Reverse Shift를 사용한 TinyJAMBU 최적 구현

정보컴퓨터공학과 권혁동

한성대학교 HANSUNG UNIVERSITY

CryptoCraft LAB

TinyJAMBU

이전 연구 기록

제안 기법

결론

TinyJAMBU

- NIST **경량 암호 공모전**에 출품된 경량 암호
- 키 크기에 따라 세 가지의 규격을 제공
  - 128 / 192 / 256
- 모든 과정에서 keyed permutation을 반복적으로 진행

$\text{StateUpdate}(S, K, i)$:
$\quad \text{feedback} = s_0 \oplus s_{47} \oplus (\sim (s_{70} \& s_{85})) \oplus s_{91} \oplus k_{i \bmod klen}$
$\quad$ for j from 0 to 126: $s_j = s_{j+1}$
$\quad s_{127} = \text{feedback}$
end

TinyJAMBU

- Keyed permutation의 구현
- 제안하는 구현은 **최적화 버전을 기반**으로 구현 진행

일반 버전

```
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 ;
        }
}
```

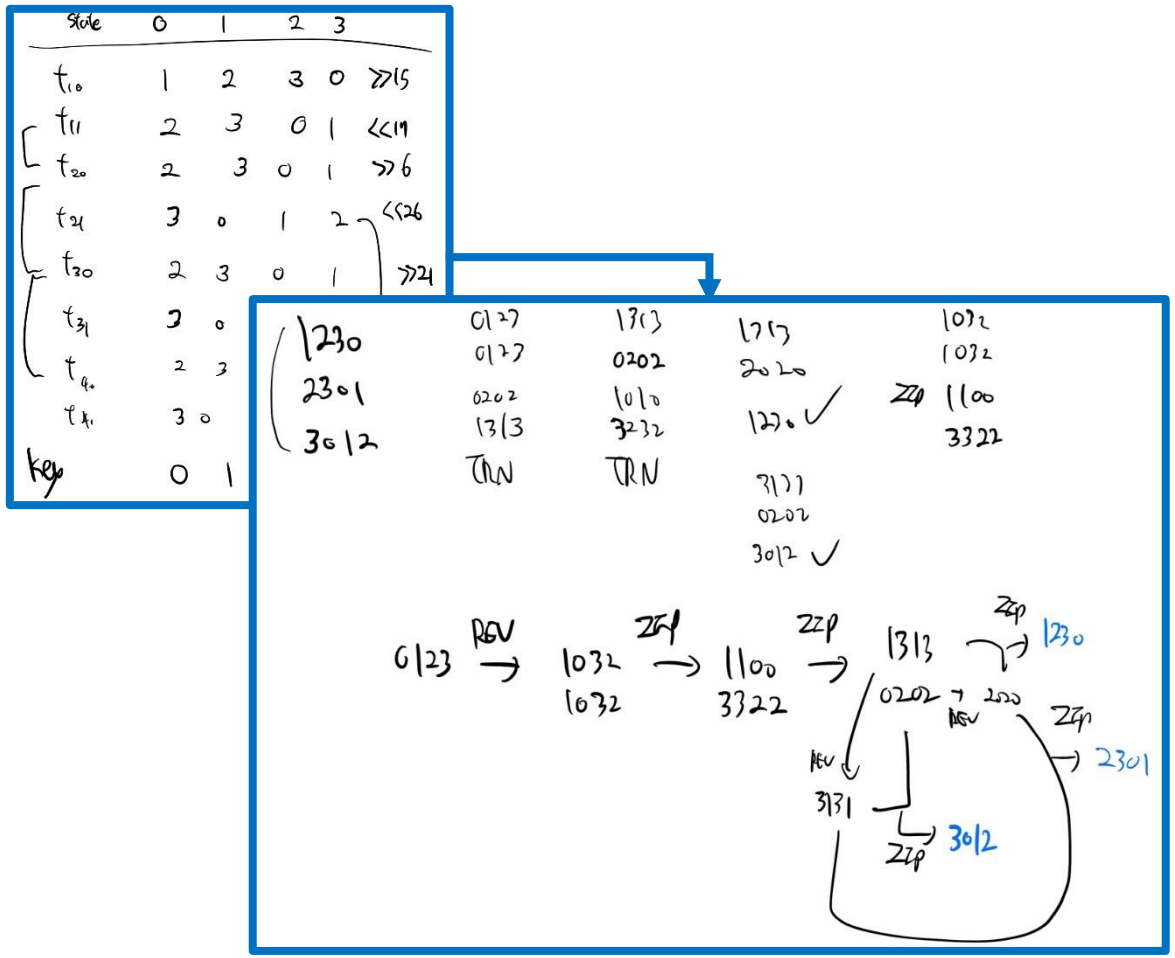
최적화 버전

```
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];
        }
}
```

이전 연구 기록

- State 0, 1, 2, 3은 서로 같은 방향으로 같은 횟수 shift가 됨
- ARMv8의 벡터 레지스터/인스트럭션으로 구현 시도
  - 하지만 **값이 누적되는 특성**으로 실패!

```
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];
        }
}
```



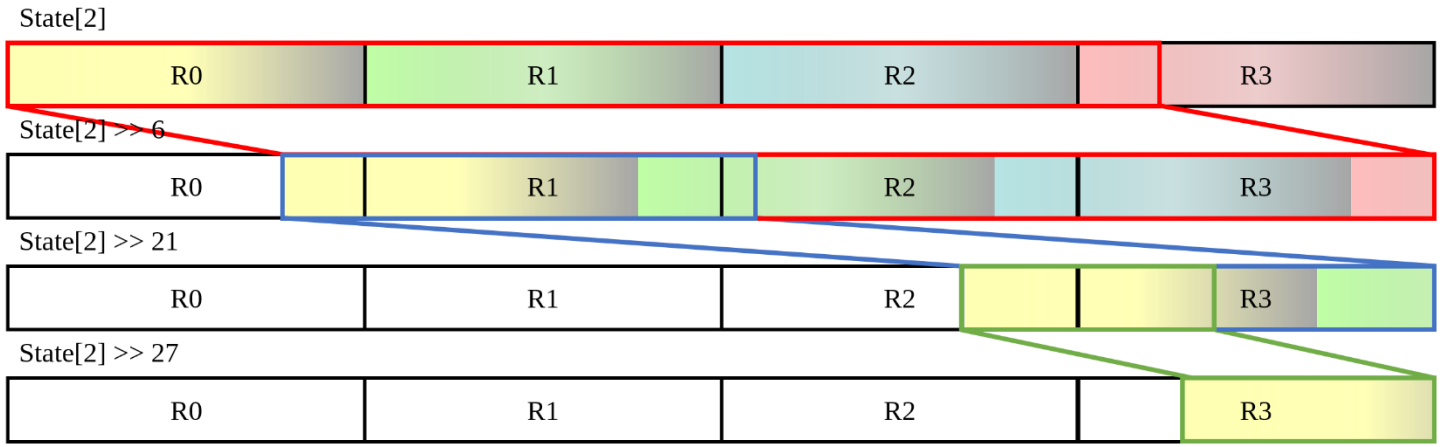
제안 기법

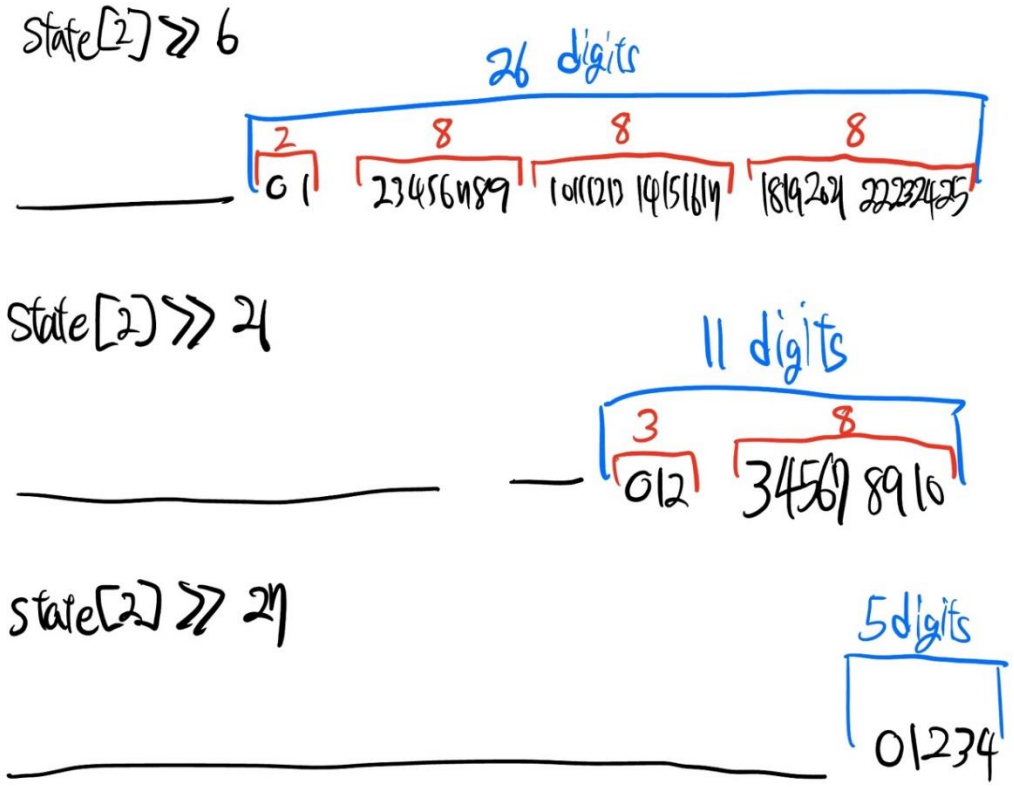
- 8-bit AVR 환경에서 새로운 구현을 제안
- 원본 state는 **32비트** → AVR 상에서는 **4개의 레지스터**에 저장
- 일정 횟수로 시프트 되는 값
- 시프트의 **방향이 일정**하며 **누적**되는 형태

```
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

- 첫 번째 시프트: 오른쪽 6회
- 두 번째 시프트: 오른쪽 15회 (누적 21회)
- 세 번째 시프트: 오른쪽 6회 (누적 27회)





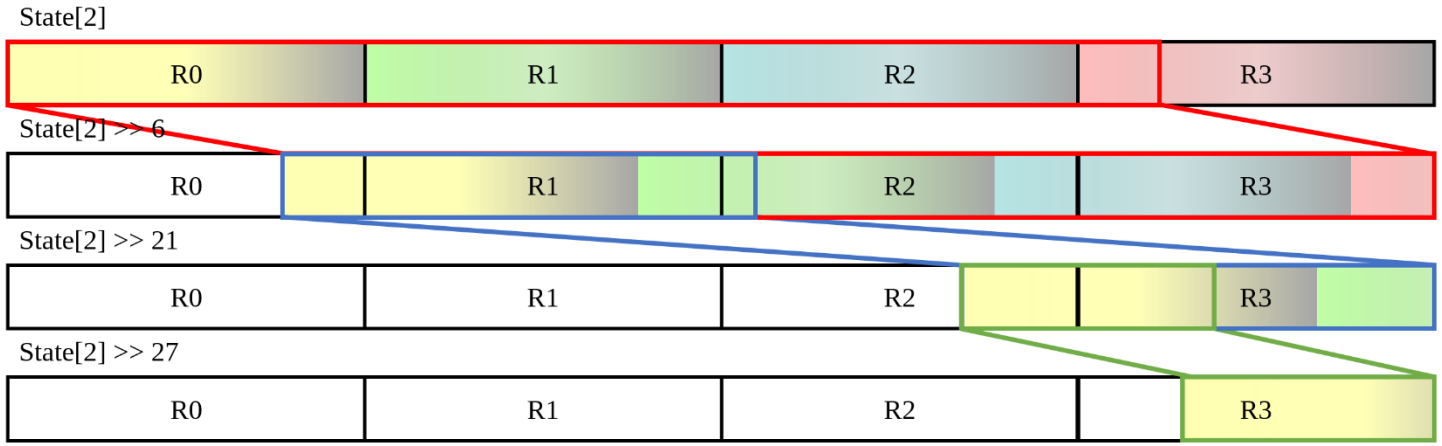
제안 기법

- SHL 대신 MOV를 통한 8비트 이동 구현

→ **가능하지만 비효율적**

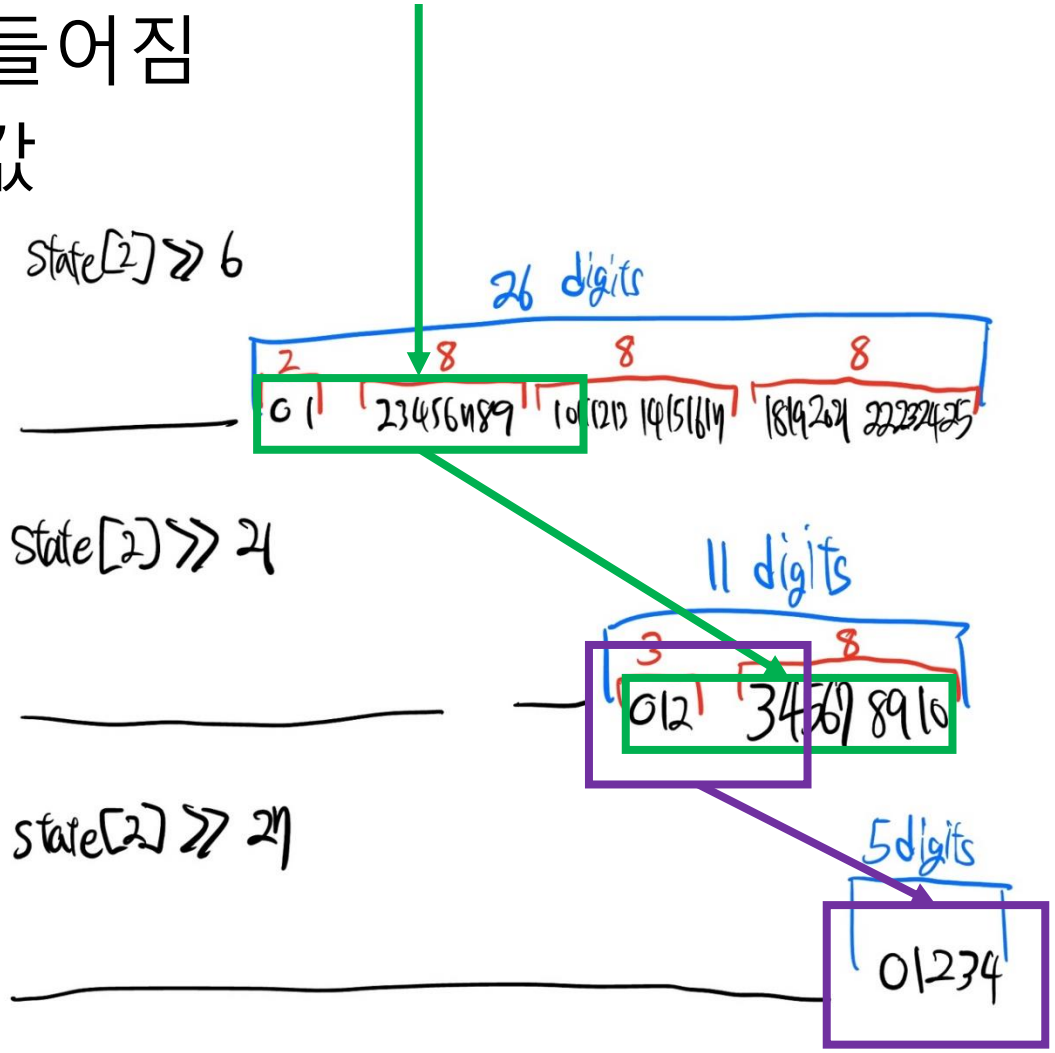
  - 6회 시프트에는 사용 불가
  - 2단계인 15회 시프트에 MOV사용 후 7회 시프트 구현 OK

- 시프트 방향과는 상관 없이, 값을 일치시키는 방법을 고안
  - **Reverse Shift**



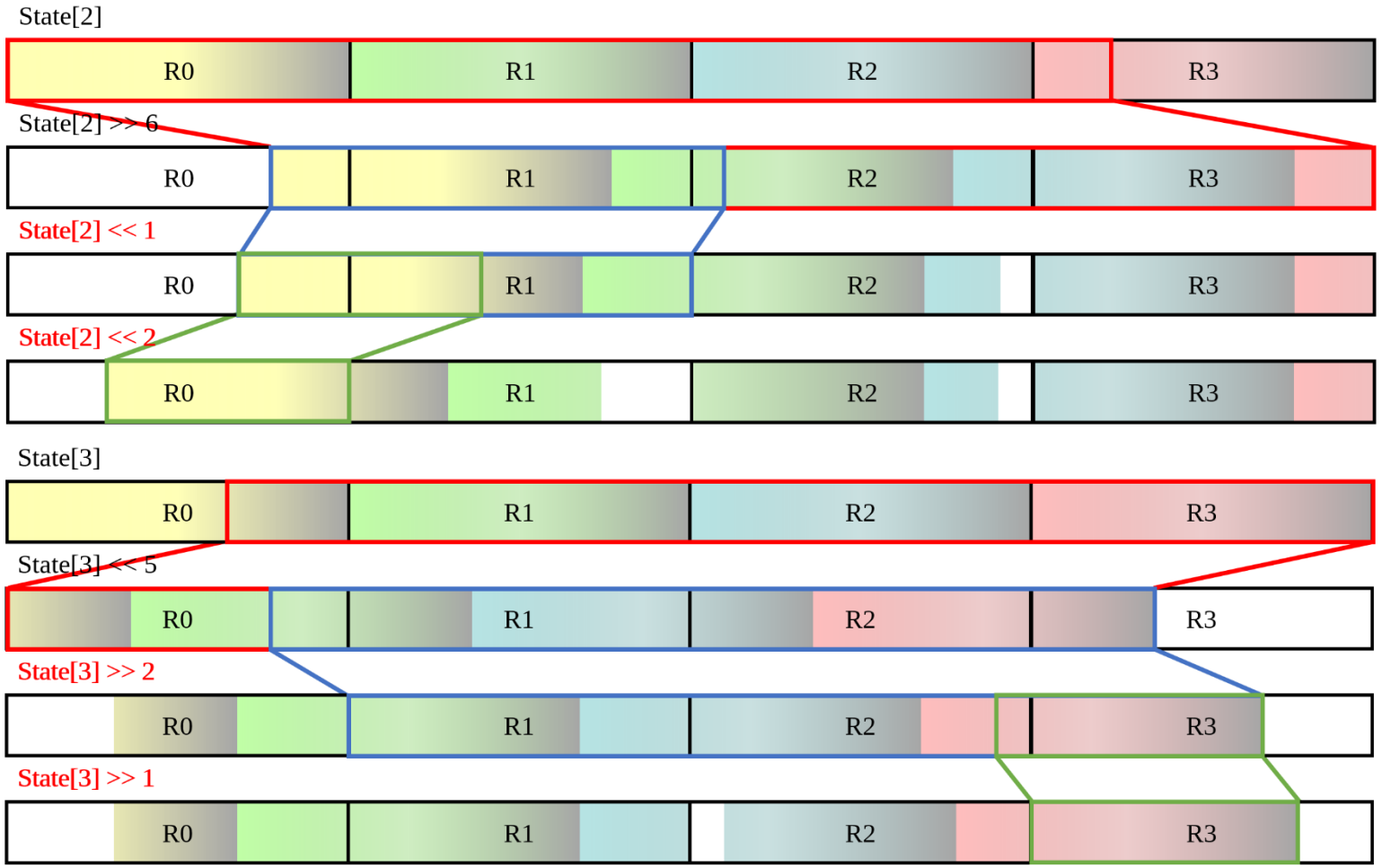
제안 기법

- 최초 6회 시프트 이후 필요한 값은 상위 11비트 값들
  → 이 값은 **좌측 시프트 1번**만 하면 만들어짐
- 마지막 값은 중간 값에서 상위 5비트 값
  → 남은 값에서 **좌측 시프트 2번**
- 기존: 누적 시프트 총 27회
- **제안: 누적 시프트 총 8회**
- 다른 유형
  - 좌측 t2,3,4 값: 26회 vs **8회**
  - 좌측 t1 값: 17회 vs **1회**
  - 우측 t1 값: 15회 vs **1회**



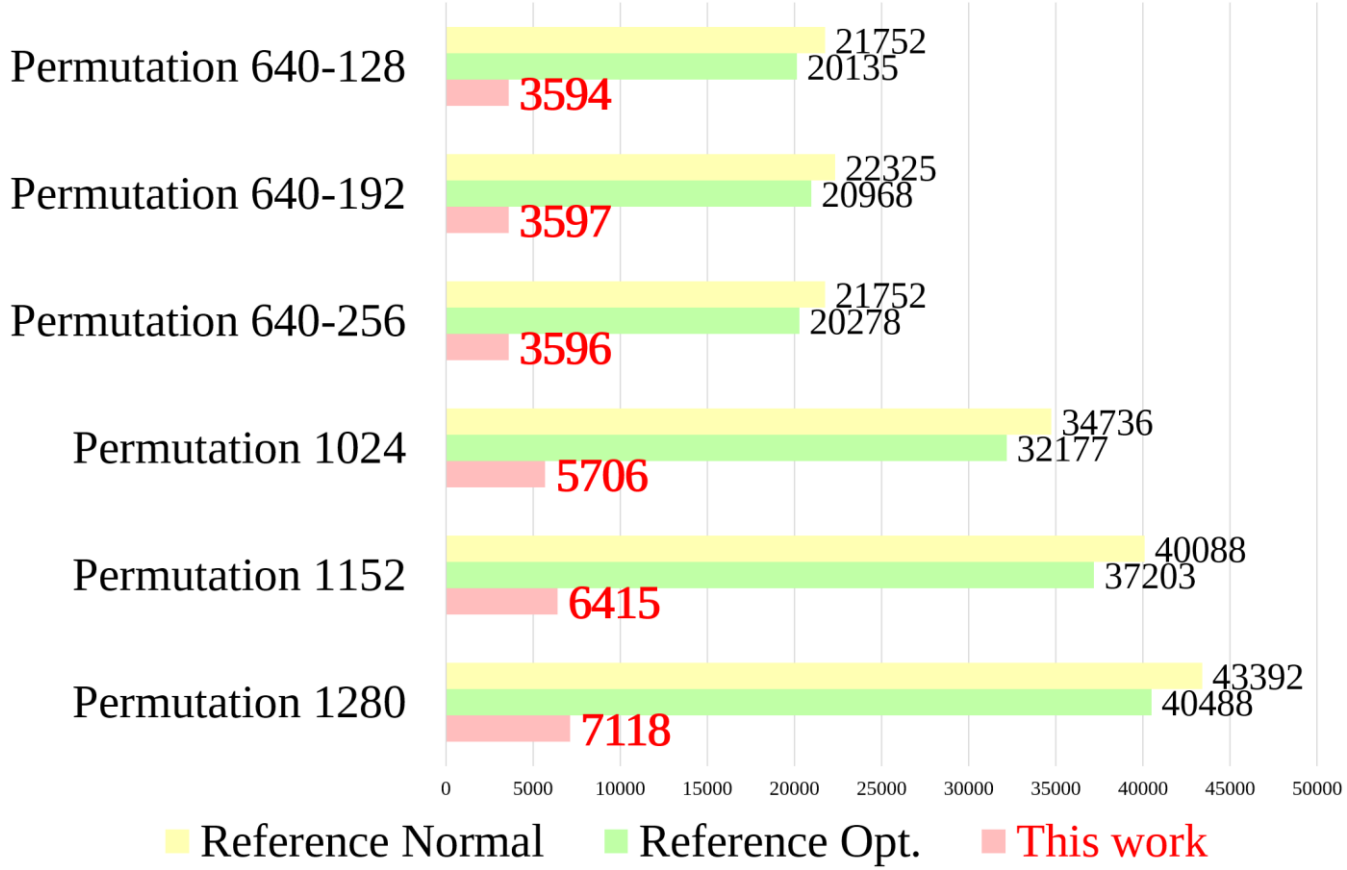
제안 기법

- 레지스터 내부 값 추적
  - 색상 선이 없는 부분은 버려지는 값



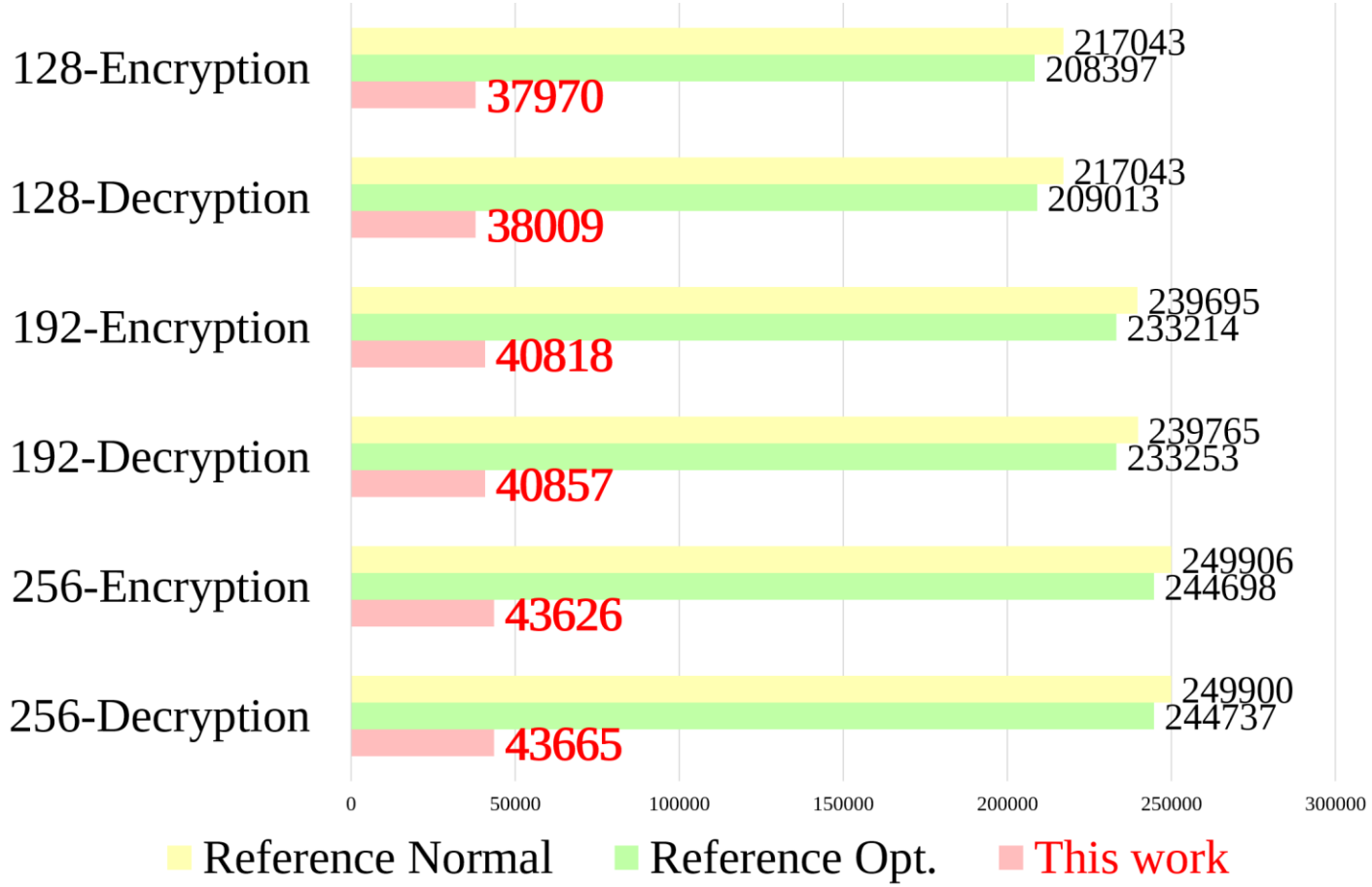
제안 기법

- Keyed permutation의 성능 평가 (단위: 클록 사이클)
- 제안하는 기법이 최소 560%에서 최대 624% 빠름



제안 기법

- 실제 TinyJAMBU에 적용
- 제안하는 기법이 최소 550%에서 최대 587% 빠름



결론

- 제안하는 기법은 TinyJAMBU의 중점 연산인 keyed permutation을 최적 구현
- **시프트 방향을 반대**로 하여 연산 횟수를 줄임
  - Permutation의 성능 최대 624% 향상
  - TinyJAMBU의 성능 최대 587% 향상
- 다른 경량 암호에도 비슷하게 적용이 가능한지 알아보기

Q & A