석사학위논문

Rust를 이용한 블록암호 LEA 메모리 보호 구현

2025년

한 성 대 학 교 대 학 원

융 합 보 안 학 과

융 합 보 안 전 공

김 상 원

석 사 학 위 논 문
지도교수 서화정

Rust를 이용한 블록암호 LEA 메모리 보호 구현

Implementation Memory Protection for block cipher LEA using Rust

2025년 6월 일

한 성 대 학 교 대 학 원

융 합 보 안 학 과

융 합 보 안 전 공

김 상 원

석 사 학 위 논 문
지도교수 서화정

Rust를 이용한 블록암호 LEA 메모리 보호 구현

Implementation Memory Protection for block cipher LEA using Rust

위 논문을 공학 석사학위 논문으로 제출함

2025년 6월 일

한 성 대 학 교 대 학 원

융 합 보 안 학 과

융 합 보 안 전 공

김 상 원

김상원의 공학 석사학위 논문을 인준함

2025년 6월 일

심사위원장 석병진 (인)

심 사 위 원 이기원 (인)

심 사 위 원 서화정 (인)

국 문 초 록

Rust를 이용한 블록암호 LEA 메모리 보호 구현

한 성 대 학 교 대 학 원
융 합 보 안 학 과
융 합 보 안 전 공
김 상 원

블록 암호 구현은 시스템 친화적인 특징이 뛰어난 C 언어로 주로 작성되어 왔다. 그러나 C 언어는 버퍼 오버플로우나 메모리 누수와 같은 고질적인 메모리 안전성 문제에 취약하다. 반면, 시스템 프로그래밍 언어 Rust는 고유한 소유권 모델을 통해 컴파일 시점에 메모리 안전성을 보장하는 강력한 대안으로 주목받고 있다. 본 논문은 Rust 언어로 LEA 블록 암호를 구현하여, 메모리 안전성을 확보하면서도 실용적인 성능을 달성할 수 있는지 실증적으로 검증하는 것을 목표로 한다. Rust 구현과 기존 구현을 최적화가 없는 환경에서 성능 비교한 결과 Rust 구현에서 성능 저하가 관찰되었다. 이를 분석하기 위해 Xcode Instruments의 Time Profiler, CPU Counter 및 디어셈블 기법을 활용하였다.

【주요어】 Rust, 메모리 안전성, LEA

목 차

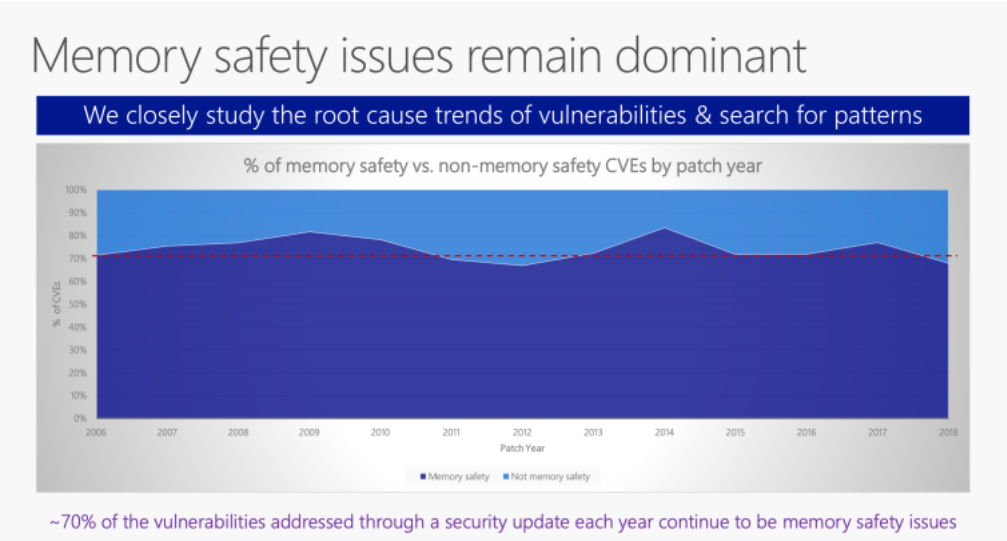
표 목 차

그 림 목 차

제 1 장 서 론

C/C++ 등 메모리 안전성이 결여된 저수준 언어는 수십 년간 심각한 보안 취약점의 근원이 되어 왔으며, 공격자는 이러한 메모리 오류를 악용하여 프로그램을 완전히 장악할 수 있다. 특히 암호 소프트웨어에서 이러한 취약점은 기밀 키 누출이나 데이터 위변조 등 치명적 결과로 직결될 수 있다. 전통적인 C 기반 암호 구현에서는 버퍼 오버플로우, use-after-free(해제 후 사용), 이중 해제(double free) 같은 메모리 오류로 인한 취약점이 다수 보고되어 왔다(Szekeres, L et al., 2013).

메모리 취약점을 완화하기 위한 다양한 방어 기법이 제안되었으나 높은 성능 오버헤드 등의 한계로 널리 채택되지 못하였고, 결국 보안성과 효율성 간 균형 문제가 대두되고 있다. 이러한 가운데 시스템 프로그래밍 언어 Rust가 메모리 안전성과 성능을 겸비한 대안으로 주목받고 있다. Rust는 소유권 기반 메모리 모델을 통해 컴파일 시점에 메모리 오류를 원천 차단하고 가비지 컬렉션 없이도 C/C++에 필적하는 실행 성능을 달성한다. Microsoft와 Google의 분석에 따르면 각각 전체 보안 취약점의 약 70%가 메모리 안전성 결함에 기인하는 것으로 나타났다(Miller, M., 2019) [그림 1-1]. 이러한 한계를 의식한 미국 국가안보국(NSA)은 Software Memory Safety 보고서(National Security Agency., 2022)에서 메모리 세이프 언어(Rust, Go, Swift 등)로의 전환을 권고하였다. 이에 발맞추어 공공 및 민간 부문에서도 Rust 등의 메모리 안전 언어를 도입하려는 움직임이 확산되고 있다.



출처 : Microsoft® (2019), "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape"

[그림 1-1] Memory Safety issues remain dominant

본 논문에서는 메모리 안전성을 확보하면서도 실용 범위 내 성능을 달성할 수 있는지를 실증적으로 검증하는 데 있다. 이를 위해 메모리 안전 언어인 Rust를 이용하여 128비트 블록 암호 알고리즘 LEA를 구현하고 그 성능을 측정하였다. LEA는 일반 프로세서 상에서 AES보다도 빠른 속도를 보이는 고속 블록 암호 알고리즘으로 제안되었으며, Rust로 구현할 경우에도 이러한 성능 우위를 유지할 수 있는지 확인하기에 적합하다. 본 연구를 통해 Rust 기반 구현이 메모리 오류를 배제하면서도 암호 알고리즘의 성능을 기존 수준으로 유지할 수 있음을 보임으로써, 향후 안전한 암호 소프트웨어 개발을 위한 실용적 방향을 제시하고자 한다. 하지만 구현물에서 C 코드 대비 성능 저하가 관측되어 오버헤드의 원인을 디어셈블 분석으로 규명하고자 한다.

제 2 장 관 련 연 구

제 1 절 LEA 블록암호

1) 개요

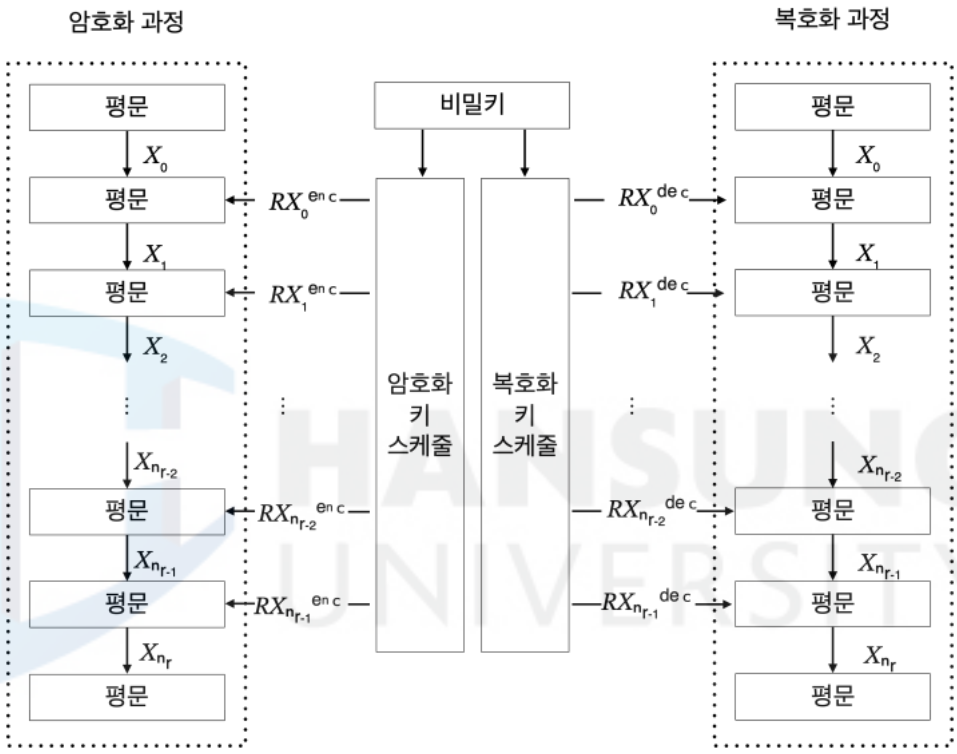
LEA(Lightweight Encryption Algorithm)는 국가보안기술연구소(NSRI, National Security Research Institute)가 2013년에 고안한 ARX(Add-Rotate-XOR) 기반 대칭키 블록암호 알고리즘이다 (Hong, D et al., 2013). LEA는 128비트 고정 블록 크기를 가지며, 128, 192, 256비트의 다양한 키 길이를 지원한다. LEA는 효율성과 경량화를 주요 목표로 삼아, 빅데이터 분석 환경이나 클라우드 컴퓨팅, 모바일 디바이스, 사물인터넷(IoT) 기기 등 자원 제약이 심한 경량 환경에서도 고속의 암호 연산을 지원할 수 있도록 한다. 이러한 특성은 소프트웨어 및 하드웨어 구현 모두에서 우수한 성능과 확장성을 보장한다.

LEA는 AES(Advanced Encryption Standard)에서 사용되는 S-box 대신 XOR 연산, 비트 회전 연산, 모듈러 덧셈과 같은 단순한 연산만으로 구성된다. 이는 하드웨어 구현 시 면적 효율성을 높여, 소프트웨어 상에서는 빠른 처리 속도를 제공한다(Seo, H. J et al., 2015). LEA는 키 길이에 따라 24, 28, 또는 32개의 라운드로 구성되며, 각 라운드에서는 모듈러 덧셈, 순환 비트 회전, XOR 연산 순서로 암호문을 생성한다. 키 스케줄링 단계에서는 키 확산(dispersion)과 상수 결합으로 다양한 공격 기법에 대한 내성을 가진다 (Seo, H. J., 2015).

이러한 간소화된 연산 구조는 저가형 임베디드 시스템이나 대량 생산 기기에서도 효율적인 하드웨어 구현이 가능하며, 전력 소모를 낮추고 충분한 암호 강도를 확보한다. 전체적인 동작 과정과 규격 등은 다음과 같다(국가보안기술연구소., 2013).

구분	Nb	Nk	Nr
LEA-128	16	16	24
LEA-192	16	24	28
LEA-256	16	32	32

표[2-1] LEA 규격



[그림 2-1] LEA 암호화 및 복호화 과정

2) 암호화

LEA의 암호화 과정은 k비트 키 K로부터 Nr개의 192비트 암호화용 라운드키 $RK_i^{enc}(0 \le i \le (Nr-1))$를 생성하는 키 스케줄 함수 $KeySchedule_k^{enc}$와, 라운드키 RK_i^{enc} 및 라운드 함수 $Round^{enc}$를 이용하여 128비트 평문 P를 128비트 암호문 C로 변환하는 암호화 함수 $Encrypt$로 구성된다.

3) 암호화 함수

LEA의 암호화 함수 *Encrypt*는 k비트 키 K에 대해 키 스케줄 함수 $KeySchedule_k^{enc}$을 수행하여 생성된 Nr개의 192비트 라운드키

$$RK_i^{enc} = (RK_i^{enc}[0], RK_i^{enc}[1], \cdots, RK_i^{enc}[5]) \quad (0 \le i \le (Nr-1))$$

와 128비트 평문 $P = (P[0], P[1], \cdots, P[15])$를 입력받아 [표 2-2]를 수행하여 128비트 암호문 $C = (C[0], C[1], \cdots, C[15])$를 출력한다.

[표 2-2] 암호화 함수

$C \leftarrow Encrypt(P, RK_0^{enc}, RK_1^{enc}, \cdots, RK_{Nr-1}^{enc})$
입력: 128비트 평문 P, Nr개의 192비트 라운드 키 $RK_0^{enc}, RK_1^{enc}, \cdots, RK_{Nr-1}^{enc}$ 출력: 128비트 암호문 C
1: $X_0 \leftarrow P$ 2: for $i = 0$ to $(Nr-1)$ *do* 3: $X_{i+1} \leftarrow Round^{enc}(X_i, RK_i^{enc})$ 4: *end* for 5: $C \leftarrow X_{Nr}$

[표 2-2]에서 $i(0 \le i \le (Nr-1))$번째 라운드의 라운드 함수 $Round^{enc}$는 128비트 내부상태 변수

$$X_i = (X_i[0], X_i[1], X_i[2], X_i[3])$$

와 192비트 라운드키

$$RK_i^{enc} = (RK_i^{enc}[0], RK_i^{enc}[1], \cdots, RK_i^{enc}[5])$$

로부터 [표 2-3]을 수행하여 새로운 128비트 내부상태 변수

$$X_{i+1} = (X_{i+1}[0], X_{i+1}[1], X_{i+1}[2], X_{i+1}[3])$$

을 생성한다.

[표 2-3] 암호화 과정의 I번째 라운드 함수

$X_i \leftarrow Round^{enc}(X_i, RK_i^{enc})$
입력: 128비트 내부상태 변수 X_i, 192비트 라운드 키 RK_i^{enc}
출력: 128비트 내부상태 변수 X_{i+1}
1: $X_{i+1}[0] \leftarrow ROL_9((X_i[0] \oplus RK_i^{enc}[0]) \boxplus (X_i[1] \oplus RK_i^{enc}[1]))$
2: $X_{i+1}[1] \leftarrow ROR_5((X_i[1] \oplus RK_i^{enc}[2]) \boxplus (X_i[2] \oplus RK_i^{enc}[3]))$
3: $X_{i+1}[2] \leftarrow ROR_3((X_i[2] \oplus RK_i^{enc}[4]) \boxplus (X_i[3] \oplus RK_i^{enc}[5]))$
4: $X_{i+1}[3] \leftarrow X_i[0]$

4) 암호화 키스케줄

키 K로부터 암호화 과정에 필요한 Nr개의 192비트 암호화 라운드키 $RK_i^{enc}(0 \le i \le (Nr-1))$를 생성하는 키 스케줄 과정을 설명한다.

키 스케줄 함수에서 사용되는 32비트 상수들 $\delta[i]\,(0 \le i \le 7)$는 다음과 같다.

$$\begin{aligned} \delta[0] &= \text{c3efe9db}, & \delta[1] &= \text{44626b02}, \\ \delta[2] &= \text{79e27c8a}, & \delta[3] &= \text{78df30ec}, \\ \delta[4] &= \text{715ea49e}, & \delta[5] &= \text{c785da0a}, \\ \delta[6] &= \text{e04ef22a}, & \delta[7] &= \text{e5c40957}. \end{aligned}$$

128비트 키 $K=(K[0],K[1],\cdots,K[15])$에 대해 LEA-128의 암호화를 위해 사용되는 키 스케줄 함수 $KeySchedule_{128}^{enc}$ 는 24개의 192비트 암호화 라운드키

$$RK_i^{enc}=(RK_i^{enc}[0],RK_i^{enc}[1],\cdots,RK_i^{enc}[5])(0\le i\le 23)$$

를 [표 2-4]와 같이 생성하며, 이 과정에서 128비트 내부상태 변수 $T=(T[0],T[1],T[2],T[3])$가 사용된다. 을 생성한다.

[표 2-4] LEA-128 암호화 키 스케줄 함수

$(RK_0^{enc},\cdots,RK_{23}^{enc}) \leftarrow KeySchedule_{128}^{enc}(K)$
입력: 128비트 비밀키 K
출력: 24개의 192비트 암호화 라운드키 $RK_i^{enc}(0\le i\le 23)$
1: $T\leftarrow K$
2: for i = 0 to 23 *do*
3: $T[0]\leftarrow ROL_1(T[0]\boxplus ROL_i(\delta[i \bmod 4]))$
4: $T[1]\leftarrow ROL_3(T[1]\boxplus ROL_{i+1}(\delta[i \bmod 4]))$
5: $T[2]\leftarrow ROL_6(T[2]\boxplus ROL_{i+2}(\delta[i \bmod 4]))$
6: $T[3]\leftarrow ROL_{11}(T[3]\boxplus ROL_{i+3}(\delta[i \bmod 4]))$
7: $RK_i^{enc}\leftarrow(T[0],T[1],T[2],T[1],T[3],T[1])$
8: end for

192비트 키 $K=(K[0],K[1],\cdots,K[23])$에 대해 LEA-192의 암호화를 위해 사용되는 키 스케줄 함수 $KeySchedule_{192}^{enc}$는 28개의 192비트 암호화 라운드키

$$RK_i^{enc}=(RK_i^{enc}[0],RK_i^{enc}[1],\cdots,RK_i^{enc}[5])(0\le i\le 27)$$

를 [표 2-5]와 같이 생성하며, 이 과정에서 192비트 내부상태 변수

$T = (T[0], T[1], \cdots, T[5])$가 사용된다.

[표 2-5] LEA-192 암호화 키 스케줄 함수

$(RK_0^{enc}, \cdots, RK_{27}^{enc}) \leftarrow KeySchedule_{192}^{enc}(K)$
입력: 192비트 비밀키 K
출력: 28개의 192비트 암호화 라운드키 $RK_i^{enc}(0 \le i \le 27)$
1: $T \leftarrow K$
2: for i = 0 to 27 *do*
3: $T[0] \leftarrow ROL_1(T[0] \boxplus ROL_i(\delta[i \bmod 6]))$
4: $T[1] \leftarrow ROL_3(T[1] \boxplus ROL_{i+1}(\delta[i \bmod 6]))$
5: $T[2] \leftarrow ROL_6(T[2] \boxplus ROL_{i+2}(\delta[i \bmod 6]))$
6: $T[3] \leftarrow ROL_{11}(T[3] \boxplus ROL_{i+3}(\delta[i \bmod 6]))$
7: $T[4] \leftarrow ROL_{13}(T[4] \boxplus ROL_{i+4}(\delta[i \bmod 6]))$
8: $T[5] \leftarrow ROL_{17}(T[5] \boxplus ROL_{i+5}(\delta[i \bmod 6]))$
9: $RK_i^{enc} \leftarrow (T[0], T[1], T[2], T[3], T[4], T[5])$
10: end for

256비트 키 $K = (K[0], K[1], \cdots, K[31])$에 대해 LEA-256의 암호화를 위해 사용되는 키 스케줄 함수 $KeySchedule_{256}^{enc}$는 32개의 192비트 암호화 라운드키

$$RK_i^{enc} = (RK_i^{enc}[0], RK_i^{enc}[1], \cdots, RK_i^{enc}[5]) (0 \le i \le 31)$$

를 [표 2-6]과 같이 생성하며, 이 과정에서 256비트 내부상태 변수 $T = (T[0], T[1], \cdots, T[7])$가 사용된다.

[표 2-6] LEA-256 암호화 키 스케줄 함수

$(RK_0^{enc}, \cdots, RK_{31}^{enc}) \leftarrow KeySchedule_{256}^{enc}(K)$
입력: 256비트 비밀키 K
출력: 32개의 192비트 암호화 라운드키 $RK_i^{enc}(0 \le i \le 31)$

1: $T \leftarrow K$
2: for i = 0 to 31 *do*
3: $T[6i \bmod 8] \leftarrow ROL_1(T[6i \bmod 8] \boxplus ROL_i(\delta[i \bmod 8]))$
4: $T[(6i+1) \bmod 8] \leftarrow ROL_3(T[(6i+1) \bmod 8] \boxplus ROL_{i+1}(\delta[i \bmod 8]))$
5: $T[(6i+2) \bmod 8] \leftarrow ROL_6(T[(6i+2) \bmod 8] \boxplus ROL_{i+2}(\delta[i \bmod 8]))$
6: $T[(6i+3) \bmod 8] \leftarrow ROL_{11}(T[(6i+3) \bmod 8] \boxplus ROL_{i+3}(\delta[i \bmod 8]))$
7: $T[(6i+4) \bmod 8] \leftarrow ROL_{13}(T[(6i+4) \bmod 8] \boxplus ROL_{i+4}(\delta[i \bmod 8]))$
8: $T[(6i+5) \bmod 8] \leftarrow ROL_{17}(T[(6i+5) \bmod 8] \boxplus ROL_{i+5}(\delta[i \bmod 8]))$
9: $RK_i^{enc} \leftarrow (T[6i \bmod 8], T[6i \bmod 8+1], T[6i \bmod 8+2],$
$T[6i \bmod 8+3], T[6i \bmod 8+4], T[6i \bmod 8+5])$
10: end for

5) 복호화

LEA의 복호화 과정은 k비트 키 K로부터 Nr개의 192비트 복호화용 라운드키 $RK_i^{dec}(0 \le i \le (Nr-1))$를 생성하는 키 스케줄 함수 $KeySchedule_k^{dec}$와, 라운드키 RK_i^{dec} 및 라운드 함수 $Round^{dec}$를 이용하여 128비트 암호문 C를 128비트 평문 P로 변환하는 복호화 함수 *Decrypt*로 구성된다.

6) 복호화 함수

LEA의 복호화 함수 *Decrypt*는 k비트 키 K에 대해 키 스케줄 함수 $KeySchedule_k^{dec}$을 수행하여 생성된 Nr개의 192비트 라운드키

$$RK_i^{dec} = (RK_i^{dec}[0], RK_i^{dec}[1], \cdots, RK_i^{dec}[5]) \quad (0 \le i \le (Nr-1))$$

와 128비트 암호문 $C = (C[0], C[1], \cdots, C[15])$를 입력받아 [표 2-7]을 수행하여 128비트 평문 $P = (P[0], P[1], \cdots, P[15])$를 출력한다.

[표 2-7] 복호화 함수

$P \leftarrow Decrypt(C, RK_0^{dec}, RK_1^{dec}, \cdots, RK_{Nr-1}^{dec})$
입력: 128비트 암호문 C, Nr개의 192비트 라운드 키 $RK_0^{dec}, RK_1^{dec}, \cdots, RK_{Nr-1}^{dec}$
출력: 128비트 평문 P
1: $X_0 \leftarrow C$
2: for $i = 0$ to $(Nr-1)$ *do*
3: $X_{i+1} \leftarrow Round^{dec}(X_i, RK_i^{dec})$
4: *end* for
5: $P \leftarrow X_{Nr}$

[표 2-7]에서 $i(0 \le i \le (Nr-1))$번째 라운드의 라운드 함수 $Round^{dec}$는 128비트 내부상태 변수

$$X_i = (X_i[0], X_i[1], X_i[2], X_i[3])$$

와 192비트 라운드키

$$RK_i^{dec} = (RK_i^{dec}[0], RK_i^{dec}[1], \cdots, RK_i^{dec}[5])$$

로부터 [표 2-8]를 수행하여 새로운 128비트 내부상태 변수

$$X_{i+1} = (X_{i+1}[0], X_{i+1}[1], X_{i+1}[2], X_{i+1}[3])$$

을 생성한다.

[표 2-8] 복호화 과정의 I번째 라운드 함수

$X_{i+1} \leftarrow Round^{dec}(X_i, RK_i^{dec})$
입력: 128비트 내부상태 변수 X_i, 192비트 라운드 키 RK_i^{dec}

출력: 128비트 내부상태 변수 X_{i+1}

1: $X_{i+1}[0] \leftarrow X_i[0]$
2: $X_{i+1}[1] \leftarrow (ROR_9(X_i[0]) \boxminus (X_{i+1}[0] \oplus RK_i^{dec}[0])) \oplus RK_i^{dec}[1]$
3: $X_{i+1}[2] \leftarrow (ROL_5(X_i[1]) \boxminus (X_{i+1}[1] \oplus RK_i^{dec}[2])) \oplus RK_i^{dec}[3]$
4: $X_{i+1}[3] \leftarrow (ROL_3(X_i[2]) \boxminus (X_{i+1}[2] \oplus RK_i^{dec}[4])) \oplus RK_i^{dec}[5]$

다음은 복호화 과정의 I번째 라운드 함수를 도식화한 것이다.

7) 복호화 키스케줄

키 K로부터 복호화 과정에 필요한 Nr개의 복호화 라운드키 RK_i^{dec} $(0 \le i \le (Nr-1))$들을 생성하는 키 스케줄 과정을 설명한다. 암호화 라운드키와 복호화 라운드키는

$$RK_i^{dec} = RK_{Nr-i-1}^{enc} \ (0 \le i \le (Nr-1))$$

인 관계를 제외하면 동일한 방법으로 생성되며, 동일한 상수가 사용된다.

128비트 키 $K = (K[0], K[1], \cdots, K[15])$에 대해 LEA-128의 복호화를 위해 사용되는 키 스케줄 함수 $KeySchedule_{128}^{dec}$ 는 24개의 192비트 복호화 라운드키

$$RK_i^{dec} = (RK_i^{dec}[0], RK_i^{dec}[1], \cdots, RK_i^{dec}[5]) (0 \le i \le 23)$$

를 [표 2-9]과 같이 생성하며, 이 과정에서 128비트 내부상태 변수 $T = (T[0], T[1], T[2], T[3])$가 사용된다. 을 생성한다.

[표 2-9] LEA-128 복호화 키 스케줄 함수

$(RK_0^{dec}, \cdots, RK_{23}^{dec}) \leftarrow KeySchedule_{128}^{dec}(K)$
입력: 128비트 비밀키 K
출력: 24개의 192비트 복호화 라운드키 $RK_i^{dec}(0 \le i \le 23)$
1: $T \leftarrow K$
2: for i = 0 to 23 *do*
3: $T[0] \leftarrow ROL_1(T[0] \boxplus ROL_i(\delta[i \bmod 4]))$
4: $T[1] \leftarrow ROL_3(T[1] \boxplus ROL_{i+1}(\delta[i \bmod 4]))$
5: $T[2] \leftarrow ROL_6(T[2] \boxplus ROL_{i+2}(\delta[i \bmod 4]))$
6: $T[3] \leftarrow ROL_{11}(T[3] \boxplus ROL_{i+3}(\delta[i \bmod 4]))$
7: $RK_{23-i}^{dec} \leftarrow (T[0], T[1], T[2], T[1], T[3], T[1])$
8: end for

192비트 키 $K = (K[0], K[1], \cdots, K[23])$에 대해 LEA-192의 복호화를 위해 사용되는 키 스케줄 함수 $KeySchedule_{192}^{dec}$는 28개의 192비트 복호화 라운드 키

$$RK_i^{dec} = (RK_i^{dec}[0], RK_i^{dec}[1], \cdots, RK_i^{dec}[5])(0 \le i \le 27)$$

를 [표 2-10]과 같이 생성하며, 이 과정에서 192비트 내부상태 변수 $T = (T[0], T[1], \cdots, T[5])$가 사용된다.

[표 2-10] LEA-192 복호화 키 스케줄 함수

$(RK_0^{dec}, \cdots, RK_{27}^{dec}) \leftarrow KeySchedule_{192}^{dec}(K)$
입력: 192비트 비밀키 K
출력: 28개의 192비트 복호화 라운드키 $RK_i^{dec}(0 \le i \le 27)$
1: $T \leftarrow K$
2: for i = 0 to 27 *do*
3: $T[0] \leftarrow ROL_1(T[0] \boxplus ROL_i(\delta[i \bmod 6]))$
4: $T[1] \leftarrow ROL_3(T[1] \boxplus ROL_{i+1}(\delta[i \bmod 6]))$
5: $T[2] \leftarrow ROL_6(T[2] \boxplus ROL_{i+2}(\delta[i \bmod 6]))$
6: $T[3] \leftarrow ROL_{11}(T[3] \boxplus ROL_{i+3}(\delta[i \bmod 6]))$

7: $T[4] \leftarrow ROL_{13}(T[4] \boxplus ROL_{i+4}(\delta[i \bmod 6]))$
8: $T[5] \leftarrow ROL_{17}(T[5] \boxplus ROL_{i+5}(\delta[i \bmod 6]))$
9: $RK_i^{dec} \leftarrow (T[0], T[1], T[2], T[3], T[4], T[5])$
10: end for

256비트 키 $K = (K[0], K[1], \cdots, K[31])$에 대해 LEA-256의 복호화를 위해 사용되는 키 스케줄 함수 $KeySchedule_{256}^{dec}$는 32개의 192비트 암호화 라운드 키

$$RK_i^{dec} = (RK_i^{dec}[0], RK_i^{dec}[1], \cdots, RK_i^{dec}[5]) (0 \le i \le 31)$$

를 [표 2-11]와 같이 생성하며, 이 과정에서 256비트 내부상태 변수 $T = (T[0], T[1], \cdots, T[7])$가 사용된다.

[표 2-11] LEA-256 복호화 키 스케줄 함수

$(RK_0^{dec}, \cdots, RK_{31}^{dec}) \leftarrow KeySchedule_{256}^{dec}(K)$

입력: 256비트 비밀키 K
출력: 32개의 192비트 암호화 라운드키 $RK_i^{dec} (0 \le i \le 31)$

1: $T \leftarrow K$
2: for i = 0 to 31 *do*
3: $T[6i \bmod 8] \leftarrow ROL_1(T[6i \bmod 8] \boxplus ROL_i(\delta[i \bmod 8]))$
4: $T[(6i+1) \bmod 8] \leftarrow ROL_3(T[(6i+1) \bmod 8] \boxplus ROL_{i+1}(\delta[i \bmod 8]))$
5: $T[(6i+2) \bmod 8] \leftarrow ROL_6(T[(6i+2) \bmod 8] \boxplus ROL_{i+2}(\delta[i \bmod 8]))$
6: $T[(6i+3) \bmod 8] \leftarrow ROL_{11}(T[(6i+3) \bmod 8] \boxplus ROL_{i+3}(\delta[i \bmod 8]))$
7: $T[(6i+4) \bmod 8] \leftarrow ROL_{13}(T[(6i+4) \bmod 8] \boxplus ROL_{i+4}(\delta[i \bmod 8]))$
8: $T[(6i+5) \bmod 8] \leftarrow ROL_{17}(T[(6i+5) \bmod 8] \boxplus ROL_{i+5}(\delta[i \bmod 8]))$
9: $RK_i^{dec} \leftarrow (T[6i \bmod 8], T[6i \bmod 8 + 1], T[6i \bmod 8 + 2],$
$T[6i \bmod 8 + 3], T[6i \bmod 8 + 4], T[6i \bmod 8 + 5])$
10: end for

8) 한계점

개요에서 언급한 다양한 장점에도 불구하고, C 언어의 구조적 특성으로 인해 LEA 구현에는 여전히 여러 한계가 존재한다(Szekeres, L et al., 2013). C 언어는 시스템 프로그래밍에서 널리 사용되는 언어로서 저수준 메모리 제어가 가능하다는 장점을 갖지만, 동시에 프로그래머에게 메모리 관리의 책임을 전적으로 위임한다. 이러한 자유도는 개발자의 실수로 인한 메모리 안전성 문제(memory safety issue)를 야기할 수 있으며, 이는 보안상 중대한 취약점으로 이어질 가능성이 크다. 특히 암호화 알고리즘과 같이 보안 민감도가 높은 응용 프로그램에서 메모리 오류는 공격자에게 치명적인 공격 벡터를 제공한다. 공격자는 이를 이용하여 프로그램의 정상 동작을 교란하거나 실행 흐름을 완전히 장악할 수 있다. 이러한 취약점은 다음 범주를 포함하나 이에 국한되지 않는다:

- 댕글링 포인터(dangling pointer) : 프로그램이 활성 객체를 실수로 해제한 후 해당 포인터가 여전히 참조될 경우, 메모리 관리자에 의해 해당 객체의 내용이 다른 데이터로 덮어써질 수 있다. 이를 악용하면 공격자가 임의의 데이터를 주입할 수 있다.
- 이중 해제(double free) : 이미 해제된 객체에 대해 여러 차례 free() 호출이 발생하면 프리리스트(freelist) 기반 할당자가 오작동할 수 있으며, 이는 메모리 할당 구조를 손상시켜 임의 코드 실행이나 시스템 크래시를 유발할 수 있다.
- 버퍼 오버플로우(buffer overflow) : 프로그램이 할당된 버퍼의 경계를 벗어나 데이터를 기록할 경우, 인접한 메모리 영역의 데이터가 손상된다. 이는 함수 포인터, 반환 주소 등을 덮어써 공격자가 프로그램 흐름을 임의로 변경하는 데 악용될 수 있다.
- 힙 메타데이터 덮어쓰기(heap metadata overwrite) : 힙 메타데이터가 객체 인근에 저장되는 경우, 범위를 벗어난 쓰기로 인해 메타데이터가 손상될 수 있다. 이는 동적 메모리 관리자의 동작을 예측 불가능하게 만들며,

공격자가 메모리 관리 구조를 조작하는 기반이 된다.

- 초기화되지 않은 읽기(uninitialized read) : 새로 할당되었거나 아직 초기화되지 않은 메모리 영역에서 값을 읽을 경우, 예상치 못한 값이 사용되어 정의되지 않은 동작(undefined behavior)이 발생할 수 있다. 이는 암호 알고리즘의 출력에 예기치 않은 영향을 미치거나 보안 강도를 약화시킬 수 있다.

이와 같은 메모리 취약점은 일회성 오류에 그치지 않고, 공격자가 정교하게 조작할 경우 원격 코드 실행(remote code execution), 정보 유출(information leakage), 권한 상승(privilege escalation)과 같은 심각한 보안 사고로 이어질 수 있다. 특히, 암호 소프트웨어는 높은 수준의 신뢰성과 정확성이 요구되므로, 이러한 취약점을 방지하기 위한 메모리 안전성 확보가 매우 중요하다. C 언어는 구조적으로 이러한 오류를 컴파일 시점에 방지할 수 없으며, 별도의 정적 분석 도구나 런타임 보호 기법에 의존해야 한다는 점도 한계로 지적된다. 따라서 메모리 안전성이 내재된 언어적 대안의 필요성이 대두되고 있다.

제 2 절 Rust(프로그래밍 언어)

1) 개요

Rust는 2010년 Mozilla Research에서 개발을 시작한 시스템 프로그래밍 언어로, 2015년 1.0 정식 버전이 공개되었다. Rust는 C 및 C++와 유사한 저수준 제어권과 성능을 제공하면서, 메모리 안전성 및 동시성 보장을 주요 목표로 설계된 언어다. 기존 C/C++ 언어는 개발자에게 메모리 관리의 자유도를 부여하지만, 이는 버퍼 오버플로우(buffer overflow), use-after-free(해제 후 사용), 이중 해제(double free) 등 치명적인 메모리 오류를 발생시키는 주요 원인이 되어 왔다. Rust는 이러한 문제를 해결하기 위해 소유권

(ownership), 빌림(borrowing), 수명(lifetime)이라는 개념을 언어 차원에서 제공한다.

Rust의 소유권 모델은 변수의 소유자를 명확하게 정의하여 소유자가 변경될 경우 컴파일러가 이를 추적한다. 또한 빌림을 통해 값에 대한 참조를 안전하게 관리하며, 수명 분석을 통해 참조 유효 범위를 엄격하게 검증한다. 이러한 메커니즘은 컴파일 타임에 메모리 오류 가능성을 원천적으로 차단하여 런타임 시 예기치 않은 동작을 방지한다. Rust는 이 과정에서 가비지 컬렉션(garbage collection)을 사용하지 않으며, C/C++ 수준의 실행 성능을 유지한다는 점에서 기존 언어와 차별화된다(Crichton, W et al., 2023).

Rust는 이러한 메모리 안전성과 고성능을 동시에 요구하는 응용 분야에서 점차 채택이 증가하고 있다. 예를 들어 Microsoft는 Windows 커널 및 일부 사용자 모드 구성요소에서 Rust를 적용하고 있으며, Google은 Chrome 브라우저의 일부 모듈을 Rust로 개발하고 있다. 또한 Amazon Web Services(AWS), Dropbox 등 주요 기업도 Rust를 사용하여 성능과 안정성이 요구되는 시스템을 구축하고 있다. 미국 국가안보국(NSA)은 2023년 발간한 Software Memory Safety보고서에서 메모리 안전 언어로 Rust를 명시하고 C/C++ 대체 언어로 적극 활용할 것을 권장하였다.

Rust는 현대 프로세서의 성능을 최대한 활용할 수 있는 제로코스트 추상화(zero-cost abstraction)를 제공하며, 다양한 병렬 프로그래밍 모델과 안전한 동시성 지원을 내장하고 있다. 이를 통해 데이터 레이스(data race) 없는 멀티스레드 프로그래밍이 가능하며, 고성능 서버, 임베디드 시스템, 블록체인 플랫폼 등에서 활용도가 높아지고 있다. 또한 Rust 생태계는 Cargo 패키지 관리 도구, Crates.io 패키지 저장소 등으로 구성되어 개발 생산성을 크게 향상시키고 있다.

이러한 기술적 우수성으로 인해 Rust는 IEEE Spectrum, Stack Overflow 등 여러 개발자 조사에서 '가장 선호하는 프로그래밍 언어'로 지속적으로 선정되고 있다. 특히 보안 취약점이 발생하기 쉬운 시스템 소프트웨어 영역에서 Rust의 채택이 빠르게 확산되고 있으며, 이는 학계와 산업계 모두에서 활발한 연구 주제가 되고 있다.

Rust는 고유한 소유권 시스템을 통해 컴파일 시점에 규칙을 검사함으로써 메모리 안전성을 달성한다. 이 시스템은 가비지 컬렉터 없이도 널 포인터 역참조, 버퍼 오버플로, 메모리 누수와 같은 일반적인 버그를 제거한다.

2) 소유권 규칙

Rust에서는 모든 데이터 조각이 단일 소유자를 가지며, 소유권은 프로그램의 한 부분에서 다른 부분으로 이전될 수 있다. 이 소유권 시스템은 변수와 같은 소유자가 스코프를 벗어날 때 해당 메모리를 자동으로 해제하여 메모리 누수를 방지한다(Crichton, W., 2020).

차용(위임)도 Rust 메모리 관리의 중요한 측면이다. Rust는 참조를 통해 데이터를 차용할 수 있게 하며, 이 참조는 소유자보다 짧은 수명을 가져야 한다. 이를 통해 참조가 가리키는 데이터가 해제된 후에도 참조가 남아 있는 댕글링 포인터를 방지한다. 또한 Rust는 참조에 대해 엄격한 규칙을 적용한다. 동시에 여러 개의 불변 참조 또는 하나의 가변 참조만 허용하며, 둘을 동시에 허용하지 않는다. 이 규칙은 데이터가 여러 곳에서 읽히는 동안 변경되지 않거나, 단일 장소에서만 변경되어 동시에 읽히지 않도록 보장하여 데이터 경쟁을 효과적으로 방지한다.

Rust 컴파일러는 차용 검사기를 통해 이러한 소유권 및 차용 규칙을 강제한다. 차용 검사기는 코드 내 모든 참조를 분석하여 수명 규칙 준수를 확인하며, 변수의 수명과 사용을 검증한다. 이를 통해 동일 데이터에 대해 두 개 이상의 가변 참조가 동시에 존재하지 않고, 참조가 원본보다 오래 지속되지 않도록 보장한다.

이러한 컴파일 시점의 소유권 및 차용 규칙 강제는 널 포인터 역참조나 버퍼 오버플로 같은 일반적인 메모리 안전성 문제를 Rust에서 사실상 불가능하게 만든다.

3) 빌림(borrowing)

Rust의 빌림(borrowing) 개념은 메모리 안전성과 효율적인 데이터 관리를 동시에 달성하기 위한 핵심 설계 요소다. Rust는 기본적으로 데이터에 대한 소유권(ownership)을 엄격하게 관리하지만, 프로그래밍 과정에서 소유권을 일일이 이전(move)하지 않고도 데이터를 참조할 필요가 빈번하게 발생한다. 이러한 요구를 충족시키기 위해 Rust는 변수의 값을 복사(copy)하거나 이동(move)하지 않고도 참조(reference)를 통해 데이터를 일시적으로 '빌려' 사용할 수 있는 메커니즘을 제공한다.

빌림은 크게 두 가지 형태로 구분된다. 첫째는 불변 빌림(immutable borrowing)이다. 이는 데이터를 읽기 전용으로 참조할 때 사용하며, 동일한 데이터에 대해 여러 개의 불변 참조가 동시에 존재할 수 있다. 이러한 구조는 다중 읽기 작업 시 안전성을 보장한다. 둘째는 가변 빌림(mutable borrowing)이다. 이는 데이터를 수정할 때 사용되며, 안전성을 위해 동일 시점에 오직 하나의 가변 참조만 허용된다. 이러한 규칙은 데이터 경쟁(data race)과 같은 동시성 오류를 원천 차단한다.

Rust 컴파일러는 빌림 규칙을 강제 적용하기 위해 빌림 검사기(borrow checker)를 내장하고 있다. 빌림 검사기는 코드 내 모든 참조를 정적 분석하여 빌림의 유효 범위(scope)를 추적하고, 규칙 위반이 발생할 경우 컴파일 단계에서 오류를 발생시킨다(Pearce, D. J., 2021). 예를 들어, 가변 참조가 존재하는 동안 불변 참조를 생성하려 하거나, 빌림이 유효 범위를 벗어난 후 참조를 사용하려 할 경우 컴파일 오류가 발생한다. 이러한 강제성은 런타임 오류 발생 가능성을 줄이고, 프로그램의 신뢰성을 크게 향상시킨다.

빌림은 메모리 안전성을 확보하면서도 성능 상의 오버헤드가 없다는 점에서 큰 장점을 지닌다. 일반적인 가비지 컬렉션 기반 언어에서는 런타임에 메모리 관리를 수행하지만, Rust는 빌림을 통해 컴파일 타임에 메모리 안전성을 검증하므로 실행 시 성능 저하가 없다. 또한 빌림을 적극 활용하면 불필요한 데이터 복사를 줄여 메모리 사용 효율성을 높일 수 있다.

빌림은 시스템 프로그래밍뿐 아니라 고성능 서버, 블록체인, 임베디드 시스템 등 성능과 안정성이 모두 중요한 분야에서 Rust의 채택을 증가시키는 중요한 요인이다. 빌림 메커니즘은 러스트 생태계 전반에 걸쳐 폭넓게 사용되

며, Rust 프로그래머가 안전하고 효율적인 코드를 작성하도록 돕는 핵심 원칙으로 자리잡고 있다.

4) 수명(lifetime)

Rust에서 수명(lifetime)은 참조(reference)의 유효 기간을 명시적 또는 암묵적으로 정의하여 메모리 안전성을 강화하는 개념이다. Rust는 참조 기반 빌림(borrowing)을 적극 활용하기 때문에, 참조가 원본 데이터보다 더 오래 유지되면 댕글링 포인터(dangling pointer)와 같은 심각한 오류가 발생할 수 있다. 이러한 문제를 방지하기 위해 Rust는 모든 참조에 대해 '수명'을 정의하고, 컴파일 타임에 참조가 안전하게 사용되는지 검증한다.

Rust에서 수명은 명시적 lifetime 파라미터('a와 같은 형식)를 사용해 표시할 수 있으며, 때로는 컴파일러가 수명 추론(lifetime inference)을 통해 자동으로 처리하기도 한다. 수명은 참조가 유효한 범위를 의미하며, 참조의 유효 범위가 원본 데이터의 유효 범위를 절대 초과할 수 없다. 이 원칙은 참조가 원본 데이터보다 오래 살아남지 않도록 보장하며, 댕글링 포인터 발생을 원천적으로 차단한다.

수명 시스템은 Rust의 메모리 안전성 보장에 있어 필수적인 요소이며, 런타임 오버헤드 없이 메모리 오류를 예방하는 강력한 도구다. Rust 프로그래머는 수명 개념을 정확히 이해하고 활용함으로써 보다 안정적이고 신뢰성 높은 소프트웨어를 구현할 수 있으며, 이는 Rust가 메모리 안전성이 필수적인 분야에서 각광받는 중요한 이유 중 하나다.

5) 불변 참조와 가변 참조

Rust는 메모리 안전성과 동시성 제어를 강화하기 위해 불변 참조와 가변 참조를 구분한다.

- 불변 참조: 데이터를 변경하지 않고 읽기만 허용한다. 동일 데이터에 대해 여러 개의 불변 참조가 동시에 존재할 수 있어 안전한 동시 읽기를 가능하게 한다. 데이터가 참조되는 동안 변경되지 않음이 보장되므로 데이터 경쟁을 방지하고 일관성을 유지한다.

```
1    let x = 5;
2    let y = &x  // x에 대한 불변 참조
3    println!("y의 값은: {}", y)
```

표[2-12] Rust 불변 참조 예시

- 가변 참조: 데이터를 읽고 수정할 수 있다. 그러나 안전성을 유지하기 위해 특정 데이터에 대해 동시에 하나의 가변 참조만 허용한다. 이 독점성은 다른 참조가 동시에 데이터에 접근하지 못하도록 하여 경쟁 상태를 방지하고, 데이터가 읽히거나 쓰이는 도중 예기치 않게 변경되지 않도록 보장한다.

```
1    let mut x = 5;
2    let y = &mut x  // x에 대한 가변 참조
3    *y += 1
4    println!("x의 값은: {}", x)
```

표[2-13] Rust 가변 참조 예시

이러한 기능을 통해 Rust 프로그램은 성능과 신뢰성을 동시에 달성한다. 특히 시스템 수준 및 임베디드 애플리케이션에서 성능 및 안전성이 핵심적인 환경에서 중요한 이점을 제공한다. Rust는 가비지 컬렉터의 런타임 오버헤드 없이 이러한 기능을 제공하는 점이 성능 중요 애플리케이션에서 큰 이점이다.

제 3 장 메모리 안전성

제 1 절 메모리 안전성의 중요성 및 배경

메모리 안전성은 현대 컴퓨터 시스템에서 가장 중요한 보안 요구 사항 중 하나로, 프로그램이 할당된 메모리 범위를 벗어나지 않도록 보장함으로써 보안 취약점 발생을 방지하는 개념이다. 소프트웨어가 의도하지 않은 메모리 공간에 접근하거나 잘못된 메모리 관리를 수행할 경우, 공격자는 이를 악용하여 시스템의 정상적인 동작을 교란하거나 원격 코드 실행, 권한 상승 등 고위험 공격을 수행할 수 있다. 따라서 메모리 안전성은 운영 체제, 애플리케이션, 암호화 알고리즘 등 모든 소프트웨어 계층에서 필수적으로 확보되어야 하는 특성이다.

현대 소프트웨어의 복잡도와 연결성이 증가함에 따라 시스템의 공격 표면(attack surface)은 지속적으로 확장되고 있다. 특히 네트워크 연결이 기본인 클라우드 서비스, IoT 기기, 모바일 플랫폼에서는 외부로부터의 공격 가능성이 높아지며, 메모리 오류가 공격 초기 단계에서 주요한 벡터로 활용되고 있다. 실제로 Microsoft는 자사 분석 결과 Windows에서 발견된 보안 결함의 70%가 메모리 안전성 부족에서 비롯되었다고 보고하였으며, Google 또한 Chromium 브라우저에서 발생한 전체 버그 중 해제 후 사용(use-after-free)으로 인한 것이 36%, 기타 메모리 관련 오류가 33%를 차지한다고 발표하였다(Miller, M., 2019; Internet Security Research Group., 2022). 이러한 수치는 메모리 오류가 여전히 심각한 보안 위협으로 자리하고 있음을 명확히 보여준다.

최근 공급망 공격(supply chain attack) 사례에서도 메모리 취약점이 주요 공격 벡터로 활용되고 있다. SolarWinds 사건, Microsoft Exchange Server 침해 사례 등에서는 메모리 오류 하나가 전체 시스템 침해로 이어지는 고도

화된 공격이 확인되었다. 이러한 사례는 메모리 안전성 부족이 단순한 소프트웨어 오류 차원을 넘어 국가 기반시설과 글로벌 서비스의 보안까지 위협하는 요소로 작용하고 있음을 보여준다.

Szekeres, Laszlo, et al.가 작성한 SoK: Eternal War in Memory 논문에서도 지적하듯, 메모리 오류 기반 공격은 수십 년간 지속되어 온 고질적 문제이며, 메모리 안전성을 확보하지 못한 시스템에서는 새로운 방어 기법이 등장하더라도 근본적인 해결이 어렵다는 점이 강조되고 있다. 특히 저수준 언어(C, C++) 기반으로 작성된 시스템 소프트웨어는 성능상의 이점을 제공하지만, 메모리 관리 책임이 개발자에게 전적으로 위임되므로 오류 발생 가능성이 높다. 이러한 구조적 한계로 인해 버퍼 오버플로, use-after-free, double free, 힙 메타데이터 손상, 초기화되지 않은 읽기 등 다양한 유형의 취약점이 지속적으로 발견되고 있다.

이러한 문제에 대응하기 위한 기존 기술들도 개발되어 왔다. 정적 분석(static analysis), 동적 분석(dynamic analysis), 컴파일러 기반 탐지(Address Sanitizer, Control Flow Integrity) 등이 대표적이다. 그러나 이러한 기법들은 런타임 오버헤드와 탐지 불완전성 등의 한계가 존재하며, 구조적인 해결에는 미치지 못하고 있다. 이에 따라 NSA는 2023년 Software Memory Safety 보고서를 통해 메모리 안전성을 보장하는 언어의 사용을 적극 권고하고 있으며, Microsoft, Google 등 주요 기업은 Rust와 같은 메모리 안전 언어를 도입하여 구조적 해결을 시도하고 있다.

요컨대, 메모리 안전성은 오늘날 소프트웨어 보안에서 단순한 기술적 고려사항이 아니라, 전체 시스템 신뢰성 확보의 핵심 요소로 자리매김하고 있다. 본 연구에서는 이러한 관점에서 메모리 안전성을 확보할 수 있는 새로운 접근법의 필요성과 그 효과성을 검증하고자 한다.

제 2 절 메모리 오류 및 공격 기법

메모리 오류는 소프트웨어가 메모리를 잘못 할당하거나 해제하거나, 잘못된 위치에 접근함으로써 발생하는 결함을 의미한다. 이러한 오류는 단순한 프로그램 충돌에 그치지 않고, 공격자가 이를 악용하여 악의적 행위를 수행할 수 있는 강력한 공격 벡터로 작용한다. 특히 C 및 C++과 같은 저수준 언어로 작성된 소프트웨어는 메모리 관리 책임이 개발자에게 전적으로 위임되므로, 이러한 오류가 발생할 가능성이 높으며, 자동 검출이 쉽지 않다.

가장 대표적인 메모리 오류는 버퍼 오버플로이다. 이는 프로그램이 할당된 버퍼의 경계를 넘어 데이터를 기록함으로써 발생한다. 초과 데이터는 인접한 메모리 공간을 덮어쓰며, 함수 포인터, 반환 주소 등 중요한 제어 흐름 데이터를 변경할 수 있다. 공격자는 이를 이용해 원격 코드 실행을 달성할 수 있다.

두 번째로 흔한 오류는 use-after-free(UAF)이다. 이는 이미 해제된 메모리 공간을 프로그램이 재사용할 때 발생한다. 해제된 메모리 영역은 다른 프로세스에 할당될 수 있으며, 공격자는 이를 악용해 해당 메모리에 악의적 데이터를 삽입한 뒤 프로그램이 이를 참조하게 유도할 수 있다. UAF는 최근까지도 심각한 위협으로 자리잡고 있으며, Google Chrome의 CVE-2021-21148 취약점(UAF 기반) 사례가 이를 잘 보여준다(Microsoft Security Response Center., 2021).

이중 해제는 이미 해제된 메모리를 다시 해제하는 오류로, 메모리 할당자의 내부 상태를 교란할 수 있다. 이를 통해 힙 구조를 조작하거나, 임의 주소에 쓰기(write-what-where)를 유도할 수 있다.

또한, 힙 메타데이터 손상은 공격자가 힙 관리 정보를 조작하여 메모리 할당 과정 자체를 장악할 수 있도록 한다. 이를 통해 메모리 내 중요한 데이터 구조를 파괴하거나, 악의적 페이로드(payload)를 삽입하는 고급 공격이 가능하다.

초기화되지 않은 메모리 읽기 역시 주요 오류 중 하나다. 초기화되지 않은 메모리에는 이전 값이 남아 있을 수 있으며, 이를 통해 민감한 정보(예: 암호화 키, 패스워드)가 유출될 수 있다. 이러한 문제는 특히 암호 알고리즘 구현

시 심각한 보안 결함으로 이어질 수 있다.

이러한 오류들은 단순한 프로그래밍 실수로 발생하는 것처럼 보일 수 있지만, 현대 공격자는 이를 체계적으로 분석하고 자동화된 공격 도구(fuzzing)를 통해 취약점을 탐색한다. SoK: Eternal War in Memory. 논문은 이러한 공격 기법이 지속적으로 고도화되고 있음을 지적하며, fuzzing 기법과 동적 분석 툴의 발전으로 메모리 오류가 과거보다 훨씬 더 쉽게 발견 및 악용되고 있음을 강조하였다(Bernhard, L et al., 2022).

최근 사례로는 SolarWinds 공급망 공격, Microsoft Exchange Server 침해 사고 등에서 메모리 오류가 공격 초기 단계에서 활용되었으며, 국가 주도의 APT(Advanced Persistent Threat) 그룹 또한 메모리 오류를 공격 벡터로 선호하는 경향이 증가하고 있다(U.S. Government Accountability Office., 2021). 이는 메모리 오류가 방어하기 어렵고, 공격 성공 시 시스템 전체 제어권을 획득할 수 있기 때문이다.

결론적으로, 메모리 오류는 단순한 프로그램 오류 이상의 심각한 시스템 보안 위협으로 자리잡고 있으며, 공격자는 이를 활용해 원격 코드 실행, 권한 상승(privilege escalation), 정보 유출(information leakage)과 같은 고위험 공격을 수행할 수 있다. 이러한 배경에서 메모리 오류 유형에 대한 체계적인 이해와 대응이 필수적이며, 메모리 안전성을 근본적으로 확보할 수 있는 새로운 접근법이 요구되고 있다.

제 3 절 기존 대응 방법과 한계

메모리 오류로 인한 보안 위협이 지속적으로 발생함에 따라, 이를 탐지하고 완화하기 위한 다양한 대응 기술이 개발되어 왔다. 전통적인 대응 전략은 크게 정적 분석(static analysis), 동적 분석(dynamic analysis), 그리고 컴파일러 기반 탐지 및 방어 기법으로 구분할 수 있다.

정적 분석(Static Application Security Testing, SAST)은 소스 코드를 실행하지 않고 분석하여 잠재적인 취약점을 탐지하는 기법이다. 컴파일 이전 단계에서 코드 전체를 분석하므로 버퍼 오버플로우, use-after-free, double

free와 같은 메모리 오류 가능성을 사전에 확인할 수 있다. 그러나 정적 분석은 프로그램의 실행 경로를 완벽히 모델링하기 어려워 거짓 양성(false positive) 발생률이 높으며, 복잡한 동적 메모리 패턴이나 런타임 조건에 따른 오류는 탐지하지 못하는 한계가 있다.

동적 분석(Dynamic Application Security Testing, DAST)은 프로그램을 실제로 실행하면서 메모리 오류를 탐지하는 방식이다. 대표적인 기법으로 fuzzing이 있으며, 공격자가 입력값을 자동 생성하여 예기치 않은 프로그램 상태를 유발하고 메모리 오류를 발견한다. 또한 Sanitizer 계열 도구들, 예를 들어 AddressSanitizer(ASan), MemorySanitizer(MSan), ThreadSanitizer(TSan) 등은 런타임 시 메모리 접근을 모니터링하여 오류를 탐지한다. 이러한 도구들은 버퍼 오버플로우, use-after-free, 힙 메타데이터 손상 등 다양한 오류를 정확히 감지할 수 있지만, 성능 오버헤드가 크며, 테스트된 경로에 대해서만 보장하므로 완전한 커버리지를 제공하지는 않는다.

Control Flow Integrity(CFI)와 같은 컴파일러 기반 방어 기법은 프로그램의 제어 흐름 무결성을 보장함으로써 RCE와 같은 공격을 방지하려는 시도다. CFI는 함수 포인터나 반환 주소를 보호하여 공격자가 제어 흐름을 변조하는 것을 어렵게 만든다. 그러나 CFI 역시 성능에 일정한 영향을 주며, 데이터 흐름(data flow) 기반 공격이나 메타데이터 조작과 같은 고급 공격에는 취약한 면이 있다(Burow, N., Carr et al., 2017).

요컨대, 기존 대응 기술들은 메모리 오류를 완전히 방지하는 것이 아니라 사후적으로 탐지하거나 일부 경로에 대해 방어하는 수준에 머무르고 있다. 특히 정적 분석과 동적 분석은 개발자의 숙련도와 테스트 품질에 따라 결과가 크게 좌우되며, 개발과정에 추가적 비용과 복잡성을 유발한다. NSA의 Software Memory Safet 보고서에서도 지적하듯, 이러한 기존 대응 기술들은 구조적으로 메모리 오류 발생 가능성을 제거하지는 못하며, 메모리 안전성이 내재된 언어적 접근이 필요하다는 방향성이 제시되고 있다.

최근 공격자는 fuzzing 기술의 고도화와 자동화된 취약점 탐지 도구를 활용하여 기존 방어 기술을 우회하는 사례가 증가하고 있다. 특히 JIT(Just-In-Time) 컴파일러 기반 시스템이나 가상화 환경에서는 기존 대응

기술이 적용되기 어려운 경우도 존재한다. 따라서 보다 근본적인 해결책으로 메모리 안전성을 언어 수준에서 보장하는 접근에 대한 필요성이 커지고 있으며, 이는 최근 산업계와 학계에서 Rust와 같은 메모리 안전 언어에 주목하게 된 주요 배경이 되고 있다.

제 4 절 메모리 안전성 위반의 영향

메모리 안전성 위반은 시스템 무결성(integrity), 가용성(availability), 기밀성(confidentiality)을 동시다발적으로 훼손한다. 버퍼 오버플로우나 use-after-free가 성공적으로 악용되면 공격자는 커널 수준 권한을 획득하여 원격 코드 실행과 서비스 거부(DoS)를 야기할 수 있다. 2021년 공개된 CVE-2021-31166(Windows HTTP.sys 힙 오버플로우)은 패치 전까지 인터넷 노출 서버를 대규모로 위험에 빠뜨렸다. 금융·의료 분야에서는 메모리 오류 한 건이 대규모 개인정보 유출과 규제 처벌로 이어질 수 있다. 예컨대 2023년 미국 의료기관 APT 공격 보고서에서는 힙 오버플로우 취약점이 악성 RAT 삽입 통로로 활용되었고, 180만 건의 진료 기록이 유출되었다.

국가 기반시설도 예외가 아니다. Windows CryptoAPI 속성 조작(Microsoft Security Response Center., 2020)은 서명 검증 과정을 우회해 악성 업데이트 배포를 가능케 했으며, 이는 전력 SCADA 시스템에 악성 코드가 유입되는 실제 사고로 이어졌다. SoK: Eternal War in Memory는 이처럼 메모리 오류가 APT 전술의 핵심으로 자리했고, 자동 fuzzing 도구의 발전으로 발견·악용 주기가 단축되고 있음을 지적한다.

클라우드·가상화 환경에서는 한 VM의 메모리 취약점이 하이퍼바이저를 경유해 논리적 경계를 무력화할 수 있다. 2022년 공개된 “Dirty Pipe”(CVE-2022-0847) 취약점은 컨테이너 격리 우회를 통해 호스트 파일시스템 오염을 가능케 했다. 이처럼 메모리 안전성 위반은 현대 ICT 인프라의 신뢰 기반을 송두리째 흔들 수 있으므로, 종합적이고 근본적인 대응이 필수적이다.

제 5 절 메모리 안전성 확보 전략

메모리 오류 대응 기술은 사후 탐지형과 사전 예방형으로 구분된다. 전자는 정적 분석(SAST)·동적 분석(DAST)·Sanitizer/CFI류 기법으로, 발견 시점이 코드 배포 이후까지 지연될 위험이 있다. 반면 사전 예방형 접근은 언어 설계 차원에서 메모리 오류 발생을 원천 차단한다. Rust는 소유권·수명 시스템으로 컴파일 시점에 UAF, double free, 데이터 레이스를 금지한다. NSA의 Software Memory Safety 보고서는 C/C++ 대신 메모리 세이프 언어 채택을 권고하며, Rust·Go·Swift를 예시로 들었다. 2025년 백악관 National Cyber Strategy Implementation Plan(White House, The., 2024).은 연방 조달 SW에 메모리 안전 언어 사용 검증 조항을 포함하도록 지침을 제정하였다.

그러나 언어 교체에는 학습 곡선, 기존 코드 이식 비용, 생태계 성숙도 같은 현실적 과제가 뒤따른다. 이에 따라 하이브리드 전략—성능·호환성이 중요한 경로는 C, 보안 경로는 Rust—이 산업계에서 점차 표준화되고 있다. 본 논문은 이러한 추세를 반영하여, Rust로 구현한 블록 암호 알고리즘 사례를 통해 메모리 안전 언어 도입의 실효성을 실험적으로 평가한다.

제 4 장 C와 Rust 구현 비교

제 1 절 Rust 키스케줄링

Rust 언어에서는 키의 크기에 따라 별도의 함수가 사용된다. round_key_gen_24, round_key_gen_28, round_key_gen_32는 각각 128비트, 192비트, 256비트 키에 대응하며, 이러한 분리된 구현은 각 함수가 해당 키 크기에 특화된 처리를 수행할 수 있도록 한다. 이에 따라 실행 시점에 키 크기를 확인하는 과정이 불필요하며, 보다 효율적인 코드 구성이 가능하다.

```
const KEY_CONST: [u32; 8] = [ ...
 /* 라운드 키 생성을 위한 키 스케줄링 과정에서 사용되는 상수들
*/
];
const MASTER_KEY: [u32; 8] = [ ...
/* 마스터 키로서, 32비트 워드들의 연결 형태로 표현된다.
키 길이가 128비트인 경우 K = (K[0], K[1], K[2], K[3])로 나타내
며, 키 길이가 192비트인 경우 K = K[0] || K[1] || ... || K[5]
로 나타낸다. 키 길이가 256비트인 경우 K = K[0] || K[1] || ...
   || K[7]로 구성된다. 여기서 || 연산은 워드 간의 연결
(concatenation)을 의미한다. */
];
fn round_key_gen_24(mk: &[u32; 8], erk: &mut
[u32; 192], drk: &mut [u32; 192]){
// 128비트 키를 위한 키 스케줄링 연산
}
fn main() {
let mut enc_round_key: [u32; 192] = [0; 192];
let mut dec_round_key: [u32; 192] = [0; 192];
round_key_gen_24(&MASTER_KEY, &mut
enc_round_key, &mut dec_round_key);
}
```

표[4-1] Rust round_key_gen_24 함수

코드의 이러한 구조는 모노모픽(monomorphic) 설계를 충족한다. 모노모픽 설계란 제네릭이나 추상 타입 없이 구체적 타입으로만 구현을 구정하여 컴파일 타임에 모든 함수와 데이터 구조가 완전히 특수화되는 설계 방식이다. 함수를 살펴보면 mk, erk, drk, 모두 컴파일 시 크기가 확정된 고정 배열 타입이며 제네릭이나 동적 할당 없이 구체적 타입만 사용해 컴파일 타임에 완전히 특수화되고, 내부 루프가 상수 범위 반복과 산술 비트 연산만으로 이루어져 컴파일 단계에서 키 길이에 따른 분기(branch)를 제거함으로써, 분기 예측 실패로 인한 페널티를 없애고 파이프라인 뒤엉킴(pipeline stall)을 최소화한다. 또한 실행 흐름이 입력 데이터와 무관하게 고정돼 런타임 오버헤드나 부채널 위험을 줄여준다.

해당 함수의 선언부에서는 Rust 언어의 주요 특성 중 하나인 불변 참조

(&) 와 가변 참조 (&mut) 를 확인할 수 있다. 이들 참조는 Rust의 메모리 안전성 보장을 위한 핵심 요소로 작용한다.

불변 참조는 여러 코드 영역에서 데이터를 읽을 수 있도록 허용하되, 데이터가 예기치 않게 변경되는 위험을 원천적으로 차단한다. 반면 가변 참조는 특정 시점에 단 하나만 존재할 수 있도록 제한함으로써 데이터 경쟁(data race) 을 방지한다.

또한 Rust 컴파일러는 함수 시그니처에 명시된 참조 타입을 바탕으로 자동으로 라이프타임(lifetime)을 추론한다. 덕분에 프로그래머가 명시적으로 메모리 해제나 소유권 이전을 기술하지 않아도, 컴파일 타임에 안전성이 검증된다. 비슷한 기능을 C/C++에서 수동으로 흉내 내려면 정적 분석 도구(예: Coverity)나 엄격한 코드 리뷰 과정이 필요하지만, Rust에서는 언어 자체가 강제한다.

1) 데이터 경쟁 방지

Rust는 한 시점에 하나의 가변 참조 또는 여러 개의 불변 참조만을 허용한다. 이로써 데이터 경쟁 상황이 발생하지 않도록 강제한다.

이는 round_key_gen_24와 같은 키 생성 함수에서 특히 중요하다. 본 함수는 민감한 암호화 키 데이터를 다루므로, 데이터 경쟁이 발생할 경우 심각한 보안 취약점으로 이어질 수 있다. 여기에 더해, Rust의 Send/Sync 트레이트 체계는 키 배열을 여러 스레드가 동시에 읽도록 허용하되, 동시에 쓰기는 불허한다. 예컨대 Arc<[u32; 192]>를 여러 암호화 스레드가 공유하여 병렬 스트림 암호 작업을 수행할 때, Arc 내부 포인터는 원자적 참조 카운터만 증가·감소할 뿐이며, 실질적인 라운드키 버퍼는 불변으로 취급된다. 그 결과 데이터 경쟁 가능성이 여전히 0에 수렴한다.

실제 현업 환경을 예로 들자면, TLS 1.3 등장 이후 Rust-TLS 라이브러리

에서는 세션 키를 Arc<LessSafeKey> 형태로 보관하고, 하위 레이어에서 여럿 스레드가 동시에 seal_in_place()를 호출하게끔 설계되어 있다. 이처럼 공유 불변 데이터 모델이 마련되어 있으면, OpenSSL 같은 전통적 C 라이브러리가 락 경합(lock contention)을 피하기 위해 CRYPTO_THREAD_run_once()와 같은 복잡한 초기화 프로시저를 두어야 하는 수고를 Rust에서는 생략할 수 있다.

2) 빌림 검사기 (Borrow Checker)

Rust 컴파일러에는 빌림 검사기(Borrow Checker) 가 내장되어 있으며, 이는 컴파일 단계에서 메모리 안전성 규칙 준수를 강제한다. 빌림 검사기는 참조가 참조 대상보다 더 오래 살아남는 상황을 방지하여 댕글링 참조(dangling reference) 가 발생하는 것을 원천적으로 차단한다.

예를 들어 round_key_gen_24 함수 내에서 erk와 drk에 대한 참조는 함수의 유효 범위를 벗어나지 않도록 보장된다. 이에 따라 use-after-free 오류, 즉 해제된 메모리에 대한 접근 오류는 발생하지 않는다.

함수 호출 시 &mut enc_round_key 와 &mut dec_round_key는 가변 참조로 전달된다. 빌림 검사기는 이들 참조가 함수 내부에서만 유효하도록 강제하며, 함수 종료 후에는 해당 참조를 사용할 수 없도록 한다.

이러한 메커니즘은 참조가 의도된 생명주기(lifetime) 외부에서 비안전하게 접근되는 상황을 방지함으로써 메모리 안전성과 데이터 무결성을 확보한다.

3) 함수 시그니처의 역할

함수 시그니처 또한 메모리 안전성을 보장하는 방향으로 설계된다. 본 함수는 키 값을 소유권 이전 없이 참조 형태로 요구함으로써, 불필요한 데이터 복사를 방지한다. 이는 성능 최적화는 물론, 메모리 관리 측면에서도 유리한

구조를 제공한다.

또한 MASTER_KEY와 같은 상수는 불변 참조를 통해 사용됨으로써 프로그램 내 여러 영역에서 안전하게 공유된다. 이로 인해 데이터 중복 사용이 최소화되고, 암호화 연산 중 원본 데이터가 실수로 변경되는 사태를 방지할 수 있다.

결론적으로, Rust 언어의 메모리 및 데이터 접근 제어 방식은 암호화 함수인 round_key_gen_24에서 높은 수준의 안전성과 보안을 제공할 뿐 아니라, 성능과 신뢰성 또한 제고하는 효과를 가져온다.

제 2 절 C 키스케줄링

```
1   typedef struct lea_key_st
2   {
3   unsigned int rk[192];
4   unsigned int round;
5   } LEA_KEY;
6   void lea_set_key_generic(LEA_KEY *key, const
7   unsigned char *mk, unsigned int mk_len) {
8   const unsigned int* _mk = (const unsigned int*)mk;
9   switch(mk_len) {
10  case 16: // 128-bit key
11  // 128비트 키를 위한 키 스케줄링 연산
12  break;
13  case 24: // 192-bit key
14  // 192비트 키를 위한 키 스케줄링 연산
15  break;
16  case 32: // 256-bit key
17  // 256비트 키를 위한 키 스케줄링 연산
18  break;
19  }
20  // 그 외...
21  }
```

표 [4-2] C lea_set_key_generic 함수

C 언어 코드에서는 입력 값인 mk가 const unsigned char* 타입으로 정의된다. 이는 이후 const unsigned int* 타입으로 캐스팅되어 4바이트 단위로 접근하게 된다. 이러한 방식은 LEA 암호화 알고리즘의 키 스케줄 설정 과정에서 사용된다.

C 언어에서는 포인터 타입 간의 캐스팅이 가능하다. mk가 const unsigned char* 타입으로 제공되는 경우, 이는 데이터에 바이트 단위로 접근함을 의미한다. 반면 이를 const unsigned int* 타입으로 캐스팅하면, 4바이트(32비트) 단위로 데이터를 접근하게 된다.

이러한 포인터 캐스팅 과정에서 발생할 수 있는 문제점은 다음과 같다:

1) 메모리 정렬(Memory Alignment) 문제

unsigned int 타입의 데이터는 일반적으로 4바이트 경계(boundary) 에 정렬되어야 한다. 그러나 unsigned char* 타입은 바이트 단위 접근이 가능하며, 반드시 4바이트 경계에 정렬되어 있을 필요는 없다. 만약 mk의 주소가 4바이트 경계에 정렬되어 있지 않은 경우, 이를 const unsigned int*로 캐스팅하여 접근하면 정의되지 않은 동작(undefined behavior) 을 초래할 수 있다.

2) 메모리 접근 오류(Memory Access Errors)

잘못된 정렬 상태에서 4바이트 단위 접근이 수행되면, 특정 CPU 아키텍처에서는 정렬되지 않은 메모리 접근(unaligned memory access) 으로 인해 예외(exception)가 발생할 수 있다. 이는 특히 일부 하드웨어 아키텍처에서 심각한 문제를 야기할 수 있다.

예를 들어, 다음과 같은 mk 바이트 배열을 고려할 수 있다.

```
1    unsigned char mk[16] ={ 0x01, 0x02, 0x03, 0x04, 0x05,
2    0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E,
3    0x0F, 0x10 };
```

표 [4-3] C unsigned char mk[16]

이러한 배열을 unsigned int* 타입으로 캐스팅하여 접근하는 경우, 데이터는 다음과 같이 해석된다:

```
1    const unsigned int* _mk = (const unsigned int*)mk;
```

표 [4-4] C unsigned int* _mk

여기서 _mk[0]는 0x04030201, _mk[1]는 0x08070605가 될 것이다. 그러나 만약 mk가 4바이트 경계에 정렬되어 있지 않은 경우, 예를 들어 다음과 같이 한 바이트 이동하여 비정렬 상태가 된다면, 예를들어:

```
1    unsigned char* mk = (unsigned char* mk)malloc(17) + 1;
```

표 [4-5] C unsigned char* mk

이 경우 mk는 4바이트 경계에 정렬되어 있지 않다. +1 연산으로 포인터가 한 바이트 이동하여 비정렬 상태가 되었으며, 이를 unsigned int*로 캐스팅하여 접근하는 경우, 주소가 4의 배수가 아니기 때문에 정의되지 않은 동작(undefined behavior)이 발생할 수 있다.

따라서 C 언어에서는 포인터 타입 캐스팅을 통한 메모리 접근 시 특별한 주의가 필요하다(Li, P., 2004). 특히 바이트 단위 포인터(unsigned char*)를 4바이트 단위 포인터(unsigned int*)로 캐스팅할 경우, 메모리 정렬 문제를 반드시 고려해야 한다.

Rust와 같은 언어는 이러한 문제를 사전에 방지하기 위해 메모리 안전성(memory safety) 을 강제한다. Rust 컴파일러는 이러한 위험한 캐스팅에 대해 경고(warning) 또는 컴파일 오류(error) 를 발생시켜, 프로그래머가 안전한

코드를 작성하도록 돕는다. 이러한 언어적 특성은 런타임 오류(runtime error) 및 보안 취약점(security vulnerability) 을 예방하는 데 중요한 역할을 한다.

더 나아가, 현대 C 컴파일러에서 -fstrict-aliasing 옵션이 기본 활성화된 경우, 서로 다른 타입의 포인터가 동일 주소를 가리킬 때 발생하는 미정의 동작(undefined behavior)은 예상치 못한 다양한 결과를 초래한다. 예컨대 _mk[0]를 읽은 뒤 이어서 *(unsigned char*)mk를 읽는 코드가 있을 경우, 컴파일러는 별칭(aliasing)이 없다고 가정하고 _mk[0]을 레지스터에 캐싱해 둔다. 결과적으로 CPU에는 존재하지 않는 '유령 값(ghost value)'이 전파돼 해시 값 계산이나 메시지 인증 코드(MAC) 검증이 틀어지는 등 치명적 오류가 발생할 수 있다.

제 3 절 Rust 암호화 및 복호화

키 스케줄링 과정뿐만 아니라, 실제 평문과 암호문을 처리하는 핵심 연산 단계인 암호화 및 복호화 함수에서도 Rust와 C의 구현 방식은 메모리 안전성에 대한 코드 구조에서 차이를 가진다. 이는 C가 가진 잠재적 위험성과 이를 Rust가 언어 차원에서 어떻게 원천적으로 방지하는지를 보여주는 대표적인 사례이다. LEA의 복호화 함수는 암호화의 역연산 과정으로, 유사한 구조와 메모리 접근 패턴을 가진다. 따라서 본 절에서는 대표적으로 enc 함수를 상세히 분석한다. 다음은 본 연구에서 구현한 Rust의 enc 함수이다.

```
1    fn enc(x: &mut [u32; 4], rk: &[u32; 192]) -> [u32; 4] {
2        let mut temp;
3        let rk_len = rk.iter().filter(|&&x| x != 0).count();
4        // 기본 반복 횟수 설정
5        let mut rounds = 24;
6        // rk_len에 따라 추가 반복 횟수 조정
7        if rk_len > 144 {
8            rounds += 4;
9        }
10       if rk_len > 168 {
11           rounds += 4;
12       }
13       for i in 0..rounds {
14           let temp1 = x[0] ^ rk[i * 6];
15           let temp2 = x[1] ^ rk[i * 6 + 2];
16           let temp3 = x[2] ^ rk[i * 6 + 4];
17           temp = x[0];
18           x[0] = temp1.wrapping_add(x[1] ^ rk[(i * 6) +
1]).rotate_left(9);
19           x[1] = temp2.wrapping_add(x[2] ^ rk[(i * 6) +
3]).rotate_right(5);
20           x[2] = temp3.wrapping_add(x[3] ^ rk[(i * 6) +
5]).rotate_right(3);
21           x[3] = temp;
22       }
23       *x
24   }
```

표[4-6] Rust enc 함수

첫째, 함수 시그니처 fn enc(state: &mut [u32; 4], round_key: &[u32; 192]) 자체가 강력한 메모리 안전 장치로 기능한다. 입력 데이터 블록을 나타내는 state 파라미터는 가변 참조(&mut) 타입으로 선언되었다. 이는 Rust의 빌림 검사기(Borrow Checker)에 의해 이 함수가 state가 가리키는 메모리에 대한 쓰기 가능한 독점적 대여권을 가짐을 컴파일 시점에 강제한다. 만약 이 함수가 실행되는 동안 프로그램의 다른 부분(예: 다른 스레드)에서 state에 접근하려는 시도가 있다면, 컴파일러는 이를 데이터 경쟁(Data Race)으로 간주하고 컴파일을 거부한다. 이는 동시성 프로그래밍에서 발생할 수 있는 가장

미묘하고 치명적인 버그 중 하나를 언어 차원에서 예방하는 것이다.

둘째, 라운드 키 round_key는 길이 정보가 타입 시스템에 명시된 슬라이스(&[u32; 192]) 타입으로 전달된다. Rust에서 슬라이스는 내부적으로 데이터 시작 주소를 가리키는 포인터와 데이터의 길이를 함께 관리하는 구조체이다. 이 덕분에 round_key[i * 6]와 같은 모든 배열 인덱싱 연산은, Release 빌드에서 컴파일러가 안전하다고 판단하지 않는 이상, 런타임에 자동으로 경계 검사(Bounds Checking)가 수행된다. 만약 루프의 반복 횟수 계산 오류나 논리적 실수로 인해 인덱스가 배열의 유효 범위를 벗어날 경우, 프로그램은 정의되지 않은 동작으로 빠져 예측 불가능한 결과를 내거나 보안 취약점으로 이어지는 대신, 즉시 패닉(panic)하며 안전하게 실행을 중단시킨다. 이는 C 구현의 가장 고질적인 취약점인 버퍼 오버플로우(Buffer Overflow)를 근본적으로 해결하는 핵심적인 메커니즘이다.

제 4 절 C 암호화 및 복호화

앞서 언급했듯이 LEA의 복호화 함수는 암호화의 역연산 과정으로, 본 절에서도 enc 함수를 대표적으로 분석한다. C 코드로 구현된 lea_encrypt 함수는 [표 4-7]과 같다. 이는 저수준 메모리 제어에 최적화되어 있지만, Rust 구현과 비교했을 때 여러 잠재적인 위험을 내포한다.

```
1    void lea_encrypt(unsigned char *ct, const unsigned char *pt, const
LEA_KEY *key)
2    {
3        unsigned int X0,X1,X2,X3;
4
5        const unsigned int * _pt = (const unsigned int *)pt;
6        unsigned int * _ct  = (unsigned int*)ct;
7
8        X0 = loadU32(_pt[0]);
9        X1 = loadU32(_pt[1]);
10       X2 = loadU32(_pt[2]);
11       X3 = loadU32(_pt[3]);
12       X3 = ROR((X2 ^ key->rk[  4]) + (X3 ^ key->rk[  5]), 3);
13       X2 = ROR((X1 ^ key->rk[  2]) + (X2 ^ key->rk[  3]), 5);
14       X1 = ROL((X0 ^ key->rk[  0]) + (X1 ^ key->rk[  1]), 9);
15       X0 = ROR((X3 ^ key->rk[ 10]) + (X0 ^ key->rk[ 11]), 3);
16       X3 = ROR((X2 ^ key->rk[  8]) + (X3 ^ key->rk[  9]), 5);
17       X2 = ROL((X1 ^ key->rk[  6]) + (X2 ^ key->rk[  7]), 9);
18       X1 = ROR((X0 ^ key->rk[ 16]) + (X1 ^ key->rk[ 17]), 3);
19       X0 = ROR((X3 ^ key->rk[ 14]) + (X0 ^ key->rk[ 15]), 5);
20       X3 = ROL((X2 ^ key->rk[ 12]) + (X3 ^ key->rk[ 13]), 9);
21       X2 = ROR((X1 ^ key->rk[ 22]) + (X2 ^ key->rk[ 23]), 3);
22       X1 = ROR((X0 ^ key->rk[ 20]) + (X1 ^ key->rk[ 21]), 5);
23       X0 = ROL((X3 ^ key->rk[ 18]) + (X0 ^ key->rk[ 19]), 9);
24       // ...
25
26       if(key->round >24)
27       {
28           //192비트 키를 위한 enc 연산
29       }
30
31       if(key->round >28)
32       {
33           //256비트 키를 위한 enc 연산
34       }
35       _ct[0] = loadU32(X0);
36       _ct[1] = loadU32(X1);
37       _ct[2] = loadU32(X2);
38       _ct[3] = loadU32(X3);
39   }
```

표 [4-7] C lea_encrypt 함수

가장 큰 차이점은 포인터 사용에 있다. unsigned char *pt와 같이 바이트 단위로 전달된 포인터를 const unsigned int *로 타입 캐스팅하여 4바이트 단위로 접근한다. 이는 제 2절에서 지적한 메모리 정렬 문제를 야기할 수 있다. 만약 입력 포인터 pt가 4바이트 경계에 정렬되어 있지 않다면, 일부 CPU 아키텍처에서는 정렬되지 않은 메모리 접근으로 인해 성능이 저하되거나 하드웨어 예외가 발생할 수 있다.

또한, key->round 값에 의존하는 for 루프는 경계 검사가 부재하다는 약점을 가진다. LEA_KEY 구조체는 스택에 할당될 가능성이 높은데, 만약 이 구조체의 round 멤버 변수가 다른 메모리 오류에 의해 손상되거나, 함수 호출 시 잘못된 key 포인터가 전달되어 비정상적인 값(예: 24보다 훨씬 큰 값)을 가지게 될 경우, key->rk[...] 접근은 할당된 라운드 키 배열의 경계를 넘어서게 된다. 이는 스택의 다른 변수나 함수 반환 주소와 같은 민감한 데이터를 읽거나 덮어쓰는 스택 기반 버퍼 오버플로우로 직결된다. 공격자는 이를 악용하여 프로그램의 실행 흐름을 완전히 장악하거나, 비밀 정보를 유출 시킬 수 있다.

제 5 장 성 능 평 가

LEA 알고리즘을 Rust로 구현한 이후, Jemalloc을 사용하고 CPU 시간 측정 방법을 적용하여 성능 비교를 수행하였다. 메모리 사용 통계 측정을 위한 실험 환경은 다음과 같은 사양의 MacBook Air에서 구성하였다:

- Model : MacBook Air
- Chip : Apple M2 (2022)
- Memory : 8 GB
- Operating System : macOS Sonoma 14.5

제 1 절 Jemalloc

jemalloc은 메모리를 효율적으로 관리하기 위해 설계된 메모리 할당 라이브러리로서, 특히 다중 스레드 환경에서 그 효과가 두드러진다. 원래는 FreeBSD 운영체제용으로 개발되었으나, 표준 메모리 할당기보다 우수한 메모리 관리 성능으로 인해 다양한 시스템에서 광범위하게 채택되고 있다.

jemalloc의 주요 장점 중 하나는 메모리 단편화(memory fragmentation)를 효과적으로 줄일 수 있다는 점이다. 이를 통해 연속적인 메모리 할당 및 해제가 가능하며, 이는 높은 수준의 동시성(concurrency)을 요구하는 환경에서 매우 중요한 요소이다(Umayabara, A et al., 2017). 전통적인 메모리 할당기의 경우, 다수의 스레드가 동시에 메모리 자원에 접근할 때 락 경합(lock contention) 으로 인해 성능 병목(bottleneck)이 발생할 수 있다(Gidenstam et al., 2010). jemalloc은 이를 완화하기 위해 멀티 아레나(multi-arena) 할당 시스템을 적용하여, 경합을 줄이고 전체적인 성능을 향상시킨다(Huang, W et al., 2004). 또한 jemalloc은 높은 구성 가능성(configurability)을 제공하여, 다양한 시나리오에 맞추어 여러 파라미터를 최적화할 수 있다. 이는 고부하(high-load) 서버 환경이나 메모리 집약적(memory-intensive) 연산 환경 모

두에 유리하게 작용한다.

jemalloc은 아울러 강력한 메모리 사용 모니터링 및 분석 도구도 제공하여, 성능 최적화 및 메모리 누수 탐지에 유용하게 활용된다(Ferreira, T et al., 2011)

jemalloc은 MongoDB 및 Redis와 같은 인기 있는 데이터베이스 시스템뿐만 아니라, Rust와 같은 현대 프로그래밍 언어에서도 널리 채택되고 있다.

이러한 시스템들은 jemalloc을 통해 기존의 전통적인 메모리 할당기보다 대규모 동시 연결 처리 또는 고속 연산 처리에서 보다 뛰어난 성능을 확보하고 있다(Berger, E et al., 2000).

1) Jemalloc Statistics

jemalloc은 메모리 사용 최적화 및 이해를 위한 포괄적인 통계 정보(comprehensive statistics) 를 제공한다. 이러한 통계에는 할당된 메모리(allocated), 상주 메모리(resident), 운영체제로부터 매핑된 메모리(mapped), 활성 메모리(active), 그리고 단편화 지표 등이 포함된다. 이러한 통계는 메모리 관리의 여러 중요한 측면에 대한 심층적인 인사이트(insight) 를 제공하며, 이를 통해 성능 튜닝, 메모리 누수(leak) 탐지, 그리고 메모리 계획을 효과적으로 수행할 수 있다.

본 연구에서는 jemalloc의 이러한 통계 정보를 활용하여 C 언어 구현과 Rust 언어 구현 간의 차이를 분석하였다. 그 결과는 아래와 같다.

	Allocated	Active	Metadata	Resident	Mapped	Retained
C	174,720	376,832	2,616,832	2,899,968	6,668,288	0
Rust	4,227,960	5,750,784	2,645,632	8,372,224	12,124,160	0

표 [5-1] Jemalloc Statistics 비교

2) Jemalloc Statistics 해석

jemalloc은 메모리 사용 현황을 최적화하고 이해하기 위한 포괄적인 통계

정보를 제공한다. 주요 지표는 다음과 같다.

Allocated (할당된 메모리)

프로그램이 jemalloc을 통해 운영체제에서 요청하여 성공적으로 할당받은 총 메모리 용량을 나타낸다. 이는 특정 시점까지 누적된 모든 메모리 블록의 크기를 합한 값으로, 애플리케이션이 실행 중에 얼마나 많은 메모리를 요구하는지 파악하는 데 유용하다.

- Active (활성 메모리)

애플리케이션이 현재 사용 중인 할당된 메모리의 크기를 의미한다. 여기에는 내부 단편화(fragmentation)나 오버헤드로 인해 실제로는 사용되지 않지만 여전히 할당된 블록 내에 남아 있는 메모리도 포함된다. 이 값이 Allocated보다 클 수 있는 것은 바로 단편화 및 내부 관리 오버헤드 때문이다.

- Metadata (메타데이터)

jemalloc 내부에서 할당 및 해제 과정을 관리하기 위해 사용되는 메모리이다. 할당된 블록과 비어 있는 블록을 추적하는 데이터 구조와 테이블 등이 포함되며, 효율적인 메모리 관리를 위해 필수적이지만 전체 메모리 사용량에는 추가적인 오버헤드를 더한다.

- Resident (상주 메모리)

프로세스의 메모리 중 실제 물리적 RAM에 상주하고 있는 부분을 의미한다. 스왑 아웃되었거나 물리 메모리에 존재하지 않는 가상 메모리 영역은 제외된다. 이는 애플리케이션이 실제로 점유하고 있는 물리 메모리 규모를 보여준다.

- Mapped (매핑된 메모리)

jemalloc이 가상 주소 공간 상에서 매핑한 전체 메모리 영역의 크기이다. 현재 사용 중이든 아니든 jemalloc이 예약(reserve)한 모든 영역을 포함하므로, 시스템 전체의 메모리 가용성에 미치는 영향을 파악할 수 있다.

- Retained (유보된 메모리)

jemalloc이 향후 할당 요청을 대비해 운영체제에서 가져와 내부에 보관 중이지만, 아직 애플리케이션에 할당되지 않은 메모리이다. 이는 시스템 콜을 다시 호출하지 않고도 빠른 할당을 가능하게 하여 성능을 향상시키지만, 그

대가로 메모리 사용량이 증가한다. 해당 실험 결과에서는 C와 Rust 모두 Retained 메모리가 0으로 나타나, jemalloc이 현재 유보된 미사용 메모리를 보유하고 있지 않음을 의미한다.

위 테이블 결과를 종합하면, Rust 구현이 C 구현보다 전반적으로 더 많은 메모리를 사용하고 있음을 확인할 수 있다. 특히 Active 메모리와 Resident 메모리에서 유의미한 차이가 나타났는데, 이는 Rust 프로그램이 전체적으로 더 많은 메모리를 할당 및 사용하고 있음을 시사한다.

제 2 절 CPU time 측정

CPU 시간 측정은 프로그램이 프로세서 자원을 얼마나 효율적으로 활용하는지를 평가한다. 실시간(wall-clock) 시간은 I/O 대기 시간을 포함하지만, CPU 시간은 프로그램의 실제 명령어 실행에 CPU가 소비한 시간을 측정한다. 이는 소프트웨어의 효율성과 성능을 이해하는 데 필수적이다(Gupta, A et al., 2014).

CPU 시간은 사용자 CPU 시간(user CPU time) 과 시스템 CPU 시간(system CPU time) 으로 구분된다. 사용자 CPU 시간은 프로세서가 프로그램 자체의 명령어를 실행하는 데 소비한 시간이다(Chen, S et al., 2011). 시스템 CPU 시간은 프로그램이 시스템 호출(system call)을 수행하는 과정에서 운영체제가 명령어를 실행하는 데 소비한 시간이다.

여러 도구가 운영체제별로 CPU 시간을 측정한다(Tallent, N. R., & Mellor-Crummey., 2009) Unix/Linux 환경에서는 time, top, ps 등이 널리 사용된다(Bianchini., & Lim, B., 1996). 특히 time 명령은 사용자 CPU 시간, 시스템 CPU 시간, 실시간(real time)을 함께 보고하여, 프로그램이 실제로 얼마나 오래 실행되었는지와 CPU 사용량이 사용자 모드와 커널 모드에서 각각 얼마인지를 상세히 보여준다.

C 언어에서는 〈time.h〉 헤더의 clock() 함수를 통해 프로세서 시간이 측정된다. clock() 함수는 프로그램 시작 이후 경과된 클록 틱(clock tick)을 반환하며, 이를 초 단위로 환산하려면 반환값을 CLOCKS_PER_SEC로 나누면

된다.

clock() 함수의 장점은 사용이 간편하고, 프로그램이 활성 상태가 아닌 동안의 시간을 제외하고 순수한 CPU 사용 시간을 제공한다는 점이다. 다만, 멀티스레드 환경에서는 모든 스레드의 CPU 시간이 합산되어 측정되므로, 스레드별 CPU 시간 측정이 불가능하다는 한계가 있다.

Rust 언어에서는 SystemTime 및 Instant 타입, 그리고 cpu_time::ProcessTime, backtrace::Backtrace 같은 크레이트(crates)를 이용하여 성능 측정과 디버깅을 수행한다. 이들 도구를 활용하면 애플리케이션 성능을 최적화하고, 문제 발생 시 디버깅에 필요한 상세 정보를 획득할 수 있다.

본 연구에서는 이러한 C 및 Rust의 CPU 시간 측정 도구를 사용하여 두 구현체 간 성능을 비교하였다. 그 결과는 다음과 같다:

Key Size (bits)	Key Schedule (ms)	Encryption (ms)	Decryption (ms)
128	3.675	2.132	1.853
192	6.348	2.111	1.813
256	6.573	2.139	1.815

표 [5-2] C의 CPU time

Key Size (bits)	Key Schedule (ms)	Encryption (ms)	Decryption (ms)
128	12.907	21.632	21.947
192	17.354	21.715	23.409
256	30.104	22.342	23.632

표 [5-3] Rust의 CPU time

제 6 장 성능 저하 원인 분석

본 연구에서 LEA 알고리즘을 Rust로 이식한 주요 목적은 C 구현과 유사한 처리량을 유지하면서도 버퍼 오버플로우나 UAF(Use-After-Free)와 같은 메모리 관련 취약점을 컴파일 시점에 원천 차단하는 것이었다. 그러나 제 5장의 표[5-1], [5-2], [5-3]에서 제시된 성능 평가 결과, Rust 구현은 C 대비 3배에서 10배에 달하는 현저한 실행 지연을 보였다. 이는 단순 런타임 오버헤드만으로 설명하기 어려운 편차로, 근본적인 원인을 규명하기 위해 본 장에서는 심층 분석을 수행했다.

성능 병목 지점을 특정하기 위해 Xcode의 Time Profiler를 사용해 Rust 구현물의 함수별 시간 점유율을 분석했다. [그림 6-1]과 같이 암·복호화 및 키 스케줄링 함수에서 전반적인 지연이 관찰되었으며, 특히 두 구현 간 구조적 차이가 가장 큰 키 스케줄링 함수 round_key_gen_24에 대한 디어셈블리 분석을 진행했다.

그림 [6-1] Xcode Instrument Time Profiler 측정 결과

분석 결과, 성능 격차의 일차적 원인은 설계 패러다임의 차이에 있었다. C 레퍼런스 코드는 키 스케줄링에 필요한 delta 상수의 모든 회전(rotate) 값을 [표 6-2]와 같이 미리 계산된 테이블로 만들어 정적 최적화(static

optimization)를 극대화했다. 런타임에는 단순히 테이블에서 값을 조회하여 더하기만 하면 됐다.

반면, 본 연구의 초기 Rust 구현을 디어셈블한 코드인 [표 6-1]은 for 루프 내에서 rotate_left 메서드를 호출하여 매 라운드마다 회전 연산을 동적으로 수행했다. 이로 인해 어셈블리 레벨에서 회전(ror), 배열 재참조(ldr), 루프 제어(cmp, b.ne) 등 다수의 명령어가 반복 실행되었고, 이것이 C 대비 약 3배 이상의 지연으로 이어진 것으로 초기에 예상하였다.

디어셈블을 통해 예상한 동적 연산과 루프 구조의 비효율성을 검증하고 성능을 개선하고자, [표 6-3]의 for 루프를 [표 6-4]처럼 24개 라운드로 완전히 나열하는 수동 최적화, 즉 루프 언롤링(Loop Unrolling)을 적용했다. 이론적으로 이는 루프 오버헤드를 제거하여 성능을 향상시켜야 했다. 그러나 최적화가 배제된 디버그 빌드(cargo build)에서 측정한 결과는 예상과 달리 오히려 성능이 저하되는 역효과를 낳았다. 이는 루프 구조 자체가 병목의 핵심 원인이 아니며, 더 근본적인 원인이 존재함을 시사했다.

이 예상 밖의 현상을 규명하기 위해, Xcode Instruments의 CPU Counters 도구를 사용하여 두 버전(for 루프와나열식')의 하드웨어 성능 카운터를 정밀하게 측정했다. 표 [6-5]는 그 측정 결과를 비교한 것이다.

표 [6-5]의 핵심 내용은 IPC(Instructions Per Cycle) 지표에 있다. enc, dec 함수를 포함한 대부분의 경우에서, for 루프 버전(A)이 나열식 버전(B)보다 현저히 높은 IPC를 기록했다. 특히 round_key_gen_24와 round_key_gen_28에서는 IPC가 각각 57.3%, 41.0% 급락하는 차이를 보였다.

이 데이터는 CPU가 for 루프 버전의 코드를 실행할 때 명령어 수준 병렬성(Instruction-Level Parallelism, ILP)을 매우 높은 수준으로 활용했음을 보여준다. 구체적으로, for 루프의 짧고 반복적인 구조는 CPU의 분기 예측기(Branch Predictor)가 루프의 반복을 거의 100% 정확하게 예측하도록 만든다. 이로 인해 명령어 파이프라인이 중단 없이 채워지고, 비순차적 실행(Out-of-Order Execution) 엔진은 루프 본문 내의 데이터 의존성이 없는 다수의 연산(예: 각기 다른 temp 값에 대한 wrapping_add와 rotate_left)을 식

별하여 여러 실행 유닛에 동시에 발행(dispatch)하는 슈퍼스칼라(Superscalar) 실행을 가능하게 한다. 그 결과, IPC가 4.47까지 상승하며 한 클럭 사이클 당 평균 4개 이상의 명령어를 처리하는 높은 처리량을 달성한 것으로 보인다.

반면, 수동으로 길게 나열된 코드는 이러한 최적화 기회를 박탈한다. 긴 직선형 코드는 분기 예측의 이점을 없애고, 연속된 데이터 의존성(예: temp[0] 계산 결과가 다음 temp[1] 계산에 바로 영향을 주는 구조)으로 인해 CPU가 병렬로 처리할 수 있는 독립적인 명령어를 찾기 어렵게 만든다. 이로 인해 파이프라인 스톨이 빈번하게 발생하여 IPC가 1.91까지 하락한 것으로 분석된다. 이 결과는 단순히 루프를 제거하는 순진한 최적화가 현대 CPU 아키텍처에서는 오히려 성능 저하를 유발할 수 있으며, 코드의 미세 구조가 하드웨어의 병렬 실행 능력과 어떻게 상호작용하는지가 성능의 핵심 요소임을 실증적으로 보여준다.

```
1   mov     x16, #0
2   add     x8, x0, #12
3   mov     w10, #53955
4   movk    w10, #61665, lsl, #16
5   mov     w9, #38535
6   movk    w9, #46245, lsl, #16
7   mov     w11, #23115
8   movk    w11, #30825, lsl, #16
9   mov     w12, #7695
10  movk    w12, #15405, lsl, #16
11  mov     w13, #-2
12  mov     w14, #-3
13  Lloh6:
14  adrp    x15, l__unnamed_3@PAGE
15  Lloh7:
16  add     x15, x15, l__unnamed_3@PAGEOFF
17  LBB8_1:
18  and     x17, x16, #0x3
19  neg     w2, w16
20  sub     w3, w13, w16
21  add     x4, x16, #1
22  ldr     w17, [x15, x17, lsl, #2]
23  ror     w2, w17, w2
24  add     w12, w2, w12
25  neg     w2, w4
26  ror     w2, w17, w2
27  add     w11, w11, w2
28  ror     w12, w12, #31
29  ror     w11, w11, #29
30  sub     w16, w14, w16
31  ror     w2, w17, w3
32  add     w9, w9, w2
33  ror     w9, w9, #26
34  ror     w16, w17, w16
35  add     w10, w10, w16
36  stp     w12, w11, [x8, #-12]
37  ror     w10, w10, #21
38  stp     w9, w11, [x8, #-4]
39  stp     w10, w11, [x8, #4]
40  add     x8, x8, #24
41  mov     x16, x4
42  cmp     x4, #24
43  b.ne    LBB8_1
44  add     x8, x1, #552
45  add     x9, x0, #16
46  mov     w10, #24
47  LBB8_3:
48  ldur    q0, [x9, #-16]
49  str     q0, [x8]
50  ldr     d0, [x9], #24
51  str     d0, [x8, #16]
52  sub     x8, x8, #24
53  subs    x10, x10, #1
54  b.ne    LBB8_3
55  ret
```

표 [6-1] Rust round_key_gen_24.s

```
#include "lea.h"
#include "lea_locl.h"
#include <stdio.h>
static const unsigned int delta[8][36] = {
        {0xc3efe9db,  0x87dfd3b7,  0x0fbfa76f,  0x1f7f4ede,  0x3efe9dbc,
0x7dfd3b78, 0xfbfa76f0, 0xf7f4ede1,
        0xefe9dbc3,  0xdfd3b787,  0xbfa76f0f,  0x7f4ede1f,  0xfe9dbc3e,
0xfd3b787d, 0xfa76f0fb, 0xf4ede1f7,
        0xe9dbc3ef,  0xd3b787df,  0xa76f0fbf,  0x4ede1f7f,  0x9dbc3efe,
0x3b787dfd, 0x76f0fbfa, 0xede1f7f4,
        0xdbc3efe9,  0xb787dfd3,  0x6f0fbfa7,  0xde1f7f4e,  0xbc3efe9d,
0x787dfd3b, 0xf0fbfa76, 0xe1f7f4eD,
        0xc3efe9db,      0x87dfd3b7, 0x0fbfa76f, 0x1f7f4ede},
 ...
};
void lea_set_key_generic(LEA_KEY *key, const unsigned char *mk, unsigned
int mk_len)
{
        if(!key)
                return;
        else if(!mk)
                return;
        const unsigned int* _mk = (const unsigned int*)mk;
        switch(mk_len)
        {
        case 16: //128비트 키스케줄 코드
        case 24: //192비트 키스케줄 코드
        case 32: //256비트 키스케줄 코드
        ...
                break;
        default:
                        return;
        }
        key->round = (mk_len >>1) +16;
}
```

표 [6-2] LEA C의 core.c

```
fn round_key_gen_24(mk: &[u32; 8], erk: &mut [u32; 192], drk: &mut [u32; 192]) {
    let mut temp: [u32; 4] = [0; 4];
    for i in 0..4 {
        temp[i] = mk[i];
    }
    for i in 0..24 {
        temp[0] = temp[0].wrapping_add(KEY_CONST[i % 4].rotate_left(i as u32)).rotate_left(1);
        temp[1] = temp[1].wrapping_add(KEY_CONST[i % 4].rotate_left((i+1) as u32)).rotate_left(3);
        temp[2] = temp[2].wrapping_add(KEY_CONST[i % 4].rotate_left((i+2) as u32)).rotate_left(6);
        temp[3] = temp[3].wrapping_add(KEY_CONST[i % 4].rotate_left((i+3) as u32)).rotate_left(11);
        erk[i *6] = temp[0];
        erk[i *6 +1] = temp[1];
        erk[i *6 +2] = temp[2];
        erk[i *6 +3] = temp[1];
        erk[i *6 +4] = temp[3];
        erk[i *6 +5] = temp[1];
    }
    for i in 0.. 24{
        drk[138 - i*6] = erk[i*6];
        drk[139 - i*6] = erk[i*6 +1];
        drk[140 - i*6] = erk[i*6 +2];
        drk[141 - i*6] = erk[i*6 +3];
        drk[142 - i*6] = erk[i*6 +4];
        drk[143 - i*6] = erk[i*6 +5];
    }
}
```

표 [6-3] 기존 Rust round_key_gen_24

```
fn round_key_gen_24(mk: &[u32; 8], erk: &mut [u32; 192], drk: &mut [u32; 192]) {
    let mut temp: [u32; 4] = [mk[0], mk[1], mk[2], mk[3]];

    // Round 0
    temp[0] = temp[0].wrapping_add(DELTA[0][0]).rotate_left(1);
    temp[1] = temp[1].wrapping_add(DELTA[0][1]).rotate_left(3);
    temp[2] = temp[2].wrapping_add(DELTA[0][2]).rotate_left(6);
    temp[3] = temp[3].wrapping_add(DELTA[0][3]).rotate_left(11);
    erk[0] = temp[0]; erk[1] = temp[1]; erk[2] = temp[2];
    erk[3] = temp[1]; erk[4] = temp[3]; erk[5] = temp[1];
    // Round 1 ~ Round 23
      …
    // 암호화 0라운드 키 -> 복호화 23라운드 키
    drk[138] = erk[0]; drk[139] = erk[1]; drk[140] = erk[2]; drk[141] = erk[3]; drk[142] = erk[4]; drk[143] = erk[5];
   // 암호화 1~23 라운드 키 -> 복호화 22 ~ 0라운드 키
   …
}
```

표 [6-4] 수정된 Rust round_key_gen_24

함수	구현 방식	총 실행 명령어 (Instructions)	총 소요 사이클 (Cycles)	L1 명령어 캐시 미스(L1I Cache Miss)	IPC (Instructions/Cycles)
enc	for 루프 버전(A)	10,903,148	5,457,760	17,028	2.00
	나열식 버전(B)	20,117,590	9,171,408	22,723	2.19
dec	for 루프 버전(A)	168,103,140	43,403,918	27,626	3.87
	나열식 버전(B)	132,462,824	34,654,564	25,924	3.82
round_key_gen_24	for 루프 버전(A)	29,732,708	6,646,872	2,684	4.47
	나열식 버전(B)	2,236,701	1,173,871	2,168	1.91
round_key_gen_28	for 루프 버전(A)	259,154,246	52,929,238	2,333	4.90
	나열식 버전(B)	52,775,985	18,255,180	13,430	2.89
round_key_gen_32	for 루프 버전(A)	698,256,535	147,815,959	3,749	4.70
	나열식 버전(B)	137,335,610	46,418,752	24,108	3.82

표 [6-5] Xcode Instruments CPU Counter 측정 결과

제 7 장 결 론

본 논문은 시스템 프로그래밍 언어 Rust로 블록 암호 LEA를 구현함으로써, 메모리 안전성을 확보하는 동시에 기존 C 기반 구현과 실용 성능을 얼마나 근접하게 달성할 수 있는지 실증적으로 검증하였다. 연구를 통해 Rust 특유의 소유권, 빌림, 수명(lifetime) 시스템이 버퍼 오버플로우, use-after-free, double free 등 다양한 메모리 오류를 언어 차원에서 차단하여 암호 알고리즘과 같이 정밀하고 보안이 중요한 소프트웨어에서 신뢰성의 근본적인 향상을 가져온다는 점을 확인했다.

실험 결과, 디버그 환경에서 Rust 구현은 C 구현에 비해 키 스케줄링과 암복호화 작업에서 각각 약 3~10배의 성능 저하가 관찰되었으나, 이는 Rust 언어 자체의 한계가 아니라, 최적화 기법과 하드웨어 마이크로아키텍처와의 상호작용 차이에서 비롯된다는 점을 성능 분석을 통해 규명하였다. 구체적으로, C 레퍼런스 코드는 라운드 상수와 회전값 테이블을 정적으로 미리 계산하는 정적 전처리와 테이블 기반 최적화를 적극적으로 활용하여, 런타임 연산량을 최소화한다는 설계상 강점을 지닌다. 반면 Rust 구현은 반복문 내에서 동적으로 회전 연산을 처리하는 구조를 따랐으며, 이 방식은 현대 CPU의 분기 예측, 파이프라이닝, 명령어 병렬화(ILP)에는 유리하지만, 총 실행 명령어와 사이클 수 측면에서는 비효율이 존재했다.

디어셈블 분석과 하드웨어 성능 카운터(Xcode Instruments, CPU Counter) 측정을 통해, 단순 루프 언롤링이나 테이블화만으로는 Rust와 C 구현의 성능 갭을 완전히 해소할 수 없음을 확인했으며, 오히려 CPU 구조에 따라 루프 기반 코드가 더 높은 IPC를 보일 수도 있음을 발견하였다. 즉, 최적의 성능을 위해서는 단순 알고리즘 이식을 넘어 Rust 환경에 맞는 빌드 스크립트, const fn, 컴파일러 힌트 등 다양한 정적 최적화 기법을 적극 도입해야 함을 시사한다.

무엇보다 중요한 성과는, Rust가 제공하는 메모리 모델이 시스템 보안 소프트웨어의 핵심 품질인 메모리 안전성을 본질적으로 보장함으로써, 장기적으

로 소프트웨어 공급망 및 인프라 보안의 수준을 한 단계 제고할 수 있는 실질적 대안임을 실험적으로 입증했다는 점이다. 최근 정부와 산업계가 Rust 등 메모리 안전 언어 도입을 적극 권고·확대하는 글로벌 정책 흐름과 맞물려, 본 연구는 암호 SW 등 고신뢰 시스템 개발에 있어 Rust와 같은 메모리 안전 언어의 실질적 도입 타당성을 제시하는 사례로서 의의가 크다.

참 고 문 헌

1. 국내문헌

국가보안기술연구소(NSR). (2013). 128비트 블록암호 LEA 규격서.

Seo, H. J. (2017). High speed implementation of LEA on ARMv8. Journal of the Korea Institute of Information and Communication Engineering, 21(10), 1929-1934.

Seo, H. J., & Kim, H. W. (2015). Implementation of lightweight cryptographic algorithm for the Internet of Things. Review of KIISC, 25(2), 12-19.

Hong, D., Lee, J. K., Kim, D. C., Kwon, D., Ryu, K. H., & Lee, D. G. (2013, August). LEA: A 128-bit block cipher for fast encryption on common processors. In international workshop on information security applications (pp. 3-27). Cham: Springer International Publishing.

2. 국외문헌

Berger, E., McKinley, K., Blumofe, R., & Wilson, P. (2000). Hoard: A scalable memory allocator for multithreaded applications. In Proceedings of the 2000 USENIX Annual Technical Conference. USENIX Association.

Bernhard, L., Scharnowski, T., Schloegel, M., Blazytko, T., & Holz, T. (2022, November). JIT-picking: Differential fuzzing of JavaScript engines. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (pp. 351-364). ACM.

Bianchini, R., & Lim, B. (1996). Evaluating the performance of multithreading and prefetching in multiprocessors. Journal of Parallel and Distributed Computing, 37, 83-97.

Burow, N., Carr, S. A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., & Payer, M. (2017). Control-flow integrity: Precision, security, and performance. ACM Computing Surveys, 50(1), 1-33.

Chen, S., Joshi, K. R., Hiltunen, M., Schlichting, R., & Sanders, W. (2011). Using CPU gradients for performance-aware energy conservation in multitier systems. Sustainable Computing: Informatics and Systems, 1, 113-133.

Chromium Projects, The. Memory safety. Retrieved July 25, 2022, from https://www.chromium.org/Home/chromium-security/memory-safety/

Crichton, W. (2020). The usability of ownership (arXiv:2011.06171).

Crichton, W., Gray, G., & Krishnamurthi, S. (2023). A grounded conceptual model for ownership types in Rust. Proceedings of the ACM on Programming Languages, 7, 1224-1252.

Cybersecurity and Infrastructure Security Agency. (2022, April 28). 2021 Top routinely exploited vulnerabilities (Alert AA22-117A). https://www.cisa.gov/news-events/cybersecurity-advisories/aa22-117a

Ferreira, T., Matias, R., Macêdo, A., & de Araujo, L. B. (2011). An experimental study on memory allocators in multicore and multithreaded applications. In 12th International Conference on Parallel and Distributed Computing, Applications and Technologies (pp. 92-98). IEEE.

Gidenstam, A., Papatriantafilou, M., & Tsigas, P. (2010). Nbmalloc: Allocating memory in a lock-free manner. Algorithmica, 58, 304-338.

Gupta, A., Sampson, J., & Taylor, M. (2014). Quality time: A simple online technique for quantifying multicore execution efficiency. In 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (pp. 169-179). IEEE.

Huang, W., Qian, Y., Srisa-an, W., & Chang, J. M. (2004). Object allocation and memory contention study of Java multithreaded applications. In IEEE International Conference on Performance, Computing, and Communications (pp. 375-382). IEEE.

Internet Security Research Group. What is memory safety and why does it matter? Retrieved May 6, 2022, from https://www.memorysafety.org/docs/memory-safety/

Ismail, N. A. (2002). Evaluation of dynamic branch predictors for modern ILP processors. 『Microprocessors and Microsystems』, 26(5), 215-231.

Li, P. (2004). Safe systems programming languages (Master's thesis). Department of Computer and Information Science, University of Pennsylvania.

Microsoft Security Response Center. (2020). CVE-2020-0601. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2020-0601

Microsoft Security Response Center. (2021). CVE-2021-21148. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-21148

Microsoft Security Response Center. (2021). CVE-2021-31166. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-31166

Miller, M. (2019). Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape [Conference presentation]. Microsoft MSRC Security Research.

https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf

National Security Agency. (2022). Software memory safety (CSI Software Memory Safety, PP-23-0782, Ver. 1.1). https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF

Pearce, D. J. (2021). A lightweight formalism for reference lifetimes and borrowing in Rust. ACM Transactions on Programming Languages and Systems, 43, Article 1-73.

Szekeres, L., Payer, M., Wei, T., & Song, D. (2013). SoK: Eternal war in memory. In 2013 IEEE Symposium on Security and Privacy (pp. 48-62). IEEE.

Tallent, N. R., & Mellor-Crummey, J. (2009). Effective performance measurement and analysis of multithreaded applications. ACM SIGPLAN Notices, 44, 229-240.

U.S. Government Accountability Office. (2021, April 22). SolarWinds cyberattack demands significant federal and private-sector response [Infographic]. https://www.gao.gov/blog/solarwinds-cyberattack-demands-significant-federal-and-private-sector-response-infographic

Umayabara, A., & Yamana, H. (2017). Mcmalloc: A scalable memory allocator for multithreaded applications on a many-core shared-memory machine. In 2017 IEEE International Conference on Big Data (Big Data) (pp. 4846-4848). IEEE.

White House, The. (2024, May). National Cybersecurity Strategy Implementation Plan (Version 2). https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/05/

National-Cybersecurity-Strategy-Implementation-Plan-Version-2.pdf

ABSTRACT

Implementation Memory Protection for block cipher LEA using Rust

Kim, Sang-Won

Major in Convergence Security

Dept. of Convergence Security

The Graduate School

Hansung University

The implementation of block ciphers has traditionally relied on the C programming language, valued for its system-level efficiency and hardware affinity. However, C is notoriously susceptible to memory safety issues such as buffer overflows and memory leaks. In contrast, Rust has emerged as a compelling alternative for systems programming, leveraging its ownership model to enforce memory safety at compile time.

This paper aims to evaluate whether implementing the LEA block cipher in Rust can achieve practical performance while guaranteeing memory safety. The study involves comparing the performance of a Rust implementation with that of existing C implementations under non-optimized conditions. The results indicate that the Rust implementation exhibits some performance degradation compared to its C counterpart. To analyze this gap, tools such as Xcode Instruments' Time Profiler, CPU Counter, and disassembly techniques were employed.

【Keywords】 Rust, Memory Safety, LEA