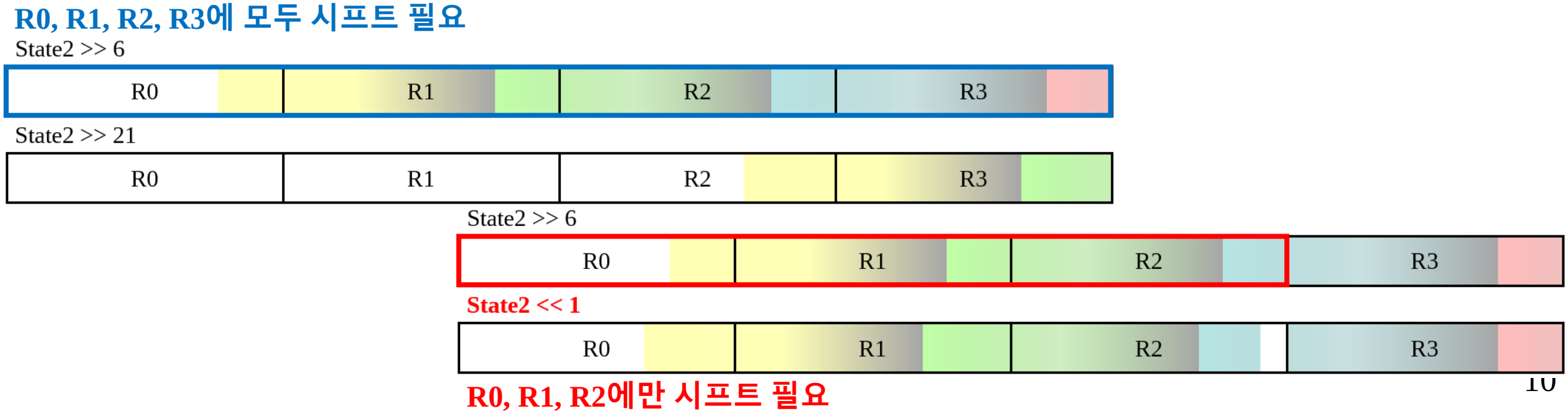
- 중간 값에서 원래 시프트 방향의 반대로 진행
  - **<u>훨씬 적은 횟수로 필요한 값을 생성할 수 있음</u>**

유형	기존	제안
좌측 t2, t3, t4	26회	8회
좌측 t1	17회	1회
우측 t1	15회	1회



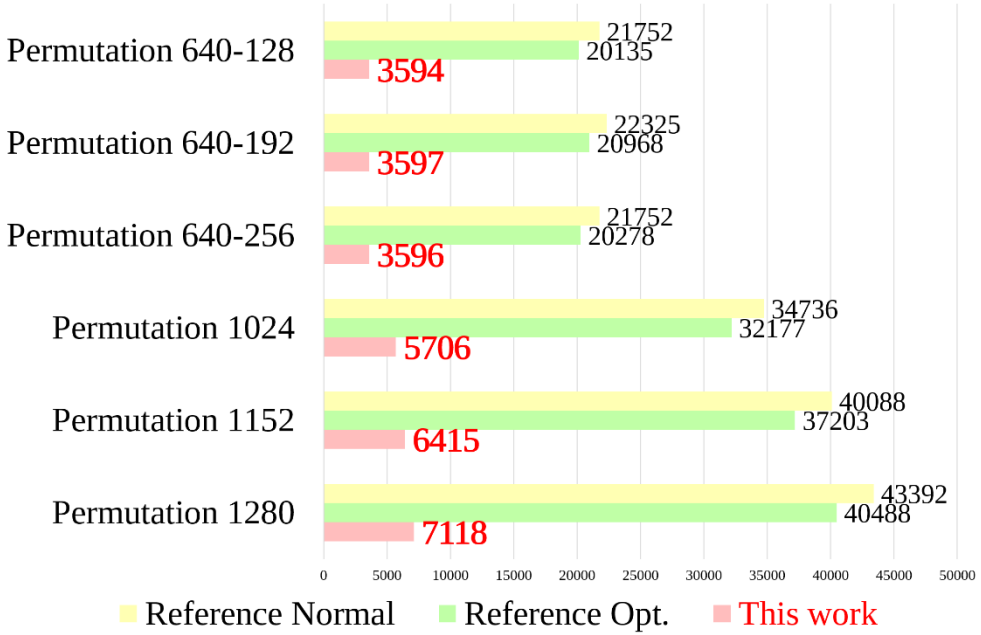
제안 기법

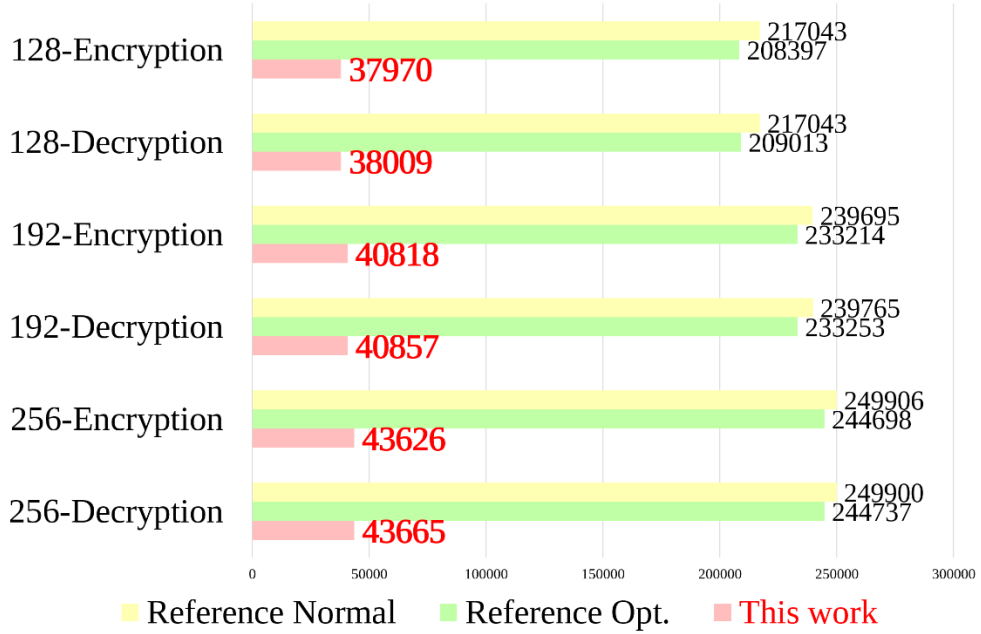
- 시프트 횟수가 줄어드는 것 이상의 효과
  - 실제 AVR 어셈블리로 구현 시에는 레지스터별로 시프트를 적용
  - Reverse Shift는 버려지는 부분은 고려하지 않음
  - **해당 레지스터에 대해서 시프트를 하지 않아도 됨**
- 시프트 중에는 레지스터 간 1비트 씩 이동이 필요
  - **시프트를 적용하는 레지스터가 1개 줄어드는 것은 큰 차이를 유발**



제안 기법

- Microchip Studio Framework를 사용하여 구현
  - ATmega128 프로세서를 대상
  - 컴파일 옵션 –O3 적용, clock cycle로 비교
- 성능 평가 결과
  - Keyed permutation: 최소 560% ~ **최대 624%** 성능 향상
  - TinyJAMBU: 최소 550% ~ **최대 570%** 성능 향상





결론

- 경량암호 공모전 후보인 TinyJAMBU를 최적 구현
  - **AVR 프로세서**를 대상
- **Reverse Shift**를 통한 시프트 연산 횟수의 최소화
  - AVR 레지스터가 8비트만 저장 가능한 특성을 활용
  - **시프트 횟수가 줄어들어 효과적**인 연산
  - 횟수 뿐만 아니라, 명령어를 적용하는 레지스터의 수도 줄어 듦
- 제안 기법의 성능은 **평균적으로 기존 대비 6배** 가량 뛰어남

Q & A