THESIS

CHALLENGES IN DISASSEMBLING RUST BINARIES

STYLIANOS SOFOKLEOUS

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2023

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

Challenges in Disassembling Rust Binaries

Stylianos Sofokleous

Supervisor

Dr. Elias Athanasopoulos

Acknowledgement

This thesis would not have been possible if it were not for the support from my supervisor instructor Elias Athanasopoulos. Dr. Athanasopoulos introduced me to the field of software analysis on my 3rd year of studies with his subject “Software Analysis” where I gained a lot of knowledge which without this thesis would not be possible. He is also the reason why I chose to pursuit a career in computer security and why I chose him as my supervisor.

I would also like to thank my family for the ethical support they gave me and reminding me that I can do whatever I set my mind to.

Finally, I would like to say a big thank you to all my friends that we spent countless hours advising each other and helping throughout the four years of our undergraduate studies.

Abstract

This paper presents a disassembly approach to analyze the Rust programming language and detect features such as Trait Objects. Trait Objects, which allow for dynamic dispatch, can be difficult to detect in Rust binaries. We propose a program that can disassemble Rust binaries and extract information about Trait Object calls. We also discuss the challenges involved in detecting Trait Objects and how our approach overcomes them. Our program provides a useful tool for developers to understand the behavior of Rust binaries and potentially improve their performance. The results of our evaluation show the effectiveness of our approach and the potential for further research in this area.

Contents

Chapter 1

Introduction

Whenever we talk about computer science or any form of computer programs, the first thing that comes to mind is the computer programming languages that developers use to implement their ideas into functional applications. There are many languages one can choose from to better match their specific needs and that suit the application. One common thing amongst developers is the requirement of their applications to run as fast as possible while being reliable and secure. One area where speed and performance really make an impact is operating system kernels and other low-level software that work very closely with the hardware. Traditionally, when developers wanted to build a performance critical system, C and C++ were the first choice that came to mind since they provide a lot of performance with a lot of flexibility in terms of memory management. All that performance and memory freedom though comes at a cost in terms of security and safety of the final products produced by these languages. Other programming languages were developed that are more secure and safe but with the disadvantage of being slower to the point where they cannot be used to write performance critical code. Thankfully, there are two other languages that seem to successfully provide the safety and security that C and C++ lack but also retain the performance of those languages. These languages are Rust and Go languages.

This thesis will focus on the Rust language and more specifically, on its disassembling and its Traits [4]. Rust language implements Traits to provide support for sharing behaviors between types. In essence, Traits is how Rust manages to implement object-oriented programming in its own way. Identifying such crucial features of a language can say a lot about the binary itself. Many reverse engineering frameworks such Marx[1] have been developed for the C++ language to extract information about the dynamic dispatch of virtual classes. As Rust gains more popularity [2] , it increases the necessity of a disassembler that can identify its core structures and help security experts and reverse engineering to better understand an application written in Rust. Rust is still a new language which is very attractive to adversaries who want to create malware that goes undetected under anti-viruses [3].

As mentioned before, not many attempts have been made to create specialized decompilers and disassemblers for Rust thus making this field relatively new. In contrast C and

C++ have existed for over 50 years thus many tools exist for these languages that not only extract features from a binary, but they can decompile the assembly code back to its source C syntax [5]. All that presents us with a challenge as there is little to no effort for creating such tools for the Rust language.

In this thesis we will explore how Rust creates Traits, how Traits are dispatched and how we can identify from a disassembled Rust binary that a Trait exists. We will then proceed onto creating a custom disassembler which will extract information from the binary file and indicate possible Traits.

Specifically, in chapter 2 we will go through a brief overview of some terminology we will be using throughout the thesis report, and we will explain what Rust is and what Traits and dispatching is.

In chapter 3 we will explain our methodology and how we approached the problem we were trying to solve. We will explain what our course of action was, and we will explore the test binaries we created to deduce the heuristics we used in our tool. We will then compare Rust's dynamic trait objects to C++'s dynamic dispatch of virtual methods. We will look at the memory layout for Rust's dynamic traits and we will proceed to the next chapter to discuss the implementation phase.

In chapter 4 we will discuss the implementation of our tool and how we built it. We will look at the heuristics we used to detect the trait object calls and the challenges we faced during the development and how we managed to solve them with the help of third-party tools and libraries.

In chapter 5 we will proceed to evaluate our tool with the toy examples we created, and we will compare our results with another evaluation framework provided by another project on Rust's dynamic traits.

Finally, we will discuss some improvements that can be made in the future and other related projects and work in this field. In the end, we will see some real-world applications of our tool and how it can be used to create other tools which will benefit developers and software reverse engineers.

Chapter 2

Background

2.1 Reverse Engineering

In computer science, reverse engineering is the process of getting a finished product, in this case a finished software product, and working backwards to discover its inner workings and how the software application works and operates [6]. In computer security, reverse engineering is used to better understand malicious software to detect their properties and uncover what their purpose is [7]. By doing that, software engineers can build heuristics into antivirus software to detect new malware and other malicious programs. Moreover, with reverse engineering and software analysis, we can re-enforce a program's defense against attacks without the need of the source code [8]. Over the years, many tools have been developed to help software engineers with reverse engineering, such as IDA Pro and Ghidra. As mentioned before, C and C++ have been studied for many years now and there are many tools and plugins for popular reverse engineering tools to help with the disassembly of those languages. In a paper published by the Blackberry Research and Intelligence Team [9] they discovered that many adversaries are using new languages such as Rust to develop their malware due to their late appearance and lack of maturity in the security community. For example, Rust binaries are more complicated a tedious to disassemble compared to traditional binaries from C/C++. The adoption of these languages by malicious developers increases the need for specialized tools to better understand the malicious code they write and prevent their attacks.

2.2 Assembly

A software engineer who analyses software and reverse engineers it, must know and understand Assembly language. Assembly is the representation of the low-level code running on the computer hardware in a more human readable way [10]. Although not as easy to read as a high-level language, Assembly is still more readable compared to 1s and 0s. As we discussed before

in “Reverse Engineering”, we need a method to learn about the behavior of a software application which most of the times we do not have the source code or any documentation or we are dealing with a legacy software without any official support. One way to “read” the software is by its assembly representation, instruction by instruction. In the computer world, everything boils down to Assembly language [11]. By reading the assembly representation of a program, a software engineer can understand a lot about the nature of a program, such as the control flow, what functions exist in the program and, ultimately, what the program does. It should be noted however that not everything can be reversed and reobtained by the assembly representation thus a full source code is not always guaranteed.

2.3 Disassembly

To obtain the assembly of a compiled binary, we need to translate the hex data of the file to valid Assembly language instructions. The process of extracting the Assembly code from a compiled binary is called disassembly. Using the machine code found in the binary, a disassembly software (known as a disassembler) decodes it into valid, architecture specific instructions that run on a CPU. Modern disassemblers, such as Ghidra, include virtual memory mapping, function identification heuristics and algorithms and entry-point detection for a binary file making it easier for the user to better understand the behavior of the program [12]. This paper will be focusing on static disassembly due to its faster performance and low overhead. There are two main static disassembly techniques one can use, i) Linear Disassembly and ii) Recursive Disassembly. Linear disassembly is the simplest approach and the easiest to implement since we are translating consecutive bytes into instructions. Although this technique can have good results, it can be tricked by bytes that do not represent instruction but data which may translate to instructions that are simply not a part of the program. Recursive disassembly on the other hand follows the control flow of the program thus it is not susceptible to inline data. Where both techniques are lacking is in following indirect jumps and calls to external dynamically linked libraries. Our aim is to build a disassembly tool that will be able to handle Rust binaries (stripped or not) and extract interesting information about them such as Rust Traits which we will discuss later. Our tool uses a combination of linear and recursive disassembly to achieve better coverage of the binary. We will talk about the inner workings of the tool in later chapters and how we utilize these two methods.

2.4 Rust

The focus of this paper is the Rust language. Rust is a compiled language that focuses on performance and memory safety. While Rust is a relatively new language, many well known services are using it to develop their platforms and applications. Some examples are Dropbox [13], Meta [14] and most notably the Mozilla Firefox web browser [15]. Rust's high performance combined with its memory safety, makes it an attractive choice for performance critical system where traditional safe languages such as Java would not be able to deliver the required performance. It is particularly well-suited for building large, complex software projects that require high performance, reliability, and security. As a replacement to system languages such as C, Rust combines low-level control over hardware with high-level abstractions that make it easier to write efficient and safe code. Most importantly, Rust is known for its code safety, part of it due to its compiler. Rust's compiler produces extra code to provide more security[16]. In this paper we will explore Rust's Traits which will be discussed later.

2.5 Traits

This paper will focus on the disassembly of Rust and most specifically on Rust's Traits. To explain what a Trait is, we need to understand OOP (Object Oriented Programming. Languages such as Java and C++ are centered around OOP. We will be focusing on C++ since it is closer to Rust in terms of performance and its real-life applications. In C++ we are defining classes which include the various attributes and methods that define that class. An object is an instance of these classes with specific attributes. We can think of the attributes of a class as the data an object can have and the methods as the functions that do a certain action on these objects. Similarly, in real life, we can think of a car as an object with some specific attributes such as its color, engine capacity and engine output and its methods, which can be applied on itself, such as "drive", "break" and "park" [17]. A key component of OOP is the property of interfaces, which are used to represent abstract concepts and are used to as a base for concreate classes. In our example with cars, we could implement an interface named "Vehicles" with some base methods and attributes which the class "Car" can inherit and specifically implement or override for the specific application. Similarly, Rust allows its programmers to share behavior using Traits. Traits can be used to define functionality a *type* can have and share it

with other *types*. Method signatures can be grouped together to define a set of behaviors needed to achieve a specific goal using trait definitions [18]. Our example with cars can be defined in Rust using a Trait "Vehicle" with the signature of the methods "drive", "break" and "park". Then we can create a struct "Car" which will implement the Trait "Vehicle" and implement each of its methods for the purposes of a car.

```
trait Vehicle {
    fn drive(&self);
    fn park(&self);
    fn breaks(&self);
}
```

Trait declaration in Rust

```
impl Vehicle for Car {
    fn drive(&self) {
        println!("Car Drive!")
    }

    fn park(&self) {
        println!("Car Park!")
    }

    fn breaks(&self) {
        println!("Car Apply Breaks!")

    }
}
```

A type Car implementing the trait Vehicle

```
impl Vehicle for Bike {
    fn drive(&self) {
        println!("Bike Drive!")
    }

    fn park(&self) {
        println!("Bike Park!")
    }

    fn breaks(&self) {
        println!("Bike Apply Breaks!")

    }
```

```
}
```

A type Bike implementing the trait Vehicle

The three figures above illustrate the structure of our example. Each type implements their own functions that they inherited from the Trait thus the functions of the trait can be applied to the types that implement them.

2.6 Dispatch

In the above section we discussed briefly about Traits and Objects in Rust and C++ respectively. Rust has the capabilities to work as an Object-Oriented Language through the use of Trait Objects. When we have multiple *types* that implement the same trait function (polymorphism in Object Oriented Languages), there needs to be a mechanism which will determine which specific version will run. Considering our Vehicle example, we chose to create a new struct called “Bike”. The “Bike” struct will then implement the methods of the Trait “Vehicle” with its own versions of “drive”, “break” and “park”. Thus, whenever we call the method “drive” on a “Car” or “Bike”, we need to know which “drive” we are referring to. Rust’s dispatch methods are either i) *static* or ii) *dynamic*, and it will prefer *static* dispatching [19]. Static dispatch is basically the creation of different versions of the same method for each of the types that implement it also known as *monomorphization*, whereas in dynamic dispatch only one version of the method is created. Consider our “Vehicle” example where we wish to create a wrapper method that will stop the vehicle. If we were to use Trait bounds (defining what types can be passed in the method), we would trigger Rust’s default behavior to create two copies of stop(x) method, one for “Car” and one for “Bike”. If we were to use trait objects, we would create only one stop(x: &dyn Vehicle) which would compute the target method “drive” in runtime using the trait object’s v-table. We will discuss the memory layout of traits and trait objects in the later chapters.

```
fn vehicle_stop_static<T: Vehicle>(x: &T){
    x.breaks();
    x.park();
    println!("The vehicle has stopped statically");

}
```

In this example, the rust compiler knows that the function vehicle_stop_static will be able to accept any `x` of type `T` that implements the trait `Vehicle` using *Trait bounds*. This will generate two functions, one for Bike and one for Car types.

```
fn vehicle_stop_Car (x: &Car){
    x.breaks();
    x.park();
    println!("The vehicle has stopped statically");

}
```

```
fn vehicle_stop_Bike (x: &Bike){
    x.breaks();
    x.park();
    println!("The vehicle has stopped statically");

}
```

The two functions shown above are the results of monomorphization creating one function for each type that implements the trait, that is, if we call `vehicle_stop_static` with an argument of type Car, the compiler will know and call `vehicle_stop_Car.` In the next chapter we will examine this behavior more closely and we will see the assembly generated from the compiler.

Chapter 3

Exploration

This section described the methodology and experimentation which helped to better understand how Traits work. Specifically, we will see some mock Rust binaries which implement Traits and we will explore the disassembled code we got from two disassemblers, `objdump` and `Cutter`. We will discuss the memory layout of traits objects and we will proceed to compare them with C++ objects. We will then explain some interesting features we have found in the disassembled code and try to create a heuristic that can identify Trait calls which will be especially important in later chapters that we will discuss our Rust disassembler.

3.1 Mock Binaries

For testing purposes, we created a few mock Rust binaries to study Traits. We wanted to create simple programs to help us figure out how Rust creates Traits, how Trait calls are done and the differences between static and dynamic dispatch. We created the "Vehicle" example in Rust as described in the previous section to observe the disassembled code. We also created a similar example in C++ to compare it with. Finally, we created another Rust binary to better understand the differences between static and dynamic dispatch and how Trait calls differ from normal function calls. To analyze the compiled binaries, we used objdump and Cutter. We compiled all the Rust binaries using the rustc compiler V 1.64.0 and the C++ binaries using the g++ compiler V 10.3. We used a Linux machine running Ubuntu 20.04.3 with an Intel Core i7 9750H CPU. All the source code files, and disassembled outputs are available at **https://github.com/stytSofo/traitor**

```
struct Car{
    color: String,
    EngineCapacity: i32,
    EngineOutput: i32,
}
```

Struct layout of Car

```
struct Bike{
    color: String,
    EngineCapacity: i32,
    EngineOutput: i32,
}
```

Struct layout of Bike

3.2 Disassembly Code

3.2.1 Vehicle Rust Binary

Our goal was to detect the assembly code produced by trait calls and determine the difference between static and dynamic dispatch. We used objdump to get an idea of the assembly code produced by the Rust binary.

```
callq  8530 <string::String >
mov    0x38(%rsp),%rax
mov    %rax,0x10(%rsp)
movups 0x28(%rsp),%xmm0
movaps %xmm0,(%rsp)
movl   $0x898,0x18(%rsp)
movl   $0x78,0x1c(%rsp)
```

Assembly code of an instance of the “Car” struct

Our first observation was that struct instances are stored in the stack, each of its attributes on neighboring stack addresses.

```
mov   %rsp,%rdi
callq  8870 <_ZN4main19vehicle_stop_static17h2b603e11d37fa859E>
```

Assembly code of the call on the wrapper function using a type “Car” as argument

By locating the call on the stop wrapper method, we can see that the argument, in this case the “Car” is loaded in the %rdi register. The %rdi register holds the head address of the struct we passed as an argument. Following the call we can in fact confirm that the method calls the

"break" and "park" methods for the "Car" struct. By locating the "Bike" struct, we can see the same behavior with the only difference being the call address for the wrapper class. As we explained before, static dispatch creates a method for each type that implements the "Vehicle" Trait thus we can see two stop methods, each calling the appropriate "break" and "park" methods.

```
fn vehicle_stop_static<T: Vehicle>(x: &T){
    x.breaks();
    x.park();
    println!("The vehicle has stopped statically");
}
```

As we can see, for a type to be able to be passed as argument in the function vehicle_stop_static, it must implement the Trait "Vehicle". We explicitly write the function signature using the <T: …> semantics to tell the compiler that this function will only accept arguments that implement the specified Trait, known as Trait Bounds [19].

```
subq    $0x38, %rsp ; main::vehicle_stop_static::h397d7081b118ec75
movq    %rdi, (%rsp) ; arg1
callq   <main::Bike as main::Vehicle>::breaks::ha944c107a0e3d1f9 ;
movq    (%rsp), %rdi
callq   <main:Bike as main::Vehicle>::park::h34a4aceb47c3f9c9 ;
```

vechicle_stop() method for Type Bike

```
subq    $0x38, %rsp ; main::vehicle_stop_static::h2b603e11d37fa859
movq    %rdi, (%rsp) ; arg1
callq   <main::Car as main::Vehicle>::breaks::h48ebe2b6f568337a ;
movq    (%rsp), %rdi
callq   <main::Car as main::Vehicle>::park::heb55b21ebd7f9c67 ;
```

vechicle_stop() method for Type Car

As we can see, on static dispatch the methods require the data of the Type as argument. For each of the methods, we can see that the appropriate Trait methods are called for the specific Type that called the method. Rust prefers static dispatch over dynamic since it does not involve points of indirections which may cause a slight drop in performance. Another benefit of static dispatching is the code in lining since the compiler knows the Types that will be passed in the method. The downside is that for each Type that implements the Trait functions, a copy of the vehicle_stop() method will be created, creating messy code and bigger binaries. As we can see, by statically dispatching a Trait, the calls on the implemented methods do not differ in any way compared to 'normal' function calls. We will demonstrate this behavior by creating the same program without using Traits but rather, we will implement each "drive", "breaks", and "park" methods for each struct. We will then implement two wrapper functions for the "breaks" and "park" methods to make our vehicles stop. In essence, we will manually do the work that the compiler does to statically dispatch the Trait.

```
fn car_stop_static(x: &Car) {
    x.breaks();
    x.park();
    println!("The vehicle has stopped statically");
}
fn bike_stop_static(x: &Bike) {
    x.breaks();
    x.park();
    println!("The vehicle has stopped statically");
}
```

Layout of the new implementation without traits

By disassembling the non-Trait code, we can see that we get almost the same binary as the statically dispatched version of our program using Traits. This is no surprise since all we did was do the work that the compiler does for us. It should be noted however that this example was done only for testing purposes to show that static dispatch is the same as implementing all the functions for each type manually. In cases where behavior can be shared across different types, Traits should be used for more concise and readable code.

Having said that, we can now draw our attention to Trait objects that utilize dynamic dispatch. In dynamic dispatch, the compiler does not know the exact type that implements the given Trait and it will only be available in runtime, thus it cannot apply 'monomorphization' and create

multiple copies of the same function. The figure following shows how to force the Rust compiler to dynamically dispatch the function `car_stop_dynamic`.

```
fn vehicle_stop_dynamic(x: &dyn Vehicle){
    x.breaks();
    x.park();
    println!("The vehicle has stopped dynamically");
}
```

Wrapper function for triggering dynamic dispatch

We can see that we define the function argument as &dyn Vehicle, signifying that we are passing a Trait Object thus the compiler does not have knowledge about the specific type of the pointer. Since the compiler does not know the type, only one copy of vehicle_stop_dynamic is created resulting in less code bloat. Looking at the disassembled code we can see that in fact there is only one copy of the function.

```
1.  leaq   data.0004e108, %rsi ; 0x4e108 ; int64_t arg2
2.  movq   %rsp, %rdi ; int64_t arg1
3.  callq  main::vehicle_stop_dynamic::hf11e4fbb557c7ebc ;
4.  jmp    0x8a36
5.  leaq   var_48h, %rdi ; int64_t arg1
6.  callq  main::vehicle_stop_static::h397d7081b118ec75 ;
7.  jmp    0x8a42
8.  leaq   data.0004e138, %rsi ; 0x4e138 ; int64_t arg2
9.  leaq   var_48h, %rdi ; int64_t arg1
10. callq  main::vehicle_stop_dynamic::hf11e4fbb557c7ebc ;
```

Disassembled code of the call of vehicle_stop_dynamic()

By looking at figure .., we can see that both calls on line 3 and 10 have the same address signifying that there is only one copy of the function. Looking into the disassembled body of the vehicle_stop_dynamic function we can see that there are not any direct calls to a function but rather, indirect calls resolved at runtime.

```
main::vehicle_stop_dynamic::hf11e4fbb557c7ebc (int64_t arg1, int64_t arg2);
; arg int64_t arg1 @ rdi
; arg int64_t arg2 @ rsi
; var int64_t var_40h @ stack - 0x40
; var int64_t var_38h @ stack - 0x38
; var int64_t var_30h @ stack - 0x30
1.    subq    $0x48, %rsp ; main::vehicle_stop_dynamic::hf11e4fbb557c7ebc
2.    movq    %rdi, var_40h ; arg1
3.    movq    %rsi, var_38h ; arg2
4.    callq   *0x28(%rsi) ;
5.    movq    var_38h, %rax; arg2
6.    movq    var_40h, %rdi; arg1
7.    callq   *0x20(%rax)
```

Disassembled body of the function vehicle_stop_dynamic

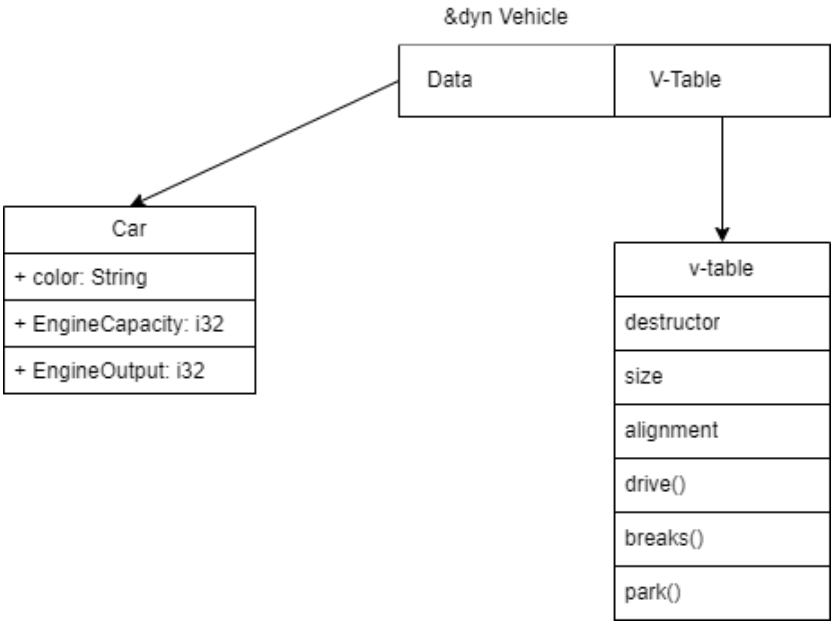
We can spot two call instructions, each with a dynamic address resolved in runtime. By examining the assembly code we can deduce that the register %rsi holds the base address on which the offset is added and then %rax has the same address as %rsi. Looking closer at the disassembled code, we can see that the disassembler deemed lines 2,3 and 5,6 as the arguments for the indirect calls. Following the control flow of the program we can determine what values are in the %rdi and %rsi registers. From the main function, we can see that an address in the data section, the v-table, is loaded into the %rsi register and the value of %rsp was transferred to the %rdi. As mentioned before, the data of a struct are stored in the stack in consecutive addresses. With that, we can deduce that the vehicle_stop_dynamic wrapper function accepts as arguments the data of a struct and the v-table for dynamic dispatch of the Trait. We will get in more depth on the v-table in the following sections.

3.2.2 Trait Object V-table

In the previous section we established how Rust creates calls on Trait objects and what parameters are used for the dynamic dispatch. In this section we will focus on the v-table and the memory structure of a Trait Object and then compare it with the memory layout of C++'s virtual classes.

A v-table is an important data structure used in OOP that allows dynamic dispatch and runtime assignment to objects of the same base class. V-tables offer us a point of indirection, as we will see later. A good, simple way to think of a v-table is as an index that has the content of the various methods and functions compatible with an object. In Rust, we do not have classes and

objects but rather, we are using Trait Objects. Similarly, we also need a mechanism that allows runtime dispatch of the supported methods of a trait object and thus Rust also uses a v-table.



Memory layout of a Trait object

In the figure (Memory Layout) we represent the memory layout of a Trait Object. There are two pointers in total, making a *fat pointer*, one that points to the data of the concrete type that implements the Trait, in this case the struct “Car”, and the second pointer that points to the v-table of the concrete type which holds the addresses of the implemented methods and functions of that concrete type. [20]. Apart from the methods and functions that the concrete type implements, the v-table also stores the size of the type, the alignment, and the destructor function of that trait object.

We will consider our “Vehicle” example and we will locate the v-table of the type “Car”, as shown in the figure above. By locating the address loaded in the main function in the %rsi register, we extracted the v-table that is in the .data section of the binary.

```
;-- data.: 0004e108
.qword 0x00000000000091c0 ; sym.core::ptr::drop_in_place_main::Car;
.qword 0x0000000000000020
.qword 0x0000000000000008
.qword 0x00000000000086f0 ; sym._main::Car_as_main::Vehicle_::drive; RELOC 64
.qword 0x0000000000008730 ; sym._main::Car_as_main::Vehicle_::park; RELOC 64
.qword 0x0000000000008770 ; sym._main::Car_as_main::Vehicle_::breaks; RELOC 64
```

V-table of the "Car" type located in the data section.

As we can see, by following the address 0x0004e108, we have obtained the v-table of "Car". We can see that the first entry is the destructor, followed by the size, alignment and then the implemented methods for the type "Car"; "drive", "park" and "breaks". Visiting each one of the three last addresses reveals the concrete implementation of each method. By looking at figure (Disassembled body of the function vehicle_stop_dynamic), we can see that the indirect calls are on the address of the %rsi register which holds the head of the v-table, and by offsetting that value, the appropriate method is called. In the code shown in figure (Disassembled body of the function vehicle_stop_dynamic), the offset is set to 0x28 pointing to 0x0004e130, where the address of the "breaks" method resides and then that method is called. Similarly, the same v-table exists for the type "Bike" which holds the appropriate addresses for its implementation of the 3 methods.

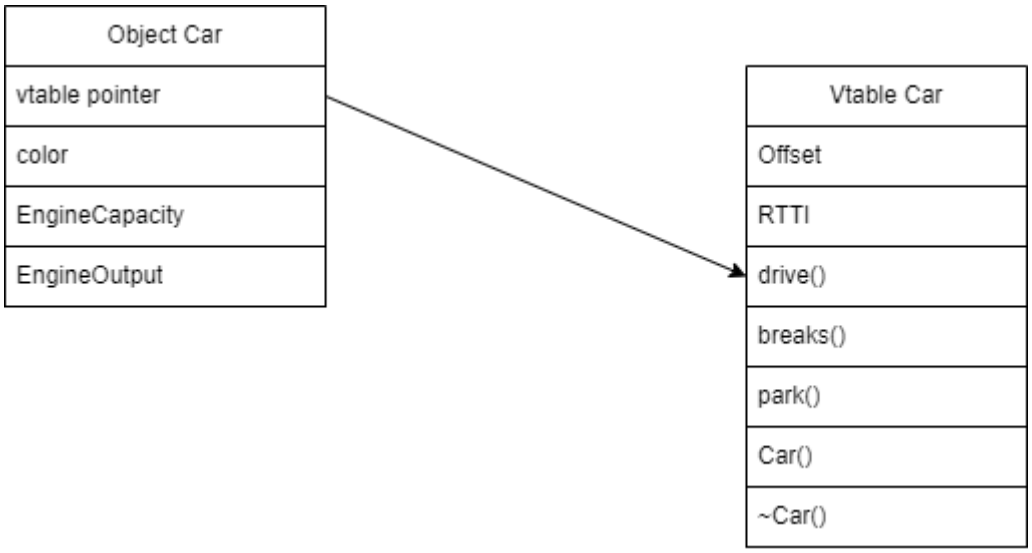
3.3 C++ Comparison

We will continue our exploration by looking at C++'s dynamic dispatch and we will compare the objects of C++ to the trait objects of Rust, and we will try to spot differences and similarities to each other. We will begin by creating the same "Vehicle" example in C++ so that we can compare the same code.

3.3.1 C++ Mock Binary

For this implementation, a base class called 'Vehicle' was created along with the 3 virtual methods 'drive', 'breaks', and 'park'. We then created two other classes that inherited from 'Vehicle' and implemented their own versions of the three virtual methods. Examples can be found on the GitHub repository.

The first thing we can see is that C++ does not separate the data and v-table of an object, but instead it stores the v-table pointer address inside the object's memory layout along with its data[21][1].



Memory layout of C++ objects

As we can see, an object in C++ points to its v-table, at the first method it can call. The first two entries are reserved for:

i) the offset that is required when there is multiple inheritance from different base classes.

ii) the Runtime Type Identification which points to metadata about the name of the class and the classes that inherit from.

The following entries are the methods and functions that the object can call, including its constructor and destructor.

As we can see, both C++'s and Rust's v-tables hold pointers to the applicable functions an object or trait object can call, and this is pretty much where the similarities end. As mentioned before, C++'s v-table holds information about the offset from the base class and metadata about its type and inherited base classes, thus allowing a C++'s object to be dynamically cast. An example of dynamic casting would be an object 'Vehicle' could be cast to an object 'Car' or 'Bike'. Moreover, C++'s objects can have two v-tables in cases of multiple inheritance as

mentioned in [1]. In contrast, Rust's trait objects are represented by a *fat pointer,* which as discussed before, is comprised of two pointers, one for the actual data and one for the v-table, meaning that the data layout is completely unaffected by any changes in the v-table. This means the type does not know that it will be dynamically dispatched. Another difference between the two v-tables is that Rust holds information about the size of that trait's type and its alignment.

Chapter 4

Implementation

The focus of this paper is the tool we propose for extracting information from a Rust binary. Our goal is to create a tool that, given a Rust binary, can tell us whether there is dynamic dispatching in the code thus presence of Trait Objects. Throughout our exploration we have discovered some properties of Trait Objects and we will apply them into our program to maximize its coverage and give us good results. In this section we will explain the difficulties we have faced in the development of the tool, how we overcame them and what techniques we used.

4.1 Description

Our tool's purpose is to identify potential Trait object calls in Rust X86-64 Linux binaries, which can either be stripped or not. We implemented our tool in Rust, utilizing two disassembly frameworks, iced-x86 [22] and the Capstone disassembly framework for Rust [23]. We are performing static analysis of the code [24] by performing linear disassembly in each body function whilst recording each call instruction for other possible functions. In conjunction with our tool, we are also using another tool, Nucleus [25] which identifies functions by examining the control flow of the code and is compiler agnostic. We will explain in later sections why Nucleus is an important addition to our tool. We should note that our tool works for Rust binaries compiled by the rustc compiler using the LLVM backend. Any other backend may result in different binary layout which our tool does not take into account.

4.2 Section Headers Detection

For our tool to be able to start identifying Trait objects, it must first learn some information about the binary it disassembles. For this task we are using the elf library that analyzes the elf binary and retrieves the section headers along with the string table to translate the names of the sections. We are also retrieving the entry point of the binary so that the analysis can start from there. We are interested in the '.data.rel.ro' section since, as we will see later, it is used to store the v-tables. Since 'data.rel.ro' holds data bytes instead of instructions, our disassembly engines will fail to correctly decode the v-tables stored in that section. Before the execution of the tool, we are collecting the data from that section using the elf library which extracts the data in a particular section. There is no need to search for the '.text' section since the control flow from the entry point to the 'main' function is in the '.text'. Moreover, Nucleus begins its function identification from the beginning of the '.text' section thus our tool completely covers the binary.

4.3 _start to main

As we have discussed in the previous section, we are starting our analysis from the entry point of the binary. We have discovered that in the entry point function of our binaries, there is a call to an external library which is dynamically hooked in runtime. Rust uses libc's '__libc_start_main' function to make call in the program's main function.

```
00000000000078d0 <_start>:
   78d0:        endbr64
   78d4:        xor    %ebp,%ebp
   78d6:        mov    %rdx,%r9
   78d9 :       pop    %rsi
   78da :       mov    %rsp,%rdx
   78dd:        and    $0xfffffffffffffff0,%rsp
   78e1:        push   %rax
   78e2:        push   %rsp
   78e3:        lea    0x39ee6(%rip),%r8  # 417d0 <libc_csu_fini>
   78ea:        lea    0x39e6f(%rip),%rcx        # 41760 <__libc_csu_init>
   78f1:        lea    0xba8(%rip),%rdi        # 84a0 <main>
   78f8:        callq  *0x4d2d2(%rip)        # 54bd0 <__libc_start_main@GLIBC_2.2.5>
   78fe:        hlt
   78ff:        nop
```

_start function that hooks the _libc_start_main function and passes the address of main as argument.

In the example shown above we can see the call on the hooked function and its parameters. Having that in mind, our tool begins from the _start function and scans the instructions until it reaches the indirect call. When we reach the call instruction, it means that the previous instruction holds the address of the main function. By examining that instruction's bytes, we can determine the address by adding the offset value to the instruction pointer's value and we return the address of main function. Following the acquired address, we discover that the 'main' (user-written main) that we wrote as programmers is in fact called by a function called 'lang_start'. The main that was called from _start function looks a lot like C's main function that in turn calls the user-written main and also sets up some panic function for error messages in case of exceptions [32]. Since the user-written main does not accept any arguments, this main is responsible for gathering any command line arguments that may be passed.

```
00000000000084a0 <main>:
   84a0:        push   %rax
   84a1:        mov    %rsi,%rdx
   84a4:        movslq %edi,%rsi
   84a7:        lea    -0x4be(%rip),%rdi        # 7ff0 <user-written main>
   84ae:        callq  8610 <_ZN3std2rt10lang_start17h8f251a561586ee14E>
   84b3:        pop    %rcx
   84b4 :       retq
```

Main function that calls the lang_start function which in turn calls the user-written main

As seen in the figure, the argument passed in the %rdi register is the address of the user-written main function. From now on, we will refer to user-written main as main.

4.4 Trait Object Detection

After we found the main function, we begin our analysis. For the analysis we need the binary itself which we obtained at the start of the tool, the main function's address and the functions' entry points and sizes which are obtained by Nucleus. Our detector uses a function call queue and an instruction queue so that we can easily refer to a discovered instruction or call. We perform linear disassembly on each function's body, and we are looking for the following hot-points:

i) Any call instructions
ii) Load or move instructions with memory address that lies in the .data.rel.ro section

4.4.1 Heuristics

From our exploration, we discovered some key features that can help us locate and distinguish trait object calls. As mentioned before, we are interested in the call instructions found in the disassembled code. For our tool we implemented the following heuristics:

i) A v-table lies in the data.rel.ro section of the ELF.
ii) The address of the v-table is loaded in one of the registers passed as arguments.
iii) The v-table has a specific layout we are looking for.

With these three heuristics we can find possible trait object calls with good enough accuracy. We will talk more about the limitations of our tool in the next chapters.

4.4.2 Methodology

We are interested in any call instructions since they will allow us to cover our code and identify more trait object calls. When we encounter a call instruction, we check its address to see if it is included in the .text section and save that instruction in a call queue which we use to visit each call's target at the end of the current function's analysis. Apart from adding the call instruction to the call queue, we are also trying to determine its arguments which would help us determine if that function is used for Trait Object call. Specifically, when we reach a call instruction, we check that we have not visited that address before and then we move up the instruction queue searching for load or move instructions that load memory addresses from the .data.rel.ro section to the rdi, rsi, rdx, rcx, r8, and r9 registers. The reason we are checking these

registers is because Rust uses the System V AMD64 ABI [26] [27] which loads any integer arguments into these registers.

To determine if a call is participating in a Trait object, we are checking its instruction information such as what registers it uses and what operand access it performs. If we discover that the instruction uses the rdi, rsi, rdx, rcx, r8, and r9 with write access and the immediate lies in the data.rel.ro section, we consider it as a Trait Object call. We will be discussing this behavior in the next section.

From our testing, we have observed that, before a Trait Object call, the v-table is loaded from the .data.rel.ro section to one of the aforementioned registers. If we find an address from that section, then we mark that call as a Trait Object call and we continue our disassembly until we reach the end of the function as determined by the size from Nucleus. At the end of the disassembly of that function we check the function call queue. If there are elements in the queue, we recursively call the trait identification function with the new function address and the process starts for that function. If there are no elements in the call queue, then it means we have finished discovering functions and the program can return and finish its execution.

To make our analysis faster, we can define the call depth the tool will search for traits making it faster to execute.

4.5 Challenges

During the development of the tool, we faced some challenges we had to deal with. The great support of Rust's documentation and crate library made the development of the tool easier. Some of our challenges consisted of:

i) Determining each function's length
ii) Making sure that already visited functions were not re-visited.
iii) Determining indirect call's target
iv) Addresses in the data.rel.ro may not always point to a v-table
v) Ground truth extraction

For us to overcome these challenges we had to rely on third-party tools such as Nucleus and use sophisticated disassembly engines like iced-x86.

i) From Dennis Andriesse's tool Nucleus, we were able to solve the first issue we had with function lengths. Since we know the entry point of a function and we are scanning the code for Traits, we needed a way to determine where to stop our disassembly. Thankfully, Nucleus' algorithm to detect functions in any binary helped us solve this problem. Initially, we load the binary we want to analyze into Nucleus, and we use the output as an input to a data structure we created to store the address of the discovered functions and its length as <u64, usize> tuples. When we pass a function's address for analysis, we check the data structure for that address, and we retrieve the length. We then know that we are analyzing from the function's address up to function's address + its size.

ii) When we are recursively analyzing a function, we may encounter another call instruction in that function's body. In case of a recursive call, our tool would fail and crash due to infinite recursion. To combat this, we had to add a Boolean flag next to the address in our data structure so that we would know if we have already visited that address and not insert it in our function call queue.

iii) We had to find a way to deal with indirect calls. Since these calls cannot be determined statically, we had to check the instruction if it was making an indirect call and not consider it at all. We are not considering indirect calls in our tool meaning that possible trait objects cannot be discovered.

iv) By looking at the data.rel.ro section, we have discovered that there are entries that do not correspond to any Trait objects v-tables. That's why, when we suspect a trait object call, we need to check the address that is in fact in the data.rel.ro section and then proceed to check whether the criteria for a v-table are met, including the format

of the v-table. Specifically, we need to check the entry of the v-table, the destructor for that trait, and check if the address that it points to is in the `.text` section of the binary. We then need to check the next 16 bytes (8+8) if they are an integer that represents the size and alignment of the Trait. The rest of the address must correspond to valid function entries in the .text section as they are the functions and methods of the implemented trait.

v) Our tool is implementing the heuristics discussed above to find trait object calls in a Rust binary. To prove our tool's correctness, we need to evaluate it against the ground truth we collect from existing binaries. One way to obtain ground truth is by looking at the source code of the binaries and determining whether dynamic traits are used. This, however, poses some challenges since some non-user defined code may use trait objects and thus would not be visible in the source code. Moreover, large projects with a lot of codebases are hard to analyze manually by looking at their source code since it might be separated into multiple files. One robust way to obtain ground truth would be to modify the rustc compiler such that it would output the uses of dynamic traits in the binary. Specifically, since rustc uses the LLVM backend, we would need to create an LLVN pass that scans for dynamic traits and prints their addresses.

Unfortunately, in our tool's implementation we have not created a ground truth extraction framework thus our evaluation will rely on mock binaries and some source code inspection alongside with a pre-build ground truth extractor. We will discuss future additions to the tool in a later chapter.

Chapter 5

Evaluation

To evaluate our work, we created a few benchmark binaries that include Trait implementations on structs to simulate real-world inheritance-like behavior and use Rust Object orientation- like features. As discussed in previous chapters, we will be considering our 'Vehicle' example and a new example we created with geometrical shapes. We will examine in more detail each test binary in this chapter. We will also look at real world applications from Rust's crates and we will try to identify any trait object calls. In addition to our empirical evaluation, we will be using Alexa VanHattum's [28] proposal for obtaining the ground truth for our binaries.

5.1 Evaluation Framework

As mentioned at the beginning of this chapter, we will be using Alexa VanHattum's proposal for obtaining the ground truth from our binaries. The proposed methodology includes two ways to obtain ground truth.

i) Scan the source code files for the keyword "dyn" that signifies dynamic trait call.
ii) Set the rust compiler to output a debug log and search for "get_vtable"

As we have mentioned before, we cannot apply the first way in many cases since we do not always have the source code files which leaves us with the second approach using the debug log of the compiler. We will use this method to extract the number of dynamic traits and compare that number with the number that our tool shows. It should be noted however that our tool is focused on finding dynamic trait calls rather than the dynamic traits themselves. For our evaluation we will consider dynamic trait calls with the same address as the same dynamic trait.

5.2 Crafted Binaries Evaluation

5.2.1 Vehicles

From the previous chapters, we have analyzed the “Vehicle” example binary, and we used objdump and Cutter to disassemble it. We will now look at the results our tool produces for the said binary and another example we created for evaluation that we will investigate later. On our “Vehicle” example we created a trait Vehicle and two structs representing a Car and a Bike. Both structs implement the trait Vehicle. We have created two wrapper functions for stopping a vehicle, one dispatched dynamically and the other statically.

```
fn vehicle_stop_dynamic(x: &dyn Vehicle){
    x.breaks();
    x.park();
    println!("The vehicle has stopped dynamically");
}


fn main(){



    let my_car = Car{
        color: String::from("Red"),
        EngineCapacity: 2200,
        EngineOutput: 120,
    };

    let my_bike = Bike{
        color: String::from("Red"),
        EngineCapacity: 2200,
        EngineOutput: 120,
    };



    vehicle_stop_static(&my_car);
    vehicle_stop_dynamic(&my_car);


    vehicle_stop_static(&my_bike);
    vehicle_stop_dynamic(&my_bike);

}
```

As we can see, each type is passed as argument in the functions, once stopping statically and once stopping dynamically. We then expect our tool to output two trait object calls.

```
0000000000009579 lea 0x51150,%rsi
0000000000009580 lea 0x10(%rsp),%rdi
0000000000009585 call 0x0000000000009450
    V-Table Address: 51150
TRAIT OBJECT
000000000000958A jmp 0x000000000000958C
000000000000958C lea 0x50(%rsp),%rdi
0000000000009591 call 0x0000000000009410
0000000000009596 jmp 0x0000000000009598
0000000000009598 lea 0x51180,%rsi
000000000000959F lea 0x50(%rsp),%rdi
00000000000095A4 call 0x0000000000009450
    V-Table Address: 51180
TRAIT OBJECT
00000000000095A9 jmp 0x00000000000095AB
00000000000095AB lea 0x50(%rsp),%rdi
00000000000095B0 call 0x00000000000096C0
```

Output of the tool for the Vehicles test binary

As we can see, our tool did in fact identify the 2 trait object calls in the main function and also marked the v-table address for that trait object. Our tool did not find any other trait object calls. This may be caused by the fact that our tool cannot resolve indirect calls of the form `call *%rax` since it does not have enough information about what address is passed in each register. As far as the user defined trait objects, the tool has successfully found both. By manually checking the addresses of the v-tables, they were indeed the addresses of each type's v-tables. Running the tool provided from Kani's repository it detected that there were 6 appearances of the "get_vtable" line from debug output of the compiler signifying there are 6 dynamic trait objects in the binary but as mentioned in the tool's paper, the number we see here is an overestimation of how many trait objects there are, especially user-defined dynamic trait objects. We should mention that any rust *closures* are implemented with v-tables so it is possible that this number includes *closures.* Since we have the source code of our test binary, we also ran the tool in its explicit mode that scans the source files for any appearance of the *dyn* keyword that signifies a dynamic trait. It turns out that the result we get from the explicit search is in fact 2, the same as our tool reports.

5.2.2 Shapes

We will now discuss our next toy binary which we used to evaluate our tool. This binary has a trait the represents shapes called *Shape* and three structs, each one representing a shape. One triangle, one rectangle and a circle. There are two wrapper functions in the program for calculating the area of each shape, one statically and one dynamically and one more function that creates circles.

```
trait Shape {
    fn perimeter(&self) -> f32;
    fn area(&self) -> f32;
}

struct Rectangle {
    verLength: f32,
    a: Point,
    b: Point,
    c: Point,
    d: Point,
}

struct Triangle {
    side1: f32,
    side2: f32,
    side3: f32,
    a: Point,
    b: Point,
    c: Point,
}

struct Circle {
    R: f32,
    a: Point,
}
```

Shapes binary layout

Above is shown the outline of our binary with the trait that is implemented by each type.

```
fn calc_area<T: Shape>(x: &T){
    x.area();
}

fn calc_area_dyn(x: &dyn Shape) {
```

```
    x.area();
}


fn create_tr(radius: f32, c_x: i32, c_y: i32)->Circle{
    let a_cricle = Circle { R: radius, a: Point { x:
c_x, y: c_y, z: 0 } };
    calc_area_dyn(&a_cricle);
    return a_cricle;
}
```

Wrapper functions which perform static and dynamic dispatch in the Shapes binary

Inside the main function we create one instance of each type and call each of the wrapper functions with those types as parameters. We then proceed to call the `create_tr` function to create a circle and calculate its area. Our goal is to show that our tool can recursively search functions and identify dynamic trait calls inside of other functions.

```
    calc_area_dyn(&x); // x is Triangle
    calc_area_dyn(&y); // y is Rectangle
    calc_area_dyn(&z); // z is Circle

    let p = Point{

        x: 3,
        y: 4,
        z: 0,
    };

    let p_b = Point{
        x: 0,
        y: 0,
        z: 0,
    };

    let acirc = create_tr(1.0, 1, 1);

    calc_area_dyn(&acirc);
```

main function of the Shapes binary

From the code above we can see that there are 4 dynamic trait calls and one inside the `create_tr` function. Similarly, our tool has correctly identified the dynamic trait calls from the

main and create_tr functions. We also ran the evaluation tool from Kani's repository which revealed 15 "get_vtable" instances.

When we manually disassembled the data.rel.ro section of the binary, there were many more v-tables corresponding to closures and traits which our tool could not find. This can be explained by the fact that our tool i) does not look for closures and ii) we cannot resolve any indirect calls which may be dynamic trait calls. As far as we are concerned, any user-defined dynamic traits can be detected by our tool since with both the toy examples we managed to find all the user defined trait objects.

5.3 Real World Binaries

Apart from our crafted binaries, we also tested our tool with real world applications which were made using Rust. Since Rust is not a very popular language, it was more difficult to find applications that were written exclusively in rust. Thankfully, we have managed to find some open-source projects written in Rust.

Another difficulty we found is that there are not many real-world uses of dynamic trait objects in Rust. From VanHattum's work and our own observations we have found that most Rust developers use Rust's default Traits with static dispatch. Instead, dynamic dispatching is used with closures which in essence are trait objects.

The binaries we found were mostly tools in UNIX such as an "ls" implementation in Rust (exa [30]) .

5.3.1 exa

Exa is an open-source application written in Rust. It is a direct replacement for the well known "ls" command in the UNIX OSes with a more modern approach. Using the latest version from GitHub, we compiled a developer build of the binary and we analyzed it. We searched the source code for any appearances of the keyword "`dyn`" signifying the use of dynamic trait objects. We found out that there was one function that used trait objects as arguments.

Our course of action was the same as all the binaries we tested.

i) Extract function entries from Nucleus
ii) Disassemble with objdump and Cutter
iii) Disassemble with our tool
iv) Verify results

Our tool was unable to detect any trait object calls.

By manually checking the data.rel.ro section we found the relatable v-tables of the trait objects used in the function we have discovered in the source code. Searching their entry point in objdump's output revealed that there exists a function that loads the v-table addresses. Searching our tool's output, it was realized that the tool did not disassemble the code section. With further investigation using Cutter and objdump, we discovered that there was no direct call on that function from the disassemble code meaning that our tool would not be able to find that function since it relies on the call instructions.

When we tried our tool with manual entry of the function's address it did in fact reveal the trait object calls as expected.

Chapter 6

Related Work

Software analysis is being utilized to get knowledge about a program's behavior and learn more about its security flaws and how we can solve them. Our work contributes to this field by providing a framework for discovering Trait Objects in Rust binaries without the need for the source code. There are many other papers and tools that contribute to this cause, not only for Rust binaries but for other languages too. We will look at some examples that inspired and motivated us into creating this project-thesis.

We will start off by mentioning a Rust related paper that it too examines Dynamic Traits. "Verifying Dynamic Trait Objects in Rust" by Alexa VanHattum is the paper that we have already mentioned in the Evaluation chapter. We are using VanHattum's evaluation framework to obtain the ground truth for our binaries. VanHattum's paper focuses on dynamic trait object reasoning. It proposes the Kani Rust Verifier [28] which is the first symbolic modeling checker for trait objects that checks the correctness and soundness of them. Kani is used as a code generator backend for the rustc compiler, just like LLVM, but it sends its output to a SAT solver for verification. Although Kani does check the validity of Trait Objects, it does not detect them in a compiled binary like our tool. It is a framework to be used while writing mission-critical code.

Another example of related work for the C++ language is the tool Marx from the paper "Marx: Uncovering Class Hierarchies in C++ Programs" by Andre Pawlowski et. Al. This is another paper that inspired us to create a tool for Rust that is like what Marx does for C++. Marx is an analysis framework for re-creating the class hierarchies of a C++ binary. For Marx to be able to create the hierarchies, it relies on detecting the v-tables of the objects and from there it builds the hierarchies. Our tool is similar to Marx in the sense that it too relies on detecting the v-tables to classify a call as Trait Object call. We are also using similar heuristics as proposed in the paper, but we modified them to fit Rust's nature.

Chapter 7

Future Work

There are many improvements that can be made to our tool to make it better and more accurate. As mentioned in the Evaluation chapter, there needs to be a valid framework to evaluate our tool against the ground truth. As we have used VanHattum's evaluation tool, we observed that it was overestimating the number of trait objects it was logging.

7.1 Function Coverage

As for now, our tool relies on Nucleus function identifier to be able to cover the binary. We need a way to identify the functions of a binary without relying on a tool that cannot run on the latest version of the operating system. A built-in solution must be implemented or at least linked to our tool in a way that the user can seamlessly enter the name of the binary they want to analyze, and all the pre-processing is done internally and correctly.

7.2 Trait Object detection heuristics

Our tool poses a proof of concept of what can be done in terms of disassemblers for Rust. There may be more heuristics that can determine if there are trait objects in the code. Such heuristics should be explored and integrated into the tool for better coverage and trait object identification. Scanning more instructions and tracking the values of the registers can help uncover more dynamic traits and even be able to extract information from indirect calls.

7.3 V-Table extraction

Our tool relies on the detection of the data.rel.ro section of the binary to determine the existence of a trait object. Since that section stores the v-tables of dynamic traits, it would be a nice extension of the tool to be able to extract the v-tables and organize them in a way that the user can follow the addresses of the functions of a v-table and provide a visualization of the v-tables and show the hierarchies between traits and types that implement them.

7.4 Evaluation Framework

Our tool needs its own evaluation framework. An evaluation framework must be build such that it will be able to extract the ground truth from the binaries during the compilation phase. It should also be able to load the files into the disassembler with the necessary information it needs to work. By creating a concreate evaluation method, it will be easier to detect any bugs in the tool's code. It will also allow us to find the weaknesses of the tool and implement better heuristics to achieve better results and more code coverage.

Chapter 8

Contributions

In this chapter, we will look at various real-world applications our tool has. We are explaining how our tool can be used an expansion to a Rust specific decompiler and as a disassembly tool for Rust binaries. We are also exploring the possibility of using it to create attack mitigation techniques in unsafe Rust applications or mixed binaries utilizing Rust along with C/C++ code.

8.1 Reverse Engineering

As we have mentioned before, this paper is focused on the development of a disassembly tool for Rust which extracts information about the presence of Trait Objects in a Rust binary. This tool is a proof of concept showing that it is possible to extract information such as v-tables and dynamic trait object calls. A language specific tool such as this will help developers and software reverse engineers to better understand a program's behavior by providing them with more information than a general disassembler. An extension to this tool's capabilities is the development of a decompiler that will transform the assembly code as close as possible to its original source code in rust. Knowing where the v-tables are and where the trait object calls are happening, a decompiler can better re-create the source code with additional information on the invocation of the functions in a trait.

8.2 Unsafe Rust

Apart from the obvious benefits in the reverse engineering segment, our tool can help mitigate vulnerabilities caused by unsafe rust. Rust supports unsafe code regions where the type and memory checkers are not used and allows the programmer to have untethered access to memory just like in C and C++ in exchange for the safety features. From our exploration, we determined the memory layout of a trait object and how a reference to a trait object happens. In unsafe rust, we can have an unsafe object with its data members in the safe region while the v-table pointer residing in the unsafe region which can lead to overflow attacks. With our tool we can detect any trait objects and apply Control Flow Integrity mechanisms to the addresses of the functions in the v-table ensuring a valid control flow [29]

8.3 Mixed Binaries

In the paper “Exploiting Mixed Binaries” it is discussed how Rust and C/C++ can co-exist in the same binary leading to unsafe indirect calls from rust code to C/C++ code. Such indirect calls are mostly trait objects with functions in the v-table leading to the unsafe C/C++ code [29]. Since they will reside in an unsafe block, as discussed previously, an attacker can point the unsafe pointers to ROP gadgets gaining execution privileges. With our tool we can detect the trait objects v-tables and ensure that the target addresses are in fact the ones leading to legitimate functions and not arbitrary code from an attacker.

Chapter 9

Conclusion

In this thesis we explored the challenges of disassembling rust binaries. We explored the fundamentals of Rust and what our objectives will be, and we made a brief introduction to the main topic of this thesis, the Rust Traits. We then discussed the different ways Rust dispatches its traits and we concluded that we will be analyzing the dynamic dispatch of trait objects.

We then began experimenting with toy programs crafted specifically for analyzing the behavior of rust's compiler creating the v-tables of the traits. We explored the v-tables that are produced by the rust compiler, and we examined their structure and memory layout and proceeded to compare them with C++'s v-tables. We observed the differences and similarities, and we explained what purpose these differences have in each language.

After gaining knowledge about the topic, we began creating a tool that would serve as a proof-of-concept regarding the trait object identification in rust binaries and the work that can be done for Rust-specific decompilers and disassemblers. In the implementation chapter we discussed the heuristics we used to detect trait object calls and how we distinguish them from normal function calls.

Finally, we used a tool from VanHattum's work on the Kani dynamic trait object verifier to evaluate our tool against the toy examples we created to explain our work and what dynamic traits are. We showed that our tool can in fact detect the majority of user defined trait objects and mark their v-table addresses.

References

1. Pawlowski, A., Contag, M., van der Veen, V., Ouwehand, C., Holz, T., Bos, H., ... & Giuffrida, C. (2017, February). MARX: Uncovering Class Hierarchies in C++ Programs. In *NDSS*.

2. https://www.binarydefense.com/digging-through-rust-to-find-gold-extracting-secrets-from-rust-malware/

3. https://www.binarydefense.com/resources/blog/digging-through-rust-to-find-gold-extracting-secrets-from-rust-malware/

4. https://doc.rust-lang.org/book/ch10-02-traits.html

5. https://ghidra-sre.org/

6. Johnson-Laird, Andrew. "Software reverse engineering in the real world." *U. Dayton l. rev.* 19 (1993): 843.

7. https://www.eccouncil.org/cybersecurity-exchange/ethical-hacking/malware-reverse-engineering/

8. http://www.cs.ucy.ac.cy/courses/EPL451/

9. G. Wassermann, and D. Svoboda, "Rust Vulnerability Analysis and Maturity Challenges," Carnegie Mellon University, Software Engineering Institute's Insights (blog). Carnegie Mellon's Software Engineering Institute, 23-Jan-2023 [Online]. Available: https://doi.org/10.58012/t0m3-vb66. [Accessed: 26-Apr-2023].

10. https://en.wikipedia.org/wiki/Assembly_language

11. https://0xinfection.github.io/reversing/pages/part-19-why-learn-assembly.html

12. Pang, Chengbin, et al. "Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask." 2021 IEEE Symposium on Security and Privacy (SP). IEEE, 2021.

13. https://www.rust-lang.org/what/wasm

14. https://engineering.fb.com/2021/04/29/developer-tools/rust/

15. https://www.rust-lang.org/what/networking

16. https://dms.cs.ucy.ac.cy/op/op.ViewOnline.php?documentid=17532&version=2

17. https://www.w3schools.com/cpp/cpp_classes.asp

18. https://doc.rust-lang.org/book/ch10-02-traits.html#traits-defining-shared-behavior

19. https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/trait-objects.html

20. https://www.eventhelix.com/rust/rust-to-assembly-tail-call-via-vtable-and-box-trait-free/

21. https://view.officeapps.live.com/op/view.aspx?src=https%3A%2F%2Fcourses.cs.washington.edu%2Fcourses%2Fcse333%2F21su%2Flectures%2F16%2F16-c%2B%2B_inheritance-I_21su.pptx&wdOrigin=BROWSELINK

22. https://crates.io/crates/iced-x86

23. https://www.capstone-engine.org/

24. https://www.intel.com/content/www/us/en/docs/inspector/user-guide-windows/2022/dynamic-analysis-vs-static-analysis.html#:~:text=Dynamic%20analysis%20is%20the%20testing,detected%20both%20dynamically%20and%20statically.

25. D. Andriesse, A. Slowinska and H. Bos, "Compiler-Agnostic Function Detection in Binaries," 2017 IEEE European Symposium on Security and Privacy (EuroS&P), Paris, France, 2017, pp. 177-189, doi: 10.1109/EuroSP.2017.11

26. https://en.wikipedia.org/wiki/X86_calling_conventions#System_V_AMD64_ABI

27. https://github.com/rust-lang/rust/blob/master/tests/codegen/abi-sysv64.rs

28. VanHattum, Alexa, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. "Verifying dynamic trait objects in Rust." In Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice, pp. 321-330. 2022.

29. Papaevripides, Michalis, and Elias Athanasopoulos. "Exploiting mixed binaries." ACM Transactions on Privacy and Security (TOPS) 24, no. 2 (2021): 1-29.

30. https://github.com/ogham/exa

31. https://github.com/BurntSushi/ripgrep

32. rust/base.rs at 38030ffb4e735b26260848b744c0910a5641e1db · rust-lang/rust (github.com)