Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.33, NO.2, Apr. 2023 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2023.33.2.345

Rust와 C/C++간 안전한 상호작용에 관한 연구의 맹점과 개선 모델 연구*

노 태 현,[1†] 이 호 준[2‡]
[1,2]성균관대학교 (대학원생, 교수)

Limitations and Future Work Suggetion on Safe Interaction Model between Rust and C/C++*

Taehyun Noh,[1†] Hojoon Lee[2‡]
[1,2]Sungkyunkwan University (Graduate student, Professor)

요 약

소프트웨어 개발이 가속화되고 프로그램들이 기하급수적으로 복잡해짐에 따라 취약점을 줄이고, 관리하는 비용도 같이 증가하였다. 이러한 흐름에서, 기존의 C/C++ 와 같이 비교적 취약점을 내포하기 쉬운 언어를 대체하고 소프트웨어의 안정성을 높이기 위해서 제시된 것이 바로 Memory Safety를 보장하는 Rust 프로그래밍 언어이다. 하지만, 구식 언어들과의 호환성 및 개발의 편리함을 높이기 위해 C/C++로 작성된 라이브러리를 Rust에서도 사용할 수 있도록 지원하고 있는데, 이러한 다중 언어 환경에서는 Rust 또한 안전하지 않다. C/C++에서 발생한 메모리 오염이 Rust 내에서 Null-pointer 역참조, Use-After-Free 및 Buffer-overflow 문제 등을 발생시킬 수 있는 원인이 된다. 이를 해결하기 위해 여러 Rust-C/C++ 격리 연구가 진행되었으나, 아직 기초 단계이다. 본 논문에서는 선행 연구들을 분석하여 공통적으로 간과된 맹점들을 실제 코드 분석과 함께 소개하고, 이를 바탕으로 Rust와 C/C++간의 안전한 상호작용 모델 연구의 올바른 방향을 제시한다.

ABSTRACT

As software development progresses and programs become increasingly complex, the cost of reducing and managing software vulnerabilities has also increased. To address this issue, the Rust programming language, which guarantees Memory Safety, has been suggested as an alternative for more error-prone languages such as traditional C/C++. However, Rust also supports the use of libraries written in C/C++ to enhance compatibility with older languages and avoid redundant development, compromising its original guarantees. For example, memory corruption happened in C/C++ can lead to exploits such as buffer overflow, Use-After-Free and null-pointer dereferecing. To tackle this problem, recent studies have been conducted to secure interactino between Rust and C/C++ by isolation. This paper uncovers areas that have not been fully explored in previous studies, following limitation analysis on each. Finally, this paper suggests the future direction of research on safe interaction between Rust and C/C++.

Keywords: Rust, C/C++, Foreign Function Interface, Software Security, Memory Safety

Received(02. 23. 2023), Accepted(03. 05. 2023)
* 이 연구는 2023년도 정부의 재원으로 한국연구재단(교육부) 기초연구사업 (NRF-2022R1C1C1010494), 정보통신기획평가원(융합보안핵심인재양성 (No. 2019-0-01343), 국제공동연구사업 (No. 2020-0-00666))의 지원을 받아 수행된 연구임.
† 주저자, dove0255@skku.edu
‡ 교신저자, hojoon.lee@skku.edu(Corresponding author)

I. 서 론

소프트웨어 개발의 역사에서 C, C++, Java 와 같은 오래된 언어들을 Rust[1], Go[2], Kotlin[3]과 같은 새로운 언어들로 대체하려는 시도들이 계속되어 왔다. 그 중 Rust는 안정성을 목표로 하는 새로운 시스템 프로그래밍 언어이다. 기존에 시스템 소프트웨어 작성에 사용되던 언어는 C와 C++인데, 다른 언어에 비해 추상화의 정도가 가볍다는 특징 덕분에 성능이 중시되는 환경에서 우선적으로 선택되었다. 하지만, 저수준의 추상화는 곧 개발자의 부담을 가중시켰고, 이는 메모리 관리 미흡으로 인한 보안 문제가 지속적으로 발생하는 배경이 되었다. Rust 언어는 앞선 문제들을 보안 기능 추상화와 컴파일러의 역량 강화를 통하여 안정적이고 지속적인 소프트웨어 개발을 돕고자 개발되었다.

이미 개발자들은 Rust를 각종 프로젝트에 활용하기 시작하고 있다. Amazon은 AWS(Amazon Web Services)에서 S3, EC2 등의 제품들에 이미 Rust를 사용하고 있으며[4], VoIP(Voice Over Internet Protocol) 프로그램 Discord도 Go에서 Rust로 주 개발 언어를 바꿨다[5]. 오픈 소스인 리눅스도 6.1버전부터 Rust를 내부에 포함하고 있다[6]. 한편, Rust의 안정성이 학문적으로 증명[7]되면서 Rust는 안전한 프로그램을 효율적으로 개발할 수 있는 확실한 대안이 되었다.

하지만 이러한 이점에도 불구하고 새로운 언어는 현실적으로 기존의 언어를 한 번에 대체할 수는 없는데, 이미 C/C++로 개발되어 성숙한 라이브러리를 Rust로 재작성하는 작업이 쉽지 않기 때문이다. 그렇기에 Rust는 그러한 라이브러리들을 활용할 수 있도록 타 언어와의 유기적인 상호 작용을 지원하는 기능을 내장하고 있는데, 이것이 바로 외부 함수 인터페이스(foreign function interface)이다. 이 기능은 타 언어로 작성된 함수 및 자료구조를 Rust가 이해할 수 있는 방식으로 재정의하여 그러한 함수들을 Rust에서 호출 가능하게 한다. 대표적으로 Rust로 작성된 Servo 브라우저[8]가 C/C++로 작성된 SpiderMonkey JavaScript Engine[9]을 외부 함수 인터페이스를 통해 사용하고 있다.

불행하게도, 이런 다중 언어 환경은 Rust가 기존에 목표로 했던 보안 모델을 깨뜨린다는 연구결과가 보고되었다[10, 11]. Rust의 보안 기능들은 Rust 단독으로 사용되는 모델을 기반으로 고안되었기 때문에, C/C++를 대상으로 알려진 공격들이 여전히 Rust-C/C++가 혼합된 환경에서도 행해질 수 있다는 것이다. 이를 보완하기 위한 제시된 기술들은 하드웨어 기술을 활용해 언어마다 사용하는 메모리 공간을 분리하거나[12, 13], 또 운영체제를 이용해 언어 간의 활동 영역을 분리하는 기술[14, 15] 등이 있다. 하지만, 상호작용 방식을 제한하거나, 사용자의 부담이 늘어나는 등 한계점 또한 존재한다.

2장에서는 Rust-C/C++ 격리 모델을 연구하는 배경에 대해 설명한다. 3장에서 최신 Rust-C/C++ 격리 기술들을 기반 격리 기술별로 비교 분석하고, 각각의 한계점에 대해서 설명한다. 4장에서는 마지막으로 선행 연구들에서 충분히 다뤄지지 않은 Rust-C/C++ 상호작용 모델 속 맹점들을 실제 오픈 소스 예제와 함께 설명하고, 더 나아가 해당 분야가 나아가야 할 방향을 제시한다.

II. 배경 및 위협 모델

Rust는 강력한 정적 타입 체계와 고유한 소유권(ownership) 및 수명(lifetime) 개념을 구문 안에 내장하고 있는데, 이들에 대한 컴파일러의 까다로운 정적 검사를 통해 Memory Safety를 보장한다. 소유권 개념에 의해 할당된 모든 메모리는 소유자가 항상 존재하며, 소유되고 있지 않은 메모리는 자동으로 할당 해제된다. 하지만 컴파일러가 할당 해제 시점을 항상 추측할 수 있는 것은 아닌데, 이런 결정할 수 없는(undecidable) 상황을 없애주는 것이 수명 개념이다. 이렇듯 Rust는 독특한 방식으로 메모리를 관리한다.

한편, Rust는 타 언어로 작성된 라이브러리를 활용할 수 있는 기능을 제공하는데, 그것이 외부 함수 인터페이스(foreign function interface)이다. 문제는 Rust의 보안 기능들은 외부 함수의 안전성을 보장하지 않는 점에서 기인하며, 이는 S. Mergendahl 등에 의해 교차 언어 공격(cross-language attack) 모델으로 정립된 바 있다[11].

본 논문의 위협 모델은 C/C++ 라이브러리에 적용할 수 있는 보안 기술과 무관하게, 그림 1과 같이 값을 공유하고 외부 함수를 호출하는 것으로부터 시작되는 모든 공격 가능성을 열어두고 있다. 각 공격은 공유되지 않은 Rust의 데이터에 접근하거나, 임의의 Rust 함수를 실행함으로서 목적을 달성한다.

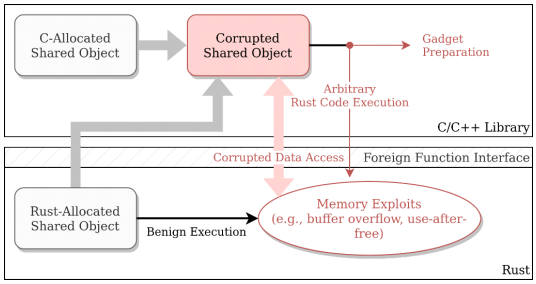

Fig. 1. Attack Scenarios between Rust and C/C++

이러한 공격들로부터 Rust의 Memory Safety를 지켜내는 것이 다음 문단에 서술된 보안 기법들의 목표이다.

III. 기반 기술 유형에 따른 Rust-C/C++ 격리 기술 비교 분석 및 평가

3.1 하드웨어 기반 격리 기술

PKRU-Safe[12]는 Rust의 힙 메모리 영역을 외부 언어들로부터 보호하는 기술이다. 하드웨어 기술인 Intel의 MPK/PKU(Memory Protection Keys for Userspace)[16]를 활용하여 Rust 와 C/C++ 영역의 접근 권한을 조작한다. 동적 분석/프로파일링 방법을 활용하여 공유 데이터를 수집하고, 최종 빌드에서 공유 데이터가 C/C++ 영역을 담당하는 메모리 할당자를 통해 할당 되도록 교체한다.

PKRU-Safe는 MPK 고유의 한계점[17]을 그대로 계승하는 것 외에도 힙 메모리 영역만을 보호한다는 한계가 있다. 즉, 스택 영역과 전역/정적 변수는 MPK의 보호 없이 C/C++ 영역에서 접근이 가능하다. 분명 PKRU-Safe의 핵심인 동적 프로파일링 기술은 지역 변수여도 공유 여부를 판단할 수 있겠지만, MPK 권한은 페이지 테이블 단위로 설정할 수 있는 기술이기 때문에 스택을 언어별로 따로 두는 등의 수정이 필요하다.

동적 프로파일링의 한계 또한 존재한다. 저자들은 메모리 객체 구분에 정적 분석을 활용하는 것은 건전성(soundness)이 문제가 되어 보호되어야 할 객체들이 위험에 노출이 된다고 하였다. 반대로 현재의 동적 분석법은 완전성(completeness)에 있어서 문제가 될 수 있으나, 그것이 좀 더 안전한 방법이라고 주장하였다. 하지만 동적 프로파일링의 완전성은 수집 단계에서 입력값에 의해 그 결과가 크게 좌우되며, 또 버전이 달라질 때마다 새로 프로파일링을 거쳐야한다는 단점이 존재한다. 이는 Code Coverage 문제와도 연결되는 부분인데, 프로그램이 커질수록 입력값이 늘어나고, 또 더 오랜 시간 프로파일링 단계를 거쳐야하는 확장성의 한계를 가진다.

Galeed[13]는 PKRU-Safe와 마찬가지로 MPK를 통하여 Rust의 힙 메모리를 다른 언어로부터 격리한다. 차이점은 PKRU-Safe의 메모리 자체를 공유하는 것보다 참조의 형태로 공유하는 방식을 선택했으며, 기존의 포인터를 대체할 유사 포인터(pseudo pointer)를 제안했다. 공유되는 Rust 내부의 포인터는 C/C++에서는 고유 ID의 형태로만 존재하며, 따라서 포인터를 인자로 가지는 함수의 선언부 또한 ID를 인자로 가지도록 변경된다. 외부 함수에서 Rust 객체를 접근하기 위해서는 포인터 역참조가 아닌 Rust 함수 호출을 하게 된다. 항상 Rust가 메모리 접근을 안전하게 대행하도록 하는 전략이다.

Galeed 고유의 한계점 중 하나는 포인터 분석 기술에 기인한다. 외부 함수로 넘겨진 포인터는 지역 변수 및 정적/전역 변수에 저장되고 나중에 다시 사용될 수 있다. 곧 포인터 Alias를 만들게 되는데, 함수 인자와 함께 alias 접근 코드 또한 역참조가 아닌 함수 호출로 수정되어야 한다. 따라서 포인터 Alias 분석의 정확도가 높아야 ID에 역참조가 일어나 오류가 발생하는 상황을 줄일 수 있다. 하지만, 포인터 Alias 분석은 결정 불가능하다고 알려져 있다[18]. 따라서 오직 한정적 상황에서만 완벽한 Alias 분석이 가능하기에 이는 기술의 안정성을 해치는 주요 원인이 된다.

3.2 운영체제 기반 격리 기술

SandCrust[14]는 운영 체제가 제공하는 프로세스 단위 격리 기능을 활용하여, Rust 코드와 C/C++ 코드를 각기 다른 프로세스에 할당하여 실행시키는 기술이다. Rust 프로세스와 C/C++ 프로세스는 RPC(Remote Procedure Call)를 통해서 상호작용하며, 프로세스 간 전달되는 함수 인자는 값을 직렬화하여 Unix pipe를 통해 전달한다.

위 기술은 운영체제의 강력한 프로세스 격리에 기반하나, 외부 함수의 활용도를 저하시킨다. 그 예로, 두 프로세스는 별도의 주소 공간을 사용하기 때문에

RPC로는 주소 값을 전달하는 것은 적합하지 않다. 또, 콜백 함수의 기능을 프로세스 간에 지원하지 못하는 등 프로그래밍 방식에 제약이 있다. 이런 제한적인 프로그래밍 모델을 극복하기 위해서는 사용 라이브러리의 API를 변형시켜야 하는데, 사용자의 입장에서 현실적이지 않은 방법이다. 마지막으로 두 언어 사이를 이동할 때 운영 체제를 통해서 문맥 교환(context switching)이 일어나는데, 이는 외부 함수를 빈번하게 부를수록 성능을 크게 저하시키는 원인이 된다.

Fidelius Charm[15] 또한 운영 체제의 기능을 활용하고 있는데, Memory Page 접근 권한을 조작하여 위험한 언어로부터 Rust 데이터에 대한 접근을 막는 방식이다. 접근 권한은 읽기/쓰기 허용, 읽기 전용, 접근 불가 순으로 제약을 강화할 수 있다.

Sandcrust에서는 RPC가 언어 간 상호작용의 경계선이 되었지만, Fidelius Charm에선 개발자가 직접 경계를 명시해야 한다. 개발자는 외부 함수 호출의 전후에 커널 모듈이 제공하는 페이지 권한 조작 함수를 추가한다. 예를 들어, 특정 변수를 외부 라이브러리에서 접근하지 못하게 하기 위해서는 해당 변수가 포함된 메모리 페이지 권한을 커널이 수정하도록 모듈을 통해 요청해야한다. 외부 호출이 끝나고 본래의 Rust 문맥으로 돌아오면 개발자의 의도에 따라 다시 이전의 페이지 접근 권한을 복구하게 된다. 이는 이론적으로 메모리 위치에 관계없이 적용할 수 있는 기술이나, 개발자가 직접 보호할 데이터와 해당 데이터를 다루는 코드를 찾아 적용(On-demand)해야 하기 때문에 활용성 측면에서 개선되어야 할 부분이 있다.

Table 1. Comparison of Security Characteristics on Recent Rust-C/C++ Protection Models

	PKRU-Safe	Galeed	Sandcrust	Fidelius Charm
Supported Memory Types	Heap	Heap	All	All, On-demand
Protection Unit	Page	Page	Process	Page
Pointer as Parameter	O	O	X	O
OS Feature	X	X	O	O
HW Dependency	O	O	X	X

IV. 코드 분석을 통한 기존 연구들의 맹점 및 향후 연구 방향 제시

기존 연구들은 공통적으로 Rust가 C/C++ 라이브러리를 사용할 때 발생하는 문제점들이 서로 다른 언어가 하나의 가상 메모리 공간을 공유하기 때문에 일어나는 현상이라고 주장한다. 이에 반해, 안전한 외부 함수 인터페이스의 중요성은 비교적 과소평가되고 있다. 그 결과 특정 데이터 타입은 외부 함수 호출로 전달할 수 없게 되거나, 외부 라이브러리 코드까지 사용자가 수정해야 했다.

다음 문단에서는 Servo 오픈 소스 브라우저의 코드와 함께 과소평가된 Rust-C/C++ 상호작용 패턴에는 어떤 것들이 있고, 안전한 외부 함수 인터페이스를 만드는데 어떤 영향을 미치는지 알아본다.

4.1 전역 상태(global state)의 존재

전역 상태 자료형은 그림 2의 Evaluate 함수의 첫 번째 인자로 넘겨지는데, 이 방식은 외부 라이브러리를 사용할 때 빈번하게 관찰된다. 이런 전역 상태의 특징은 저장되는 정보가 그 형태와 사용법에 있어서 공통점을 찾기 어렵고, 생성과 소멸에 있어서도 규칙이 존재하지 않는다는 것이다. 예를 들어, 상태의 일부는 한번 저장되면 절대 수정되지 않거나, 데이터 주소가 아닌 코드의 주소(함수 포인터)를 저장하거나, 또는 용도상 일부는 의존성을 서로 가지는 등 다양한 양상을 보인다. 따라서 그러한 내부의 규칙 또는 특성을 언어 경계를 넘어서도 지킬 수 있는 수단이 필요하다.

예를 들어, 반투명 구조체(opaque structure)는 두 언어 중 나머지 한 쪽에서는 절대 내부에 접근하지 않는 구조체를 의미하는데, 이를 외부 함수 인터페이스와 결합, 확장하여 전역 상태처럼 복합적인 구조체의 요소들에 대해서 각각 접근 방식을 서술하

```
extern "C" {
    pub fn Evaluate(
        cx: *mut root::JSContext,
        options: *const root::JS::ReadOnlyCompileOptions,
        srcBuf: *mut root::JS::SourceText<u16>,
        rval: root::JS::MutableHandle<root::JS::Value>,
    ) -> bool;
}
```

Fig. 2. A FFI definition of "Evaluate" function, which is one of SpiderMonkey API.

고 각 요구사항을 컴파일러 에러를 통해서 통제하는 방식을 고안할 수 있다.

4.2 Rust Semantic의 단절

Rust는 할당 영역 외 접근과 같은 메모리 취약점을 없애기 위해 모든 동적 크기의 자료형에 대해서 그 크기 정보를 같이 보관하고 있다. 그림 3의 6번째 줄처럼 Rust의 고유 동적 자료형(Vector)들은 C/C++가 이해할 수 있는 저수준의 자료형(u16)으로 전달되어 일련의 바이트 배열로 취급되곤 한다. 이런 방식으로 상호작용 할 때, 동적 자료형의 포인터와 같이 저장되는 할당 크기 값을 보호하지 않으면 외부 함수의 실행이 끝난 후 Rust 내에서 할당 영역 외 접근이 발생하는 원인이 될 수 있다. 즉, Rust의 실행에 있어서 판단 기준이 되는 정보들은 C/C++가 수정할 수 없어야 하며, Rust 언어 체계가 이런 중요 정보를 신뢰할 수 있도록 보호 체계가 필요하다.

수명과 불변성 또한 상실되는 Rust semantic인데, 두 언어가 공유하는 데이터들은 생성 및 소멸 위치가 확정되어 있지 않다. 곧 C/C++로부터 받은 값이 이미 할당 해제 되어있는 등의 문제가 생길 수 있는데, 이를 공유 데이터의 무결성의 관점에서 해석할 수도 있으나 수명에 대한 확실성과 더불어 최소한 외부 언어가 읽기만 하는 공유 데이터는 그 무결성이 보장되어야 한다. 예를 들어, 공유 데이터의 생성과 소멸 권한을 Rust 내부에서만 가능하게 하거나, 가상의 교차 소유권 개념으로 메모리의 책임 소재를 명확히 하여 Rust로 하여금 그 정보들을 활용할 수 있게 하여야 한다.

```
1  let mut json_text = Vec::with_capacity(capacity);
2  // manipulate json_text
3  unsafe {
4      if !JS_ParseJSON(
5          *cx,
6          json_text.as_ptr(), /* *const u16 */
7          json_text.len() as u32,
8          rval.handle_mut(),
9      ) {
10         JS_ClearPendingException(*cx);
11         return NULLValue();
12     }
13 }
14
```

Fig. 3. Simplified examples that pass a pointer of dynamically sized types from Servo(line 6). JS_* functions are FFI functions.

4.3 콜백(Callback) 함수

그림 4의 Rust 코드에 정의된 JSSecurity Callbacks 구조체는 Rust의 함수포인터를 포함하고 있는 자료형이다. 해당 구조체는 C/C++ 외부 함수 JS_SetSecurityCallbacks의 인자로 넘겨지는데, 이를 통해 Rust의 함수포인터가 JSContext라는 C/C++의 전역 상태안에 저장된다. 이렇게 저장된 Rust 함수는 나중에 C/C++에 의해 불릴 수 있는데, 안전하지 않은 문맥(C/C++)에서 호출된 안전한 언어로 작성된 함수는 그 실행의 안전성을 보장할 수 없다. 이런 방식의 상호작용을 자주 찾아볼 수 있는데, 격리 기반 기술에서는 콜백 함수의 존재가 까다로울 수밖에 없다. Rust의 입장에서 함수 포인터를 받는다면 해당 주소가 임의의 공격자가 유도하는 주소인지 판단할 수 없고, 또 함수 포인터를 인자로 넘겨주는 경우에도 현재 줬던 함수가 올바른 실행 코드에서 호출될 것인지도 짐작할 수 없다. 게다가 메모리 접근 권한을 통제하는 방식에서 콜백 함수들은 공격자의 관점에서 접근 권한이 없는 데이터에 접근을 가능케 하는 Confused Deputy 문제에 취약하다. 이는 다중 언어 환경에서 외부 함수 인터페이스에 선언된 함수 signature를 C/C++ 의 signature와 관계 지을 수 있는 컴파일러 상의 CFI(Control-Flow Integrity) 기술을 통해 공격 가능성을 낮출 수는 있으나, 완벽한 방법이라 보기 어렵다. 유연하면서도 안전한 Rust-C++ 프로그래밍 모델을 지원하려면, C/C++ 영역에서 호출되어도 안전한 함수는 무엇인지, 호출된 Rust 함수를 어떤 원칙에 따라 통제할 것인지 반드시 고려해야 한다.

```
1  pub type JSSubsumesOp = ::std::option::Option<
2      unsafe extern "C" fn(
3          first: *mut root::JSPrincipals,
4          second: *mut root::JSPrincipals
5      ) -> bool,
6  >;
7  pub type JSCSPEvalChecker = ::std::option::Option<
8      unsafe extern "C" fn(
9          cx: *mut root::JSContext,
10         kind: root::JS::RuntimeCode,
11         code: root::JS::HandleString,
12     ) -> bool,
13 >;
14 #[repr(C)]
15 #[derive(Debug, Copy, Clone)]
16 pub struct JSSecurityCallbacks {
17     pub contentSecurityPolicyAllows: root::JSCSPEvalChecker,
18     pub subsumes: root::JSSubsumesOp,
19 }
```

Fig. 4. Rust-defined callbacks that are stored in Servo's FFI types.

V. 결 론

본 논문에서는 Rust와 C/C++의 상호작용 모델의 안정성을 향상시키는 연구들을 비교 분석하고 그 한계점을 설명했다. 그리고 제시된 해결책들은 외부 함수 인터페이스의 중요성에 대해서 고려하지 않아, 프로그래밍 및 상호작용 모델의 다양성이 저하되는 결과를 가져왔다. 그렇다고 해서 기존의 C/C++ 취약점 연구의 성과를 해당 문제에 그대로 적용한다면, 그것 또한 메모리 취약점의 근본적인 원인을 새로운 보안 개념 추상화를 통해 제거하려 했던 Rust의 개발 의도와 무관한 방향으로 가게 된다. 따라서 효과적으로 Rust의 독특한 기능들을 살리면서 안전한 외부 상호작용을 하기 위해서는 전역 상태와 그에 종속된 데이터들의 연관성이 안전성을 해칠 수 있는 수단이 되지는 않는지, C/C++에서 사용되는 불안전한 데이터를 어떻게 Rust의 수명 및 불변성 개념과 이을 것인지, 새로운 격리 프레임워크에서는 콜백 함수들에 대해 어떤 원칙을 적용할 것인지 등을 하나씩 짚어보아야 다한. 앞서 말한 외부 함수 인터페이스와 상호작용 모델을 꼼꼼히 고려한다면 Rust-C/C++ 문제를 해결하는 유기적인 방안이 나올 수 있다.

References

[1] N. D. Matsakis and F. S. Klock, "The rust language," ACM SIGAdaAda Letters, vol. 34, no. 3, pp. 103 - 104, Dec. 2014.

[2] J. Meyerson, "The go programming language," IEEE software, vol. 31, no. 5, pp. 104 - 104, Sept-Oct. 2014

[3] Kotlin Language Documentation, https://kotlinlang.org/docs/kotlin-pdf.html, 02. 12. 2023,

[4] "Sustainability with Rust | AWS Open Source Blog", . https://aws.amazon.com/blogs/opensource/sustainability-with-rust/, 02. 11. 2023

[5] Discord Inc., Discord, https://discord.com, 02. 11. 2023

[6] "[GIT PULL] Rust Introduction for v6.1-Rc1 - Kees Cook.", https://lore.kernel.org/lkml/202210010816.1317F2C@keescook/, 02. 20. 2023

[7] Jung, Ralf, et al. "RustBelt: Securing the foundations of the Rust programming language," Proceedings of the ACM on Programming Languages2, POPL, pp. 1-34. Jan. 2018.

[8] Servo, https://github.com/servo. 02. 20. 2023

[9] Mozilla Spidermonkey JavaScript Engine, https://spidermonkey.dev/, 02. 20. 2023

[10] M. Papaevripides and E. Athanasopoulos, "Exploiting mixed binaries," ACM Transactions on Privacy and Security (TOPS), vol. 24, no. 2, pp. 1 - 29, May. 2021.

[11] Mergendahl, Samuel, Nathan Burow, and Hamed Okhravi, "Cross-language attacks," Proceedings 2022 Network and Distributed System Security Symposium. NDSS, Vol. 22, pp. 1-17, 2022.

[12] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz, "PKRU-safe: automatically locking down the heap between safe and unsafe languages. In Proceedings of the Seventeenth European Conference on Computer Systems," Association for Computing Machinery, New York, NY, USA, pp. 132 - 148, 2022.

[13] Rivera, Elijah, et al, "Keeping safe rust safe with galeed," Annual Computer Security Applications Conference, pp. 824-836, 2021.

[14] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. "Sandcrust: Automatic Sandboxing of Unsafe Components in Rust. In Proceedings

of the 9th Workshop on Programming Languages and Operating Systems," Association for Computing Machinery, New York, NY, USA, pp. 51 - 57, 2017.
[15] Hussain M. J. Almohri and David Evans, "Fidelius Charm: Isolating Unsafe Rust Code," Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy (CODASPY '18), Association for Computing Machinery, New York, NY, USA, pp. 248 - 255, 2018.
[16] Jonathan Corbet. 2015. Intel Memory Protection Keys. https://lwn.net/Articles/643797/. 02. 20. 2023
[17] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard, "PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems," 29th USENIX Security Symposium (USENIX Security 20), USENIX Association, pp. 1409 - 1426, 2020.
[18] Landi, William, and Barbara G. Ryder, "Pointer-induced aliasing: A problem classification," Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 93-103, 1991

〈저자소개〉

노 태 현 (Taehyun Noh) 학생회원
2021년 8월: 성균관대학교 소프트웨어학과 졸업
2021년 9월~현재: 성균관대학교 소프트웨어학과 석사과정
〈관심분야〉 정보보호, 시스템보안

이 호 준 (Hojoon Lee) 정회원
2010년 12월: The University of Texas at Austin 졸업
2013년 8월: KAIST 석사
2018년 2월: KAIST 박사
2019년 11월~현재: 성균관대학교 소프트웨어대학 교수
〈관심분야〉 정보보호, 프로그램 분석, 소프트웨어보안, 시스템보안, TEE, 클라우드 AI보안