C/C++ 대비 Rust의 메모리 안전성 보장 메커니즘 분석

안승민, 김예진, 김예찬, 권다은, 김동찬*
국민대학교

{bryan0126, alice1225kim, tomking0820, ekdms3809, dckim*}@kookmin.ac.kr

A Comparative Analysis of Memory Safety Guarantee Mechanisms in Rust and C/C++

Ahn Seung-Min, Kim Ye-Jin, Kim Yae-Chan, Kwon Da-Eun, Kim Dong-Chan*
Kookmin University

요 약

C/C++는 운영체제 및 임베디드 시스템 등 저수준 소프트웨어 개발에 널리 활용되어 왔으나, 메모리 관련 취약점에 기인한 보안 문제가 지속적으로 보고되고 있다. 선행 보고에 따르면 buffer overflow, use-after-free, dangling pointer와 같은 메모리 안전성 결함은 전체 취약점의 약 2/3을 차지한다. 본 논문은 이러한 문제를 완화하기 위한 대안으로 제안된 Rust 언어의 개발 배경을 분석하고, 소유권·빌림·수명 규칙과 Option 타입, 경계 검사 및 슬라이스 기반 접근 제어 등 핵심 특성이 컴파일 타임에 메모리 안전성을 보장하는 방식과 그 의미를 정리한다. 또한 Rust는 정적 검증을 통해 가비지 컬렉터 없이도 메모리 오류와 데이터 레이스를 예방하며, 추가적인 런타임 오버헤드 없이 C/C++에 준하는 성능과 메모리 안전성을 동시에 달성할 수 있음을 주장한다.

Ⅰ. 서 론

저수준(Low-Level) 프로그래밍 언어는 운영체제, 임베디드 시스템, 네트워크 인프라와 같이 하드웨어 자원에 대한 직접 제어가 요구되는 분야에서 사용된다. 1970년대에 개발된 C 언어와 이를 확장한 C++는 이러한 요구를 충족하는 언어로서 시스템 소프트웨어 개발에 활용되어 왔다.

그러나 하드웨어 자원에 대한 직접 접근을 허용하는 언어에서는 메모리 관리가 개발자에게 크게 의존하며, 그 결과 메모리 관련 취약점이 지속적으로 보고되고 있다. 2023년 미국 CISA와 NSA가 공동 발표한 “The Case for Memory Safe Roadmaps” 보고서에 따르면, 메모리 안전성을 언어 차원에서 보장하지 않는 언어로 구현된 소프트웨어에서 보고되는 취약점의 약 3분의 2가 메모리 관련 결함에서 기인한다[1]. 2024년 미국 백악관 보고서 또한 메모리 안전 언어(Memory safe language)로의 전환을 권고하면서, 그 예시로 Rust를 제시하였다[2].

Rust는 소유권(Ownership) 기반의 메모리 관리 체계를 도입하여, 가비지 컬렉터 없이도 컴파일 타임에 메모리 안전성을 검증할 수 있도록 설계되었다. 본 논문은 C/C++에서 보고된 메모리 관련 취약점의 유형과 대표 사례를 제시한 뒤, secure coding 관점에서 Rust의 개발 배경을 정리하고, Rust의 메모리 안전성 보장 메커니즘을 분석한다.

Ⅱ. C/C++의 보안 취약점 유형 및 대표 사례

C/C++로 구현된 소프트웨어에서는 다양한 메모리 관련 보안 취약점이 보고되고 있다. buffer overflow는 배열 경계를 초과하는 쓰기 동작으로 인해 인접 메모리가 덮어써지는 취약점이며, 스택 또는 힙 영역에서 제어 흐름이 변조될 가능성을 초래한다. 2023년 보고된 CVE-2023-4863은 libwebp 라이브러리의 힙 버퍼 오버플로우 취약점으로, Chrome, Firefox, Edge 등 주요 브라우저와 다수의 Electron 기반 애플리케이션에 영향을 미쳤고 실제 공격에 악용된 사례로 보고되었다[3].

use-after-free는 해제된 메모리 영역에 대한 접근이 발생하는 취약점으로, 악용될 경우 임의 코드 실행으로 이어질 수 있다. 2024년 보고된 Chrome 브라우저의 CVE-2024-3914는 V8 JavaScript 엔진의 use-after-free 결함으로 인해 원격 코드 실행이 가능했던 사례로 보고되었다[4].

double free는 동일 메모리 블록을 중복으로 해제하는 오류로, 힙 메타데이터 손상을 유발할 수 있다. 2024년 보고된 CVE-2024-1086은 Linux 커널의 netfilter nf_tables 구성요소에서 double free 결함이 확인된 사례로, 로컬 권한 상승이 가능했으며 랜섬웨어 공격에 악용된 것으로 보고되었다[5].

null pointer dereference는 널 포인터를 역참조함으로써 프로그램 비정상 종료를 유발하는 취약점이다. 2024년 Linux 커널의 CVE-2024-49921은 AMD GPU 드라이버에서 null pointer dereference로 커널 패닉을 유발할 수 있는 것으로 확인되었다[6].

race condition은 멀티스레드 환경 또는 시그널 처리 과정에서 동기화 없이 공유 자원에 동시 접근할 때 발생할 수 있다. 2024년 OpenSSH의 CVE-2024-6387(regreSSHion)은 시그널 핸들러의 race condition으로 인해 인증 없이 원격에서 root 권한으로 코드 실행이 가능하다고 보고되었다[7]. 이에 따라 이러한 취약점을 사전에 예방하기 위한 secure coding 방법론의 도입과 발전이 중요한 과제로 논의되어 왔다.

Ⅲ. Rust의 개발 배경

앞서 제시한 취약점 유형을 사전에 예방하기 위한 방법으로 secure coding 방법론이 활용되어 왔다. secure coding은 소프트웨어 개발 과정에서 보안 취약점을 사전에 방지하기 위한 코딩 원칙 및 기법을 의미한다[8]. OWASP 등 주요 보안 기관은 secure coding 가이드라인을 제시하고 있으며, 메모리 관리, 입력 검증, 오류 처리 등을 주요 항목으로 다룬다[8]. 그러나 전통적인 secure coding 접근 방식은 개발자의 주의, 코드 리뷰, 정적 분석 도구 등에 의존하므로 인적 오류를 배제하기 어렵다는 한계가 있다.

Rust는 이러한 한계를 완화하기 위해 secure coding 원칙이 언어 설계 차원에서 강제되도록 설계되었다. Rust는 2006년 Mozilla 소속 개발자 Graydon Hoare의 개인 프로젝트로 시작되었으며, 2009년 Mozilla가 공식 후원하기 시작한 이후 Servo 프로젝트의 구현 언어로 채택되었다. Rust의 설계 목표는 C/C++와 동등한 성능을 유지하면서도 메모리 안전성과 스레드 안전성을 컴파일 타임에 보장하는 데 있다. 다음 절에서는 이러한 설계 목표를 실현하기 위한 Rust의 구체적인 메모리 안전성 보장 메커니즘을 분석한다.

Ⅳ. Rust의 메모리 안전성 보장 메커니즘

소유권은 Rust의 핵심 개념으로, 각 값에는 단 하나의 소유자가 부여되며 소유자의 스코프가 종료되면 해당 값이 자동으로 해제된다. C에서는 해제된 메모리에 대한 접근 또는 중복 해제가 런타임에 발생할 수 있으나, Rust에서는 소유권이 이전된 값을 다시 사용하려 할 때 컴파일 오류가 발생하므로 use-after-free와 double free를 방지할 수 있다. 관련 예시는 [그림 1]에 제시하였다.

```rust
fn main() {
    let ptr = Box::new(42);
    drop(ptr);  // 소유권 해제
    // println!("{}", ptr); // 컴파일 에러: value borrowed after move
}
```

〔그림 1〕 소유권 예제 코드

빌림은 동일한 데이터에 대해 하나의 가변 참조 또는 다수의 불변 참조만 존재하도록 강제함으로써 데이터 레이스를 컴파일 타임에 방지한다. 이를 통해 동기화 누락에 따른 공유 자원 동시 접근 문제를 컴파일 단계에서 차단한다.관련 예시는 [그림 2]에 제시하였다.

```rust
fn main() {
    let mut data = vec![1, 2, 3];

    let r1 = &data;      // 불변 참조 1
    let r2 = &data;      // 불변 참조 2 (허용)
    println!("{:?}, {:?}", r1, r2);

    let r3 = &mut data;  // 가변 참조
    // println!("{}", r1); // 컴파일 에러: 불변 참조와 가변 참조 동시 사용 불가
    r3.push(4);
}
```

〔그림 2〕 빌림 예제 코드

수명은 참조의 유효 범위를 컴파일 타임에 검증하여 dangling pointer를 방지한다. 예를 들어, C에서 허용되는 지역 변수 참조 반환은 Rust에서 컴파일 오류로 처리된다. 관련 예시는 [그림 3]에 제시하였다.

```rust
fn get_local_ref() ->&i32{
    let local =42;
    &local      // 컴파일 에러: cannot return reference to local variable
}
fn main() {
    // let ptr = get_local_ref();
}
```

〔그림 3〕 수명 예제 코드

Option 타입은 널 포인터를 타입 시스템으로 대체하여 null pointer dereference를 방지한다. Rust는 값의 부재를 Option<T>로 표현하며, 내부 값에 접근할 때 명시적 분기 처리를 요구한다. 관련 예시는 [그림 4]에 제시하였다.

```rust
fn find_value(arr:&[i32], target:i32) -> Option<&i32> {
    arr.iter().find(|&&x|x == target)
}
fn main() {
    let arr = [1, 2, 3];
    match find_value(&arr, 5) {
        Some(val) => println!("{}", val),
        None => println!("Not found"),  // 명시적 처리 강제
    }
}
```

〔그림 4〕 Option 예제 코드

buffer overflow는 경계 검사와 슬라이스를 통해 완화된다. Rust에서는 인덱스 범위 초과 접근이 런타임 패닉으로 이어질 수 있으며, get()을 사용하면 Option<T>로 반환되어 값의 존재 여부를 명시적으로 처리할 수 있다. 관련 예시는 [그림 5]에 제시하였다.

```rust
fn main() {
    let arr = [1, 2, 3];
    // println!("{}", arr[10]);  // 런타임 패닉: index out of bounds
    match arr.get(10) {    // 안전한 접근
        Some(val) => println!("{}", val),
        None => println!("Index out of bounds"),
    }
}
```

〔그림 5〕 Buffer overflow 방지 예제 코드

이와 같은 특성으로 Rust는 개발자의 실수에 의존하지 않고 컴파일러 수준에서 secure coding을 강제하는 “memory safe by default” 패러다임을 제시한다. Rust는 2025년 Linux 커널의 공식 언어로 확정되어 실험 단계를 종료하였으며, Android, AWS, Microsoft Windows 등에서 적용이 확대되고 있다[9]. Google의 Android 보안 팀 보고에 따르면 Rust 도입 이후 Android의 메모리 안전성 취약점 비율은 2019년 76%에서 2024년 24%로 감소하였다[10].

성능 측면에서도 Rust는 C/C++와 대등한 수준의 결과를 보인다. Computer Language Benchmarks Game의 측정 결과, Rust는 C와 비교하여 대부분의 알고리즘에서 5-10% 이내의 실행 시간 차이를 보인다[11]. 이는 Rust의 메모리 안전성 검증이 컴파일 타임에 수행되어 런타임 오버헤드가 발생하지 않는 제로 비용 추상화(Zero-Cost Abstraction) 원칙과 연관된다. 또한 Discord는 Rust 전환 이후 가비지 컬렉터로 인한 지연이 감소했다고 보고하였다[12].

다만 Rust의 메모리 안전성 보장에는 한계가 존재한다. unsafe 블록 내에서는 소유권 규칙이 적용되지 않으므로 개발자가 직접 메모리 안전성을 책임져야 한다. FFI(Foreign Function Interface)를 통해 C 라이브러리를 호출하는 경우에도 unsafe 사용이 필요하며, 이때 외부 C 코드의 취약점이 Rust 코드로 전이될 가능성을 배제하기 어렵다. 2024년 기준 crates.io에 등록된 크레이트 중 19.11%가 unsafe 키워드를 사용하고 있으며, 34.35%는 unsafe를 사용하는 다른 크레이트를 직접 호출하는 것으로 나타났다[13].

V. 결론

본 논문은 C/C++ 기반 소프트웨어에서 보고되는 메모리 관련 취약점의 대표 유형을 실제 사례를 통해 제시하고, 이러한 취약점이 보안 침해로 이어질 수 있음을 정리하였다. 또한 개발자 중심의 secure coding 접근이 인적 오류를 완전히 배제하기 어렵다는 한계를 바탕으로, Rust가 소유권·빌림·수명 규칙과 Option 타입, 경계 검사 및 슬라이스 등을 통해 컴파일 타임에 메모리 안전성을 강화하는 메커니즘을 갖는다는 점을 제시하였다.

아울러 관련 자료를 근거로 Rust가 메모리 안전 언어로의 전환 흐름에서 중요한 대안으로 제시되고 있으며, 적용 사례와 성능 비교 결과를 통해 보안 개선과 성능 유지가 양립 가능함을 주장하였다. 다만 unsafe 사용과 외부 코드 연동에서는 안전성 보장이 제한될 수 있으므로, 도입 시 해당 구간에 대한 관리와 점진적 전환 전략이 병행되어야 한다.

ACKNOWLEDGMENT

이 논문은 2025년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임(No. RS-2024-00397105, KCMVP 보안수준 3 암호 모듈 제작을 위한 핵심기술 개발).

참 고 문 헌

[1] CISA, NSA, "The Case for Memory Safe Roadmaps", 2023.
[2] The White House, "Back to the Building Blocks", ONCD, 2024.
[3] NIST NVD, "CVE-2023-4863: libwebp heap buffer overflow", 2023.
[4] NIST NVD, "CVE-2024-3914: Chrome V8 use-after-free", 2024.
[5] NIST NVD, "CVE-2024-1086: Linux nf_tables double-free", 2024.
[6] NIST NVD, "CVE-2024-49921: AMDGPU NULL pointer deref", 2024.
[7] Qualys, "CVE-2024-6387 (regreSSHion): OpenSSH race condition", 2024.
[8] OWASP, "Secure Coding Practices Quick Reference Guide" 2024.
[9] LWN.net, "The end of the kernel Rust experiment", 2025.
[10] Google, "Eliminating Memory Safety Vulnerabilities", 2024.
[11] The Benchmarks Game, "Rust vs C clang - Which programs are fastest?", Debian, 2024.
[12] Discord Engineering, "Why Discord is switching from Go to Rust", Discord Blog, 2020.
[13] Rust Foundation, "Unsafe Rust in the Wild: Notes on the Current State of Unsafe Rust," 2024.
[14] Google Open Source Blog, "Rust fact vs. fiction: 5 Insights from Google's Rust journey in 2022", 2023.