The Indonesian Journal of Computer Science
www.ijcs.net
Volume 14, Issue 2, April 2025
https://doi.org/10.33022/ijcs.v14i2.4640

C vs Rust: Manual vs Automatic Spatial and Temporal Memory Safety

Amril Syalim, Dewangga Putra Sheradhien
amril.syalim@ui.ac.id[1], dewangga.putra@ui.ac.id[2]
[1,2] Fakultas Ilmu Komputer, Universitas Indonesia

Article Information

Received : 12 Jan 2025
Revised : 20 Feb 2025
Accepted : 15 Apr 2025

Keywords

Rust Programming Language, Software Security, Memory Safety

Abstract

The C programming language is commonly used for creating high-performance and low-level applications such as device drivers and operating systems due to its efficiency. However, despite its performance capabilities, C is known for its vulnerabilities and unsafe coding practices. Rust is presented as an alternative to C, with a focus on improved safety without compromising performance. Rust employs ownership and borrowing concepts to manage memory usage, ensuring that the memory cannot be manipulated freely without adhering to specific rules designed to prevent security attacks. In Rust, the memory restrictions are implemented either at compile time or runtime without requiring the programmer's direct involvement; however, the programmer must adhere to a strict coding standard. In contrast, C programs can be secured by manually implementing similar restrictions on memory access and adding checks for unpredictable runtime behavior. While this approach offers some protection against attacks, it requires the developer to have detailed knowledge of memory management and programming best practices. This research focuses on evaluating memory safety issues in terms of spatial and temporal safety, comparing Rust's security mechanisms (or lack thereof) to C. Spatial safety involves securing vulnerable memory locations, while temporal safety ensures safe access to memory at different times. These concepts are frequently exploited by attackers to access data or inject attack payload. Our analysis demonstrates that Rust offers stronger guarantees for memory safety compared to manual security implementations in C. Nevertheless, C remains a viable option for performance-critical applications, as it can still be secured through careful coding practices.

A. Introduction

Nowadays, computer programs are typically written by developers using high-level programming languages, which allow them to focus on solving problems rather than delving into the intricacies of hardware. These high-level languages provide an abstraction layer that simplifies the development process. Compilers play a crucial role in translating these high-level program codes into machine-readable instructions, making it possible for computers to execute the programs. In addition, the compilers are also instrumental in various aspects of program development (i.e. optimization, debugging and linting, security and protection and portability).

C is a high-level programming language commonly used to develop applications, designed collaboratively by Dennis Ritchie and Ken Thompson [9, 13]. The C programming language offers abstractions from computer hardware, making it easier for programmers to write understandable code. Despite this, C still supports low-level manipulation of hardware, such as pointer operations. Its flexibility comes with certain risks, as C does not prevent known security threats like buffer overflow and use after free. Multiple compilers are available for the C programming language, including pcc (Portable C Compiler), msvc, clang, and gcc (GNU Compiler Collection).

The Rust programming language was developed by Graydon Hoare in 2006 along with a compiler called rustc [2]. Considered memory-safe and thread-safe, Rust enforces these memory safety requirements using rules such as ownership, borrowing, and lifetime during both the compilation and runtime processes. These safety measures help prevent several vulnerabilities commonly found in C from being compiled or executed in Rust programs. However, Rust provides an *unsafe* keyword that allows programmers to bypass some of these security restrictions. When using the unsafe Rust, the security of Rust is comparable to the plain C [6,7].

Rust's security models are based on many lessons learned from memory safety issues in C that are frequently caused by the programmers. Researchers have also conducted various studies to understand how Rust's memory rules apply in practice. One of such studies employed type theory to create a model of the Rust language called Patina [12], which demonstrated that the Rust's memory rules successfully prevent issues like double frees and dangling pointers. In another study, the researchers have developed abstract models for the parts of the Rust language to create the Featherweight Rust (FR) language [11].

B. Objective and Methodology

The main objective of this research is to investigate the effectiveness of the security mechanisms in Rust by examining how the mechanisms improve the memory safety compared to traditional C programming approaches. Specifically, we focus on understanding and evaluating the spatial and temporal aspects of memory management offered by the Rust compiler. In our experiments, we design and develop two sets of programs: one in C language that violates the spatial and temporal memory safety requirements, and an equivalent, functionally similar program written in Rust where Rust should automatically prevent such issues without requiring manual security checking. Our methodology involves the following steps:

1. Write a C program that violates spatial and temporal memory safety requirements, such as bounds checking, null pointer dereferencing, use-after-free errors, etc., to serve as our baseline for comparison.
2. Develop an equivalent Rust program with similar functionality, which is expected to automatically prevent such issues due to the built-in protection offered by Rust's memory management mechanisms.
3. Compare and analyze the results from compiling and running each program, checking on errors or warnings produced during compilation, as well as the behavior of the compiled programs when executed.

C. Background on C and Rust Memory Management

C is a widely used programming language for system programming tasks such as creating operating systems or embedded devices. To create a program that processes and stores data we need to access the main memory in the form of variables. In C, variables can have different scopes (local, global, or static) and data types. Local variables are declared within functions or blocks of code and have a lifetime limited to the execution of that function or block. Global variables are visible throughout the program and maintain their lifetime for the duration of its execution, while static variables retain their values between function calls. Local variables in C can be initialized at declaration with either a default value or explicitly assigned [9].

In Rust, variables have similar scopes and data types, but unlike in C, local variables do not need to be explicitly initialized due to Rust's strong type system allowing for automatic deduction of variable types. Rust also manages their lifetimes through ownership, borrowing, and lifetimes. Global variables can retain their values throughout the entire program execution using the `static` keyword like their counterparts in C. Rust constants (declared with the `const` keyword) have a fixed value at compile-time and cannot be modified during runtime. Variables in Rust can be mutable or immutable by default [7].

A notable feature of C is its use of pointers, which store memory addresses. To declare a pointer in C, we prefix the variable name with an asterisk (`*`), such as `int *p` for a pointer to an integer data type. The 'address-of' (`&`) operator retrieves the memory address of a specific variable, while the dereference operator (`*`) on a memory address produces the value stored at that location. A characteristic of pointers in C is aliasing, when multiple variables access the same data in memory [1]. Aliasing can provide flexibility and dynamic behavior for object-oriented programming languages, but it also introduces potential risks such as security vulnerabilities due to side effects [1,3,8].

Rust takes a more stringent approach to pointers than C, with the primary goal of preventing memory safety issues such as aliasing and null-pointer dereferencing. In Rust, pointers are referred to as references, represented by the '`&`' symbol, and come in three types: (1) Immutable references denoted by the '`&`' symbol, (2) Mutable references signified by the '`&mut`' symbol, (3) Reference-to-an-iterator denoted by '`iter`'.

In C, memory is dynamically allocated on the heap using functions such as `malloc()`, `calloc()`, `realloc()`, and `free()`. Allocated memory has an undefined lifetime until it's freed using the `free()` function. Rust uses smart

pointers, such as `Box<T>`, to allocate memory on the heap. These pointers manage the lifetime of the allocated memory and deallocate it when the box goes out of scope. To access the value stored in a Box, we can use the dereference operator (*) as in C. Rust introduces ownership to ensure proper memory management by having one owner at a time for each value, with deallocation occurring when the owner goes out of scope. The `drop` function in Rust serves a similar purpose to the `free()` function in C and allows explicit deallocation of memory through the concept of ownership. References in Rust also have specific lifetimes to prevent dangling references from being used after their valid lifetimes end, offering stricter controls over pointers than in C [10].

D. Spatial and Temporal Memory Safety

In this research, we use the concepts of spatial and temporal safety to represent the memory safety. Spatial memory safety deals with the appropriate allocation, initialization, and deallocation of memory during runtime to avoid unintended accesses to memory addresses such as buffer overflows and use-after-free [1,8]. In C programming, this is achieved through a combination of manual memory management using functions like `malloc()`, `calloc()`, `realloc()`, and `free()` and careful handling of pointers [5]. On the other hand, Rust employs strict ownership and borrowing rules to prevent the vulnerabilities related to spatial safety.

Temporal memory safety relates to managing object lifetimes and access patterns during runtime, preventing dangling references or data races that may cause unintended behavior or security vulnerabilities. In C programming, this is achieved through techniques like reference counting or manually controlling the creation and destruction of objects [1,8]. Tools like MemSafe [14,15] can help C programmers ensure spatial and temporal memory safety. Meanwhile, Rust ensures temporal safety using a concept called ownership that restricts the number of owners for each value and deallocates objects when their owner goes out of scope.

In the following, we list the specific memory violations identified by spatial and temporal memory safety [14,15]:

1. *Bounds violations*: Incorrectly accessing memory outside the designated boundaries of an allocated block can lead to buffer overflows, reading or writing unintended data, and crashing the application. Spatial memory safety ensures that memory is accessed within its bounds during runtime by enforcing proper allocation, initialization, and deallocation.
2. *Uninitialized pointers*: Using a pointer without first initializing it to a valid address can lead to undefined behavior or crashes in the application. Spatial memory safety ensures that pointers are initialized before use.
3. *Null pointers*: Assigning null values to pointers indicates that they are not currently pointing to any data. Using null pointers without proper checks can lead to unintended behavior or crashes in the application. Spatial memory safety ensures that null pointers are handled properly.
4. *Manufactured pointers*: Manually constructing pointers without using functions like `malloc()` or `new()` can lead to incorrect memory allocation, resulting in bounds violations and other memory-related issues.

Spatial memory safety ensures that memory is allocated correctly by either manually managing it with functions such as `malloc()` or relying on smart pointers.

5. *Dangling stack pointers*: Stack pointers point to data stored on the stack during function execution. When a function returns, the memory associated with that stack pointer may become invalid if not properly deallocated or reused. Accessing dangling stack pointers can lead to unintended behavior or crashes in the application.
6. *Dangling heap pointers*: Heap pointers point to memory allocated on the heap using functions such as `malloc()`. Once the memory associated with a dangling heap pointer is deallocated, accessing it can lead to unintended behavior or crashes in the application.
7. *Multiple deallocations*: Deallocating memory more than once can lead to undefined behavior or memory leaks. Temporal memory safety ensures that memory is only deallocated once and properly managed during its lifetime.

E. Design of Experiments

We design experiments to compare the two programming languages – C and Rust – with respect to their handling of common memory-related bugs that can lead to vulnerabilities. Our experiment consists of the following steps.

Step 1: Preparing Test Cases. To start with, we create a pair of test program codes for each type of memory security violation based on relevant examples provided in Table 1 [14]. Initially, we write the test case in the C programming language. Next, we rewrite an equivalent test case in the Rust programming language that performs similar computations. These sets of program codes serve as a basis for further refinement and testing to explore various scenarios while still containing the same memory safety violations.

Table 1. Memory security violations and examples [14].

Type of Violation	Example (with C programming language)

Bounds violations	`struct { ... int array [100]; ... } s;`
	`int *p;`
	`p = &( s.array [101]);`
	`... *p ... // bounds violations`

Uninitialized pointers	`int *p;` `... *p ... // uninitialized pointers dereference`

Null pointers	`int *p;`
	`p = NULL;`
	`... *p ... // null pointers dereference`

Manufactured pointers	`int *p;`
	`p = (int*) 42;`
	`... *p ... // manufactured pointers dereference`

Dangling stack pointers	`int *p;`
	`void f() {`
	`int x;`
	`p = &x;`
	`}`
	`void g() {`
	`f();`
	`... *p ... // dangling stack pointers dereference`
	`}`

Dangling heap pointers	`int *p, *q;`
	`p = (int*) malloc (10* sizeof (int));`
	`q = p;`
	`free (p);`
	`... *q ... // dangling heap pointers dereference`

Multiple deallocations	`int *p, *q;`
	`p = (int*) malloc (10* sizeof (int));`
	`q = p;`
	`free (p);`
	`free (q); // multiple deallocations`

Step 2: Compiling Test Cases. Once prepared, each pair of the program code is compiled using appropriate compilers – gcc (C) and rustc (Rust). After compilation, both versions should be ready for execution and comparison.

Step 3: Comparing Outputs. We compare the output produced by the compiler and the running program. The analysis involves providing an explanation on how the compiler and the execution of the compiled program produce the respective outputs. We derive a conclusion about the differences in the way C and Rust handle memory-related operations, which affect the resulting output.

F. Results

In this section, we present the experimental results obtained from the testing of seven memory violations: (a) *Bounds violations*, (b) *Uninitialized pointers*, (c) *Null pointers*, (d) *Manufactured pointers*, (e) *Dangling stack pointers*, (f) *Dangling heap pointers*, (g) *Multiple deallocations*.

a. Bound violations

Simple Test Cases: First, we compare simple test cases represented by Code 1 written in C and Code 2 written in Rust. These test cases are examples of bounds violations, which occur when a program accesses memory beyond its defined bounds, such as an array index that is greater than the maximum index. Both test codes are derived from the code snippets presented in Table 1, where we have intentionally introduced a bounds violation by attempting to access an array index outside of its allowed range.

```c
#include < stdio.h >
struct Struct {int array [5];};
void main(){
    struct Struct s = {
      .array = {2, 3, 5, 7, 11}};
    int* p;
    p = &( s.array [6]);
    printf ("%d\n", *p);
}
```

Code 1. C program code for *bounds violations*

```rust
use std :: print ;
struct Struct { array : [i32; 5] }
fn main(){
    let mut s = Struct {
        array : [2, 3, 5, 7, 11]};
    let mut p : * mut i32;
    p = & mut ( s.array [6]);
    print !("{}\n", *p);
}
```

Code 2. Rust program code for *bounds violations*

Test Results: Code 1 is compiled successfully by gcc and executed without errors. The output produced when running Code 1 is a value printed to the standard output. In this program, the structure Struct is created on the stack, including an array named array. Accessing elements within `s.array` involves interaction with memory on the stack. When attempting to access an element outside the boundaries

of `s.array`, the value retrieved is not part of the allocated memory for the structure. The outputted values are random due to ASLR (Address Space Layout Randomization) in the operating system. On the other hand, Code 2 is not accepted by rustc due to a dereference operation on raw pointers `p`. In Rust, performing an unsafe operation such as a dereference on raw pointers is not allowed by the compiler. This means that the program code in Code 2 cannot be compiled and executed successfully.

Advanced Cases: The modifications made to Code 1 resulted in the creation of Code 3, which reveals that accessing out-of-bounds elements in `s.array` can also affect other fields within a structure (`struct`) in C program. Using pointers `p` and `q`, we can read the values of `secret1` and `secret2` by accessing the `s.array` elements at indices `-1` and `6`, respectively. The retrieved value is then printed to the standard output. Code 4 is a Rust code modified from Code 2, so that the variable `p` is a *reference* compared to *raw pointers.*

Test Results: Code 3 is accepted by gcc and can be executed to access out-of-bounds elements while Code 4 is not accepted by rustc because the compiler can see that there is an out-of-bounds access to an `s.array element.`

```c
#include < stdio.h >
struct Struct {
int secret1;
int array [5];
int dummy ;
int secret2;
};
void main(){
    struct Struct s = {
        .secret1 = 1337,
        .array = {2, 3, 5, 7, 11},
        .dummy = 0,
        .secret2 = 1337
    };
    int* p;
    int* q;
    p = &( s.array [-1]);
    printf (" s.array [-1] = %d\n", *p);
    q = &( s.array [6]);
    printf (" s.array [6] = %d\n", *q);
}
```

Code 3. Accessing other fields via `s.array`

```rust
use std :: print ;
struct Struct { array : [i32; 5] }
fn main(){
    let mut s = Struct {
        array : [2, 3, 5, 7, 11]
    };
    let mut p : & mut i32;
    p = & mut ( s.array [6]);
    print !("{}\n", *p);
}
```

Code 4. Results of modification of the `p` variable data type

More Advanced Cases: If the index value used to access an `s.array element` is dependent at *runtime,* the Rust compiler may not be able to detect the out-of-bounds access. One way to make values in Rust dependent at *runtime* is to return them from functions. Code 5 and code 6 are a pair of program codes resulting from modification of code 1 and code 2 so that the index for accessing `s.array` elements is obtained from a function call.

```c
#include < stdio.h >
struct Struct { int array [5]; };
void main(){
    struct Struct s = {
       .array = {2, 3, 5, 7, 11}
    };
```

```rust
use std :: print ;
struct Struct { array : [i32; 5] }
fn main(){
    let mut s = Struct {
        array : [2, 3, 5, 7, 11]
    };
```

```
    int* p;
    p = &( s.array [6]);
    printf ("%d\n", *p);
}
```

Code 5. C program code for more advanced case

```
    let mut p : * mut i32;
    p = & mut ( s.array [6]);
    print !("{}\n", *p);
}
```

Code 6. Rust program code for more advanced case

Test Results: The two codes above are accepted by each compiler. If the program from compilation code 5 is executed, the program still has the same behavior, namely printing values outside the boundaries of `s.array.` Execution of the program from compiling Code 6 causes *panic. Panic* is a mechanism in the Rust programming language that stops the program when an unrecoverable error occurs like *exceptions* in other programming languages.

Conclusion: We observe that C does not provide memory safety guarantees in cases where bounds violations occur. Rust compiler prevents the unsafe operations from being compiled and provides enhanced memory safety and reducing the likelihood of runtime errors.

b. **Uninitialized pointers**

Simple Test Cases: Code 7 and Code 8 are a pair of program codes created based on the code snippets in Table 1 regarding *uninitialized pointers dereference* in the C and Rust programming languages. Both of them did a simple thing, namely making a *pointer* without initializing any value, then performing a *dereference operation* on the *pointer*.

```
#include < stdio.h >
void main(){
    int* p;
    printf ("%d\n", *p);
}
```

Code 7. C program code for *pointers dereference*

```
use std :: print ;
fn main(){
    let mut p : * mut i32;
    print !("{}\n", *p);
}
```

Code 8. Rust program code for *pointers dereference*

Test Results: Code 7 is accepted by gcc. When the program is executed, the program produces the message "Segmentation fault" and exits. This occurs because a dereference operation was performed on pointer `p` which in this case has a value in the form of a memory address that cannot be accessed by the program. Meanwhile, Code 8 will not be accepted by the rustc compiler. The cause is that the compiler detects the use of a variable without initializing a value. Another cause is the *dereference operation* on the *raw pointers.*

Advanced Cases: For certain cases, *uninitialized pointers* can point to data that should not be accessed as in Code 9 as follows.

```
#include < stdio.h >
void f(){
int secret = 1337;
int* p = & secret;
    printf ("*p = %d\n", *p;
}
```

```
void g(){
    int dummy;
    int* q;
    printf ("*q = %d\n", *q;
}
void main(){ f(); g(); }
```

Code 9. ***Uninitialized pointers*** **`q` points to the `secret value in a C program`**

Test Results: In Code 9, there are two functions: `f` and `g`. The main function initiates the call to function `f` initially. Function `f` contains two local variables: a secret variable with a value of 1337 and a pointer `p` pointing to this secret variable. The value stored in pointer p (i.e., 1337) is printed to standard output. Following the execution of function `f`, function `g` is invoked. Function `g` has two local variables: dummy variables and an uninitialized pointer `p`. Interestingly, these two variables retain residual values from the call of function `f`. Consequently, a different pointer `q` points to the secret variable that can only be accessed via function `f`. The value stored in pointer q is then printed to standard output.

Conclusion: By examining the above test scenario, we can conclude that C programming language still lacks memory safety guarantees when uninitialized pointers occur. This vulnerability could be exploited to gain unauthorized access to sensitive data. In contrast, Rust ensures enhanced memory safety by preventing such unsafe operations from even being compiled.

c. **Null pointers**

Simple Test Cases: Code 10 and Code 11 were designed based on the code snippets in Table 1 pertaining to *null pointers dereference* in both C and Rust programming languages. The purpose of these program codes was to perform a straightforward operation: creating a *null pointer* and then attempting to execute a dereference operation on it. The intent behind these test scenarios was to compare the behavior of both languages when dealing with *null pointers*.

```
#include < stdio.h >
void main(){
    int* p = NULL;
    printf ("%d\n", *p);
}
```

Code 10. C program code for ***null pointers dereference***

```
use std ::{ ptr, print };
fn main(){
    let mut p : * mut i32 = ptr :: null_mut ();
    print !("{}\n", *p);
}
```

Code 11. Rust program code regarding ***null pointers dereference***

Test Results: Code 10 is accepted by gcc but produces a "Segmentation fault" and terminates upon execution. This occurs due to pointer p being a null pointer, referring to an invalid object. In contrast, Code 11 is rejected by rustc. It utilizes the `null_mut` function, which returns mutable raw null pointers. In Rust, raw pointers are not permitted to perform dereference operations as the compiler does not accept the code in question.

Conclusion: In these straightforward test scenarios, the gcc compiler accepts the provided C code without any issues, even though the code contains a null pointer dereference operation that is likely to cause runtime segmentation faults. On the

other hand, Rust rejects the code during compilation because it attempts to perform an invalid dereference operation on a null pointer.

d. **Manufactured pointers**

Simple Test Cases: Code 12 and Code 13 were created based on the code snippets in Table 1 pertaining to *manufacturing pointers dereference* in both C and Rust programming languages. These program codes aim to create *a pointer* that points to a self-generated address by type casting an integer into a pointer. In this case, the generated pointer points to memory with address 0x1337, which is chosen to not point to any valid object. These test scenarios compare the behavior of both languages when dealing with manually created pointers that could potentially dereference invalid memory addresses.

```c
#include < stdio.h >
void main(){
    int* p = (int*)0x1337;
    printf ("%d\n", *p);
}
```

Code 12. C program code for *manufacturing pointers dereference*

```rust
use std :: print ;
fn main(){
    let mut p : * mut i32 = 0x1337 as * mut i32;
    print !("{}", *p);
}
```

Code 13. Rust program code for *manufacturing pointers dereference*

Test Results: As in the *null case pointers dereference,* code 12 received by gcc. Execution of the program produces the message "Segmentation fault" and exits. Meanwhile code 13 is not accepted by the rustc compiler due to a *dereference operation* in *raw pointers.*

Conclusion: We conclude that when a pointer created manually points to an invalid memory address, Rust's built-in safeguards help prevent errors or potential security issues that could occur at runtime. This is very different from what happens when using the C programming language under comparable circumstances.

e. **Dangling stack pointers**

Simple Test Cases: Code 14 and Code 15 are two test scenarios created based on the code snippets from Table 1 that pertain to dangling stack pointers dereference in both C and Rust programming languages. In these program codes, a global pointer named `p` is declared, while a local variable `x` is defined within function `f`. The value of pointer `p` is set to the memory address of variable `x`. In the main function, the function f is called, followed by a dereference operation on pointer `p`. The intention behind creating these test scenarios was to examine how both languages manage dangling stack pointers, which could potentially cause errors or security vulnerabilities if not handled properly.

```c
#include < stdio.h >
int* p;
void f(){
    int x;
```

```rust
use std :: print ;
static mut p : * mut i32 = 0 as * mut i32;
fn f(){
    let mut x : i32;
```

```
    p = &x;
}
void main(){
    f();
    printf ("%d\n", *p);
}
```

Code 14. C program code for *dangling stack pointers dereference*

```
    p = & mut x;
}
fn main(){
    f();
    print !("{}\n", *p);
}
```

Code 15. Rust program code for *dangling stack pointers dereference*

Test Results: Code 14 is accepted by gcc. When the program is executed, a random value is printed to the standard output. Each execution produces a different random value. The value comes from the stack frame function `f`. More specifically, this value is the value of the variable `x` in the function `f`. After the function `f` is called, pointers `p` points to the variable `x` that has been deallocated. *Dereference* operation on pointer*s* `p` produces the value of the variable `x` which is then printed to standard output. Meanwhile, Code 15 is not accepted by rustc. One of the error messages the compiler produces indicates that the variable `x` was read before it was initialized. Another error message is the use of raw pointers with mutable properties static because it can cause undefined behavior, namely aliasing and data races.

Advanced Test Cases: To avoid the use of mutable static in Rust, the test code is modified so that pointer `p` is updated through the function parameter `f`. Additionally, the value of the variable `x` can be initialized with a known value initially. The data type owned by the variable `p` in Code 17 is a *reference* rather than *raw pointers*, which are *mutable static*. To provide a comparative analysis, we also create a similar C program in Code 16.

```
#include < stdio.h >
void f(int** p_to_p ){
    int x = 1337;
    *p_to_p = &x;
}
void main(){
    int dummy = 0;
    int* p = & dummy ;
    f(&p);
    printf ("%d\n", *p);
}
```

Code 16. Advanced test case for C program code

```
use std :: print ;
fn f( mut p_to_p : & mut & mut i32){
    let mut x : i32 = 1337;
    *p_to_p = &mut x;
}
fn main(){
    let mut dummy : i32 = 0;
    let mut p : &mut i32 = &mut dummy ;
    f(& mut p);
    print !("{}\n", *p);
}
```

Code 17. Advanced test case for Rust program code

Test Results: As previous modification, Code 16 is accepted by the gcc compiler. This time, the value printed by the program will always be 1337. Meanwhile, Code 17 is not accepted by the rustc compiler. However, the error message produced by the Rust compiler is different from before. The new error message says that variable `x` does not have a long enough *lifetime.* This is because the variable `x` will be deallocated at the end of the function `f`, while *the reference* that points to it can still be used outside the function `f`. In other words, *the lifetime* of *the mutable references* `p_to_p` will exceed *the lifetime* of the variable `x`, which does not satisfy the *lifetime rules* in the Rust programming language.

Conclusion: Through the provided test scenarios, Rust compiler successfully prevents errors or potential security vulnerabilities caused by dereferencing

dangling stack pointers in comparison to the behavior of C programming language when dealing with comparable situations.

f. Dangling heap pointers

Simple Test Cases: Code 18 and code 19 are a pair of program codes created based on the code snippets in Table 1 related to dangling heap pointers dereference in the C and Rust programming languages. The allocation process is carried out using the malloc function in the C programming language and the `Box:: new` function in the Rust programming language. Pointers `p` points to that memory allocation and is copied to become the value pointers `q`. Through pointers `p`, the allocation is then deallocated. The deallocation process is carried out using the `free` function in the C programming language and the `drop` function in the Rust programming language. Finally, a dereference operation is carried out on pointers `q`.

```c
#include < stdio.h >
#include < stdlib.h >
void main(){
    int* p = malloc ( sizeof (int)); *p = 2;
    int* q = p;
    free (p);
    printf ("%d\n", *q);
}
```

Code 18. C program code for *dangling heap pointers dereference*

```rust
use std :: print ;
use std ::{ boxed ::Box, mem ::drop};
fn main(){
    let mut p : Box<i32> = Box:: new (2);
    let mut q : Box<i32> = p;
    drop(p);
    print !("{}\n", *q);
}
```

Code 19. Rust program code for *dangling heap pointers dereference*

Test Results: Code 18 is accepted by gcc. The program will print the value 0 when executed. Meanwhile, Code 19 is not accepted by rustc. When the Box value `p` is copied to become the Box value `q`, what happens is the value of Box `p` is "moved" to Box `q`. This happens because the Box data type has *a move semantics.* Box `p` is considered to have no value and using it as an argument to the drop function will cause an error.

Conclusion: Based on the above cases, we conclude that when dealing with dangling heap pointers, like with dangling stack pointers, the Rust compiler prevents the compilation of the code, thereby ensuring that a potentially vulnerable program is not executed. This is in contrast to C programming language where similar vulnerabilities can exist without detection by the compiler.

g. Multiple deallocations

Simple Test Cases: Code 20 and Code 21 are a pair of program codes created based on the code snippets in Table 1 related to multiple deallocations in the C and Rust programming languages. The two codes are similar to Code 18 and Code 19. The difference is that the deallocation is carried out through pointers `q` rather than the dereference operation on pointers `q`.

```c
#include < stdlib.h >
void main(){
    int* p = malloc ( sizeof (int)); *p = 2;
    int* q = p;
    free (p);
    free (q);
}
```

Code 20. C program code for *multiple deallocations*

```rust
use std ::{ boxed ::Box, mem ::drop};
fn main(){
    let mut p : Box<i32> = Box:: new (2);
    let mut q : Box<i32> = p;
    drop(p);
    drop(q);
}
```

Code 21. Rust program for *multiple deallocations*

Test Results: Code 20 is accepted by gcc. When the program is executed, the program detects a *double free* (*multiple deallocations*) and exit. This protection occurs thanks to *double checking free* in *tcache,* which is a data structure used to store allocated memory that has been deallocated. For most systems, *tcache* can only store seven memory allocations. If more than seven memory allocations are deallocated, the remainder will be stored in other data structures such as *unsorted bin, fast bin, small bin,* and *large bin.* Meanwhile, Code 21 is not accepted by the rustc compiler. Similar to the case in Code 19, the value currently held by Box `p` is now owned by Box `q`, thereby forbidding reading the value indicated by Box `p` by the compiler.

Advanced Test Cases: Code 22 is a modified version of Code 20 that executes the deallocation process for eight distinct memory allocations. The initial memory allocation takes place using the malloc function, with the allocation address stored as an *array element* r. Afterward, the deallocation process is carried out for seven allocations and stored in *tcache*, which triggers the use of a different data structure (in this case, *fast bin*) to store the remaining allocation due to *tcache* being full. The *fast bin* data structure includes *double protection free,* ensuring that it checks if the last two allocated memory allocations are identical to prevent *double free*. Given that the program deals with memory designated by *pointers* `p` and *pointers* `q` sequentially, the program will identify a *double free* and terminate accordingly.

```c
#include < stdlib.h >
void main(){
int* r[7];
    for (int i = 0; i < 7; i++){ r[i] = malloc ( sizeof (int)); *r[i] = 1; }
int* p = malloc ( sizeof (int)); *p = 2;
int* q = p;
    for (int i = 0; i < 7; i++) free (r[i]);
    free (p); free (q);
}
```

Code 22. Modification of C program code

More Advanced Cases: Unfortunately, the protection in the previous cases might not suffice. By manipulating other memory locations between the deallocation processes involving pointers `p` and pointers `q`, a bypass can occur. To put it more explicitly, Code 23 provides an example of this. The deallocation process takes place on the memory allocation designated by pointer `t`.

```c
#include < stdlib.h >
void main(){
```

```
    int* r[7];
    for (int i = 0; i < 7; i++){ r[i] = malloc ( sizeof (int)); *r[i] = 1; }
    int* p = malloc ( sizeof (int)); *p = 2;
    int* q = p;
    int* t = malloc ( sizeof (int)); *t = 0;
    for (int i = 0; i < 7; i++) free (r[i]);
    free (p); free (t); free (q);
}
```

Code 23. Modification of C program for more advanced case

Test Results: Code 23 is accepted by gcc without any issues but does not produce any output since there is no code to print to standard output. This also suggests that a *double free* has occurred in the program.

Conclusion: In these last cases, we can conclude that by preventing multiple deallocations of heap memory during the compilation stage, Rust's programming model yet again show its ability to defend against more sophisticated attacks on heap memory management compared to C. The ownership model in the Rust programming language prohibits the double free vulnerabilities from being incorporated into the program during the compilation process.

G. Analysis and Discussion

In this section, we describe our analysis and discuss the test results.

1. Our first conclusion is that the testing results show the fact that the C compiler is allowing code containing the possibility of out-of-bounds array accesses without checking, while the Rust compiler is detecting these issues at compile time and asking the developer to modify their code accordingly to prevent runtime problems. The Rust compiler also prevents other potential memory safety violations such as uninitialized stack memory reading via pointers (uninitialized pointers), null pointer dereferences, and manipulation of fabricated pointers during the compile time.
2. We also observe that the C compiler accepts codes that may read data from other functions or access deallocated memory, potentially leading to unpredictable behavior and potential security vulnerabilities. The Rust compiler mitigates these risks by enforcing strict lifetime and ownership rules at compile time that prevent such issues occurring and guaranteeing more predictable and secure program behavior.
3. Rust clearly wins from the security perspective. However, it is worth noting that while Rust reduces the likelihood of security vulnerabilities being exploited at runtime by enforcing stricter rules, this may come with a steeper learning curve due to the programmers need to focus more on memory safety. In contrast, C's less stringent enforcement of memory safety rules allows for greater freedom and potential efficiency in programming, but with a higher risk of security vulnerabilities arising at runtime due to the compiler's acceptance of code containing potential security problems without any warnings or errors. To be fair, however, Rust still accommodates the programmer's choice to engage in unsafe operations, if necessary, by using unsafe Rust [6].

H. Comparison of C and Rust Compilation Times

In the previous sections, we have shown that Rust's compiler can automatically check the security problems by rejecting vulnerable codes, whereas C's compiler does not check many security issues, raising questions about the compilation overhead of the Rust compiler. Theoretically, due to its lower level of abstraction, the C programming language should have faster compilation times compared to the Rust programming language. However, it's important to note that to write secure C programs, additional efforts may be required through manual memory safety checks that must be performed by programmers. This overhead/cost can be comparable to the time needed by Rust programmers adhering to Rust's strict standard due to its higher learning curve.

In this section, we present experimental comparisons of compilation times between C and Rust programming languages using twelve cases (a to l). Seven cases are derived from codes used in the previous section: (a) Bounds violations, (b) Uninitialized pointers, (c) Null pointers, (d) Manufactured pointers, (e) Dangling stack pointers, (f) Dangling heap pointers, and (g) Multiple deallocations. We also include five more cases from comparable C and Rust programs related to data structures and algorithms that are often needed in a computer program. The cases are: (h) Linked list, (i) Binary search tree, (j) Extended binary search tree, (k) Double linked list, and (l) Extended double linked list. For each case, we compare the compilation time of C vs Rust when utilizing the similar code.

Figure 1 illustrates the results from our experiment comparing compilation times for C and Rust programs using a set of twelve pairs of test codes. The tests were executed on an Intel x86 computer (Linux Ubuntu 24, Intel i3 12100, 16GB RAM).

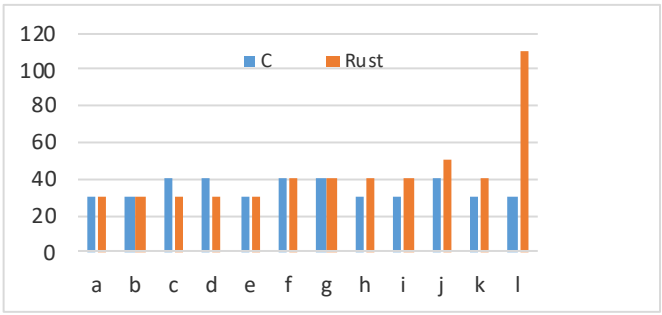

Figure 1. Compilation time for 12 cases (a to l) in milliseconds (ms)

Based on the experimental result, we conclude the following:

1. Experiments a, b, e, f, and g have very close or identical compilation times for both C and Rust, with differences generally within below 10 ms. This suggests that for these scenarios, there is little performance advantage in using either language (C or Rust) during the compilation phase.
2. Experiments h, i, j, and k demonstrate slightly faster compilation times for C compared to Rust while in experiments c and d, the Rust compiler is slightly faster. However, the differences are still very small (below 20 ms). This indicates that Rust's memory safety features may contribute to slightly

longer compilation times in some scenarios, but in many cases, this feature does not impact the compilation time.

3. In experiment l, Rust has a noticeably slower compilation time compared to C (110 ms versus 30 ms). This suggests that in some specific scenarios the C compiler is significantly faster over Rust. We believe that the number of these scenarios should be small, and the potential benefits of using Rust for memory safety, security, and modern language features may still outweigh the compilation time differences.

I. Conclusion

Our research demonstrates that the Rust compiler outperforms the C compiler in detecting memory security vulnerabilities during early development stages (compilation). While C offers greater flexibility, creating a secure C program requires manual security checks, which necessitate additional effort and expertise. Adopting Rust's security model allows developers to significantly reduce common memory-related bugs risks. Our research also shows that the performance overhead associated with using Rust is minimal. Rust can serve as an applicable and secure choice for many modern software development projects due to its enhanced security without compromising performance. However, as Rust is a stricter language, there is a learning curve cost involved in transitioning to its ecosystem.

J. References

[1] Aldrich, J., Kostadinov, V., & Chambers, C. (2002). Alias annotations for program understanding. SIGPLAN Not., 37(11), 311–330. doi : 10.1145/583854.58244

[2] Avram, A. (2012). Interviews on rust, a system programming language developed by mozilla. Retrieved 08-03-2012, from https://www.infoq.com/news/2012/08/Interview-Rust/

[3] Clarke, D. G., Potter, J. M., & Noble, J. (1998). Ownership types for flexible alias protection. In Ownership types for flexible alias protection (p. 48–64). New York, NY, USA: Association for Computing Machinery. doi : 10.1145/286936.286947

[4] CWE Content Team. (2023). Common weakness enumeration. Accessed from https://cwe.mitre.org/data/definitions/1000.html

[5] Dhurjati, D., Kowshik, S., Adve, V., & Lattner, C. (2003). Memory safety without run- time checks or garbage collection. In Proceedings of the 2003 acm sigplan conference on languages, compilers, and tools for embedded systems (p. 69–80). New York, NY, USA: Association for Computing Machinery. doi : 10.1145/780732.780743

[6] GitBook. (2023a). The rustonomicon. Accessed from https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html

[7] GitBook. (2023b). The rust reference. Accessed from https://doc.rust-lang.org/stable/reference

[8] Hogg, J., Lea, D., Wills, A., deChampeaux, D., & Holt, R. (1992). The geneva convention on the treatment of object aliasing. SIGPLAN OOPS Mess., 3(2), 11 16. doi : 10.1145/130943.130947

[9] Kernighan, B. W., & Ritchie, D. M. (1988). The C programming language. prentice-Hall.
[10] Klabnik, S., & Nichols, C. (2018). The rust programming language. USA: No Starch Press.
[11] Pearce, D.J. (2021). A lightweight formalism for reference lifetimes and borrowing in rust. ACM Trans. Program. Lang. Syst., 43(1). doi : 10.1145/3443420
[12] Reed, E.C. (2015). Patina : A formalization of the rust programming language. In Patina : A formalization of the rust programming language.
[13] Ritchie, D. M. (1996). The development of the c programming language. In History of programming languages —ii (p. 671-698). New York, NY, USA: Association for Computing Machinery.
[14] Simpson, M. S. (2011). Runtime enforcement of memory safety for the c programming language. ProQuest Dissertations and Theses, 218. Matsakis, N.D., & Klock, F.S. (2014). The rustic language. Ada Lett., 34(3), 103-104. doi : 10.1145/2692956.2663188.
[15] Simpson, M. S., & Barua, R. K. (2013). MemSafe: ensuring the spatial and temporal memory safety of C at runtime. Software: Practice and Experience, 43(1), 93-128.