Project report: Language security of Rust

Simon Johansson, Emrik Lindahl

June 19, 2024

Contents

1 Motivation

For a long time, C has been seen as the standard machine oriented programming language. It has been regarded as one of the best programming languages when it comes to performance, development and security. However during later years, languages such as the Rust language has emerged as a potential competitor to C. Rust also offers fast performance but most important of all it is said to be more secure, because of the way Rust implements memory management.

With Rust gaining more and more popularity by the day, it would be interesting from a security perspective to see how Rust manages memory safety and also compare the performance of compiled Rust code but also the development stages compared to the C language. Is Rust a true contender with C primarily when it comes to security, or is Rust just another language for low-level programming that can co-exist with C?

Since buffer overflows still are a common security problem for multiple applications even to this day, it would be interesting to see if Rust has anything new to offer when it comes to just buffer overflows. If Rust is able to efficiently withstand against buffer overflows, it would also be interesting to see if there are any performance loss because of the added security.

1.1 Problem Statement and Goal

In this project, we will investigate buffer-overflow attacks in the Rust language, and compare the implementations with that of the C language. The problem could be summarized with one question: Is the Rust language more secure when it comes to buffer-overflows and if it is, does it sacrifice any performance in doing so?

When this project has come to an end, we hope to have gained a greater understanding of both the Rust and C languages and also use this wisdom when developing applications in the future.

2 Background

In this chapter, we will explore the different ways memory management is implemented in both the C language and the Rust language, and then compare both languages.

2.1 Memory in the C language

Memory in C is either allocated by the compiler or allocated manually by the programmer by using functions. These will be described further in section 2.1.1.

The memory layout in C programs consists of a text segment, initialized data segment, uninitialized data segment, heap and stack. The text segment contains the program instructions to be executed and is usually placed above the heap or stack to prevent overflows from affecting it. The initialized data segment is allocated for static variables that are initialized by the programmer. The initialized data can be changed in run time and the segment can be split into a read-only and a read-write area. The uninitialized data segment instead contains the variables that has been initialized without a specified value, e.g int i;.

The stack and heap are parts of the memory layout which can grow and change in size. The stack, as discussed by Aleph One[1], comes from the abstract data type with the same name and works like a LIFO queue. To this queue so called stack frames are pushed and stored, these frames contain a set of values and also at minimum a return value. The stack frames are generated when a function is called and dynamically allocates memory for the local variables used in the function, the parameters passed to the function and the return values of the function. To keep track of the stack as it changes a pointer called the stack pointer is used. The stack pointer points to the end of the stack and the beginning is a fixed point in memory.

Since the size of the data to be stored are not always known there needs to be a way to allocate memory in run time, this is were the heap comes in. To allocate space in memory on the heap some functions are used, these are described further in 2.1.1.

2.1.1 Dynamic allocation of memory

Memory allocation in C are mostly handled through dynamic allocation of memory, which is done manually by the creator of the program. In modern programming languages this allocation of memory to the heap does not need to be handled by the programmer. The approach that C has gives a lot of freedom when writing programs in terms of memory management. But it also means that it is up to the programmer to make sure that all allocated memory is freed up after it has been used, otherwise the machine will run out of memory and it will crash.

In C the dynamic allocation of memory in the heap is most commonly handled with the methods Malloc, Calloc, Realloc and free is used to free up the allocated memory. The implementation of the allocation method may vary and there are multiple existing implementations. But it is also possible to allocate memory by using a function like *sprintf*, which writes a string to a buffer.

2.2 Memory in Rust

The memory structure in Rust is similar to that of in C. Although in Rust, there are no manual memory allocations. Instead memory is allocated and freed according to the ownership and borrowing rules[2]. Rust stores data-types that have a static size, for example the data-type int, on the stack. Data-types that has a dynamic size, like vectors, are stored on the heap. To explore more in-depth how memory management is done, we will explain mutability in Rust and then explore the two concepts ownership and borrowing more deeply.

2.2.1 Mutability

In Rust, variables are not mutable by default. This means that it is not possible to modify a value once it has been instantiated. In order to make a variable mutable, one needs to add the *mut* keyword before like in the following code snippet.

```rust
fn main() {
    let mut number = 3;
    number += 4;
}
```

2.2.2 Ownership

One of the core fundamental rules of the Rust language is ownership. According to the Rust book[3], there are three rules to ownership:

- Each value must have an owner.
- There can only be one owner at a time.
- The value is dropped once the owner goes out of scope.

To explain this in simpler terms, each value is associated with an owner, where an owner is typically a variable. Once that owner goes out of scope, the value will be dropped and automatically removed from memory. Although this is only applied to dynamic data-types since those data-types are the only data-types that are using the heap. Let's consider the following example using both a static data-type and a dynamic data-type.

```
1 fn main() {
2     let s1 = String::new("hello");
3     let s2 = s1;
4     println!(s1);
5 
6     let i1 = 3;
7     let i2 = i1;
8     println!("{}", i1);
9 }
```

This code will panic at line 4, but not at line 8. The reason for this is that when the string is created and assigned to *s1* on line 2, the variable *s1* becomes the owner of the value "hello". Once *s1* is assigned to *s2*, the ownership of the string is transfered from *s1* to *s2*, dropping *s1* in the process. Thus it is not allowed to use *s1* after line 3, which we try to do at line 4. For data-types with static size, like the integer at line 6, this is allowed. The number 3 will simply just be cloned to the variable *i2* and both variables will be valid.

The reason for this can be explained using pictures. The pictures are borrowed from the Rust book[3]. In figure 1, one can see how *s1* is represented in memory.

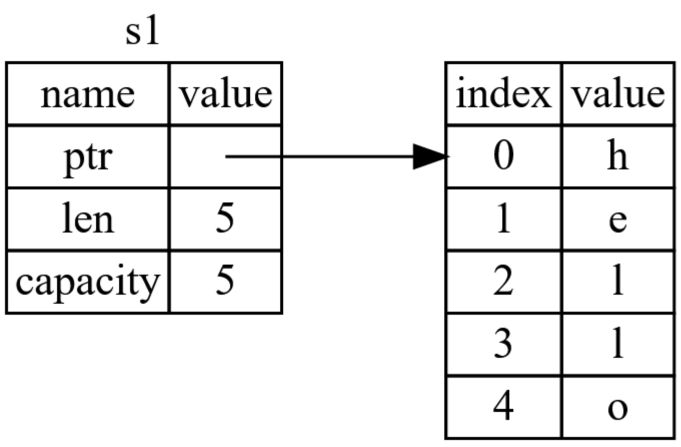

Figure 1: Representation of the variable *s1* in memory.

When we execute line *s2* we have three options:

- Create the variable *s2* and have it point to the same value, as seen in figure 2
- Create a clone of the value and let *s2* point to that clone, as seen in figure 3.
- Create the variable *s2* and have it point to the value and dropping *s1* in the process, as seen in figure 4

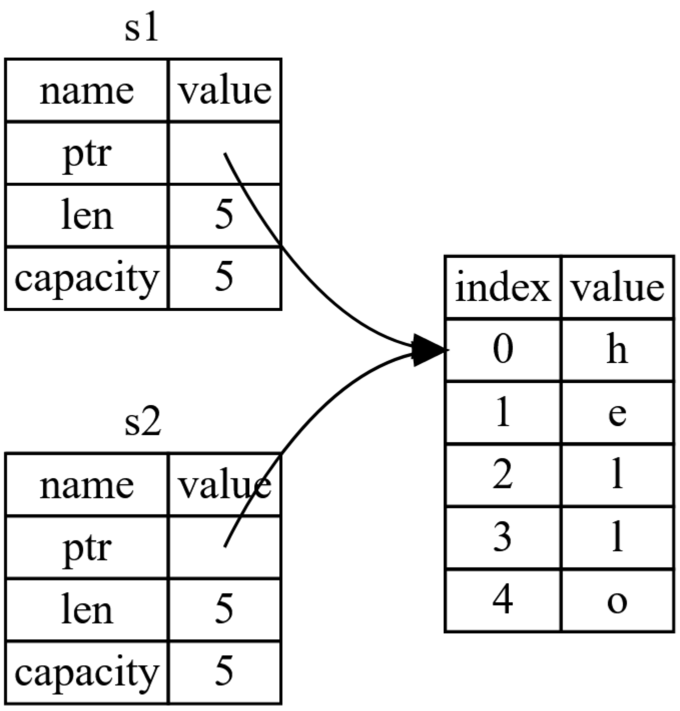

Figure 2: *s2* simply creates a new pointer to the same value, while *s1* keeps it's pointer.

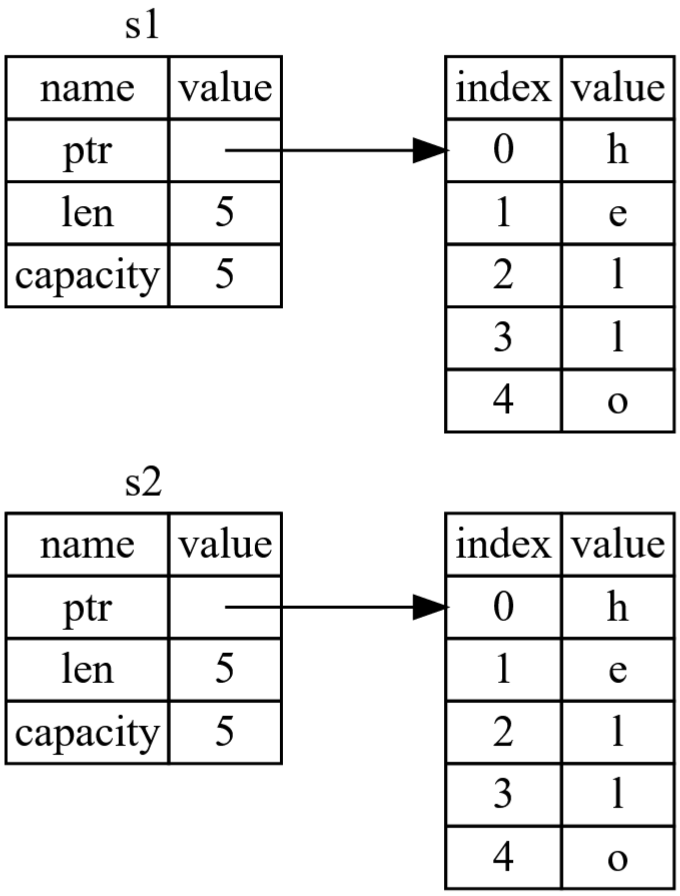

Figure 3: The value is cloned, and *s2* starts pointing towards that new value.

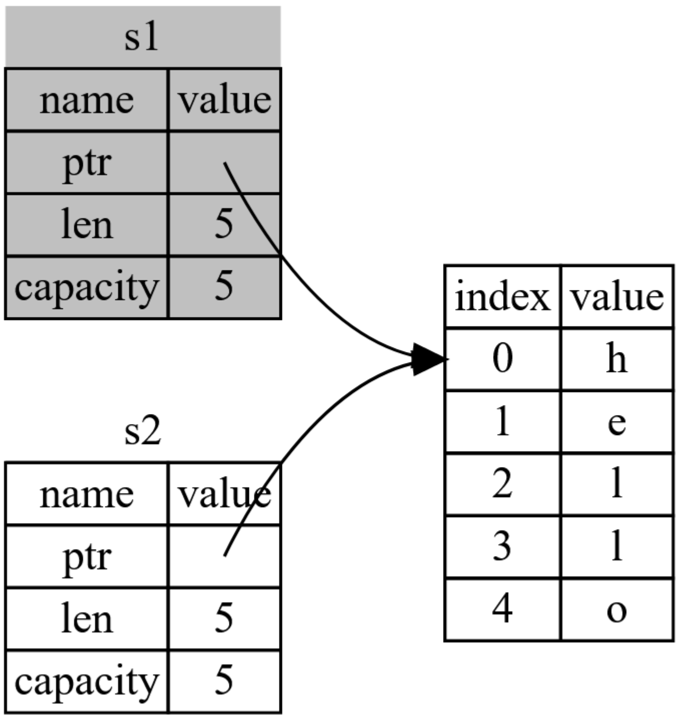

Figure 4: The variable *s2* starts pointing towards the value and*s1* is dropped.

The first option is problematic because of the classic C problem double-free[4]. If Rust attempts to free both variables at two different times, it could potentially lead to memory leaks as one instance had then already freed the memory from the heap. It creates inconsistency, so this option is not viable. The second option is doable but this could be a very expensive operation if the data-type is big enough. When it comes to values with static sizes this is doable as they are often very small. This leaves us with the last option which is also the way Rust implements memory management. *s1* is dropped and the value is *moved* to *s2*, which becomes it's new owner. This is the most memory secure way and leaves no inconsistency. The compiler will ensure that the value *s1* can't be used again, and there will be a compile error if the programmer tries to use *s1* again. The second option could be done manually if one wishes, utilizing the *clone* function for Strings as seen in the following snippet:

```rust
fn main() {
    let s1 = String::new("hello");
    let s2 = s1.clone();
    println!(s1);

    let i1 = 3;
    let i2 = i1;
    println!("{}", i1);
}
```

Ownership can also be transferred to other functions. Consider the following code:

```rust
1 fn another_function(transfer: String) {
2     println!("Recieved transfer {}", transfer);
3 }
4
5 fn main() {
6     let transfer = String::new("I will be transferred!");
7     another_function(transfer);
8     println!("{}", transfer);
9 }
```

This function will panic at line 8, since we cannot use the variable *transfer* after we have transferred the ownership to the function *another_function*. Once the function *another_function* goes out of scope, the value of transfer will be dropped and no longer in memory. Although if *transfer* was a data-type with static size, this code would not panic as the variable *transfer* would just be cloned instead. If the ownership was not transferred, it would create ambiguity of when the value should be dropped. It could either be at the end of the functions scope, or at the end of the current scope. By transferring the ownership the ambiguity is removed and the value is dropped at the end of the functions scope.

2.2.3 Borrowing

There are situations where one would like to use a variable in another function without passing that function ownership of the object. This can be done with borrowing, which creates a reference to the object instead of transferring ownership. Using the same example as earlier, we can rewrite it with borrowing in order to make it work.

```rust
fn another_function(transfer: &String) {
    println!("Recieved transfer {}", transfer);
}

fn main() {
    let transfer = String::new("I will be transferred!");
    another_function(&transfer);
    println!("{}", transfer);
}
```

The & symbol creates a reference to the object after the & symbol, in this case *transfer*. Thus a reference to transfer is sent into the function instead of transferring the ownership to the function. In order to make this work, the argument for *another_function* needs to be changed to allow references. Mutable variables can also be borrowed, but there are some rules to keep in mind.

- At any given time, you may only have one mutable reference or multiple unmutable references to the same value
- References must always be valid

The rule that references must always be valid is enforced by the compiler so it is not possible to create by accidents. Unless the programmers is writing "unsafe" Rust code, which we will explain further in the next sub-chapter.

2.2.4 Unsafe Rust

It is possible in Rust to make code "unsafe"[5]. This means that some safety-checks can be disabled and the responsibility of memory safety falls onto the programmer. The reason to be able to do this, according to the book, is that the compiler could sometimes be too restrictive with what it allows. Sometimes it can not compile code which is undeniably safe since the compiler could not determine the safeness. Unsafe Rust is more of a side note in this project, as it will not be used for benchmarking. But it is important for the reader to know about unsafe Rust as it will be discussed later in the report.

2.3 Buffer overflow attack

A buffer overflow attack is an attack where a malicious actor tries to insert vulnerable code into a system, crash a service or corrupt data by overflowing a buffer according to OWASP[6]. OWASP also states that buffer overflows are very hard to discover and is one of the greatest challenges in creating a memory safe program.

The buffer overflow attacks are made possible by not employing any bounds checks when writing to a buffer. In a classic buffer overflow attack, a buffer is overflowed in such a way where the return pointer of a function instead points to a malicious function that exploits the system in numerous ways, like gaining unlimited superuser access. In the following sub-chapters we will explore how buffer overflows are made in the C and Rust language, and also ways to mitigate these attacks in respective languages.

Buffer overflows can come from other places than a malicious actor finding an exploit in code. This is described by William Stalings and Lawrie Brown in Computer Security: Principles and Practice [7] when they point out that buffer overflows does not always have to come from an attack. They can also occur in a normal program in which any buffer is given more data than the space allocated for it. This could lead to corruption of data, unexpected transfer of control of the program, memory access violation and program termination.

2.3.1 Buffer overflow in the C language

There are multiple ways to do a buffer overflow attack in C. This is accomplished by writing more data to memory than the fixed-length size of memory which was allocated for it. C does not have any built in checks to make sure that the data written to memory is not larger than the allocated space. This means that it is up to the programmer to add these checks and make sure that inputs is of the right size. An example of this is some standard functions in C such as *strcpy*, *gets* and *sprintf*. These functions does not check the length of the inputs and just writes it to memory.

Example code where a buffer overflow attack would be possible:

```c
#include <string.h>

int main(int argc, char **argv)
{
   char input[20];
   gets(input);
   printf("%s\n",input);
   return 0;
}
```

The above code shows one of the most common buffer overflow attacks which is the stack buffer overflow attack. The goal with overflowing the stack is usually to change the return address of the stack frame to point at some malicious code. This is done by writing data of a size greater than the buffer with enough data to in turn change the return address. This address could then point to a shellcode, which was included in the data written to the buffer.

Other buffer overflows could arise from programs not deallocating memory when it has been used with the free() method. This means that the program will allocate new parts of memory while the old allocations is still there unused, which will lead to the system memory being full and the system crashing.

2.3.2 Buffer overflow in Rust

In safe Rust, it is not possible to create a buffer overflow attack. There are bounds checks built into the language to make sure that this does not happen. The borrow checker is also very good at making sure you are not leaving dangling references or forgetting to deallocate memory, which is done automatically. This is further improved by having all variables immutable by default, as an immutable variable cannot cause a buffer overflow (since it cannot be written to). In unsafe Rust however, it is possible to create a buffer overflow attack.

An example of a buffer overflow in unsafe Rust can be seen in the following snippet:

```rust
#[repr(C)]
struct Hackvist {
    buffer: [u8; 16],
    point: *const fn(),
}

fn main() {
    let mut args: Vec<OsString> = env::args_os().into_iter().collect();
    let first_arg: OsString = args.remove(1);
    let input_bytes: &[u8] = first_arg.as_bytes();
    let mut hackvist = Hackvist {
        buffer: [0; 16],
        point: 0 as *const fn() -> (),
    };

    unsafe {
        std::ptr::copy(
                input_bytes.as_ptr(),
                hackvist.buffer.as_mut_ptr(),
                input_bytes.len(),
            )
        }

    if hackvist.point as usize == 0 {
        println!("Try again");
    } else {
        let code: fn() = unsafe { std::mem::transmute(hackvist.point) };
        code();
    }
}
```

This code snippet was created by Tomasz Guz[8]. What is being done here is that the program takes a string as an input, and then does an unsafe copy of that input into the *hackvist* variables buffer field. The last argument to this function specifies the size of the variable being copied. Since we are running inside an *unsafe* scope, there are no checks to see if this would overflow the variable or not. When code inside a *unsafe* scope is run, Rust trusts that the programmer knows what is being done and that the programmer can vouch for that it is safe. This leaves us with an overrun of the buffer field of the *hackvist* variables, which leaks into the point field. This may enable us to run arbitrary code as we then later execute the point field. If we were running this outside the *unsafe* scope, we would get a compiler error since we would not be allowed to manipulate the memory manually.

2.4 Buffer overflow prevention

Aside from the security mechanisms given in Rust and the additions to C other implementations have been developed to prevent or detect buffer overflow attacks. Some of these implementations are discussed by Stallings and Brown [7] and are categorized as compile time defenses and run time defenses.

2.4.1 Compile time defenses

Compile time defenses focuses on preventing buffer overflow attacks before the code is being run. The prevention mechanism could therefore be built into the compiler or in some other way try to detect or prevent the overflow.

The choice of programming language can help in preventing buffer overflows. When using a language like Rust with mechanisms to handle allocation of memory and to prevent buffer overflows. In contrast languages like C instead leaves this to be handled by the programmer and trusts that the programmer knows safe coding techniques. But as Stallings and Brown writes it has been shown that programmers can't always be trusted to write safe code. This is were the use of languages with safety mechanisms to ensure memory safety are useful to ensure that programs are safe.

Another part of programming safe code is to use libraries that are memory safe and can handle buffer overflow attacks. As described above C has standard functions that allow for an arbitrary amount of data as input that is then written to the buffer. One way to battle the use of unsafe standard functions has been to switch to using safe versions of the standard functions. This solutions has one problem though: they only works for new programs. While this is good to have there are solutions needed for already existing programs with bugs and weaknesses that do not require rewriting code.

Other ways of protecting from buffer overflow attacks are so called stack protection mechanisms. One of these mechanisms is called a stackguard and it writes a canary below the frame pointer in the stack. When the function returns, a check is made to see if the canary has been modified and will terminate execution if a modification is detected. Of course the stackguard method is not perfect and Stalling and Brown explain that the mechanism can interfere with debugger when analyzing the stack. Therefore stackguard only works with updated debuggers. The stackguard can also be overcome since for example an attacker could write the canary in the code and then no difference would be found. So Stallings and Brown discuss different variants of the stack protection that has been developed. One method saves the return pointer when the allocation has been made and saves it in a safe part of memory. Then, just like with the canary, the return address is checked when the function returns against the saved return address and if they don't match the process is terminated.

2.4.2 Run time defenses

Since code has already been written with bugs and exploits in it there has been a need for ways to defend against buffer overflow from outside the code. Run time defenses tries to solve this problem by using OS mechanisms to protect from buffer overflows.

One of the ways to protect in run time is to make the stack non-executable. This means that the OS is told that the data on specified stack frames are not to be executed. By blocking execution of the stack an attacker cannot place code when overflowing the buffer and then using the return address to execute that code.

To make a buffer overflow attack harder Stallings and Brown also presents address space randomization. Address space randomization is a method in which the location of the stack is moved in a randomized manner based on the process. This makes the prediction of the return address much harder for an attacker. An extension of the address space randomization that is mentioned is to randomize the order in which standard libraries are loaded for programs. Since these are usually loaded in the same order and manner, their addresses are easier to predict and attackers could instead try to use the libraries instead of the standard stack overflow attack with code in the buffer.

3 Method

3.1 Prototypes

The approach is to create equal safe prototypes in both C and Rust, and then to compare both the build time and and the run time with each other and then analyse the difference. The prototypes needs to do the exact same thing to make it as fair a comparison as possible, and there should not be any unnecessary code in any of the prototypes unless they have the same unnecessary code.

The prototypes should also be memory safe. This is because we want to compare the languages when both are considered safe and in an optimal state, which is what it (at least in theory) should look like in a real scenario. In this way it is possible to test if Rust is either slower or faster than C when C is using safe functions.

All C prototypes will use the safe versions of functions that could cause a buffer overflow if the unsafe version was used. This is also to be able to test if the extra mechanics introduced by C in these safe versions makes the program considerably slower when compared to Rust. Some programs should also be running what they are doing inside a loop. This is to be able to compare the run-time performance more easily, as it would be harder to detect any differences if the functions were only called once per run. To be able to detect compilation differences in the same way, some prototypes will have multiple calls to the same function as well.

Along with everything else mentioned, the prototypes should be compiled at the most optimized level since it is also how it will be compiled in real applications. There is no use to compare debug compilations, as C and Rust may differ in what the languages determines to be optimized enough for debugging.

3.1.1 The *shelloverflow* prototype

The purpose of this program is to mimic how a shell login could look like, and will also act as the most basic prototype with only a few calls to a secure function. The function being benchmarked in C here is the *fgets* function which sibling, the *gets* function, could introduce a buffer overflow. Rust does not have a one-to-one function to easily replace this one, but we have implemented the most basic equivalent of this behaviour using the *std::io* package in Rust. The C code for this prototype is borrowed from the course Computer Security (EDA263), but is heavily modified.

3.1.2 The *print* prototype

The purpose of this prototype is to test the compilation times. This program takes an command-line argument and then prints the argument given. The secure function being used here is a function which we implemented ourself, the *safe_copy* function. What this funcion does is using the insecure function *strcopy* to copy a string to a buffer, but with the added functionality of checking the bounds. Otherwise this function would introduce a buffer overflow. The Rust equivalent function being used here is *format!*.

To be able to check of this impacts the compilation time, we will compile the code using different amounts of calls to the function. A loop should not be used, as it would not be the same as having all the calls after one another. Or it will make no difference if the compiler uses the concept known as “loop unrolling”.

3.1.3 The *addhostalias* prototype

This prototype will mainly check the run time. This program takes three inputs, and then writes the inputs sequentially to a local file. The secure function in C being used here is *snprintf*, with it’s sibling *sprintf* being able to introduce a buffer overflow. The Rust equivalent function being used is once again *format!*.

To test the run time, we will run the function that formats the string and writes to the file multiple times using a loop. This will not only check if the secure function is fast, but also how fast Rust is compared to C when it comes to writing to a file. Thus we also get a more generic benchmark. There is also multiple versions of this prototype, where the number of iterations in the loop differ. The C code for this prototype was also borrowed from lab 2 in this course as well.

3.2 Benchmarking

To benchmark the prototypes there will exist two scripts: one to benchmark the compilation-time and another one to benchmark the run-time. The scripts will utilize “Makefiles” to do the compilation. In order to make the comparison as fair as possible all prototypes must use a Makefile to compile, since the Makefile could add some overhead to the build process. If all prototypes then use a Makefile, the additional overhead would be the same for all builds. The script will also accept as input the number of iterations the prototype should run/compile, and will then write the results of each iteration to a file.

To further ensure that the comparison is fair, all benchmarks must run on the same machine. If the benchmarks took place on different machines we would get wildly different times. Another problem is that it is hard to ensure that the machine is having the same workload during each benchmark, which we will return to in the discussion chapter.

For the benchmarking purposes we will also disable warnings generated by each application. The reason for this is that we do some stuff that the programs do not like. For example in our *print* prototype for Rust, the compiler is not happy with us calling *format!* with the same arguments multiple times. To not make the printing have any effect on the timings, we thus disable all warnings in both Rust and C.

4 Result

Here the results from the benchmarks are presented. All benchmarks from the same prototypes are put into the same diagram, to be able to compare them easier. Each benchmark used 200 iterations.

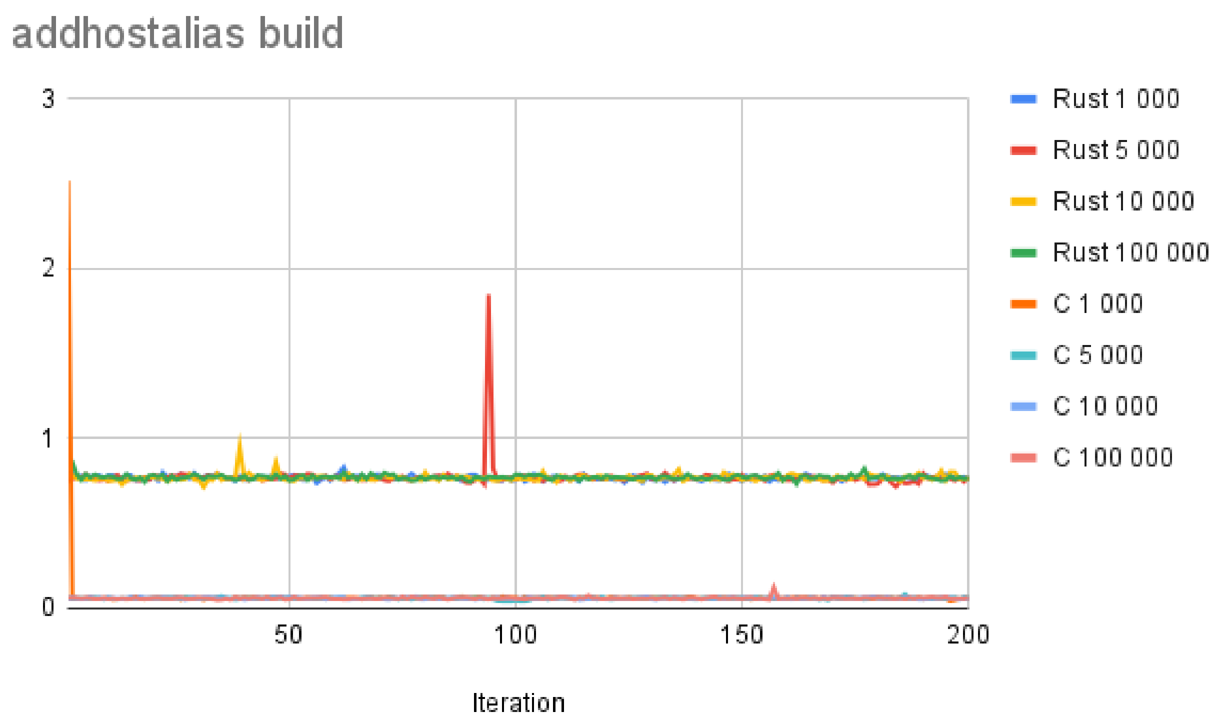

Figure 5: The build times for the *addhostalias* prototype. The y-axis represents the time in seconds the build took, and the x-axis is the iteration number. The number after the language for the series represents how many iterations there are in the loop that calls the function.

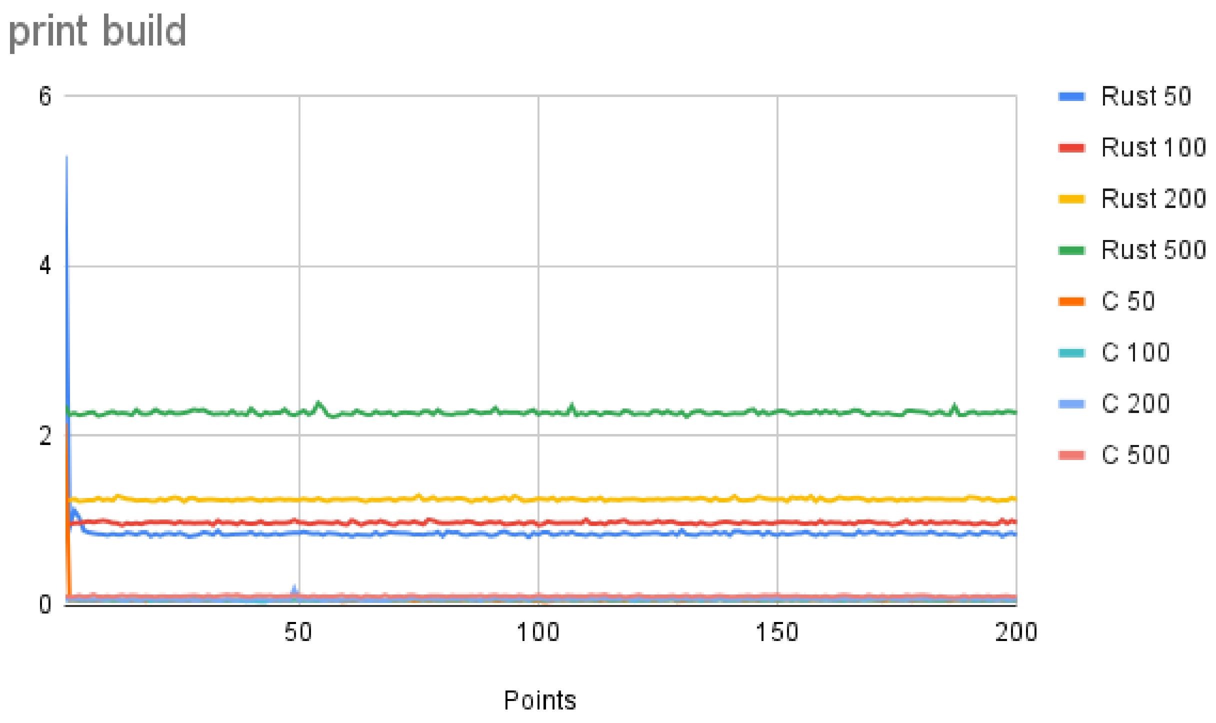

Figure 6: The build times for the *print* prototype. The y-axis represents the time in seconds the build took, and the x-axis is the iteration number. The number after the language for the series represents how many calls to the safe function is programmed.

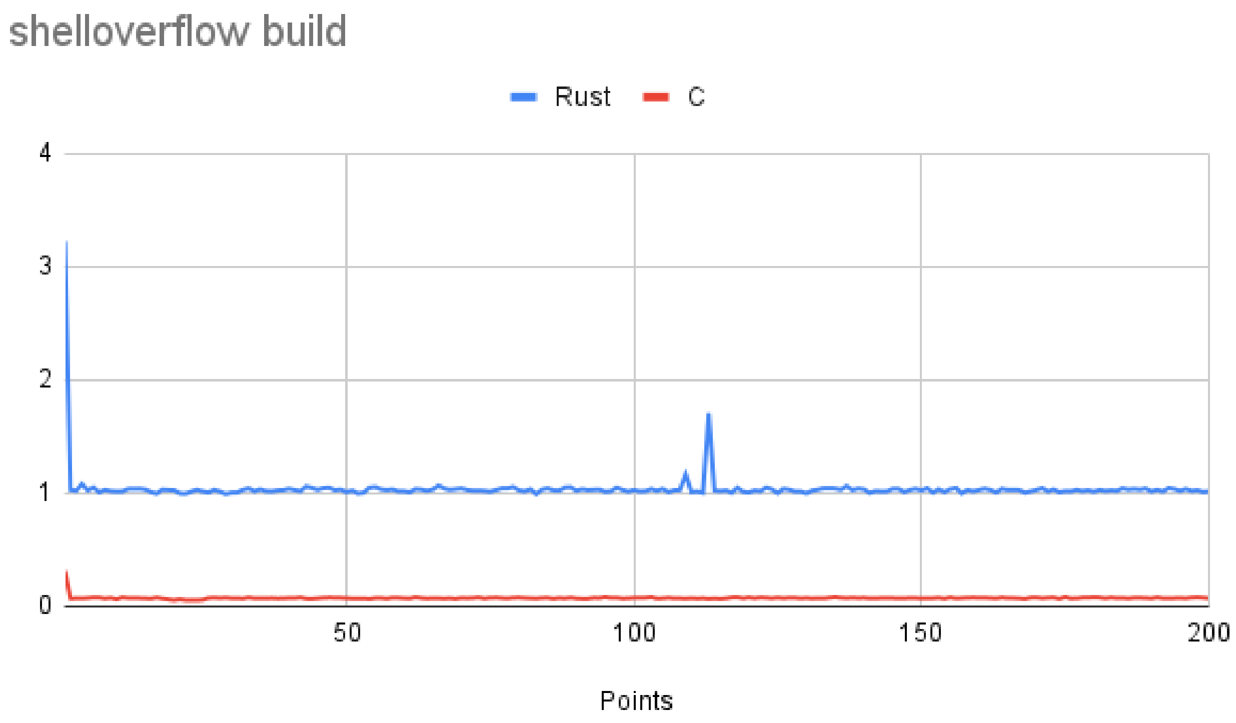

Figure 7: The build times for the *print* prototype. The y-axis represents the time in seconds the build took, and the x-axis is the iteration number.

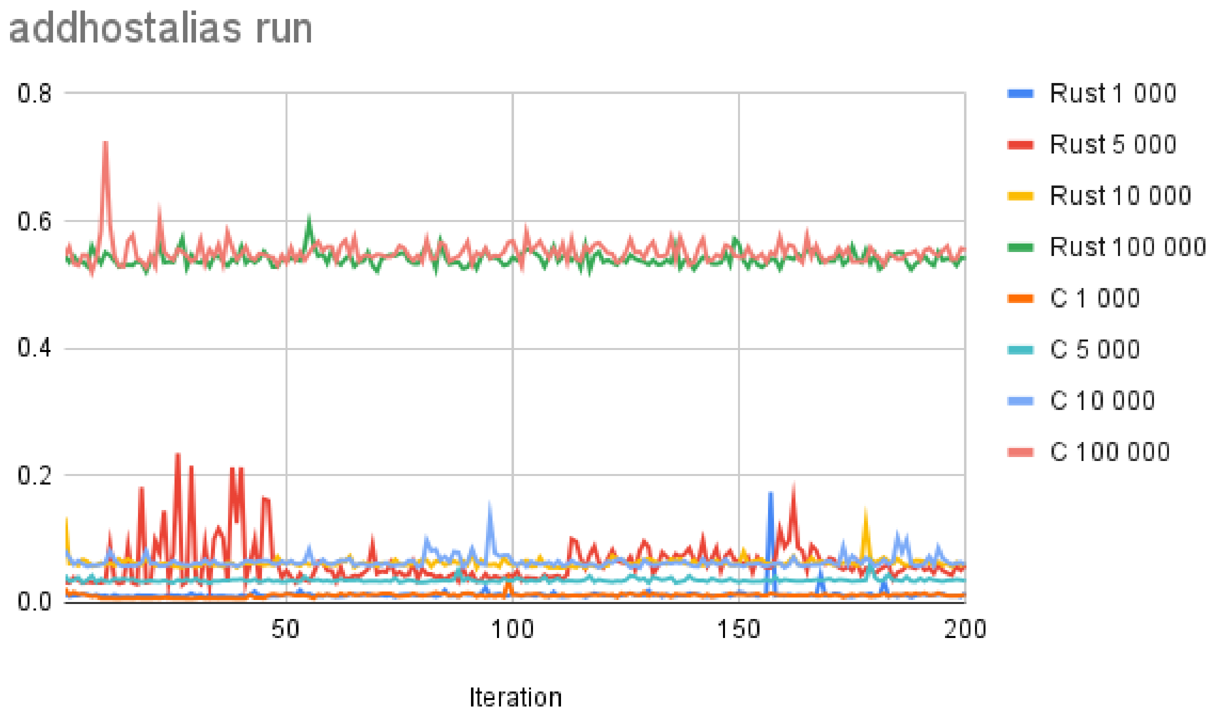

Figure 8: The run times for the *addhostalias* prototype. The y-axis represents the time in seconds the build took, and the x-axis is the iteration number. The number after the language for the series represents how many iterations there are in the loop that calls the function.

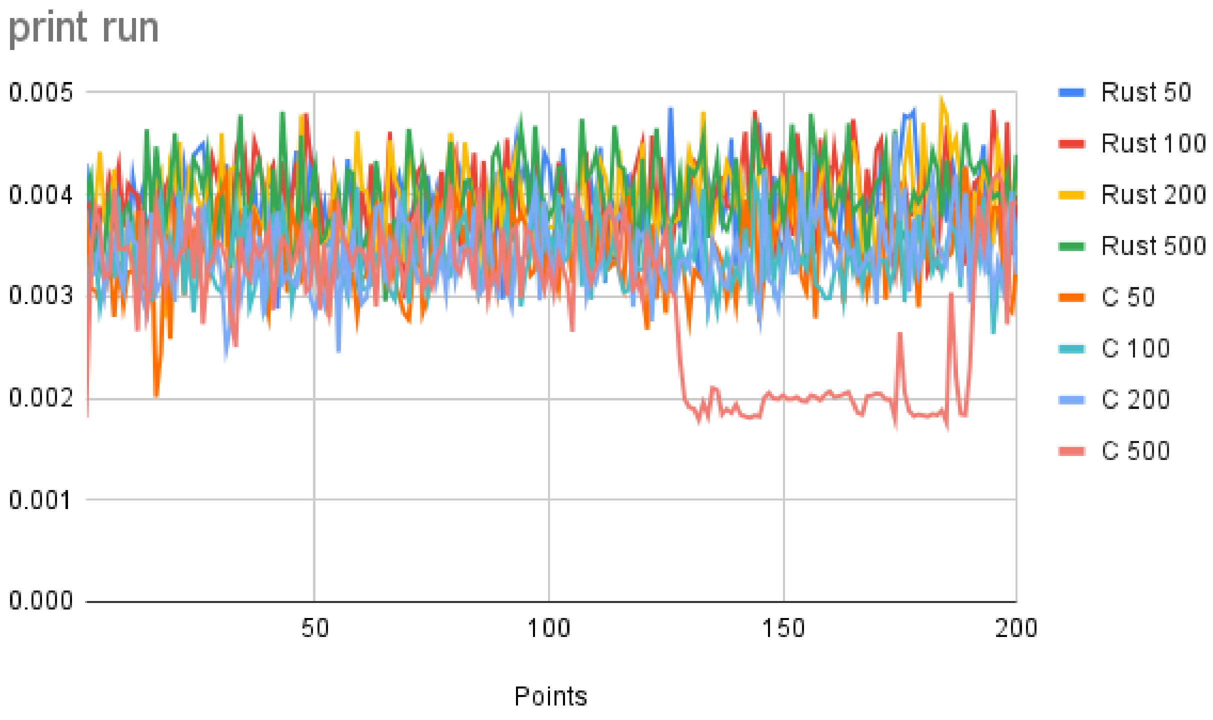

Figure 9: The run times for the *print* prototype. The y-axis represents the time in seconds the build took, and the x-axis is the iteration number. The number after the language for the series represents how many calls to the safe function is made during the run.

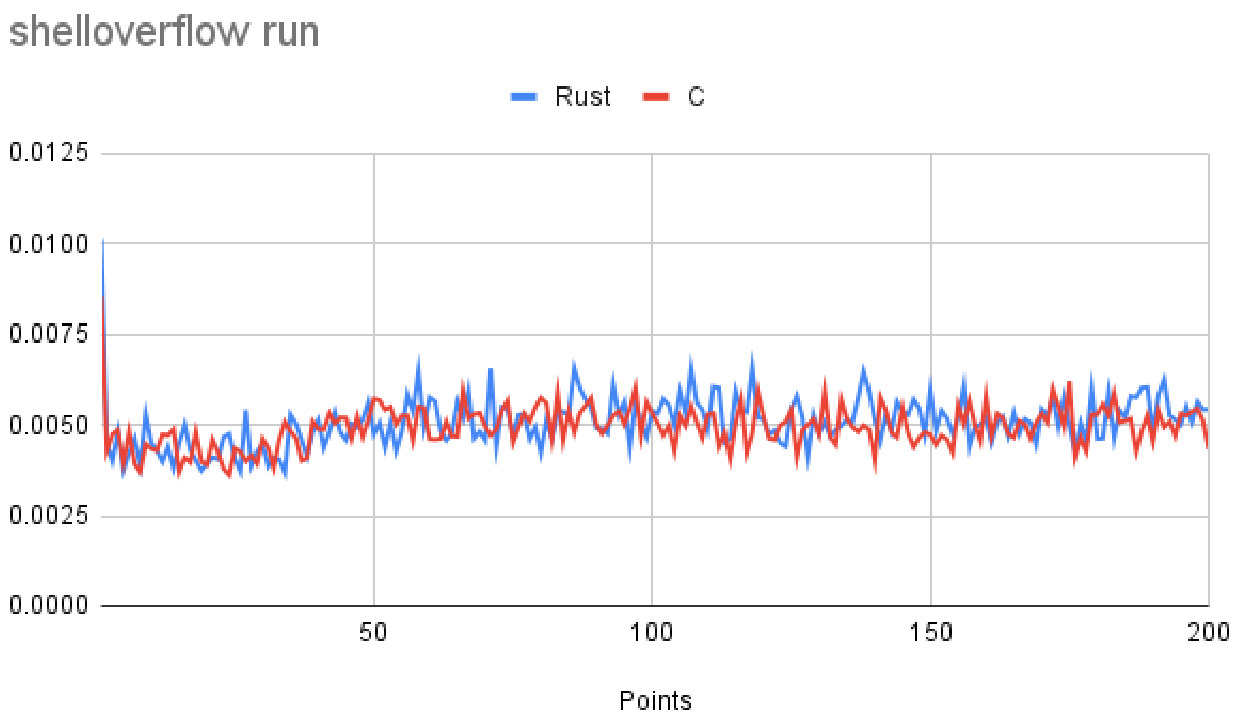

Figure 10: The run times for the *print* prototype. The y-axis represents the time in seconds the build took, and the x-axis is the iteration number.

5 Discussion

5.1 Observations from the results

In this section we will discuss the results when building the prototypes and their run time. We will also discuss the differences found between C and Rust.

5.1.1 Build times

As we can see from the results, C code generally builds much faster than Rust code. This was expected, as the primitives Rust use to ensure memory safety does add another layer of abstraction. The difference in build times are quite large, with the average build times being about 10x greater than C build times. One can also notice a spike in the beginning for the build times for both C and Rust. This could be because of some preparation being done, for example fetching the code to the cache that does not need to be fetched on later compiles.

In the print prototype build times see a significant increase for Rust when increasing the number of calls to the safe function. In the case of Rust the time almost doubles when the function calls goes from 200 to 500, while it only increases by about 20% in C. This could be significant when it comes to projects in a larger scale, and where C clearly has an advantage.

5.1.2 Run times

Regarding the running speed, they are on quite a similar level. This means that the cost of the added abstractions of Rust comes in build time, while run time is similar. The spike in the first runs most likely comes from having to fetch new data to the cache, the same reason for the build times being slow in the first iterations. On the "addhostalias" prototype we can observe that the run times are about the same, except for that the Rust variants had some more jitter. This could be because of the language, or because of the machine running other tasks in the background. It is hard to tell from these results alone.

5.1.3 The machine used

The machine that was used was a home server that apart from running these tests, also ran some generic web servers in the background. This could have impacted the results, such as making the build- or run times take longer. This could be a reason for why there sometimes is a spike in the graphs, and should just be regarded as some random noise.

To make the benchmarking as accurate as possible, one could run this on an isolated machine whose sole purpose is to do this benchmarking. This was unfortunately not possible due to the nature of the project of us not been given any funds or resources.

5.2 Development time

We can conclude that there is an increase of development time using Rust, since the build times takes longer. Although one important aspect to consider is that when building the Rust programs for this project we are using the highest level of optimization. While developing a program, the programmer most likely is using the debug level of optimization which is quicker than building with the highest optimization. The same could although be said about the C builds, but those builds are already quite fast.

In a conducted survey[9], it was made clear by the participants that it could take time to understand how Rust works. The Rust compiler is known to complain a lot and does not always permit code that the programmer knows is secure. This is one of the features which makes the Rust language more secure, but could also be a headache since it could be hard to understand why the code doesn't build. The Rust compiler can also be very helpful in guiding the programmer to fix the code, but for a beginner this could be a hurdle at first.

In the same survey it was made clear that there is a steep learning curve in learning Rust. For companies that already have a team fluent in C, there might be a huge time- and money cost in trying to adapt those developers to Rust which might not be worth it for the added security given by Rust. For some projects it might not even be feasible, depending on deadlines and customer expectations. This was also mentioned by the participants. However, in the same survey the respondents felt that their development skills in other languages increased as they learnt Rust.

Stallings and Brown[7] also brings up the problem regarding business deadlines, but gives a new perspective to it. To be able to create secure application, one needs to apply defensive programming. When developing a program a programmer typically focus on what is needed here and now to make the application work. They thus focus on the normal flow of execution, but leave out certain security aspects. Using defensive programming one should exhaust all possible usages of the code, which could take time. This might go against business pressure to deliver a program in time, which ultimately might lead to no defensive programming and thus maybe introducing critical bugs in the code. This is a situation in which Rust could save both time and money in the long run, by forcing the programmer to program in a memory safe manner as well as not letting the programmer implement code that could cause some sort of undefined behaviour. Stallings and Brown goes on about how there needs to be a change of mindset in the world of computer engineers. As a society, we are not very keen about having bridges collapse or planes falling from the sky, but software development has not yet reached this level of maturity in the society which leads to software faults being much more tolerated.

5.3 Assumptions about input

Buffer overflows are commonly caused by programmers having an assumption about the input, according to Stallings and Brown[7]. When dealing with inputs, programmers typically create a buffer with an arbitrary size like 512 or 1024 and then not creating any measures to ensure that the input is not greater than this buffer.

Many C functions, for example those we have covered in the prototypes like *sprintf*, provides no mechanism to prevent a buffer from being overflowed. To be able to handle these inputs in a safe way, the programmer needs the mindset that input is dangerous and to also take action if the size of the input is to great. When using a language such as Rust, some of these problems are abstracted away and will cause either a run- or build time error instead if they are not handled properly. It is naive to expect every programmer to be aware of what functions are safe or not safe to be used. There will always be a junior programmer who is naive about the use of for example the function *sprintf*, and by having that abstracted away in Rust it leaves a smaller room for errors.

5.4 Security

An important aspect when comparing Rust and C is in the trade off in development and build time compared to the security benefits that comes with the Rust language natively. As programmers are pressed for time the benefit of having a compiler and language that makes sure that your code is memory secure is quite huge. Altough Rust is slower than C, we can undeniably argue that as a whole the Rust language is much more secure than the C language. There are simply more hoops that one needs to jump through to make a memory flaw in Rust.

It is a trade-off one should consider. What is more valuable, fast development time or higher grade of security? One important aspect to not forget is that Rust is not safe against bad programming practices. While Rust guarantees that buffer overflow attacks are not possible, Rust does not safeguard against different types of injection attacks or bugs in the code. This is still up to the programmer to safeguard against. According to William Stallings and Lawrie Brown[7] it is also possible for programmers to forget code used for debugging in the production stages, which can introduce unforeseen vulnerabilities. Neither Rust nor C protects against this natively, and it is up to the programmer to keep this in mind.

For example, in our *addhostalias* prototype we are able to write whatever we want into a specific file. Rust does not take any action whatsoever to ensure that the input is valid, because Rust could not possibly know what type of input it should expect. The file in the prototype is harmless to write to, but if we consider the case that the program writes to a file where the owner is root with the S and X bits set one could do enormous damage to the system. But again, this is not expected of neither the Rust nor C languages to handle. It is just an example that one should not be naive and expect Rust to handle every possible security flaw. Good programming practices are still needed and expected.

5.5 Unsafe Rust

One feature that speaks against Rust being a memory safe language is the option to disable much of Rusts memory safety by wrapping the code inside an *unsafe* block. Some argue that this makes the memory safety features of Rust pointless, since an unknowing programmer could still make a memory insecure function that could cause leaks or other undefined behaviour. However, unsafe Rust is necessary in the language to be able to run certain code on a lower level where memory manipulation is necessary in order to use certain features of the machine. For example the microcomputer MD407 used by Chalmers needs the programmer to access some very specific memory addresses to be able to manipulate certain registers on the microcomputer. If unsafe Rust didn't exist, it would not be possible to run Rust code on the MD407.

An argument in favour of unsafe Rust is that even though it allows the programmer to program unsafely, it is another hurdle one has to go over in order to program unsafely. As a programmer you need to explicitly tell the compiler that you are about to do something that may be unsafe, and that you are also aware that you are doing something unsafe. Unsafe code also cascade upwards, with unsafe code only being callable by an unsafe function. This means that it is hard to accidentally call an unsafe function, and thus the programmer is always aware of when something that might be undefined is being done. If a program introduce some memory problems it's easier to just check the code that is marked *unsafe*, in contrary to C where every function could potentially be an unsafe function.

This line of argument does however not negate the fact that unsafe Rust is still as the name implies unsafe, and reduces the safety of the language as a whole just because of it's presence. But compared to the alternative (C) it is still, in our opinion, a better way to handle it. Rust as previously mentioned should also not be assumed to be 100% secure. Rust is a great tool, but not a solution for every problem.

6 Conclusion

To conclude this project report we can see that C and Rust are two languages that have been built with different goals in mind. Rust with its strong standards to ensure memory safe and C that leaves the handling of memory to the programmer which gives freedom but also puts a lot of trust in the programmer. It can also be seen that fixing bugs related to buffer overflow can be hard since code can't always be rewritten in a new language and therefore other solutions needs to be used. But from the results we conclude that using a language like Rust can improve the security of a program by having the compiler handle memory safety and not the programmer. Although the compile time and handling of the compiler might take more time in Rust, this could be saved by not having to search for buffer overflow weaknesses.

A Contribution

Simon has mainly worked on refining the prototypes and doing the benchmarks. He has also worked on integrating the results and prototypes into the report, as well as doing research on the Rust language. He has also worked on discussing the implications of the results.

Emrik has mainly worked on researching the aspects of the Rust language considering development and productivity, as well as the implications. He has also done research on the buffer overflow subject. He has also worked on researching the C language.

Both feel that the work has been equal.

B shelloverflow prototypes

Rust prototype

```rust
use std::io;

#[allow(warnings)]
fn main() {

    let important1 = "**IMPORTANT1**".to_string();

    let salt = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789./";

    let important2 = "**IMPORTANT2**".to_string();

    println!("Starting..");

    println!("user: ");
    let mut stdin = io::stdin();
    let input = &mut String::new();

    input.clear();
    stdin.read_line(input);

    let user = input.trim().to_string();
    if user.is_empty() {
        return;
    }

    println!("password: ");
    input.clear();
    stdin.read_line(input);

    let password = input.trim().to_string();

    println!("Your password was {}", password);
}

```

C prototype

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <stdio_ext.h>
#include <string.h>
#include <sys/types.h>

#define LENGTH 16

int main(int argc, char *argv[]) {
        char important1[LENGTH] = "**IMPORTANT 1**";

        char salts[] =
                        "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789./";

        char user[LENGTH];

        char important2[LENGTH] = "**IMPORTANT 2**";

        printf("Starting..\n");

        //char   *c_pass; //you might want to use this variable later...
        char prompt[] = "password: ";
        char user_pass[LENGTH];

        printf("user: ");
        fflush(NULL); /* Flush all  output buffers */
        __fpurge(stdin); /* Purge any data in stdin buffer */

        //Use fgets instead of gets
        if(fgets(user, LENGTH, stdin) == NULL){
                exit(0);
        }

        //To prevent buffer overflow, we replace the enter key with end string
        for(int i = 0 ; i < LENGTH ; i++){
                if(user[i] == '\n'){
                        user[i] = '\0';
                }
        }

        if(user == NULL) {
                exit(0);
        }

        printf("password: ");
        fflush(NULL); /* Flush all  output buffers */
        __fpurge(stdin); /* Purge any data in stdin buffer */

        //Use fgets instead of gets
        if(fgets(user_pass, LENGTH, stdin) == NULL){
                exit(0);
        }

        //To prevent buffer overflow, we replace the enter key with end string
        for(int i = 0 ; i < LENGTH ; i++){
                if(user_pass[i] == '\n'){
                        user_pass[i] = '\0';
                }
        }

        if(user_pass == NULL) {
                exit(0);
        }
        printf("Your password was %s\n", user_pass);

        return 0;
}


```

C print prototypes

Rust prototype

```rust
use std::env;

#[allow(warnings)]
fn main() {
    let args: Vec<String> = env::args().collect();
    if args.len() != 2 {
        println!("Usage: {} text", args[0]);
        return
    }

    let mut formatbuffer: String;

    formatbuffer = format!("{}", args[1]);
    formatbuffer = format!("{}", args[1]);
    formatbuffer = format!("{}", args[1]);
    formatbuffer = format!("{}", args[1]);
    formatbuffer = format!("{}", args[1]);

    ...

    formatbuffer = format!("{}", args[1]);
    formatbuffer = format!("{}", args[1]);
    formatbuffer = format!("{}", args[1]);
    formatbuffer = format!("{}", args[1]);
    formatbuffer = format!("{}", args[1]);

    println!("You wrote: {}", formatbuffer);

}


```

C prototype

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

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s text", argv[0]);
        exit(0);
    }

    char buffer[1024];
    safe_copy(buffer, argv[1], 1024);
    safe_copy(buffer, argv[1], 1024);
    safe_copy(buffer, argv[1], 1024);
    safe_copy(buffer, argv[1], 1024);
    safe_copy(buffer, argv[1], 1024);

    ...

    safe_copy(buffer, argv[1], 1024);
    safe_copy(buffer, argv[1], 1024);
    safe_copy(buffer, argv[1], 1024);
    safe_copy(buffer, argv[1], 1024);
    safe_copy(buffer, argv[1], 1024);

    printf("You wrote: %s \n", buffer);
    exit(0);
}

void safe_copy(char* buffer[], char* string[], int size) {
    if (strlen(string) < size) {
        strcpy(buffer, string);
    }
}

```

D addhostalias prototypes

Rust prototype

```rust
use std::env;
use std::io::Write;
use std::fs::OpenOptions;

const HOST_FILE: &str = "./file.txt";

#[allow(warnings)]
fn add_alias(ip: &str, host: &str,  alias: &str) -> std::io::Result<()> {
    let formatbuffer: String;

    formatbuffer = format!("{} {} {}", ip, host, alias);

    let mut file = OpenOptions::new()
        .write(true)
        .append(true)
        .open(HOST_FILE)?;

    writeln!(file, "{}", formatbuffer)?;
    Ok(())
}

#[allow(warnings)]
fn main() -> std::io::Result<()> {
    let args: Vec<String> = env::args().collect();
    if args.len() != 4 {
        println!("Usage: {} ipaddress hostname alias", args[0]);
        return Ok(());
    }

    for n in (1..101) {
        add_alias(&args[1], &args[2], &args[3])?;
    }
    Ok(())
}
```

C prototype

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


#define HOSTNAMELEN 256
#define IPADDR      1
#define HOSTNAME    2
#define ALIAS       3

```

```c
#define HOSTFILE "./file.txt"


void add_alias(char *ip, char *hostname, char *alias) {
  char formatbuffer[256];
  FILE *file;

  snprintf(formatbuffer, "%s\t%s\t%s\n", ip, hostname, alias, 256);

  file = fopen(HOSTFILE, "a");
  if (file == NULL) {
    perror("fopen");
    exit(EXIT_FAILURE);
  }

  fprintf(file, formatbuffer);
  if (fclose(file) != 0) {
    perror("close");
    exit(EXIT_FAILURE);
  }
}


int main(int argc, char *argv[]) {
  if (argc != 4) {
    printf("Usage: %s ipaddress hostname alias \n", argv[0]);
    exit(EXIT_FAILURE);
  }

  for (int i = 0; i< 100; i++) {
    add_alias(argv[IPADDR], argv[HOSTNAME], argv[ALIAS]);
  }
  return(0);
}


```

References

[1] A. One. *Smashing The Stack For Fun And Profit.* Fetched 04-05-23. URL: https://insecure.org/stf/smashstack.html.

[2] S. Klabnik and C. Nichols. *Understanding Ownership.* Fetched 18-04-23. URL: https://doc.rust-lang.org/stable/book/ch04-00-understanding-ownership.html.

[3] S. Klabnik and C. Nichols. *What is Ownership?* Fetched 18-04-23. URL: https://doc.rust-lang.org/stable/book/ch04-01-what-is-ownership.html.

[4] *Doubly freeing memory.* Fetched 18-04-23. URL: https://owasp.org/www-community/vulnerabilities/Doubly_freeing_memory.

[5] S. Klabnik and C. Nichols. *Unsafe Rust.* Fetched 18-04-23. URL: https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html.

[6] T. O. foundation. *Buffer Overflow.* Fetched 18-04-26. URL: https://owasp.org/www-community/vulnerabilities/Buffer_Overflow.

[7] L. B. William Stallings. *Computer Security: Principles and Practice, Fourth Edition.* Pearson Education, 2018.

[8] T. Guz. *Playing with buffer overflow in Rust.* Fetched 18-05-04. URL: https://tgrez.github.io/posts/2022-06-19-buffer-overflow-in-rust.html.

[9] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek. "Benefits and drawbacks of adopting a secure programming language: rust as a case study". In: *Symposium on Usable Privacy and Security.* 2021.