Journal of the The Korean Institute of Defense Technology Vol. 08, No. 2, pp. 007-013, 2026

유도탄 소프트웨어 적용을 위한 Rust 언어의 성능 및 안정성 분석

Performance and Stability Analysis of Rust for Application to Missile Software Systems

김주원*,1) . 황지호1) . 박주현1)

Juwon Kim*,1) . Jiho Hwang1) . Juhyeon Park1)

[초 록]

본 논문은 유도탄 체계에서 사용되는 시스템 소프트웨어 환경을 대상으로 Rust 프로그래밍 언어의 적용 가능성을 분석한다. 기존 C/C++ 기반 시스템 소프트웨어는 메모리 관리, 동시성 제어 문제와 같은 안정성 한계를 가진다. 이에 Rust와 C/C++의 구조적 차이를 이론적으로 비교 분석하고, 시스템 프로그래밍 환경에서의 성능 및 안정성 특성을 검토하여 방산 소프트웨어에서 Rust의 활용 가능성을 제시한다. 본 연구는 유도탄 소프트웨어의 성능 향상에 기여할 것으로 예상된다.

[ABSTRACT]

This study evaluates the applicability of Rust to missile system software to address the memory management and concurrency limitations of conventional C/C++. By theoretically comparing structural characteristics and examining performance and reliability in system programming environments, we demonstrate Rust's potential in defense systems. The findings are expected to significantly enhance the reliability and safety of missile system software.

Key Words : Rust(러스트), System Software(시스템 소프트웨어), Memory Safety(메모리 안전성)

1. 서 론

현재 방산 분야에서 소프트웨어의 역할이 지속적으로 확대되고 있으며, 유도탄과 같은 무기체계에서도 다양한 소프트웨어 시스템이 핵심적인 기능을 담당하고 있다. 유도탄 체계에는 자세 제어, 센서 데이터 처리, 통신 등의 기능을 수행하는 내부 임베디드 소프트웨어뿐만 아니라, 유도탄의 상태 확인과 기능 검증을 수행하는 점검 장비 소프트웨어가 함께 활용된다.[1]

유도탄 소프트웨어는 일반적인 무기체계 소프트웨어와 비교하여 제한된 운용 시간 동안 고주기 센서 데이터 처리, 유도·항법·제어 연산, 상태정보 송수신, 발사 전·후 점검 및 시험 지원 기능이 밀접하게 연계된다는 특성을 가진다. 특히 비행 중에는 짧은 주기 내에 수신된 데이터를 처리하고 제어 명령 또는 상태 정보를 안정적으로 전달해야 하며, 점검 장비 소프트웨어 역시 유도탄 내부 장치와의 통신을 통해 실시간 상태 확인과 기능 검증을 수행해야 한다. 따라서 유도탄 소프트웨어에서는 처리 성능뿐만 아니라 실시간 통신 지연, 메모리 오류 방지, 장시간 운용 중 안정적인 자원 관리가 중요한 요구사항으로 작용한다.

이러한 소프트웨어 시스템은 대부분 실시간 데이터 처리와 안정적인 통신 기능을 요구하는 시스템 소프트웨어 형태로 구현되며, 현재까지는 주로 C/C++ 언어를 기반으로 개발되어 왔다. 그러나 C/C++ 기반 시스템 소프트웨어에서는 메모리 관리와 동시성 제어를 개발자가 직접적으로 수행하는 구조를 가지고 있으며, 이로 인해 다양한 메모리 오류가 발생할 가능성이 존재한다.

대표적으로 Dangling pointer, Use-after-free, Double free와 같은 오류가 발생할 수 있으며, 이러한 문제는 시스템 안정성과 보안 취약점의 주요 원인이 될 수 있다. 실제로

1) ㈜LIG 디펜스앤에어로스페이스 미사일시스템 체계종합연구소(Missile Systems, Naval Surface-to-air Missile Development, LIG Defense&Aerospace)
* Corresponding author, E-mail: juwon.kim7@ligdna.com

Received : May 19, 2026 Revised : June 18, 2026
Accepted : June 30, 2026

Microsoft 보안 분석에 따르면 대규모 소프트웨어 시스템에서 발생하는 취약점(CVE)의 약 70%가 메모리 안전성 문제와 관련되어 있는 것으로 보고되고 있다.[2]

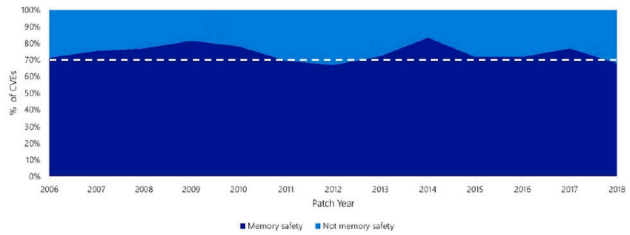

그림 1. Microsoft CVE 취약점 중 메모리 안전성 관련 취약점 비율 (2006-2018) [MSRC, 2019]

Fig. 1. Proportion of memory safety vulnerabilities in Microsoft CVEs (2006-2018) [MSRC, 2019]

이러한 문제는 복잡한 시스템 소프트웨어 환경에서 더욱 두드러지며, 유도탄 소프트웨어에서도 중요한 기술적 과제로 인식되고 있다.

최근 이를 해결하기 위한 대안으로 Rust가 주목받고 있다. Rust는 2010년 Mozilla에서 개발된 시스템 프로그래밍 언어로, 가비지 컬렉터 없이도 메모리 안전성을 보장하도록 설계되었다. Rust는 소유권 기반 메모리 관리 모델과 컴파일 단계의 Borrow Checking 메커니즘을 통해 메모리 오류를 사전에 방지하며, 동시성 처리 과정에서 발생할 수 있는 데이터 경합을 컴파일 단계에서 검증한다는 특성을 가진다.[3] 이에 따라 Rust는 최근 시스템 소프트웨어 분야에서 적용이 확대되고 있으며, Linux Kernel에서도 Rust 언어 지원이 도입되는 등 활용도가 높아지고 있다.

따라서 본 논문에서는 유도탄 소프트웨어에서 요구되는 고주기 통신, 파일 입출력, 연산 처리, 메모리 안전성 관점에서 Rust와 C++의 특성을 비교하고, Rust가 기존 C/C++ 기반 개발 방식의 보완 또는 대안으로 적용될 수 있는지 검토한다.

2. Rust 개요 및 산업적 동향

2.1 Rust의 설계 철학 및 특징

Rust는 2010년 Mozilla Research에서 개발된 시스템 프로그래밍 언어로, 가비지 컬렉터 없이도 메모리 안전성을 보장할 수 있도록 설계되었다. Rust의 핵심 설계 철학은 크게 3가지로, 메모리 안전성 확보, 동시성 안전성 보장, 성능 저하 없이 고수준 언어 기능을 제공하는 Zero-cost abstraction으로 요약할 수 있다.

2.1.1 메모리 안전성 확보

Rust의 가장 큰 특징은 소유권(Ownership) 기반 메모리 관리 모델이다. Rust에서는 모든 데이터가 단일 소유자를 가지며, 소유자가 스코프를 벗어나면 메모리가 자동으로 해제된다. 또한 컴파일 단계에서 Borrowing과 Lifetime 개념을 통해 데이터 참조 관계를 검증한다. 이러한 구조는 Dangling pointer나 Use-after-free와 같은 메모리 오류를 사전에 방지할 수 있도록 하며, 메모리 안전성을 확보할 수 있도록 한다.[4]

2.1.2 동시성 안전성 보장

또한 Rust는 동시성 안전성을 중요한 설계 목표로 한다. C/C++에서는 멀티스레드 환경에서 데이터 경합(Data Race)이 발생할 가능성이 있으며 이러한 문제는 시스템 안정성에 큰 영향을 미칠 수 있다. Rust는 Send와 Sync 트레이트를 기반으로 스레드 간 데이터 공유의 안정성을 컴파일 단계에서 검증하며, 이를 통해 동시성 오류를 사전에 방지할 수 있도록 설계되었다.

2.1.3 Zero-cost Abstraction

마지막으로 Rust는 Zero-cost abstraction 설계 원칙을 기반으로 한다. 이는 고수준 언어 기능을 제공하면서도 런타임 오버헤드를 최소화하는 것을 의미한다. Rust는 LLVM 기반 컴파일러를 사용하여 제네릭, 트레이트, 패턴 매칭과 같은 고수준 기능을 제공하면서도 C/C++과 유사한 수준의 실행 성능을 유지할 수 있도록 설계되었다. 이러한 특징으로 인해 Rust는 시스템 프로그래밍 언어로서의 성능을 유지하면서도 안정성을 강화한 언어로 평가되고 있다.[5]

2.2 Rust의 개발환경과 생태계

Rust는 통합된 개발 환경 또한 제공한다. Rust 프로젝트에서는 Cargo라는 공식 빌드 및 패키지 관리 도구가 기본적으로 제공되며, 이를 통해 프로젝트 빌드, 의존성 관리, 테스트 실행, 문서 생성 등의 기능을 통합적으로 수행할 수 있다. Cargo는 Rust 개발 과정에서 핵심적인 역할을 수행하며, 개발 환경의 일관성과 재현성을 확보하는 데에 중요한 요소로 작용한다.[6]

Cargo는 Cargo.toml 파일을 통해 프로젝트의 의존성과 설정 정보를 관리하며, 필요한 외부 라이브러리의 경우 crates.io 저장소를 통해 자동으로 다운로드된다. 또한 Cargo.lock 파일을 통해 의존성 버전을 고정함으로써 동일한 빌드 환경을 유지할 수 있다. 이러한 구조는 대규모 프로젝트에서 발생할 수 있는 라이브러리 의존성 문제를 줄이고, 안정적인 빌드 환경을 유지할 수 있도록 한다.

또한 Rust는 표준 라이브러리를 통해 다양한 시스템 프로그래밍 기능을 제공한다. 파일 입출력, 네트워크 통신, 스레드 관리 등의 기능이 표준 라이브러리로 제공되며, 추가적인 기능은 외부 크레이트(crates)를 통해 확장할 수 있다. Rust의 패키지 생태계는 지속적으로 성장하고 있으며, 다양한 분야에서 활용 가능한 라이브러리가 제공되고 있다.

이러한 통합된 개발 환경은 시스템 소프트웨어 개발 과정에서 생산성과 유지보수성을 향상시키는 요소로 평가된다. 특히 의존성 관리와 빌드 환경의 재현성을 제공한다는 점에서 Rust 개발 환경은 기존 C/C++ 기반 개발환경과 차별화되는 특징을 갖는다.

2.3 Rust의 산업 및 시스템 분야 채택 동향

Rust는 언어 차원에서 메모리 안전성과 동시성 안전성을 제공하는 특성으로 인해 최근 다양한 산업 분야에서 활용이 확대되고 있다. 특히 운영체제, 클라우드 인프라, 네트워크 시스템과 같은 시스템 소프트웨어 분야에서 Rust의 채택이 증가하고 있으며, 기존 C/C++ 기반 개발 환경의 대안으로 주목받고 있다.[7]

대표적으로 Google은 Android 운영체제의 시스템 레벨 코드에 Rust를 도입하고 있으며, 메모리 안전성을 확보하기 위한 목적으로 그림 2와 같이 C++을 대체하여 Rust를 적용하고 있다.

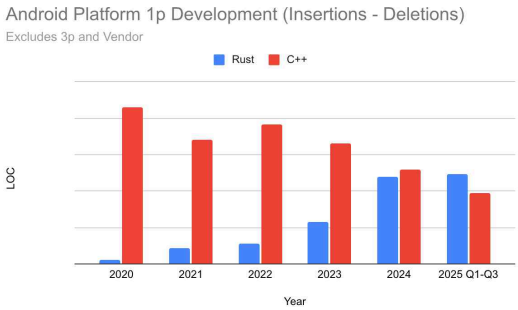

그림 2. Android 플랫폼 내 Rust와 C++ 코드 증가량 비교 [Google Security Blog]
Fig. 2. Comparison of rust and c++ code growth in the android platform [Google Security Blog]

Google의 보안 관련 보고에 따르면 Rust는 C++ 대비 취약점 발생 밀도가 1,000배 이상 낮은 것으로 추적되고 있으며, Rust 도입 이후 그림 3과 같이 메모리 관련 취약점이 최초로 20%까지 감소하는 효과가 나타난 것으로 보고되었다.

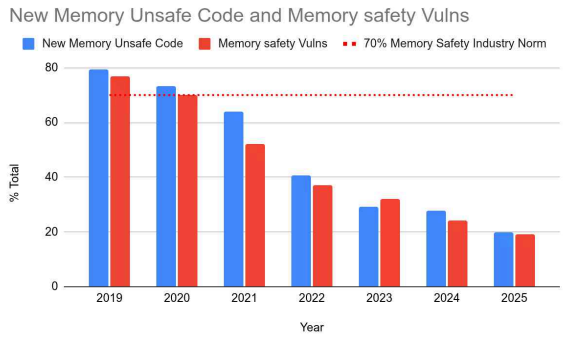

그림 3. Android 플랫폼의 메모리 안전 취약점 감소 추이 [Google Security Blog]
Fig. 3. Decline in memory safety vulnerabilities in the android platform [Google Security Blog]

또한 Linux Kernel은 6.x 버전부터 Rust를 공식적으로 지원하기 시작하였으며, 일부 커널 모듈 개발에 Rust를 적용하고 있다. 이는 기존 C 기반 커널 개발 환경에서 발생할 수 있는 메모리 안전성 문제를 완화하기 위한 시도로 보여지고 있다.

클라우드 인프라 분야에서도 Rust의 활용이 확대되고 있다. Amazon은 AWS 환경에서 경량 가상화 기술인 Firecracker를 Rust로 개발하였으며, 이를 통해 높은 성능과 안정성을 동시에 확보하였다. 이와 함께 Cloudflare와 같은 네트워크 서비스 기업에서도 Rust를 활용하여 고성능 네트워크 서비스를 구현하고 있다.

Rust는 산업적 활용뿐만 아니라 개발자 커뮤니티에서도 높은 관심을 받고 있다. Stack Overflow Developer Survey에 따르면 Rust는 여러 해 동안 “Most Loved Programming Language” 상위권을 유지하고 있으며, 이는 기존 사용자들이 지속적으로 Rust를 선호하고 있음을 의미한다. 이러한 결과는 Rust가 단순한 연구 단계 언어가 아니라 실제 개발 환경에서 높은 만족도를 보이는 언어임을 나타낸다.

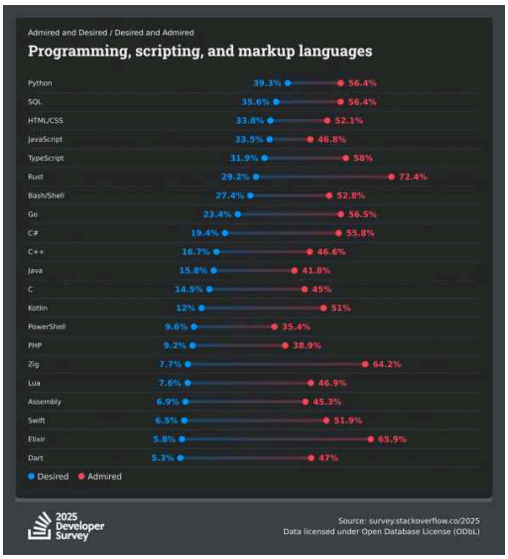

그림 4. 프로그래밍 언어별 개발자 선호도 및 사용 의향 [Stack Overflow Developer Survey]
Fig. 4. Developer admiration and desire by programming language [Stack Overflow Developer Survey]

이와 같이 Rust는 운영체제, 클라우드, 네트워크 등 다양한 시스템 소프트웨어 분야에서 실제로 활용되고 있으며, 산업적 채택과 개발자 선호도 측면에서 모두 긍정적인 흐름을 보이고 있다. 이러한 동향은 Rust가 기존 C/C++ 기반 시스템 프로그래밍 언어의 대안으로 활용될 수 있는 가능성을 보여준다.

2.4 Rust의 제한사항 및 고려사항

Rust는 메모리 안전성과 동시성 안전성을 컴파일 단계에서 강화할 수 있다는 장점이 있으나, 기존 C/C++ 기반 유도탄 소프트웨어를 즉시 대체하기에는 몇 가지 고려사항이 존재한다. 첫째, 기존 C/C++ 코드, 장비 드라이버, 외부 라이브러리와 연동하기 위해 FFI 적용이 필요할 수 있으며, 이 과정에서 unsafe 코드 사용이 불가피할 수 있다. 둘째, 소유권과 Lifetime 개념은 초기 학습 비용을 증가시킬 수 있다. 따라서 Rust 적용 시에는 기존 소프트웨어 자산과의 연동성, unsafe 코드 관리, 개발 환경 전환 비용을 고려한 단계적 도입 전략이 필요하다.

3. Rust VS C++ 성능 비교 실험 및 결과 분석

본 논문에서는 Rust와 C++의 성능 비교를 위해 Windows Desktop 환경과 Linux 기반 가상환경에서 동일한 조건으로 실험을 수행했다. Linux 환경의 경우 자원 사용을 제한하여 임베디드 시스템과 유사한 실행 환경을 구성했다. 시험 항목은

유도탄 소프트웨어에서 요구되는 고주기 통신, 대용량 파일 입출력(File I/O), 연산 알고리즘 처리를 대상으로 각 항목별 성능을 비교 분석했다.[8]

본 실험 항목은 유도탄 소프트웨어의 주요 기능 요구와 대응되도록 구성하였다. UDP 통신 실험은 유도탄 내부 장치 및 점검 장비 간 상태정보 송수신에서 요구되는 고주기 통신 특성을, File I/O 실험은 시험 데이터 및 운용 로그 저장 특성을, 알고리즘 실험은 CRC 검증, 비트 필드 기반 상태 해석, 행렬 연산과 같은 유도·항법·상태 데이터 처리 특성을 반영한다. 또한 메모리 안전성 실험은 제한된 운용 환경에서 런타임 오류를 사전에 방지할 수 있는지를 확인하기 위해 수행하였다.

표 1. 실험 환경
Table 1. Experimental environment

구분	Windows Desktop	Linux VM
운영체제(OS)	Windows 10 Pro	Ubuntu 64bit
실행 방식	Native	Virtual Machine
CPU	Intel Core i7-4790K @ 4.00GHz	4 vCPU
메모리(RAM)	32.0GB	2048MB
컴파일러	rustc, GCC	

3.1 UDP 통신 성능 비교

3.1.1 실험 방법

본 실험에서는 Rust와 C++의 고주기 통신 성능을 비교하기 위해 UDP 기반 송수신 실험을 수행했다. 송신 주기 800Hz, 패킷 크기 1024 bytes, 실험 시간 60초 조건에서 수행했으며, 동일 조건의 실험을 5회 반복하여 평균값과 표준편차를 산출하였다. 각 패킷은 Sequence Number와 송신 Timestamp를 포함하도록 구성하였다. Packet Loss는 누락된 Sequence Number를 기준으로 산출했으며, Latency는 송신 Timestamp와 수신 시각의 차이로 계산하였다. Jitter는 이상적인 수신 주기인 1.25ms와 실제 수신 간격의 차이를 기준으로 측정하였다.

그림 5. 패킷 구조
Fig. 5. Packet structure

3.1.2 실험 결과

그림 6, 그림 7은 800Hz UDP 송수신 환경에서 Rust와 C++의 UDP Jitter를 비교한 결과이다. 그림 6은 Windows 환경, 그림 7은 자원이 제한된 Linux VM 환경에서의 측정 결과이다. 실험 결과 두 환경 모두에서 Rust와 C++은 대부분의 패킷 구간에서 유사한 Jitter 분포를 보였다. 특히 평균 Jitter 값의 차이가 크지 않아, Rust가 C++과 비교하여 큰 성능 저하 없이 800Hz 수준의 고주기 UDP 통신을 안정적으로 수행할 수 있음을 확인하였다. 표준 편차 또한 두 언어 모두 1.25ms 대비 제한적인 수준으로 나타났다. 다만 Linux VM 환경에서는 일부 구간에서 순간적인 Jitter 증가가 나타났으며, 이는 언어 자체의 차이가 아닌 가상환경의 자원 제한과 운영체제 스케줄링 영향으로 판단된다. 또한 Packet Loss의 경우 두 언어 모두 없었으며, Latency도 Rust가 근소하게 작았으나 유사한 수준으로 측정되었다.

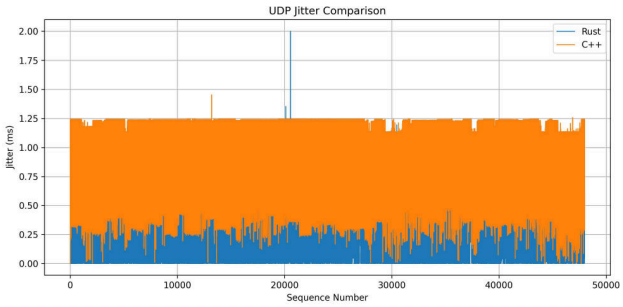

그림 6. Windows 환경에서의 UDP Jitter 비교
Fig. 6. UDP jitter comparison in windows environment

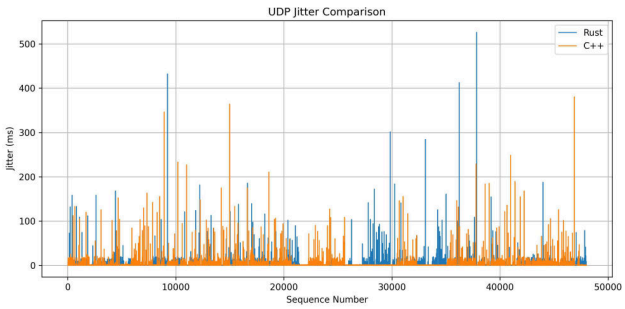

그림 7. Linux VM 환경에서의 UDP Jitter 비교
Fig. 7. UDP jitter comparison in linux vm environment

3.2 File I/O 성능 비교

3.2.1 실험 방법

File I/O 성능 비교 실험은 동일한 크기의 바이너리 파일을 대상으로 순차 쓰기 및 순차 읽기 작업을 수행하는 방식으로 구성했다. 각 언어는 1MB 크기의 버퍼를 사용하여 데이터를 반복적으로 기록하거나 읽었으며, 파일 크기는 500MB, 1000MB로 설정했다. 각 실험은 5회 반복 수행하였으며, 평균 수행 시간과 평균 처리량을 산출했다.

3.2.2 실험 결과

실험 결과 Rust와 C++은 순차적인 파일 쓰기 및 읽기 작업에서 유사한 수준의 처리량을 보였다. 이는 Rust가 시스템 소프트웨어에서 요구되는 대용량 파일 입출력 작업에서도 C++과 비교하여 큰 성능 저하 없이 동작할 수 있음을 의미한다.

표 2. File I/O 성능 비교 결과
Table 2. Experiments results of file I/O

실행 환경	파일 크기	작업	Rust 처리량 (MB/s)	C++ 처리량 (MB/s)
Windows	500MB	Write	2517.280	2578.521
		Read	2983.484	2929.925
	1000MB	Write	2594.083	2316.395
		Read	3335.925	3211.745
Linux VM	500MB	Write	687.112	737.489
		Read	1156.375	1153.887
	1000MB	Write	513.718	609.48
		Read	1138.101	1114.826

3.3 알고리즘 처리 성능 비교

3.3.1 실험 방법

알고리즘 처리 성능 비교 실험에서는 CRC 계산, 행렬 연산 등의 연산 알고리즘을 구현하여 처리 시간을 측정했다.
각 알고리즘은 동일한 입력 데이터에 대해 5회 반복 수행되었으며, 실행 시간을 측정하여 두 언어 간 성능 차이를 비교했다. 특히 반복 연산이 많은 환경에서의 처리 성능을 확인하기 위해 일정 횟수 이상의 반복 실행을 수행했다.

이러한 실험을 통해 Rust와 C++의 연산 처리 성능을 비교하고, 시스템 소프트웨어 환경에서의 적용 가능성을 평가했다.

3.3.2 실험 결과

알고리즘 처리 성능 비교 결과, Windows 환경에서는 Rust와 C++이 CRC32와 Matrix Multiplication에서 유사한 실행 시간을 보였으며, Bit-field Parsing에서는 Rust가 더 빠른 결과를 보였다. 이는 비트 연산 구현 방식, 반복문 최적화, 컴파일러의 인라인 처리 차이가 복합적으로 작용한 결과로 판단된다. Linux VM 환경에서는 CRC32에서 Rust가 C++보다 빠른 반면, Matrix Multiplication에서는 C++이 Rust보다 약 3.6배 빠른 결과를 보였다. Matrix Multiplication의 경우 메모리 접근 패턴, 배열 표현 방식, 컴파일 옵션, 표준 라이브러리 및 최적화 수준에 민감하므로, 해당 결과만으로 언어 자체의 우열을 단정하기는 어렵다. 따라서 본 실험 결과는 Rust가 모든 알고리즘에서 항상 우수하다는 의미가 아니라, 유도탄 소프트웨어에서 사용될 수 있는 일반적인 연산 처리 항목에서 C++과 비교 가능한 성능을 보였으며, 특정 연산에서는 구현 및 최적화 방식에 따라 성능 차이가 발생할 수 있음을 의미한다.[9]

표 3. 알고리즘 처리 성능 비교 결과

Table 3. Experiments results of execute algorithm

실행 환경	알고리즘	Rust (ms)	C++ (ms)
Windows	CRC32	7761.6792	7890.2545
	Bit-field Parsing	186.4193	631.3739
	Matrix Multiplication	21.6279	22.0308
Linux VM	CRC32	2140.8384	6671.6184
	Bit-field Parsing	216.9116	189.3600
	Matrix Multiplication	22.5755	6.2530

3.5 메모리 안전성 비교

3.5.1 실험 방법

메모리 안전성 비교의 경우 C++과 Rust의 메모리 관리 방식의 차이를 분석하기 위해 수행했다. 대표적인 메모리 오류 상황인 Use-after-free, Double free를 구현하여 각 언어에서의 동작을 비교했다.

C++ 환경에서는 이러한 오류가 런타임에서 발생할 수 있으며, 프로그램의 비정상 동작이나 시스템 오류를 유발할 수 있다.[10] 반면 Rust는 소유권(Ownership) 기반 메모리 관리 모델을 통해 이러한 오류를 컴파일 단계에서 검출하여 사전에 차단한다.

본 실험에서는 동일한 메모리 오류 상황을 구현하여 Rust에서 컴파일 오류로 검출되는지 확인하고, 이를 통해 Rust의 메모리 안전성 확보 매커니즘을 분석했다.

3.5.2 Use-after-free 실험

그림 8과 그림 9는 Use-after-free 오류 상황을 Rust와 C++에서 각각 실행한 결과이다. C++에서는 이미 해제된 메모리에 다시 접근하는 코드가 컴파일 및 실행되었으나, 실행할 때마다 서로 다른 값이 출력되었다. 이는 해제된 메모리 영역을 참조함으로써 정의되지 않은 동작(Undefined Behavior)이 발생했다는 것을 의미한다.

Rust에서는 동일한 Use-after-free 상황을 구현한 코드가 컴파일 단계에서 오류로 검출되었다. Rust는 소유권(Ownership) 규칙에 따라 drop 이후 해당 값의 사용을 허용하지 않으며, 이를 통해 해제된 메모리에 접근하는 오류를 실행 전에 차단한다.

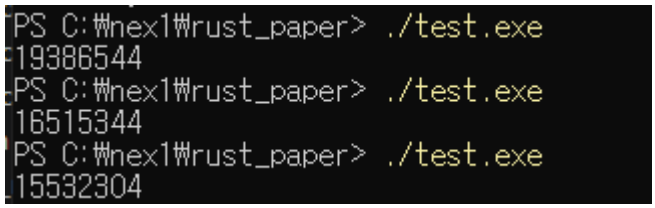

그림 8. C++에서의 use-after-free 실행 결과

Fig. 8. Execution result of use-after-free in c++

그림 9. Rust에서의 use-after-free 컴파일 결과

Fig. 9. Compilation result of use-after-free in rust

3.5.3 Double free 실험

그림 10과 그림 11은 Double free 오류 상황을 C++과 Rust에서 각각 실행한 결과이다. C++에서는 동일한 메모리 영역을 두 번 해제하는 코드가 컴파일되었으며, 실행 또한 가능하였다. 이 경우 출력값은 발생하지 않았으나, 동일 포인터에 대해 중복 해제가 수행되었으므로 정의되지 않은 동작(Undefined Behavior)이 발생할 가능성이 존재한다.

반면 Rust에서는 동일한 Double free 상황을 구현한 코드가 컴파일 단계에서 오류로 검출되었다. Rust에서는 drop 호

출 이후 해당 값의 소유권이 더 이상 유효하지 않으므로, 동일한 값을 다시 해제하려는 시도가 컴파일러에 의해 차단된다.

따라서 본 실험을 통해 C++에서는 Double free와 같은 메모리 오류가 컴파일 단계에서 완전히 차단되지 않을 수 있으나, Rust는 소유권 규칙을 통해 중복 해제를 사전에 방지할 수 있음을 확인하였다.

그림 10. C++에서의 Double free 실행 결과
Fig. 10. Runtime Result of Double free in C++

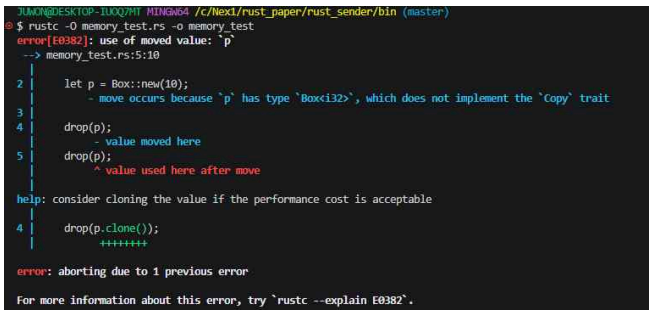

그림 11. Rust에서의 double free 컴파일 결과
Fig. 11. Compilation result of double free in rust

3.5.4 실험 결과

Use-after-free와 Double free 실험 결과, C++에서는 두 오류 상황 모두 컴파일이 가능했으며 실행 단계에서 정의되지 않은 동작이 발생할 가능성이 있음을 확인하였다. Use-after-free의 경우 해제된 메모리에 접근함으로써 실행할 때마다 서로 다른 값이 출력되었고, Double free의 경우 동일한 메모리 영역을 두 번 해제하는 코드가 컴파일 및 실행되었다.

반면 Rust에서는 두 오류 모두 컴파일 단계에서 차단되었다. Use-after-free는 해제 이후 값에 접근하려는 시도가 오류로 검출되었으며, Double free는 drop 이후 소유권이 사라진 값을 다시 해제하려는 시도가 컴파일 오류로 처리되었다. 이를 통해 Rust는 소유권 기반 메모리 관리 구조를 통해 대표적인 메모리 안전성 오류를 실행 전에 방지할 수 있음을 확인하였다.

4. 결 론

본 논문에서는 유도탄 소프트웨어 분야에서 Rust가 기존 C/C++ 기반 개발 방식의 대안으로 활용될 수 있다는 가능성을 분석하였다. Rust는 소유권(Ownership), Borrowing, Lifetime 기반의 메모리 관리 구조를 통해 가비지 컬렉터 없이도 메모리 안전성을 확보할 수 있으며, 동시성 오류를 컴파일 단계에서 검출할 수 있다는 장점을 가진다. 이로 인해 운영체제, 클라우드 인프라 등 다양한 산업 분야에서 채택되고 있으며, Cargo 기반의 통합 개발 환경을 제공하여 빌드, 의존성 관리, 테스트, 문서화 측면에서 개발 생산성과 유지보수성 또한 향상시킬 수 있다. 이러한 특성은 안정성과 장기 운용성이 중요한 유도탄 소프트웨어 환경에서 C/C++ 기반 개발 방식의 한계를 보완할 수 있는 요소로 판단된다.

Rust와 C++의 성능 비교 실험에서는 800Hz UDP 송수신, 파일 입출력(File I/O), 알고리즘 처리 성능을 중심으로 비교 분석을 수행하였다. 실험 결과 Rust는 C++과 유사한 수준의 통신 성능과 파일 처리 성능, 연산 처리 성능을 보였으며, 큰 성능 저하 없이 시스템 소프트웨어에 적용할 수 있음을 확인하였다. 특히 메모리 안전성 비교에서는 C++에서 런타임 오류 또는 정의되지 않은 동작으로 이어질 수 있는 오류를 Rust가 컴파일 단계에서 차단할 수 있음을 확인하였다. 따라서 Rust를 활용하면 C++과 유사한 성능을 유지하면서도 메모리 안전성을 강화한 시스템을 개발할 수 있을 것으로 기대된다.

향후 연구에서는 Rust를 기반으로 한 유도탄 시험 지원 소프트웨어를 개발하고, 실제 운용 환경에서의 성능, 안정성, 실시간성 및 기존 C/C++ 코드와의 연동성을 추가적으로 분석할 예정이다. 이를 통해 Rust가 유도탄 점검 장비뿐만 아니라 내부 임베디드 소프트웨어 영역에서도 신뢰성 향상과 유지보수성 개선에 기여할 수 있는지 검증하고자 한다.

References

[1] Sang Hoon Koh, Ho Jin Song, Nam Ho Keum, Pil-joong Yoo, Se Kwan Oh and Young-Sung Kim, “The algorithm design and the test bed construction method of processing for periodic delayed data”, Journal of Advanced Navigation Technology, pp. 102-110, 2023.

[2] Gavin Thomas, “A proactive approach to more secure code”, Microsoft Security Response Center Blog, July 16, 2019.

[3] William Bugden and Ayman Alahmar, “Rust: The Programming Language for Safety and Performance”, 2nd International Graduate Studies Congress (IGSCONG'22), pp. 1-9, June 8-11, 2022.

[4] Michal Sudwoj, “Rust Programming Language in the High-Performance Computing Environment”, Master’s Thesis, ETH Zurich, Zurich, September, 2020.

[5] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers and Derek Dreyer, “RustBelt: Securing the Foundations of the Rust Programming Language”, Proceedings of the ACM on Programming Languages, Vol. 2, No. POPL, Article 66, pp. 1-34, January 2018.

[6] Hao Li, Filipe R. Cogo and Cor-Paul Bezemer, “An Empirical Study of Yanked Releases in the Rust Package Registry”, IEEE Transactions on Software Engineering, pp. 348-363, January 2023.

[7] Mehmet Emre, Ryan Schroeder, Kyle Dewey and Ben Hardekopf, “Translating C to Safer Rust”, Proceedings of the ACM on Programming

Languages, pp. 1-29, October 2021.
[8] Woon-Sik Kim, Byung-Sun Lee and Sang-Ha Kim, “A study on the development of HWIL simulation control system for high maneuver guided missile system”, The Journal of Korean Institute of Communications and Information Sciences, pp. 1659-1666, 2010.
[9] N. Dashdamirli, “Analyzing the Performance and Practicality of C, Rust, Python, and Lua Programming Languages for Developing Microcontroller Applications”, 2025 6th International Conference on Problems of Cybernetics and Informatics (PCI), pp. 1-5, 2025.
[10] Juan Caballero, Gustavo Grieco, Mark Marron and Antonio Nappa, “Undangle: Early Detection of Dangling Pointers in Use-after-free and Double-free Vulnerabilities”, Proceedings of the 2012 International Symposium on Software Testing and Analysis, pp. 133-143, July 2012.