ARMing-sword
ARMv8 상의 Scabbard 최적 구현

정보컴퓨터공학과 권혁동

한성대학교 HANSUNG UNIVERSITY

CryptoCraft LAB

Scabbard

ARMing-sword

성능 평가

결론

Scabbard

- 2021년에 제안된 Saber의 개선판
- Ring-learning with rounding을 사용하는 격자 기반 암호
- 3가지 버전 제공
  - **Florete**: Saber의 하드웨어 아키텍쳐와 소프트웨어 모듈 재사용
    NTT 기반 곱셈기 대신 Toom-cook 곱셈기 적용, 곱셈 횟수 줄임
  - **Espada**: 다항식의 길이를 짧게 줄인 버전
    병렬 연산이 효과적, 하드웨어 구현에 적합
  - **Sable**: Saber의 형태를 변형한 격자 기반 암호
    연속 균등 분포를 통한 rounding error 사용

Scabbard

- 곱셈기는 3단계로 구성됨
  - Evaluation: 입력 값을 256-bit 길이로 맞추고, 일정 수 만큼 값을 확보
  - Multiplication: 곱셈 연산을 진행하는 단계
  - Interpolation: 마지막 단계에서 값을 보간
- Evaluation은 모든 배열을 순회하며 **값을 복사**
- Multiplication은 **곱셈을 반복**적으로 진행

Scabbard

- ARMv7 대상의 구현물은 존재
- 제안하는 기법은 ARMv8 상에서 최적 구현을 시도
- ARMing-sword
- 두 가지 최적화 기법을 제안
  - Direct Mapping
  - Sliding Window

ARMing-sword

- Evaluation은 값을 복사하는 과정
- Espada는 2번 연속으로 진행하는 부분이 존재
- **값 이동이 효율적이지 못함**

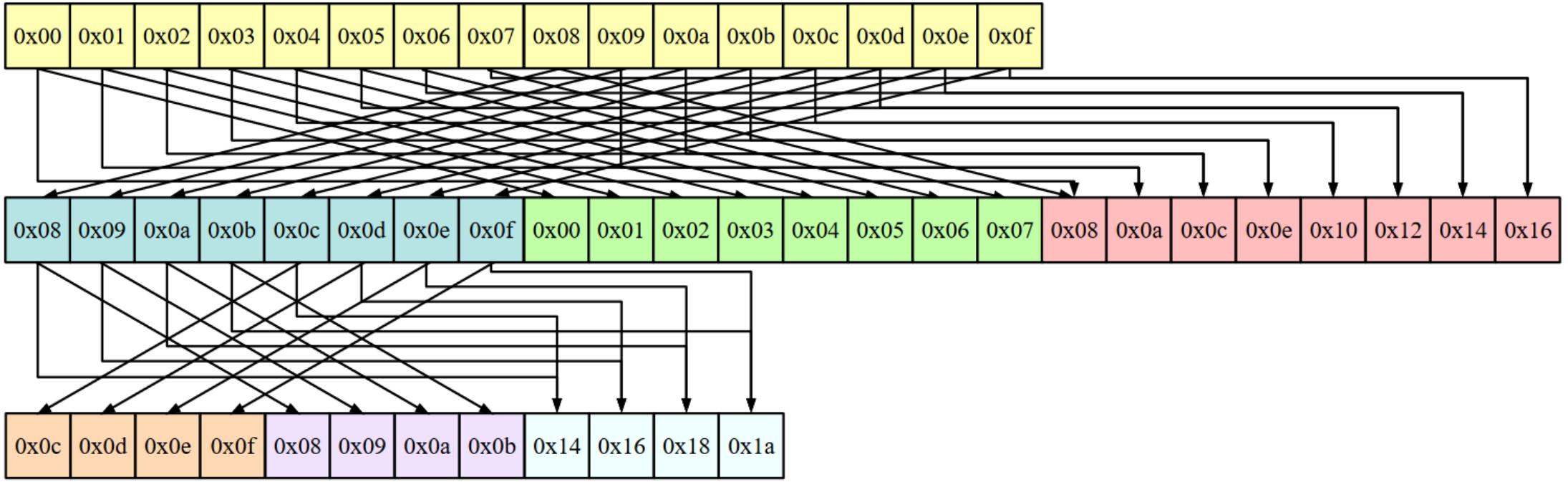
Algorithm 1 Pseudo-code of Scabbard Espada Evaluation step.

Input: N length of 16-bit array $A1$, middle length $M = N/2$, output length $L = N/4$.
Output: L length of 16-bit array $AW00$, $AW01$, $AW02$, $AW10$, $AW11$, $AW12$, $AW20$, $AW21$, $AW22$.

1: $i \leftarrow 0$
2: **for** i to M **do**
3: $AW0[i] \leftarrow A1[i + L]$
4: $AW2[i] \leftarrow A1[i]$
5: $AW1[i] \leftarrow A1[i] + A1[i + L]$
6: $i \leftarrow i + 1$
7: **end for**
8: $j \leftarrow 0$
9: **for** j to L **do**
10: $AW00[j] \leftarrow AW0[j + M]$
11: $AW01[j] \leftarrow AW0[j] + AW0[j+M]$
12: $AW02[j] \leftarrow AW0[j]$
13: $AW10[j] \leftarrow AW1[j + M]$
14: $AW11[j] \leftarrow AW1[j] + AW1[j+M]$
15: $AW12[j] \leftarrow AW1[j]$
16: $AW20[j] \leftarrow AW2[j + M]$
17: $AW21[j] \leftarrow AW2[j] + AW2[j+M]$
18: $AW22[j] \leftarrow AW2[j]$
19: $j \leftarrow j + 1$
20: **end for**
21: **return** $AW00$, $AW01$, $AW02$, $AW10$, $AW11$, $AW12$, $AW20$, $AW21$, $AW22$

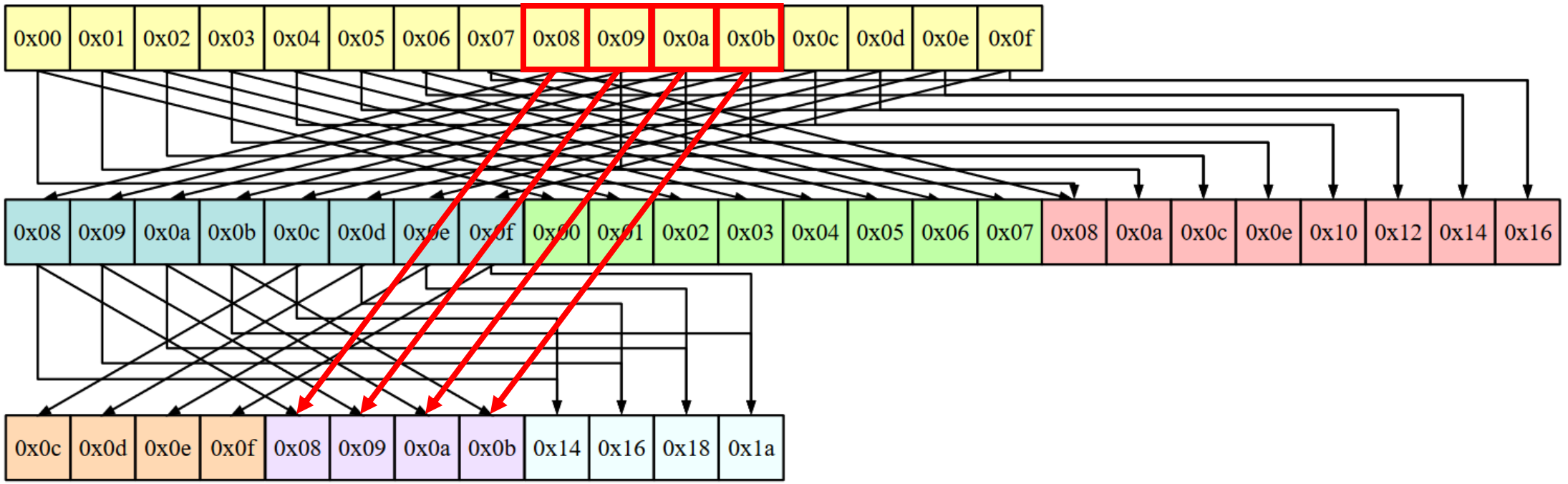
ARMing-sword

- Scabbard의 Evaluation 단계
- 최종적으로 초기 변수의 ¼길이인 배열이 9개 생성



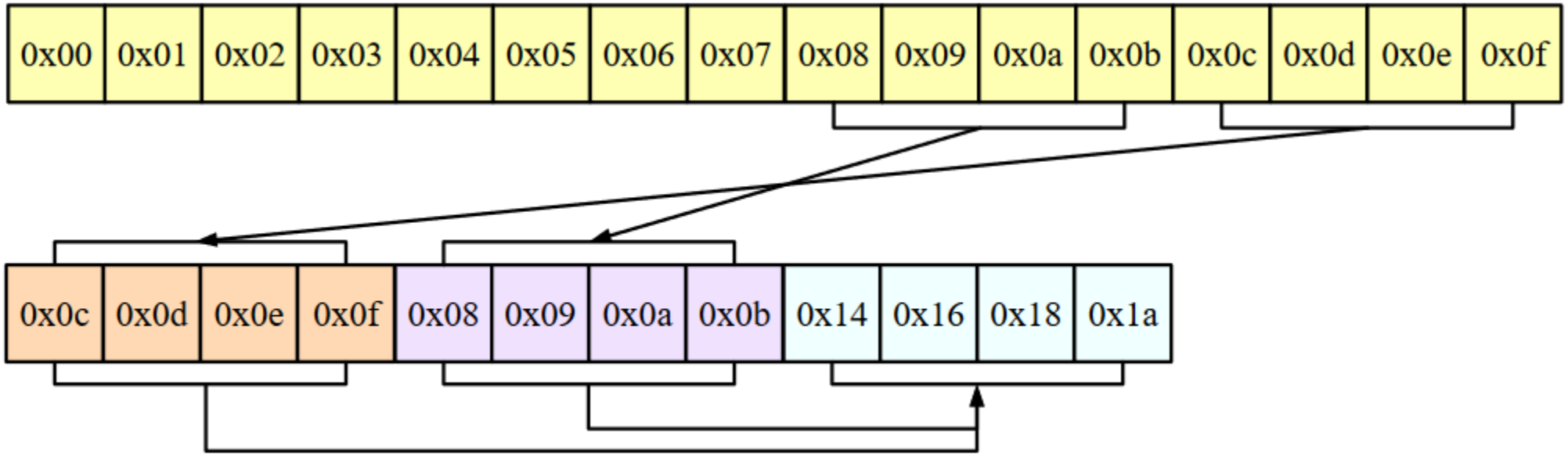
ARMing-sword

- 값의 이동을 살펴보는 것으로, 초기 **입력 값을 출력 값에 직접 대입**
- 중간 단계를 생략하는 것으로 더 빠른 속도로 값 생성 가능



ARMing-sword

- 벡터 레지스터/인스트럭션을 사용하여 **병렬화**
  - **반복 횟수를 줄일 수 있음**
- 합산을 통해서 생성되는 값은 레지스터에 **잔류한 값으로 연산**
  - **메모리 접근 최소화**
- Direct Mapping 기법
  - Florete/Sable은 구조는 조금 다르지만 거의 동일하게 구현 가능



ARMing-sword

- Multiplication은 곱셈을 진행하는 과정
- 변수를 일일히 순회하면서, 결과 값을 누적시키는 형태
- 순회에 시간이 오래 듦
- 병렬 구현으로 순회를 줄일 수 있으나,
  ARMv8 명령어가 지원하지 않음

Algorithm 5 Pseudo-code of Scabbard Espada Multiplication stage.

Input: Evaluation result array A, input array B
Output: output result C

```
1: i ← 0
2: j ← 0
3: for i to 32 do
4:     for j to 63 do
5:         C[0][0][i] ← A[i] * B[0][0][j]
6:         C[0][1][i] ← A[i] * B[0][1][j]
7:         C[0][2][i] ← A[i] * B[0][2][j]
8:         C[1][0][i] ← A[i] * B[1][0][j]
9:         C[1][1][i] ← A[i] * B[1][1][j]
10:        C[1][2][i] ← A[i] * B[1][2][j]
11:        C[2][0][i] ← A[i] * B[2][0][j]
12:        C[2][1][i] ← A[i] * B[2][1][j]
13:        C[2][2][i] ← A[i] * B[2][2][j]
14:        j ← j + 1
15:    end for
16:    i ← i + 1
17: end forC
```

ARMing-sword

- ARMv8 명령어는 **연산 후 주소 값을 이동**시키는 부분이 존재
  - e.g.) LD1.16b {v0}, [x0], #16 // LD1.8h {v0}, [x0], #32
- Scabbard의 Multiplication은 **2-byte씩 이동이 필요**
- ARMv8의 명령어는 **최소 8-byte 단위**로 이동
- 따라서 명령어의 기능으로는 구현이 불가능
- 주소 값을 직접 수정
- 현재 위치한 주소에 누적까지 동시에 진행하도록 구조 변경
- Sliding Window 기법

ARMing-sword

- 구현에 사용한 소스 코드
- 반복적인 부분은 생략 됨

Algorithm 3 Implementation codes for Direct Mapping technique.

Input: $A1$ address = x0, AW address = x1
Output: $AW00$, $AW01$, $AW02$, $AW10$, $AW11$, $AW12$, $AW20$, $AW21$, $AW22$ values

```
1: LD1.8h {v16, v17}, [x0], #32
2: LD1.8h {v12, v13}, [x0], #32
3: LD1.8h {v4, v5}, [x0], #32
4: LD1.8h {v0, v1}, [x0], #32
5: ADD.8h v2, v0, v4
6: ADD.8h v3, v1, v5
7: ADD.8h v14, v12, v16
8: ADD.8h v15, v13, v17
9: ADD.8h v6, v0, v12
10: ADD.8h v7, v1, v13
11: ADD.8h v10, v4, v16
12: ADD.8h v11, v5, v17
13: ADD.8h v8, v2, v14
14: ADD.8h v9, v3, v15
15: ST1.8h {v0, v1, v2, v3}, [x1], #64
16: ST1.8h {v4, v5, v6, v7}, [x1], #64
17: ST1.8h {v8, v9, v10, v11}, [x1], #64
18: ST1.8h {v12, v13, v14, v15}, [x1], #64
19: ST1.8h {v16, v17}, [x1], #32
```

Algorithm 6 Implementation codes of Sliding Window technique for ARMing-sword.

Input: B address = x1, A address = x2
Output: multiplication result C

```
1: LD1.8h {v18, v19}, [x1], #32
2: LD1.8h {v21, v22}, [x2]
3: MUL v23.8h, v18.8h, v0.h[0]
4: MUL v24.8h, v19.8h, v0.h[0]
5: ADD.8h v21, v21, v23
6: ADD.8h v22, v22, v24
7: ST1.8h {v21, v22}, [x2]
8: ADD x2, x2, #2
```

성능 평가

- 곱셈기의 성능을 clock cycle 단위로 측정
- **최대 6.34배**의 성능 향상이 존재
- 특히 Espada의 곱셈기는 Florete/Sable보다 뛰어남

알고리즘	Scabbard	ARMing-sword
Evaluation single	272	**137.6**
Evaluation 3-way	1740.8	**329.6**
Evaluation 4-way	588.8	**92.8**
Multiplier Espada	29286.4	**8736**
Multiplier Florete/Sable	496	**425.6**

성능 평가

- 곱셈기를 사용한 성능 비교를 clock cycle x10^4로 측정
- 대부분의 경우에서 ARMing-sword의 성능이 우세
- **최대 2.17배**의 성능 개선

버전	알고리즘	Scabbard	ARMing-sword
Sable	키생성	80.8	**75.7**
	암호화	89.8	**84.1**
	복호화	93.4	**88.4**
	키+암+복	263.2	**247.5**
Espada	키생성	475.6	**230.7**
	암호화	505.4	**239.7**
	복호화	521.2	**241.7**
	키+암+복	1497.4	**720.4**
Florete	키생성	53.2	**50.8**
	암호화	72.5	**72.6**
	복호화	86.0	**78.9**
	키+암+복	222.1	**203.8**

결론

- Scabbard를 ARMv8 프로세서 상에서 최적 구현한

  ARMing-sword 구현

- **Direct Mapping, Sliding Window 최적화 기법** 제안 및 구현
- 곱셈기의 성능은 최대 6.34배 개선
- 암호 알고리즘의 성능은 최대 2.17배 개선
- 후속 과제로 Interpolation에 맞는 기법을 개발

Q & A