Rust 언어와의 외부 함수 인터페이스 격리 연구방향에 관한 연구

커욘도마틴[1,4], 유준승[1,4], 최진명[2,4], 백윤흥[3,4]
[1]서울대학교 전기정보학부 박사과정
[2]서울대학교 전기정보학부 석사과정
[3]서울대학교 전기정보학부 교수
[4]반도체 공동연구소
kymartin@sor.snu.ac.kr, jsyou@sor.snu..ac.kr, jmchoi@sor.snu.ac.kr ypaek@snu.ac.kr

Assessing The Landscape: A Survey on Foreign Function Interface Isolation in Rust

Martin Kayondo[1,2], Junseung You[1,2], Jinmyeong Choi[1,2], Yunheung Paek[1,2]
[1]Dept. of Electrical and Computer Engineering, Seoul National University
[2]Inter-University Semiconductor Research Center, Seoul National University

Abstract

Rust has gained recognition for its emphasis on and commitment to providing memory safety. However, seamlessly integrating it with Foreign Function Interfaces (FFIs) written in unsafe languages remains a significant challenge towards achieving complete memory safety. To address this challenge, researchers have proposed Foreign Function Isolation as a potential solution, leading to emergence of various approaches in this domain. This paper critically evaluates existing solutions and illuminates the gaps that need to be addressed to realize practical foreign function isolation in Rust.

1. Introduction

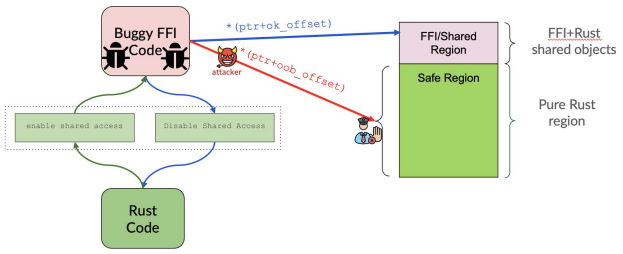
Memory bugs occupy a large proportion of software vulnerabilities, and related attacks are commonplace. Most existing programming languages are inherently unsafe, and do not guarantee memory safety out-of-the-box. Patch-based solutions such as compiler-based static analysis and instrumentation have been proposed but most of them either incur high overhead or are too resource intensive to be applied in production. In 2017, Rust emmerged as a safe programming language that guarantees memory safety out-of-the-box, while offering competitive performance, close to unsafe languages. It achieves this by relying on the compiler to enforce memory safety rules, which when broken, the compilation process completely fails. For this, it has gained popularity, to the extent the United States NSA recommended future software be written in Rust.

However, incorporating Rust in existing codebases has become a challenge. The amount of software in circulation already in production is too much to rewrite in Rust. For example, most existing operating systems, including the Linux Kernel, libraries and performant programs were written in C/C++, a language plagued by memory bugs. As a result, it became inevitable for Rust to coexist and interoparate with existing unsafe languages. Rust provides a separate mode, *unsafe Rust*, which not only allows programmers to write code that foregoes memory safey rules, but also allows for calling functions written in other languages - the foreign function interface (FFI). With this, Rust written functions can call and be called by, foreign functions.

However, foreign functions operate outside of Rust's memory safety rules. Thus, their unsafety can permeate Rust's otherwise safe code, compromising its safety guarantees. Therefore, isolation between Rust and FFI execution becomes paramount to maintaining Rust's memory

safety.

In this paper, we analyze existing FFI isolation schemes and shed a light on the gaps yet to be addressed. In particular, we study XRust[1], TRust[2], PKRU-Safe[3], SandCrust[4] and Galeed[5] for our assessment. Our goal is to offer a comprehensive assessment of the practicality and feasibility of current solutions in effectively isolating FFI, and thus their ability to enhance Rust's memory safety.



(Figure 1) FFI Isolation and Rust Memory Protection

2. Foreign Function Isolation Schemes

Existing FFI isolation mechanisms isolation FFI execution through memory isolation. The rationale is that memory used exclusively by Rust should be protected and isolated from the used by FFI as shown in Figure 1. Depending on the policy, memory shared between FFI and Rust maybe always accessible to FFI or controlled environment where FFI makes shared memory access requests to Rust maybe offered. All isolation solutions aspire to solve the following challenges:

C1: Identifying memory exclusively used by Rust - the safe Vs unsafe objects.

C2: Allocating memory in isolated spaces.

C3-1: Realizing memory isolation during execution.

C3-2: In case of controlled shared access, identifying memory access points in FFI that require access permission from Rust, and implementing a request/response mechanism for access permissions between Rust and FFI.

2.1 Solutions to the Challenges:

C1: Identifying memory exclusively used by Rust: To solve this challenge, XRust requires the programmer to manually identify objects shared by Rust to FFI, and thus, indirectly identifying safe memory objects. This approach has no false positives, as the developer only selects objects they have confirmed to be unsafe. However, for large codebases, it proves impractical, as it requires too many manhours to analyze code manually. As a result, many unsafe objects remain unclassified, presenting false negatives. TRust and Galeed perform compiler-based static data-flow analysis to automatically identify safe and unsafe objects. While this approach theoretically outperforms manual identification, static data-flow analysis surffers from imprecision and incompleteness. Worse more, for large codebases, static analysis tools raise the compile time budget. For example, TRust takes days to fully analyze servo, and as reported, it suffers from false positives and negatives even for mere libraries. Finally, PKRU-Safe performs a dynamic profiling test-run to identify unsafe objects before actually executing the program. On surface, this seems like the best solution, but it is only practical in development. It is impossible to anticipate user input to build credible profile input at development phase. SandCrust requires the developer to manually sandbox blocks of code that are deemed unsafe and m,ay interact with FFI. Like XRust, this is manual and suffers from similar drawbacks.

C2: Allocating memory in isolated spaces: After identifying safe and unsafe objects, the next challenge is allocating their memory in isolated regions. XRust, PKRU-Safe and Galeed rely on a specialized allocator that allocates memory in two separate regions depending on a developer set flag. One region is reserved for safe memory. This design requires developers intending to apply any of these solutions to use the accompanying allocator or modify one following the specifications in these works. TRust uses two

memory allocators, one for safe objects and the other for unsafe ones. It then instruments rust programs to redirect safe object allocation to the safe allocator. Unsafe object allocations will are left to call the original *malloc* functions and hence go to the unsafe allocator. TRust mininally modifies the allocators to allocate only from specific address spaces, and the safe allocator to additionally protect its pages with Intel MPK. SandCrust allocates Rust/FFI shared objects in a process shared memory region, and executes FFI in a separate process. Thus SandCrust provides the easiest adoptable solution in this aspect as no modification to the allocator is required.

C3: Memory Isolation and Access Control

C3-1: A solution to this challenge defines the security level provided and the performance guarantees. XRust provides no further isolation mechanism beside allocating unsafe objects in a separate region. TRust, PKRU-Safe and Galeed protect the safe memory region using hardware-based Intel MPK. The isolation provided is performant as it is enforced through register modification and special instructions. Since Intel MPK provides for intra-process isolation, TRust and PKRU-Safe guarante low performance overhead. Galeed's performance overhead due to memory isolation is not guaranteed, because it depends on the shared memory accesses in FFI as explained under C3-2. SandCrust relies on sandboxing FFI in a separate process and lets the operating system handle memory isolation. However, this design requires IPC for shared memory, rendering SandCrust performance heavy.

C3-2: Among all solutions above, only Galeed attempts to reduce the granularity of access to shared memory per object. Once an object is placed in the shared/unsafe memory region by TRust, XRust, PKRU-Safe or SandCrust, no further guarantees on its safety are provided. In otherwords, any memory bug in FFI can be used to illegally access all memory objects in the unsafe region using authorization given from one memory object. Therefore, these solutions only guarantee that FFI cannot access the safe memory region, but no further safety guarantees are provided for objects stored in the unsafe/shared region. Galeed identifies that allowing FFI to have access to the whole unsafe region is problematic. For this, officially, Galeed has no shared objects. Memory used by FFI is strictly isolated from that used by Rust. If Rust needs to share a pointer to its memory with FFI, Galeed instead shared a pseudo-pointer, and instruments all FFI code that accesses the intended pointer to use the pseudo-pointer. Memory access through the pseudo-pointer is a request/response process, where FFI code requests access from Rust, and Rust performs necessary safety checks before granting access. Such checks include bounds and liveness checking on the pointer, while access is granted by enable MPK access (read or write as requested by FFI). Although this ensures finer granular protection and temporal sharing, repeatedly requesting access increases performance overhead. Additionally, instrumenting FFI requires availability of FFI source code. For cases where FFI source is unavailable, analysis and instrumentation is impossible, and thus Galeed must share the pointer as is, instead of a pseudo-pointer. Furthermore, FFI instrumentation requires data-flow analysis, which may not be complete. In this case, FFI will raise access errors on uninstrumented pointers, rendering Galeed only applicable for debug purposes and not in production.

3. Missing Pieces and Discussion

Among the solutions assessed above, none of them provides complete isolation between safe and unsafe objects. XRust not only relies on the developer to identify objects shared with FFI but also does not provide any further memory protection of safe memory objects from FFI. TRust relies on LLVM-based data-flow analysis to identify safe and unsafe objects. This analysis, as reported is incomplete and imprecise, thus some objects maybe incorrectly classified. PKRU-Safe relies on program profiling to identify

unsafe objects, which is impractical for production purposes. XRust, TRust, PKRU-Safe and SandCrust provide no further protection for objects shared between FFI and Rust, while the solution provided by Galeed is incomplete and has a high performance overhead.

Based on this analysis, we identify several challenges yet to be tackled in this area. Firstly, identifying unsafe and safe objects requires a balance between applicability and completeness. Profiling is accurate and complete, but only depends on specific input, while static analysis is both incomplete and imprecise. Thus, there is need for a solution that strikes a balance between static instrumentation and dynamic classification of unsafe and safe objects. Secondly, there is need for a finer granular solution, that can protect individual objects instead of a whole class. Our speculation is that TRust and PKRU-Safe fail to provide such a solution due to their reliance on MPK, which is page granular. Although Galeed attempts to do better than the former two, its solution does not apply to binary FFI and data flow analysis on FFI source code may be incomplete or imprecise. Thus, a performant solution that can protect shared objects individually and temporarily, and is applicable in production is necessary. Finally, none of these works pays attention to Rust smart pointers. Smart pointers are a special part of Rust, providing runtime memory bug prevention, yet relying on metadata stored with program data. Therefore, a full solution enhancing Rust memory security in this direction must account for protection of smart pointer metadata.

4. Conclusion

In this paper we assess existing solutions for isolating Rust and FFI memory objects. We study and evaluate four existing works on the topic and provide a direction for future works. We find that existing solutions are either incomplete, impractical or incur high overhead. We therefore identify focus points for any future works attempting to provide solutions or improvements on existing works.

References

[1] Liu, Peiming, Gang Zhao, and Jeff Huang, "Securing unsafe rust programs with XRust.", Proceedings of the ACM/IEEE International Conference on Software Engineering, 42nd, 2020, pp 234-245.

[2] Bang I, Kayondo M, Moon H, Paek Y, {TRust}: A Compilation Framework for In-process Isolation to Protect Safe Rust against Untrusted Code, USENIX Security Symposium (USENIX Security 23), 32nd , 2023, pp. 6947-6964.

[3] Kirth P, Dickerson M, Crane S, Larsen P, Dabrowski A, Gens D, Na Y, Volckaert S, Franz M, PKRU-Safe: Automatically locking down the heap between safe and unsafe languages, Proceedings of the European Conference on Computer Systems, 17th, pp. 132-148.

[4] Lamowski B, Weinhold C, Lackorzynski A, Härtig H, Sandcrust: Automatic sandboxing of unsafe components in rust. Proceedings of the Workshop on Programming Languages and Operating Systems, 9th, 2017, pp. 51-57.

[5] Rivera E, Mergendahl S, Shrobe H, Okhravi H, Burow N. Keeping safe rust safe with galeed. Proceedings of the Annual Computer Security Applications Conference, 37th, pp. 824-836.

Acknowledgement

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (RS-2023-00277326). This work was supported by the BK21 FOUR program of the Education and Research Program for Future ICT Pioneers, Seoul National University in 2024. This work was supported by Inter-University Semiconductor Research Center (ISRC). This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) under the artificial intelligence semiconductor support program to nurture the best talents (IITP-2023-RS-2023-00256081) grant funded by the Korea government(MSIT).