라운드 키 선행 로드를 사용한 CHAM 고속 구현

정보컴퓨터공학과 권혁동

한성대학교 HANSUNG UNIVERSITY

CryptoCraft LAB
https://crypto.modoo.at

Contents

CryptoCraft LAB

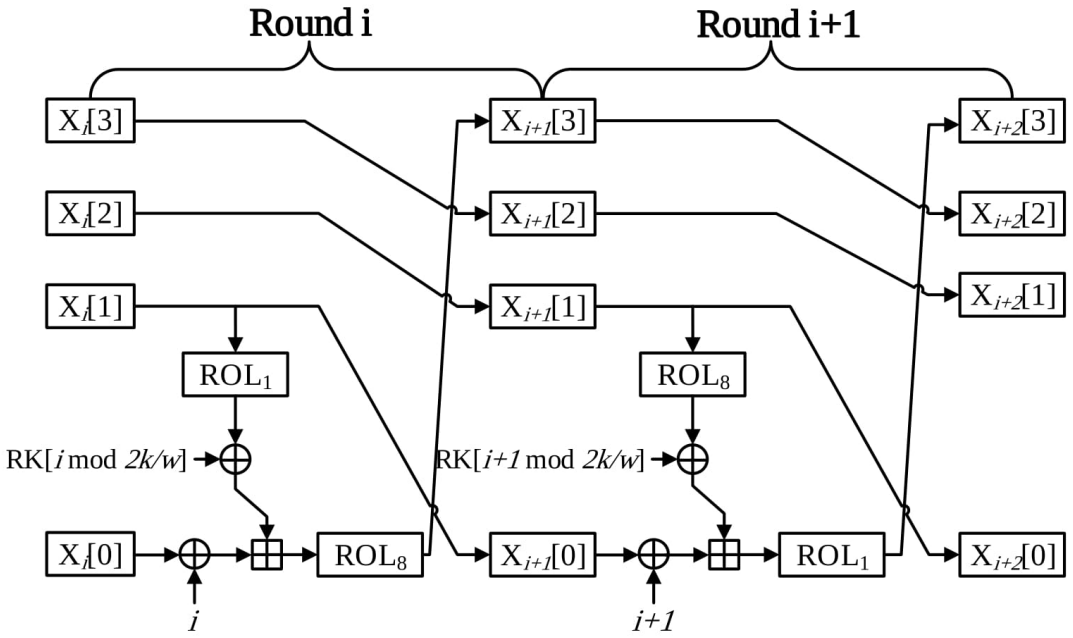
CHAM

- ICISC'17에서 발표된 국산 경량 블록암호
- Revised 버전이 ICISC'19에서 발표
  - 기존과 **라운드 수만 다르고 모든 구조가 동일**

분류	n(평문)	k(키)	r(라운드)	old-r
64/128	64	128	88	80
128/128	128	128	112	80
128/256	128	256	120	96

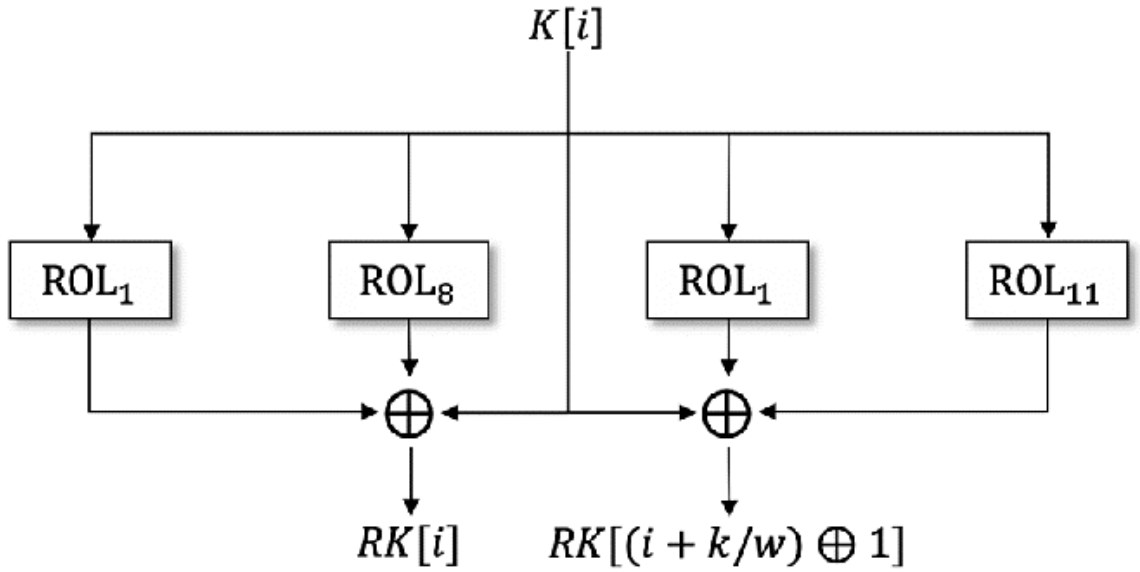
CHAM

- 4-branch Feistel 구조
- ARX 구조
- **연산과 구조가 단순하여 쉬운 구현**



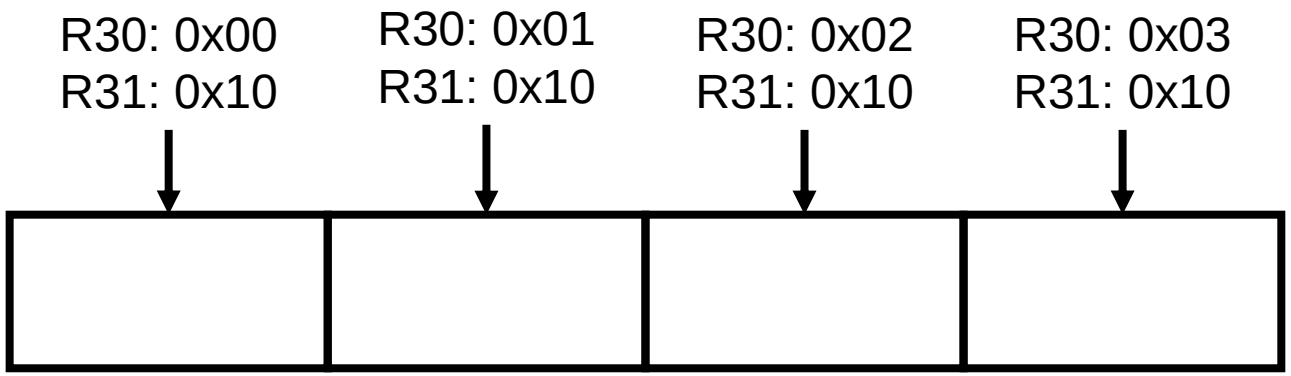
CHAM

- stateless 키 스케줄링
- 키의 상태를 저장하지 않는 기법
  - **키 저장 공간을 줄임**



8-bit AVR 최적 구현

- 라운드 키 접근 최적화
  - CHAM의 키 사이즈는 32 또는 64 바이트
  - 메모리의 접근 범위는 최대 64
  - 하위 주소가 0x00인 경우, **1바이트 오프셋 연산으로 모든 키 접근 가능**



8-bit AVR 최적 구현

- 카운터 접근 최적화
  - CHAM의 라운드 수는 최대 96
  - 이는 1바이트 레지스터 표현 범위에 들어옴
  - **레지스터 하나로 카운터 표현 가능**

8-bit AVR 최적 구현

- 메모리 접근 최적화
  - 메모리 접근 이후 자동으로 주소 이전
  - **메모리 자동 계산 명령어 활용**
- 이 구현물은 CHAM의 최적 구현

```
LD R22, X
INC R30
LD R23, X
INC R30
```

```
LD R22, X+
LD R23, X+
```

CHAM-CTR fixed-key 구현

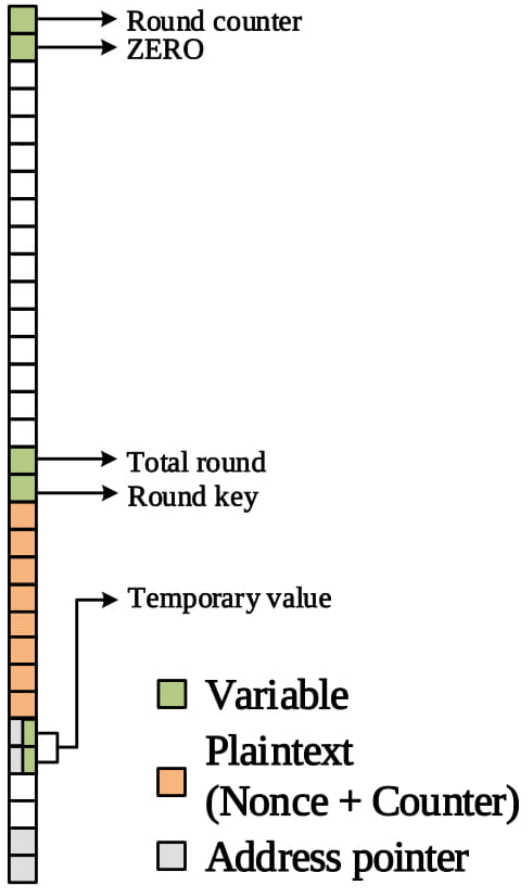
- Revised CHAM을 대상으로 한 구현
- 블록암호 운용모드 중 하나인 카운터 모드를 적용
- **입력의 논스 부분은 고정 값임을 활용한 사전 연산 기법**
- 라운드 키를 고정 상태로 알고리즘 가동
  - 라운드 키가 갱신될 때 사용이 불가능하다는 단점

CHAM-CTR variable-key 구현

- CHAM-CTR fixed-key의 개량형
- 키가 변동되는 상황에서도 유연하게 동작 가능
  - 사전연산만 진행하는 모델
  - **사전연산을 하며 1개의 블록을 암호화하는 모델**

Furious CHAM-CTR-64/128 구현

- CHAM-CTR variable-key 64/128의 개량형
- 기존 구현물에 미사용 레지스터가 16개 존재
  - 미사용 레지스터에 미리 일부 라운드 키를 로드
  - **라운드 키 로드 시간만큼 최적화 가능**



Furious CHAM-CTR-64/128 구현

- CHAM의 라운드 키
  - 64/128: 32바이트
  - 그 외: 64바이트
- **사용 가능한 레지스터는 16바이트이므로 64/128만 구현**

Furious CHAM-CTR-64/128 구현

- Fixed-key 버전 대상
- R27을 제외한 모든 레지스터가 한가지 목적으로 사용
  - R27: 평문 블록 주소 레지스터 + 라운드 키 레지스터
- **전체 라운드 숫자 레지스터는 삭제**
  - CPI 명령어를 통해 즉시 비교
- 따라서 구현에 어려움이 크게 없음

```
#define ZERO R1

#define X00 R18 // X0
#define X01 R19 // X0
#define X10 R20 // X1
#define X11 R21 // X1
#define X20 R22 // X2
#define X21 R23 // X2
#define X30 R24 // X3
#define X31 R25 // X3

#define RK0 R2
#define RK1 R3
#define RK2 R4
#define RK3 R5
#define RK4 R6
#define RK5 R7
#define RK6 R8
#define RK7 R9
#define RK8 R10
#define RK9 R11
#define RK10 R12
#define RK11 R13
#define RK12 R14
#define RK13 R15
#define RK14 R16
#define RK15 R27

#define RC R17
#define RK R0

#define XT0 R28
#define XT1 R29
```

Furious CHAM-CTR-64/128 구현

- Variable-key 버전 대상
- 사전 연산 테이블을 위한 포인터 레지스터가 두개 더 필요
- 함수의 매개변수는 R24, R22, R20 … 순서로 입력
- **평문을 로드하는 순간, 입력 받은 포인터 정보가 사라짐**

```
#define ZERO R1

#define X00 R18 // X0
#define X01 R19 // X0
#define X10 R20 // X1
#define X11 R21 // X1
#define X20 R22 // X2
#define X21 R23 // X2
#define X30 R24 // X3
#define X31 R25 // X3

#define RK0 R2
#define RK1 R3
#define RK2 R4
#define RK3 R5
#define RK4 R6
#define RK5 R7
#define RK6 R8
#define RK7 R9
#define RK8 R10
#define RK9 R11
#define RK10 R12
#define RK11 R13
#define RK12 R14
#define RK13 R15
#define RK14 R16
#define RK15 R27

#define RC R17
#define RK R0

#define XT0 R28
#define XT1 R29
```

Furious CHAM-CTR-64/128 구현

- 포인터 정보가 덮어씌워지지 않도록 **평문 레지스터를 변경**
  - 또는, 라운드 키 -> 평문 순으로 로드하는 방법도 가능

```
#define ZERO R1

#define X00 R2 // X0
#define X01 R3 // X0
#define X10 R4 // X1
#define X11 R5 // X1
#define X20 R6 // X2
#define X21 R7 // X2
#define X30 R8 // X3
#define X31 R9 // X3
```

- 평문 포인터 레지스터는 라운드 함수 동안에는

라운드 키 포인터 레지스터로 동작

```
MOVW R26, R24   // plain
MOVW R30, R22   // RK

LD X00, X+
LD X01, X+
LD X10, X+
LD X11, X+
LD X20, X+
LD X21, X+
LD X30, X+
LD X31, X+

PUSH R26
PUSH R27

MOVW R26, R20   // table
```

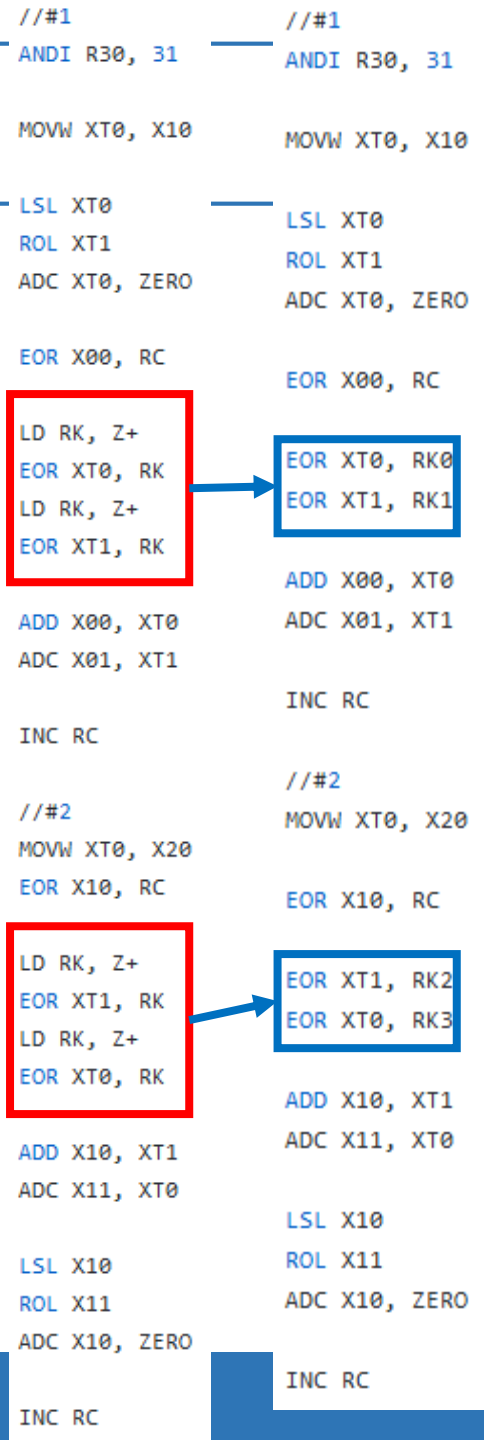
```
#define RK0 R18
#define RK1 R19
#define RK2 R20
#define RK3 R21
#define RK4 R22
#define RK5 R23
#define RK6 R24
#define RK7 R25
#define RK8 R10
#define RK9 R11
#define RK10 R12
#define RK11 R13
#define RK12 R14
#define RK13 R15
#define RK14 R16
#define RK15 R27

#define RC R17  // Round Counter
#define RK R0   // Round Key

#define XT0 R28
#define XT1 R29
```

Furious CHAM-CTR-64/128 구현

- 라운드 함수 구현 (모든 버전 공용)
- 32바이트 라운드 키 중에서 16바이트만 선행 로드
- 라운드 함수 부분이 두 가지로 구현 됨
  - 선행 로드 라운드 키를 활용하는 부분
  - 개별로 라운드 키를 로드해서 연산하는 부분
- **16-way 기반 구현으로 확장**
  - 8-way 기반 구현 시: 10회 반복
  - 16-way 기반 구현 시: 5회 반복



Furious CHAM-CTR-64/128 구현

- 전체적으로 약 7%의 성능 향상
- 코드 사이즈가 길어지는 단점
  - 레지스터를 모두 사용
  - 저장 공간은 최적 관점에 두지 않음