Temporal Memory Safety 보장을 위한 Rust 프로그램 보호 기법

양수민*, 진홍주*, 이동훈*

*고려대학교

*uint24@korea.ac.kr, *realredwine@korea.ac.kr, *donghlee@korea.ac.kr

Protection techniques guaranteeing temporal memory safety for Rust Programming

Sumin Yang*, Hongjoo Jin*, Dong Hoon Lee*

*Korea Univ.

요 약

안전하고 효율적인 Rust 언어의 등장으로 인하여 기존의 C/C++ 코드와 Rust 코드가 통합된 다언어 응용프로그램이 많아지고 있다. 그러나, 공격자는 다언어 응용프로그램을 대상으로 하는 Cross-Language Attacks (CLA) 기법을 이용하여 temporal memory corruption을 수행할 수 있고, 따라서 임의 코드 실행이 가능해진다. 본 논문에서는 CLA 기반의 temporal memory corruption에 대응하기 위하여 Rust 및 C/C++로 작성된 다언어 응용프로그램을 대상으로 설계된 프레임워크를 제안한다. 제안하는 기법은 자체적인 정적 분석을 통해 댕글링 포인터가 될 수 있는 후보만을 선별하고, 빠른 시간 내에 식별한 포인터에 대한 런타임 검증를 수행하여 프로그램의 temporal memory safety를 보장한다.

Ⅰ. 서론

Rust 언어는 새롭게 개발된 안전하고 빠른 시스템 프로그래밍 언어[1]로, 많은 프로그램의 C/C++ 코드가 메모리 안전성과 빠른 성능을 제공하는 Rust 코드로 대체되고 있다. 하지만 C/C++로 작성된 모든 코드를 Rust로 대체하는 것은 비효율적이며 오랜 시간이 걸리기 때문에, 기존의 C/C++ 코드와 통합하여 Rust 코드를 작성하는 추세이다. 따라서 Rust 프로그램은 단일 언어가 아닌, 여러 언어로 구성된 다언어 응용 프로그램(Multi Language Application, MLA)으로 존재하는 경우가 많다[3].

Cross-Language Attacks(CLA)[2]은 MLA에서 가능한 새로운 공격 방법으로, 각각의 언어에 적용된 메모리 보호 기법들을 우회하는 공격이다. 공격자는 CLA를 통해 기존 C/C++의 Use-After-Free(UAF)나 Double-Free(DF)와 같은 메모리 취약점을 활용하여 Rust와 C/C++로 구성된 프로그램에 대한 공격이 가능하다[4]. CLA가 가능한 이유는 MLA를 구성하는 각각의 언어들에 적용되는 메모리 안전하다고 여겨지는 조건들이 일관되지 않기 때문이다.

본 논문은 CLA를 활용한 UAF나 DF기반 공격을 방어하기 위한 프레임워크를 제안한다. 해당 기법은 Rust 프로그램에서 댕글링 포인터(dangling pointer)의 사용을 방지함으로써 temporal memory safety를 제공한다.

Ⅱ. 배경지식

Rust는 소유권, 생명주기, 대여 등 새로운 개념을 도입하여 메모리 안전성을 보장하며[5], 프로그래머들은 이러한 규칙들에 맞춰 코드를 작성해야 한다. 하지만, 이 규칙들은 Rust만의 개념이므로, MLA에 존재하는 C/C++와 같은 다른 언어에는 해당 규칙들을 적용할 수 없다. 따라서, Rust는 *unsafe* 라는 키워드를 제공하며, unsafe 키워드 블록 안에서는 기존의 안전한 Rust에서 불가능한 기능(e.g., 로우포인터 역참조)을 수행할 수 있다. 프로그래머는 unsafe block 안에서 다른 언어로 작성된 *안전하지 않은 함수*를 호출할 수 있으며, 외부 함수 인터페이스(Foreign Function Interface, FFI)를 이용하여 Rust와 다른 언어 간 데이터를 주고받을 수 있다.

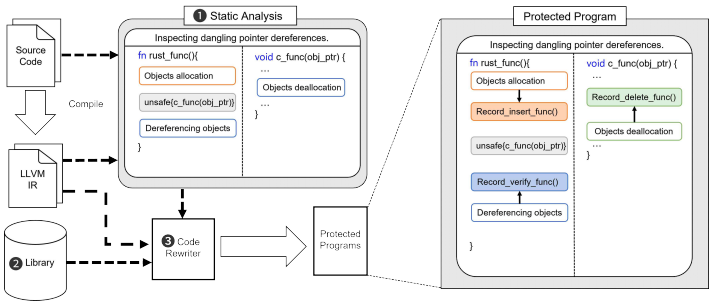

Figure 1: 제안하는 프레임워크의 구조 및 동작 과정.

Rust는 자체적인 규칙을 활용하여 가비지 컬렉터없이 객체 및 메모리를 관리한다. 하지만 FFI를 이용하여 객체를 C/C++ 코드로 전달할 경우, 해당 객체에 대한 메모리 안전성을 보장할 수 없게 된다[3]. 예를 들어, Rust 코드에서 할당된 객체가 C/C++ 코드로 전달된 후 해당 객체가 C/C++ 코드에서 해제될 수 있다. 하지만 Rust는 해당 객체가 해제되었음을 인지하지 못하며, 이후 Rust 코드에서 해당 객체에 접근할 수 있다. 이 경우 댕글링 포인터를 사용하게 되어 UAF나 DF와 같은 취약점이 발생할 수 있다.

Ⅲ. 설계

본 논문에서 제안하는 프레임워크는 Rust와 C/C++언어로 작성된 MLA에 대해 temporal memory safety를 보장한다. 제안하는 기법의 구조는 (1)정적 분석기, (2) 동적 라이브러리, (3) 코드 변환기로 구성되어 있으며, Figure 1은 전체 시스템의 구조 및 동작 과정을 보여준다.

정적 분석기는 Rust 코드와 LLVM IR 파일을 이용하여 unsafe block과 FFI 유무를 판단한다. 만약 FFI가 존재한다면, Rust 코드에서 (1) unsafe block 이전에 메모리 할당 지점, (2) unsafe block 이후의 메모리 역참조 지점과, C 코드에서 (3)메모리 해제 지점을 식별한다. Unsafe block 이후에 메모리 할당되거나, unsafe block 이전에 메모리 역참조 지점은 Rust의 메

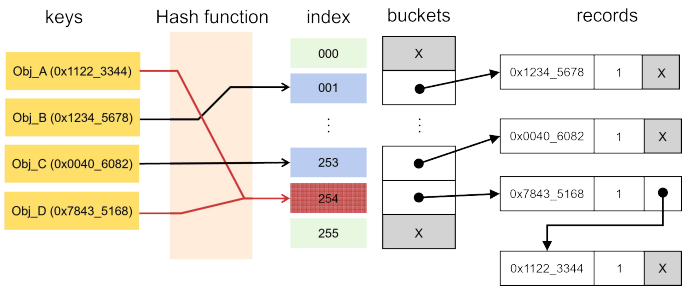

Figure 2: 제안하는 프레임워크에서 해시 테이블을 활용하는 예시.

모리 안전성에 의해 보호되므로 식별하지 않아도 된다.

제안하는 프레임워크는 메모리 관리를 위해 해시 테이블(hash table)을 사용하며, 라이브러리에는 해시 테이블을 관리하는 함수들이 존재한다. Figure 2와 같이 해당 기법에서 사용하는 해시 테이블은 메모리 주소를 키로 사용하며, 해당 메모리 주소의 유효 여부를 데이터로 저장한다. 해시 테이블을 생성하는 초기화 함수, 해시 테이블에 레코드를 추가하는 삽입 함수, 레코드 검증 함수, 레코드 삭제 함수가 라이브러리에 포함된다.

레코드 삽입 함수는 해당 메모리의 주소를 키로 사용하여 레코드를 생성하며, 레코드의 값은 유효함을 나타내는 '1'로 설정한다. 레코드 검증 함수는 메모리 주소가 입력으로 주어졌을 때, 해당 주소에 해당하는 레코드를 검색하고 해당 값이 '1'인지 여부를 확인한다. 만약 레코드가 존재하지 않거나 레코드의 값이 '0'이라면 검증에 실패하게 되고, 프로그램 충돌을 발생시킨다. 레코드 삭제 함수는 더 이상 유효하지 않은 메모리 영역에 해당하는 레코드를 삭제하거나 값을 '0'으로 변환한다.

코드 변환기는 입력으로 받은 코드를 컴파일 타임에 적절히 수정한다. 먼저, 프로그램 엔트리 포인트에 초기화 함수를 삽입하여 프로그램이 시작할 때 해시 테이블을 생성 및 초기화한다. 그리고 정적분석 단계에서 식별한 메모리 할당 지점의 직후에 레코드 삽입 함수를 추가하여, 해당 메모리가 유효하다는 정보를 해시 테이블에 저장한다. 그리고, 식별한 unsafe block 이후의 메모리 역참조 지점에는 레코드 검증 함수를 추가하여, 접근하는 메모리가 유효한지 검증하여 유효한 경우에만 메모리 접근을 허용한다. 또한 C 코드의 메모리 해제 지점 직전에는 레코드 삭제 함수를 삽입하여, 유효하지 않은 메모리에는 접근하지 못하도록 레코드 정보를 삭제한다.

제안하는 기법이 적용된, 함수가 추가된 코드는 Figure 1의 오른쪽 의사코드와 같다. 먼저, Rust의 객체 할당 지점 직후에 Record_insert_func()를 호출하여 해당 객체에 대한 레코드를 해시 테이블에 저장한다. 이후 C언어로 작성된 외부 함수가 호출되고 해당 객체에 대한 free()가 수행되기 전에, 해당 객체에 대한 레코드 정보를 삭제하는 Record_delete_func()를 호출한다. 외부 함수의 실행이 끝나고 다시 Rust 코드에서 해당 객체에 대한 접근을 수행하는데, 그 전에 Record_verify_func()를 호출하여 해당 객체가 유효한지 검증한다. 이 경우 레코드가 존재하지 않으므로, 레코드 검색에 실패하여 검증을 통과하지 못한다. 프로그램 충돌을 발생시킴으로써, 제안하는 기법은 MLA에서 발생할 수 있는 UAF나 DF와 같은 temporal memory corruption으로부터 프로그램을 보호할 수 있다.

Ⅲ. 구현

본 논문에서 제안하는 프레임워크는 메모리의 유효 여부를 나타내는 메타데이터를 해시 테이블에 저장하는데, 그 이유는 해시 테이블에 대한 검색, 삽입, 삭제가 빠른 시간에 수행되어 성능 저하를 최소화할 수 있기 때문이다. 하지만 해시 테이블을 적절하게 관리해야 하며 해시 테이블과 관련된 문제를 해결해야 한다. 먼저 해시 테이블은 gs 레지스터를 활용하여 접근할 수 있는데, 공격자는 gs 레지스터에 접근할 수 없으므로 해시 테이블의 정보를 안전하게 보관할 수 있다. 또한, Figure 2와 같이 해시 함수의 특징에 의해 충돌 문제가 발생할 수 있다. 예를 들어 Figure 2에서 Obj_A와 Obj_D는 다른 메모리 주소이지만 해시 함수의 결과값인 인덱스는 동일하므로 같은 bucket에 포함된다. 따라서, Separate chaining 기법을 사용하여 동일한 bucket에 해당하는 데이터를 링크드 리스트로 연결하며, 해당 bucket으로 검색을 수행할 때 링크드 리스트를 따라가며 입력값과 저장된 키를 비교하는 과정이 추가된다.

정적분석 단계에서는 Rust 코드와 LLVM IR을 이용하여 분석에 필요한 정보를 수집한다. Rust 코드에서 C/C++ 함수를 호출하는지 알아내기 위해서는 extern 키워드를 식별한 뒤, extern 키워드가 FFI를 위해 사용되는지 확인하는 과정을 거쳐야 한다. 또한 Rust의 경우 프로그래머가 직접 메모리 할당과 해제를 수행하지 않으므로, LLVM IR에서 *alloca()* instruction을 확인하여 메모리 할당 지점을 식별한다[6].

Ⅳ. 결론

본 논문에서 제안하는 프레임워크는 Rust와 C/C++로 구성된 MLA에서 CLA 기반의 temporal memory corruption을 막기 위한 기법이다. 제안하는 기법은 CLA에 의해 댕글링 포인터가 될 수 있는 포인터만을 선별하여 보호하며, 메타데이터를 해쉬 테이블에 저장하여 검증함으로써 비교적 적은 성능 저하를 유발한다. 또한 제안하는 기법은 하드웨어에 독립적이므로 확장성이 뛰어나다.

ACKNOWLEDGMENT

이 논문은 2023년도 정부(산업통상자원부)의 재원으로 한국산업기술진흥원의 지원을 받아 수행된 연구임(P0023522, 2023년 산업혁신인재성장지원사업)

참 고 문 헌

[1] Benchmarks Game, 2023, (https://benchmarksgame-team.pages.debian.net/benchmarksgame/box-plot-summary-charts.html).

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

[3] Li, Z., Wang, J., Sun, M., & Lui, J. C. (2022, September). Detecting Cross-language Memory Management Issues in Rust. In Computer Security–ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part III (pp. 680–700). Cham: Springer Nature Switzerland.

[4] Yosef E Mihretie. 2022. Automatic Exploit Generation for Cross-Language Attacks. Ph. D. Dissertation. Massachusetts Institute of Technology

[5] Steve Klabnik and Carol Nichols. 2018. The Rust Programming Language. NoStarch Press, USA.

[6] Bang, Inyoung, et al. "TRUST: A Compilation Framework for In-process Isolation to Protect Safe Rust against Untrusted Code." 32nd USENIX Security Symposium (USENIX Security 23). Baltimore, MD: USENIX Association. 2023.