Rust 언어의 FFI로 인한 취약에 대한 연구

카욘도 마틴*, 방인영*, 유준승*, 서지원*, 백윤흥*
*서울대학교 전기정보학부, 반도체 공동연구소
{kymartin, iybang, jsu, jwseo}@sor.snu.ac.kr, ypaek@snu.ac.kr

A Study on Security Issues Due to Foreign Function Interface in Rust

Kayondo Martin*, In-Young Bang*, Jun-Seung You*, Ji-Won Seo*, Yun-Heung Paek*
*Department of Electrical and Computer Engineering and Inter-University Semiconductor Research Center, Seoul National University.

Abstract

Rust is a promising system programming language that made its debut in 2010. It was developed to address the security problems in C/C++. It features a property called ownership, on which it relies to mitigate memory attacks. For this and its many other features, the language has consistently gained popularity and many companies have begun to seriously consider it for production uses. However, Rust also supports safe and unsafe regions under which the foreign function interface (FFI), used to port to other languages, falls. In the unsafety region, Rust surrenders most of its safety features, allowing programmers to perform operations without check. In this study, we analyze the security issues that arise due to Rust's safety/unsafety property, especially those introduced by Rust FFI.

I. Introduction

Over the years, C/C++ has gained continued employment in system programming majorly due to its robustness. C language, especially, is preferred by embedded systems engineers citing its closeness to hardware, speed and lightweightness [1]. For such reasons, most of the systems today have C/C++ as their core, on top of which other languages are built. However, despite its advantages, C/C++ faces a significant problem of memory security [2]. Programmers have to be extremely vigilant to avoid bugs that would result in memory corruption, rendering systems vulnerable to attacks. Major memory safety violations include:

1) Use after free (UAF)
2) Double free
3) Buffer overflow
4) Using uninitialized memory

In cases where the programmer forgets that a certain object has been freed and tries to use it, a UAF violation is a possibility. This of course can happen following a double free, where the programmer forgets that a given object has been freed and frees it again. A buffer overflow may arise if the programmer accesses memory out of bounds of the allocated object. Finally, a programmer may attempt to use a pointer to memory that they have not initialized yet.

Memory errors may result in:

i) a crash, if invalid memory is accessed, leading to termination of the program.

ii) information leakage, especially if the error arises from an overflow. In such cases, an attacker may read privileged information.

iii) arbitrary code execution, usually as a result of UAF and double free errors. An attacker may rely on such errors to redirect the program to execute unintended code. This method is mostly used to spawn reverse shells from servers.

Years of research have been invested in devising means to mitigate such attacks without affecting the functionality and robustness of the language but to little success. Considerable hardware designs have attempted to help with the issue. These include the memory tagging mechanism (MTE) in ARMv8.5+, Pointer Authentication Code (PAC) mechanism by ARMv8.3+ [3], memory protection keys (MPK) in Intel chips [4] and AMD Memory Guard in AMD chips. These protection mechanisms, however, are mostly effective in mitigating stack memory attacks and require additional crafting for utilization in dynamic memory corruption. Compile optimizations, such as safe stack, have also been utilized but attackers have found corresponding bypasses.

Rust programming language presents a distinct approach while offering relatively the same advantages as C/C++. It guarantees memory safety while maintaining the same robustness as or even better than C/C++. Even though still under development, Rust promises a considerably safe system programming environment, and its employment is predicted to surpass C/C++ in future production. The language relies on properties such as ownership and mutability to ensure memory safety.

II. Body

2.1 Memory Management in C/C++

Aside from stack memory, which is allocated on variable declaration in a given function and deallocated when the function exits, dynamic memory is allocated manually by the program as a buffer. In C, allocation is done by calling the malloc related functions. Advanced allocation can be done by mapping memory directly through the mmap system call. In C++, both the malloc related functions and the new operator can be used for allocation. To deallocate a memory object, both C and C++ can rely on the free function or munmap (depending on the allocation function used), or C++ can use the delete operator to free an object allocated using the new operator. In most cases, a program relies on the system allocation library - the allocator, rather than direct memory mapping. Such libraries include dlmalloc, ptmalloc, jemalloc, tcmalloc, ffmalloc etcetera. They are designed differently based on the needs of the program, and provide different performance and memory overhead guarantees. The major determinants of the allocator to use are: memory needs of the program, multithreading requirements and lightweightness. A C/C++ programmer is tasked with manually allocating and deallocating memory as they need it, but they are also responsible for any possible memory errors.

2.2 Memory management in Rust

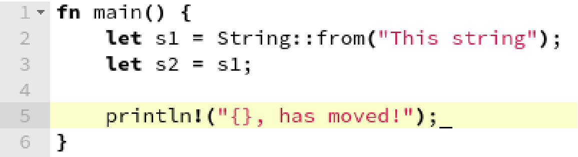
Rust, in the background, relies on the same memory allocation and deallocation functions, but does a lot more to ensure memory safety. It utilizes the ownership property to archive both performance and memory safety. This way, without incurring performance overhead, rust outperforms C/C++ by ensuring memory safety at negligible cost.

2.3 Rust Ownership

The concept of ownership in Rust restricts values from sharing 'owners', and is bound by the following rules:

a) Each value has a variable, the owner.
b) A value shall not have more than one owner at a time.
c) A value shall die when its owner goes out of scope.

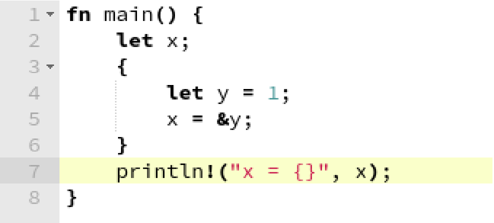
Because these rules are too strict, and may appear very aggressive to programmers, ownership is accompanied by two other concepts; moving and borrowing to support charing of values among owners. A part of the Rust compiler, the borrow checker, enforces these rules and ensures memory violation errors do not arise. Figure 1 below shows Rust's ownership in action. In the example, both s1 and s2 point to the same memory object. When they go out of scope due to rule (c), the system attempts to free s1 and s2 at the same time, resulting in a double free memory error. Therefore, Rust enforces the moving concept, where in line 3, ownership is transferred (moved) from s1 to s2, invalidating s1. Due to rule (a), any references to s1 thereon will be invalid and the compiler will throw and error. A deep copy can be made to avoid this, depending on the desires and intentions of the programmer.



```
1 fn main() {
2     let s1 = String::from("This string");
3     let s2 = s1;
4
5     println!("{}, has moved!");_
6 }
```

Figure 1. Rust's ownership in action.

The borrow checker also verifies the lifetimes of the variables to mitigate errors due to using uninitialized memory. Figure 2 below provides an example of such an incident.



```
1 fn main() {
2     let x;
3     {
4         let y = 1;
5         x = &y;
6     }
7     println!("x = {}", x);
8 }
```

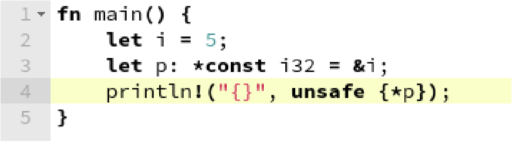
Figure 2. Rust enforcing rule (c)

Variable x is declared at line 2, but is initialized in the next block. The initialization, however, refers to a variable y, which 'dies' when it goes out of scope. For that reason, x is uninitialized on reaching line 7 and any attempt to use it will result in error (4). The compiler rejects this code after tracking scopes. This way, dangling pointers are avoided.

Like C/C++, Rust allows for uninitialized variables, but contrary to C, it does not allow using them until they are initialized. Additionally, Rust deters dangling pointer and null pointer dereference by avoiding nullable pointers and raw pointer dereference.

2.4 Safe Vs Unsafe Rust

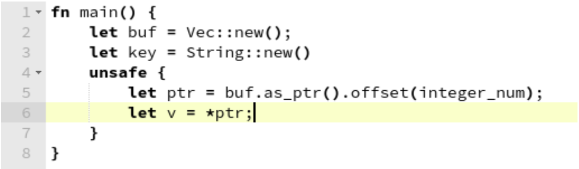
Topics discussed above entail what is referred to as safe Rust. The language also allows for an unsafe option [5]. Normally, rust programs comprise safe and unsafe regions. Inside an unsafe region, the programmer may go against some of Rust's safety rules. Unsafe rust, declared by using the 'unsafe' keyword to wrap code, allows for operations suchs raw pointer dereferencing, aliasing, foreign function calls etcetera. Figure 3 shows a code snippet in which raw pointer dereferencing is done (just like in C/C++).



```
fn main() {
    let i = 5;
    let p: *const i32 = &i;
    println!("{}", unsafe {*p});
}
```

Figure 3. Raw pointer dereferencing in unsafe Rust.

Such operations are considered unsafe because the pointer p is dereferenced without any sanity check. Rust, therefore, entrusts the programmer and assumes they consider the operation safe enough. Figure 4 shows how unsafe Rust can be used to violate Rust rules, hence leading to memory violations.



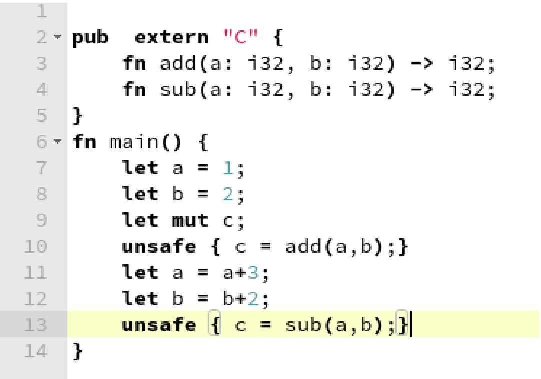
```
1  fn main() {
2      let buf = Vec::new();
3      let key = String::new()
4      unsafe {
5          let ptr = buf.as_ptr().offset(integer_num);
6          let v = *ptr;
7      }
8  }
```

Figure 4. Unsafe Rust resulting in memory corruption

In this example, if 'integer_num' is larger than the allocated size of the buffer, 'buf', then the information stored in 'key' can be leaked to 'v' in line 6. This way, Rust is hacked just like C, and therefore ceases to be a safe language.

2.5 Rust Foreign Function Interface (FFI)

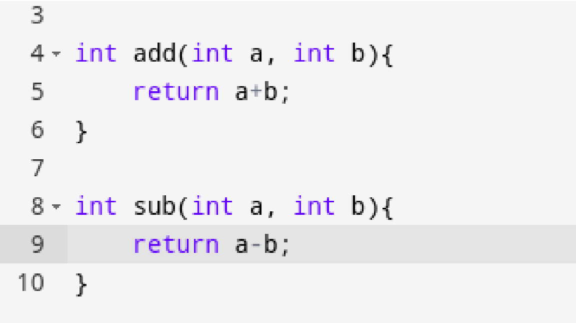
The FFI allows Rust to interact with other languages. Libraries written in C, for example, can be used with Rust by using the 'extern' keyword. This is a big advantage because there is no need to rewrite all existing C libraries in Rust to attain the same functionality. External blocks declared with the 'extern' keyword are annotated with the '#[link]' attribute containing the name of the foreign library to make it available to Rust. This way, Rust only needs to declare the foreign functions the exact same way they are declared in the foreign library, and can use their definitions from the foreign library through its external linkage. Figures 5 and 6 show how Rust FFI is used to import C code to Rust. Here, the add and sub functions take two integers and return an integer. Therefore, Rust must declare the same functions with the same names, same types of the arguments and same return type.



```

pub  extern "C" {
     fn add(a: i32, b: i32) -> i32;
     fn sub(a: i32, b: i32) -> i32;
}
fn main() {
     let a = 1;
     let b = 2;
     let mut c;
     unsafe { c = add(a,b);}
     let a = a+3;
     let b = b+2;
     unsafe { c = sub(a,b);}
}
```

Figure 5. Calling foreign functions from Rust.



```

int add(int a, int b){
    return a+b;
}

int sub(int a, int b){
    return a-b;
}
```

Figure 6. Functions to be exported to Rust defined C

The challenge, however, is that there is no means to enforce Rust's safety in such imported libraries. It is therefore a Rust rule that all calls to foreign functions must be wrapped with the 'unsafe' keyword. Because of this unsafety, pointer variables sent to foreign function calls as arguments may be used to corrupt Rust's safety.

Considering Rust treats foreign functions as black boxes, there seems to be no measures to possibly take to tackle this problem indigenously. Programmers are therefore left with a choice of either rewriting C libraries in Rust or ensuring only safe C code is imported. This may be done by avoiding any external functions that take pointers as arguments, a restriction that renders FFI virtually useless.

2.6 Possible Mitigations

There are numerous research works that have been done in attempt to solve the problem, but current solutions provide either partial safety, require an enormous effort to write supplementary code to the Rust compiler or memory allocator, or protect only a subset of programs.

Liu et al. present XRust [6], a mechanism to mitigate security threats due to unsafe Rust. They ensure that memory allocation in safe regions is done separately from that in unsafe regions, thereby restricting access to the safe region from the unsafe region. This approach provides a fair solution to the problem, but the incurred overhead is significant. It also requires an enormous effort to apply, requiring programmers to use other means of memory allocation than originally learned in Rust programming. The learning curve is not as significant as the overhead issue. The high overhead incurred in applying XRust to important libraries like vec and string pose a big challenge to the adaptation of the solution for production use.

Other possible solutions to this problem include MemSentry by Koning et al [4], where safe regions are isolated by relying on Intel's MPK. A similar approach may be ERIM as presented by Vahldiek-Oberwagner et al [7], where Intel's MPK is used to isolate domains. These mechanisms work for Intel processor based systems, but adopting ARM's MTE or PAC may be used similarly to achieve the same goal. Farkhani et al. present PTAuth [3], a memory protection mechanism based on ARM's PAC. Such a mechanism may be used to apply particular Authentication Codes to objects allocated in safe regions, making them inaccessible to unsafe code.

At the moment, there is no confirmed solution to this problem and a lot of research is being done to find an applicable measure to tackle it.

III. Conclusion

Rust programming language is still young but its popularity is on the rise than ever before. It is no doubt it can be a lot better, but its promising security features are giving it a head start and perhaps preference over its C/C++ counterpart. Its foreign function interface is another feature that has made it easy to adopt, yet on the other hand, could possibly be its nemesis if not paid special attention. The language attempts to provide a safe environment for system programming, and in order for its success, programmers will have to fully acknowledge its security features without the need for special patches. Finding a means to use completely safe Rust FFI will be a big step ahead for both language users and developers. There have been many attempts to solve the memory safety problems in C/C++ but most of the suggested solutions have turned out costly either performance wise or memory wise. This poses a challenge for Rust, if it is to replace C/C++ for good. Just like the C/C++ memory safety problem, will Rust's unsafety (and thus FFI) issues become its eternal weakness that software engineers will aim to circumvent forever?

IV. Acknowledgement

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (NRF-2020R1A2B5B03095204) and the BK21 FOUR program of the Education and Research Program for Future ICT Pioneers, Seoul National University in 2021.

V. References

[1] Sangehul, Lee and Jae, Wook, Jeon, Evaluating performance of Android platform using native C for embedded systems, ICCAS, 2010

[2] Shin, Jangseop and Kwon, Donghyun and Seo, Jiwon and Cho, Yeongpil and Paek, Yunheung, CRCount: Pointer Invalidation with Reference Counting to Mitigate Use-after-free in Legacy C/C++, NDSS: Network and Distributed System Security Symposim, 2019

[3] Farkhani, Reza and Ahmadi, Mansour and Lu, Long, PTAuth: Temporal Memory Safety via Points-to Authentication, Cryptography and Security, 2020

[4] Koning, Koen and Chen, Xi and Bos, Herbert, No Need to Hide: Protecting Safe Regions on Commodity Hardware, EuroSys: European Conference on Computer Systems, 2017

[5] Astrauskas, Vytautas and Matheja, Christoph and Poli, Federico and Muller, Peter and Summers, Alexander, How do Programmers Use Unsafe Rust?, IEEE: Proceedings of the ACM on Programming Languages, 2020

[6] Liu, Peiming and Zhao, Gang and Huang, Jeff, Securing Unsafe Rust Programs with XRust, IEEE: International Conference on Software Engineering, 2020

[7] Anjo, Vahldiek-Oberwagner and Eslam, Elnikety and Nuno, Duarte and Michael, Sammler and Peter, Druschel and Deepa, Garg, ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK), USENIX Security Symposium, 2019