On the Translation Between C and Rust

1st Daniel Dean
The College of Information Sciences and Technology
The Pennsylvania State University
State College, PA, United States
dpd5518@psu.edu

***Abstract*—C and C++ have been dominant languages in the systems programming space since the 1970s. Both of these languages are immensely powerful, but this power comes at the cost of security. Memory security issues have plagued this style of programming since its inception. The tradeoff between speed and security is not necessary, however. Rust is a programming language with the speed of C/C++ and the memory safety guarantees of a garbage collected languages. The tradeoff that Rust makes is for complexity and a slower speed of development. The problem: how can we transfer from a C dominated space to a safer Rust dominated space? This paper will give a background on the security mechanisms of rust, the current state of automatic translation from C to Rust, and discuss what the future may entail for security minded individuals in the systems programming space.**

***Index Terms*—C/C++, Rust, Memory Safety, C2Rust**

I. Introduction

In the realm of systems development, C/C++ are the typical choice when it comes to writing complex yet highly efficient systems. While this offers fast software with relatively expedient speed of development, it comes with one major cost– safety. More specifically, C can be written in such a way that allows for memory issues which in the best case could lead to slower software and in the worst case could lead to a bad actor hijacking the execution of the application. Some of these memory issues include buffer overflow attacks, dangling pointers, and memory leaks.

There have been many proposed solutions to these memory safety issues. But the simplest is re-implementing code bases written in C in a memory safe language. While this could be any memory safe language, Rust has recently gained popularity, evident in its appearance in the Linux kernel [1] and recent United States Government incentives [2]. In early 2024 the United States Government released a report on the dire nature of software security as it relates to national cyber security goals. The report emphasizes the importance that programming languages play in the development of secure software. They offer the Rust programming language as a potential solution to decades of dubious memory management caused by the C languages [2]. This may signify a push from the government to require that new applications must be written in Rust, or legacy C applications must be rewritten in Rust – a simple solution, but a logistical nightmare.

While Rust offers high performance memory safety, many of the use-cases in which it is needed already have a C implementation. The transition from C to Rust poses a massive logistical problem. This problem is compounded by Rust's relatively steep learning curve, limiting access to talent that could feasibly implement these C programs in Rust. Attempts to fully automate this process have not yet proven fruitful.

II. Background

The reason that code-bases such as the Linux kernel are written in C is the high level of control these languages offer over the machine, allowing for highly performant code. Two principal aspects of C that aid in performance are the manual memory management and low runtime overhead. The traditional approach to ensure memory safety is to automate away memory management with a garbage collector; Lisp and Java are examples of garbage collected languages. Notwithstanding bugs in the implementation of the garbage collector, this guarantees memory safety, but results in more overhead, as the garbage collector must exist at runtime, causing performance degradation. While there are a plethora of algorithms for achieving garbage collection, all of which are outside the scope of this paper. Some of the traditional techniques can cost four times the amount of CPU cycles as opposed to the non-garbage-collected program, while others can have negligible effects on the program performance [3]. More recent techniques have brought this upper bound down to closer to two times the CPU cycles, which is still not ideal [4]. Outside of the performance costs, garbage collectors add a runtime to the execution of a language, which could result in unintended program side effects when developing at the systems-level. This approach to memory management creates a false dichotomy between memory safety and performance.

A. The Borrow Checker

Rust achieves memory safety without a garbage collector through the use of strict compile time rules: Lifetimes, Ownership, Borrowing, and Exclusive Mutability, all of which are checked at compile time [5]. Lifetimes simply denote how long a variable is valid in a program, which the rust compiler's borrow checker typically implicitly derives. For more complex lifetimes, the borrow checker will fail to automatically denote the lifetime and thus it must be done manually. This is a point of contention for new Rust programmers, as demonstrated through an analysis of stack overflow questions [6]. Borrowing Ownership and Borrowing are two related ideas in Rust where each variable owns a value, which changes when another variable takes ownership, or the value is passed to a function.

Borrowing is the passing of a reference to a variable or function, without changing ownership. Exclusive mutability is ensured through the compiler by enforcing that there is either one mutable reference or an arbitrary amount of immutable references [6], [7]. There are also checks that are compiled into the machine code for each Rust program, which could be considered overhead. Performance wise, the most impactful of these is the bounds checker, which checks for out-of-bounds accesses [5]. This suite of compile time checks guarantee memory safety so long as they are not deactivated using an unsafe block [7].

B. Unsafe Rust

Rust's Borrow Checker was designed to be conservative by nature. This assures that any program that passes the Borrow Checking phase of compilation (Borrow Checkable) will be memory safe. It is not the case, however, that every memory safe program will pass by the Borrow Checker's standards; this is intended [8](see Fig. 1). This unsafe feature can be encapsulated in safe functions, which is extensively used in the Rust project itself [9].

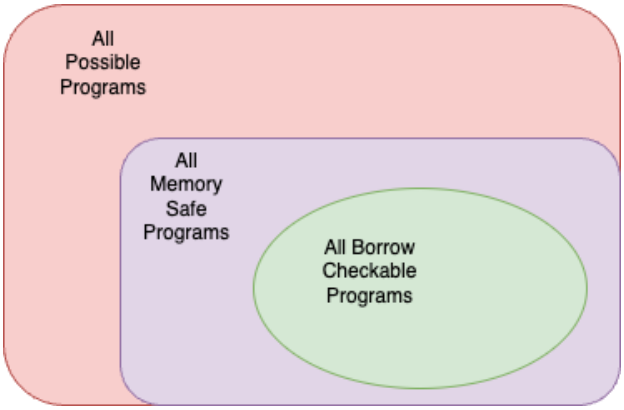

Fig. 1. Conceptual map of the Borrow Checkers conservative nature

III. Example of Vulnerabilities

In this section, we will explore examples of some common C vulnerabilities. We will consider two examples: stack buffer overflows, and double-fee. This section will include code, compilation errors, and explanations. All code for this section is available on GitHub [10].

It is also worth noting the difference between crashing an throwing exceptions, or errors, as this is a crucial difference between Rust and C/C++. Behavior like crashing and throwing exceptions may allow for a programmer to easily find and triage the bug, or allow for the analysis of a crash report. The difference between a crash and an exception is that a crash is completely out of the program's control–it is a system level problem. Exceptions, on the other hand, are still within a program and can often be caught and handled inside the program at runtime. An example of a crash would be a buffer overflow, whereas a an exception would be failing a bounds check for an array. Crashes can be used to mount attacks, whereas errors cannot.

A. Stack Buffer Overflow

A buffer overflow is a vulnerability where the bounds of a buffer are overwritten. This can be used to overwrite the return address of the current stack frame, allowing for arbitrary code execution. Consider the following proof of concept that takes in a user-provided string, and compares it with a hard-coded password. It is certainly feasible that a C programmer may assume that the **const char secret** will remain what they set it to at compilation time.

```
#include <stdio.h>
#include <string.h>

#define MAX_BUF 20

int main(int argc, char **argv) {
 const char secret[] = "letmein";
 char buf[MAX_BUF];
 if (argc < 2){
    printf("Usage: ./run <guess>\n");
    return 1;
 }
 if (argc > 1) {
    strcpy(buf, argv[1]);
 }
 if (0 ==
    strncmp(buf,secret,sizeof(secret))) {
    printf("You guessed correctly!\n");
    printf("The secret was: %s
        \n",secret);
    return 0;
 }
 printf("'%s' was not the secret.\nTry
    again later!\n",buf);
 return 0;
}
```

Fig. 2. Example C code with a buffer overflow vulnerability

We can see in Fig.3, that the program works as expected with sufficiently short input. As stated earlier, this program is vulnerable to a buffer overflow attack. This is due to the lack of bounds checking on the buffer **buf**, which can be overwritten by **strcpy**. By the construction of this program, we are able to overwrite **secret** in such a way that the comparison between the two strings always passes (Fig.4). Importantly in Fig.4, the **secret** variable has been completely overwritten by the user's input–which is a side effect of the attack, and my have down-stream effects in the program (i.e. if this variable is used multiple times). This type of vulnerability is especially nefarious, as the program doesn't crash or throw an exception.

B. Use After Free

A use after free vulnerability is when a programmer allows for the use of heap allocated memory after it has already been

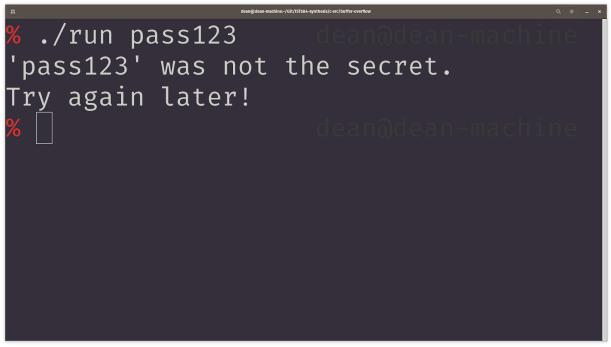

Fig. 3. The program working as expected

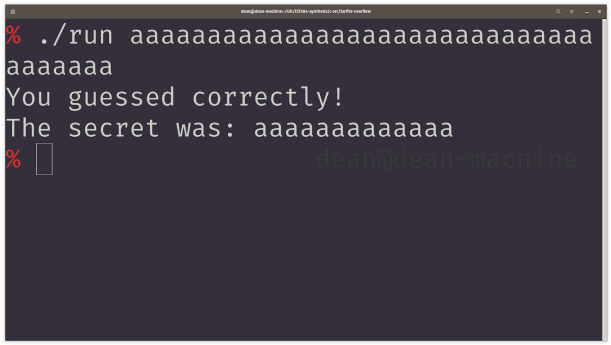

Fig. 4. A successful overflow attack

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct String {
  char *s;
  int len;
};
void consume(struct String * s){
  printf("string struct: {'%s',
      len=%d}\n",s->s,s->len);
  free(s);
}
int main(int argc, char **argv) {
  struct String *s =
      malloc(sizeof(struct String));
  if (argc < 2) {
    printf("Usage: ./run <string>\n");
    return 1;
  }
  s->s = argv[1];
  s->len = strlen(s->s);
  consume(s);
  consume(s);
  return 0;
}
```

Fig. 5. Example C code with use after free bug

Fig. 6. The code from Fig. 5 unconditionally crashing

freed, this may result in a crash, or a malicious corruption of memory [11]. Consider the code block in Fig. 5. This is a simple example that always crashes, but it compiles with no warnings. The reason it crashes is the the function **consume**, consumes the variable **s** in the sense that it frees the allocated variable. It succeeds the first time, but fails the second time, because the memory that the pointer **s** points to is already free (from the first call). We see the code failing in Fig. 6

C. Rust Implementations

We can rewrite these programs to show how they some of these crashes are solved by re-implementing the code in Rust. In Fig 7, the resulting program does not have a buffer overflow vulnerability. This is because it uses the Rust String type, which is different than the array of characters that C uses. Furthermore, this String type is dynamically allocated on the program's heap [12]. Next, we see in Fig. 9, that the code provided does not compile. The reason for this is that the ownership of **s** is moved into consume, and not returned, thus dropped. The purpose of this example is to show that writing the bug the same way as we wrote in C is not possible in safe Rust. This also means that any automatic translation must use unsafe Rust to correctly capture the functionality of the program, or lack thereof.

IV. Literature Review

A. C2Rust

C2Rust is a designed with the express purpose of generating a usable Rust implementations from code written in C [13][c2rust](https://c2rust.com/). To that end, the project is not concerned with implementing the C code in safe Rust, rather it is principally concerned with the primary translation between the languages. It succeeds in this goal. It achieves this through extensive use of unsafe Rust, which allows the automatic translation to mirror exactly what the C code is

```rust
use std::env;
fn main() {
    let argv: Vec<String> =
        env::args().collect();
    if argv.len() < 2 {
        println!("Usage: cargo run
            <guess>");
        return;
    }
    let secret: String =
        String::from("letmein");
    let guess: String = argv[1].clone();
    if secret == guess{
        println!("You guessed
            correctly!");
        println!("The secret was: {}"
            ,secret);
    }else{
        println!("'{}' was not the
            secret.",guess);
        println!("Try again later!\n");
    }
}
```

Fig. 7. Example Rust code equivalent to Fig. 2

doing, barring some limitations [14].

The next logical step is to transform the unsafe Rust into safe Rust if it is possible. This, however, is not trivial and is the target of ongoing research.

V. C++ Security

References

[1] L. Torvalds, "Linux/Rust at master," GitHub, 2024. Available: https://github.com/torvalds/linux/tree/master/rust. [Accessed: Mar. 18, 2024]

[2] The United States Government, "Back to The Building Blocks: A Path Towards Secure and Measurable Software," 2024. Available: https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

[3] M. Hertz and E. D. Berger, "Quantifying the performance of garbage collection vs. explicit memory management," Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Oct. 2005, doi: https://doi.org/10.1145/1094811.1094836[4]Z.

[4] Cai, S. M. Blackburn, M. D. Bond, and M. Maas, "Distilling the Real Cost of Production Garbage Collectors," 2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), May 2022, doi: https://doi.org/10.1109/ispass55109.2022.00005. Available: https://users.cecs.anu.edu.au/ steveb/pubs/papers/lbo-ispass-2022.pdf. [Accessed: Feb. 24, 2023]

[5] Y. Zhang, Y. Zhang, Georgios Portokalidis, and J. Xu, "Towards Understanding the Runtime Performance of Rust," 37th IEEE/ACM International Conference on Automated Software Engineering, Oct. 2022, doi: https://doi.org/10.1145/3551349.3559494

[6] S. Zhu, Z. Zhang, B. Qin, A. Xiong, and L. Song, "Learning and programming challenges of rust," Proceedings of the 44th International Conference on Software Engineering, May 2022, doi: https://doi.org/10.1145/3510003.3510164


```rust
use std::env;

fn consume(s: String) {
    println!("String: ('{}', {})",
        s,s.len());
}

fn main() {
    let argv: Vec<String> =
        env::args().collect();
    if argv.len() < 2 {
        println!("Usage: cargo run
            <string>");
        return;
    }
    let s: String = argv[1].to_string();
    consume(s);
    consume(s);
}
```

Fig. 8. Example Rust code equivalent to Fig. 5

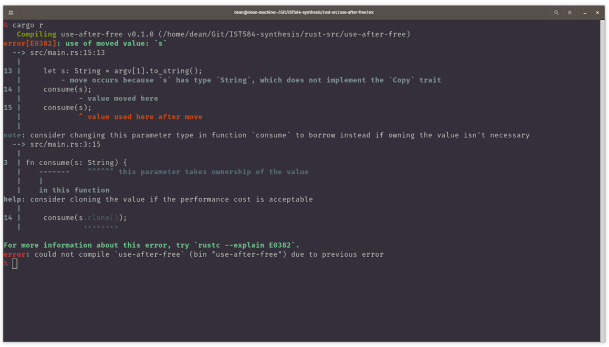

Fig. 9. A Rust program failing to compile

[7] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. R. Lyu, "Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs," ACM Transactions on Software Engineering and Methodology, vol. 31, no. 1, pp. 1–25, Jan. 2022, doi: https://doi.org/10.1145/3466642

[8] Rust Foundation, "Unsafe Rust - The Rust Programming Language," doc.rust-lang.org. Available: https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html

[9] Rust Foundation, "rust-lang/rust," GitHub, Apr. 15, 2024. Available: https://github.com/rust-lang/rust/tree/master. [Accessed: Apr. 15, 2024]

[10] D. Dean, "dandeandean/masters-capstone," GitHub, Apr. 15, 2024. Available: https://github.com/dandeandean/masters-capstone. [Accessed: Apr. 15, 2024]

[11] OWASP, "Using freed memory — OWASP," owasp.org. Available: https://owasp.org/www-community/vulnerabilities/Using_freed_memory

[12] The Rust Programming Language, "Strings - Rust By Example," doc.rust-lang.org. Available: https://doc.rust-lang.org/rust-by-example/std/str.html. [Accessed: Apr. 16, 2024]

[13] C2Rust, "C2Rust Demonstration," c2rust.com, Jun. 2019. Available: https://c2rust.com/

[14] Immunant, "Known Limitations of Translation," GitHub. Available: https://github.com/immunant/c2rust/wiki/Known-Limitations-of-Translation. [Accessed: Apr. 16, 2024]