universität wien

# BACHELORARBEIT / BACHELOR’S THESIS

Titel der Bachelorarbeit / Title of the Bachelor’s Thesis

„Reverse Engineering of Rust Programs through Language Features“


verfasst von / submitted by

Lamies Abbas


angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Bachelor of Science (BSc)

Wien, 2024 / Vienna, 2024

| | |
|---|---|
| Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet: | UA 033521 |
| Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet: | Informatik |
| Betreut von / Supervisor: | Univ.-Prof. Dipl.-Ing. Mag. Dr.techn. Edgar Weippl |

# Abstract

Modern software development often requires robust, secure programming languages like *Rust* or *Go*. However, the code complexity introduced by *Rust* makes it challenging to reverse engineer the code. A common problem is *Rust's* ownership and borrowing model which complicates reverse engineering by making it difficult to monitor and comprehend memory allocations and deallocations during binary analysis. In contrast, *C* is known for producing easily readable decompiled code due to its straightforward memory management and simpler syntax. This thesis shows the shortcomings of the existing *Rust* reversing tools, and introduces *Rusteronies*, a collection of scripts designed to improve the process of reverse engineering *Rust* binaries. The tool set allows reconstructing both static and dynamic strings in *Rust* by searching the read-only memory and identifying the values of registers. These scripts offer better performance and accuracy compared to the Ghidra inbuilt analyzer, making the reverse engineering of Rust binaries more efficient and comprehensive. Furthermore, it also includes a script that can reconstruct macros, such as the simple *println*, *print*, and *fmt* functions.

# Kurzfassung

Die moderne Softwareentwicklung ist häufig auf robuste, sichere Programmiersprachen angewiesen, zu denen unter anderem *Rust* und *Go* zählen. Die durch *Rust* eingeführte Codekomplexität erschwert es jedoch, den Code durch Reverse Engineering zu analysieren. Ein häufig zu beobachtendes Problem ist das *Ownership*- und *Borrowing-Modell* von *Rust*, welches das das Reverse Engineering erschwert, da das Überwachen von *memory allocations and deallocations* während *binary analysis* mit Schwierigkeiten verbunden ist. Im Gegensatz dazu ist *C* dafür bekannt, dass es aufgrund seiner unkomplizierten *Memory-Management*-Funktionen und einer einfacheren Syntax leicht lesbaren dekompilierten Code erzeugt. Die vorliegende Arbeit zeigt die Limitierungen der gegenwärtig verfügbaren *Rust-Reverse-Engineering-Tools* auf und stellt *Rusteronies* vor, eine Sammlung von Skripten, die den Prozess des Reverse Engineerings von *Rust-Binaries* verbessern sollen. Die Skripte ermöglichen die Rekonstruktion sowohl statischer als auch dynamischer Strings in *Rust*. Dazu wird der *read-only* Speicher durchsucht und die Werte von Registern identifiziert. Die Skripte bieten eine optimierte Leistung und Genauigkeit im Vergleich zum integrierte Ghidra-Analyzer. Zudem beinhaltet das Paket ein Skript, welches die Rekonstruktion von Makros ermöglicht, darunter einfache Funktionen wie *println*, *print* und *fmt*.

# Contents

**Abstract** i

**Kurzfassung** iii

**1. Introduction** 1

**2. Background** 5
2.1. Rust . . . . . . . . . . 5
2.2. Ghidra . . . . . . . . . . 6
2.3. ANGR . . . . . . . . . . 8

**3. Related Work** 9
3.1. Rust Analysis . . . . . . . . . . 9
3.2. Go Analysis . . . . . . . . . . 10
3.3. Other Analysis Methods . . . . . . . . . . 11

**4. Rusteronies** 13
4.1. String Representation . . . . . . . . . . 14
4.2. Function ID Analysis . . . . . . . . . . 19
4.3. Macro Analysis - ANGR . . . . . . . . . . 20

**5. Evaluation** 27
5.1. Setup . . . . . . . . . . 27
5.2. Comparison of Script Performance . . . . . . . . . . 28
5.3. Comparison of String Count . . . . . . . . . . 29
5.4. Comparison Across Architectures . . . . . . . . . . 31
5.5. Comparison of Length Distribution . . . . . . . . . . 34
5.6. Comparison of Baselines . . . . . . . . . . 36
5.7. Format Macros . . . . . . . . . . 39

**6. Conclusion** 45
6.1. Future Work . . . . . . . . . . 45

**List of Tables** 47

**List of Figures** 49

**List of Listings** 51

**Bibliography** **53**

**A. Appendix** **57**

# 1. Introduction

Nowadays, Rust is gaining popularity among developers because of its emphasis on memory safety, concurrency, and performance. One of the key features that distinguishes Rust from other programming languages is its robust type system, which makes it possible to write safe and efficient code without sacrificing performance. However, the same features that make Rust popular with developers also create challenges for reverse engineers attempting to analyze Rust binaries due to the complex decompiled code produced by reverse engineering tools such as Ghidra.

Today's cybersecurity techniques rely on software reverse engineering. This process allows them to investigate software binaries, find vulnerabilities, understand the system, and improve the overall system security. The growing popularity of Rust, combined with the challenges that come with it, is increasing the amount of malware being written in Rust. This highlights the importance of the lack of tools specifically designed to reverse engineer Rust binaries.

One of the first steps in reverse engineering binaries is to look at the defined strings. While in some programming languages strings are easily reconstructed by reverse engineering tools, existing methods and scripts for recovering strings from Rust binaries often fail to effectively deal with the challenges that arise from Rust strings such as ownership semantics and the lack of a terminating null byte at the end of each string. Another major challenge in reverse engineering Rust binaries is the powerful compile-time code generation through macros. Unlike simple function calls, macros are expanded at compile-time, making it difficult to reconstruct them. As a result, current reverse engineering tools are unable to identify and analyze macros within Rust binaries.

Listing 1 and Listing 2 provide a clear example of the large expansion of Rust macros at compile-time, converting the short source code into a much larger block of decompiled *C*-code. The example is the `#[tokio::main]` macro, which simplifies asynchronous programming in Rust. Examining the decompiled output includes many function calls, variable declarations, and exception handling mechanisms. Furthermore, the decompiled code produces detailed functionality for the resource management for example by calling `core::ptr::drop_in_place<>` for local variables.

```
undefined  [16] __rustcall hello::main(void)
{
  undefined auVar1 [16];
  undefined local_178 [8];
  /* ... */
  /* ... */
  /* ... */
  undefined local_d0 [181];
  undefined2 local_1b;

  tokio::runtime::builder::Builder::new_multi_thread(local_d0);
  local_1b = 0x101;
                    /* try { // try from 00111ad5 to 00111ae2 has its CatchHandler @ 00111bac */
  tokio::runtime::builder::Builder::build(local_120,local_d0);
  if (local_120[0] != 2) {
    local_138 = local_e0;
    uStack_134 = uStack_dc;
    uStack_130 = uStack_d8;
    uStack_12c = uStack_d4;
    local_148 = local_f0;
    uStack_144 = uStack_ec;
    uStack_140 = uStack_e8;
    uStack_13c = uStack_e4;
    local_158 = local_100;
    uStack_154 = uStack_fc;
    uStack_150 = uStack_f8;
    uStack_14c = uStack_f4;
    local_168 = local_110;
    uStack_164 = uStack_10c;
    uStack_160 = uStack_108;
    uStack_15c = uStack_104;
    uStack_170 = uStack_118;
    uStack_16c = uStack_114;
                    /* try { // try from 00111b21 to 00111b31 has its CatchHandler @ 00111b8c */
    auVar1 = tokio::runtime::runtime::Runtime::block_on(local_178,0,&PTR_s_src/main.rs_00198bc0);
                    /* try { // try from 00111b32 to 00111b3f has its CatchHandler @ 00111bac */
    core::ptr::drop_in_place<>(local_178);
    core::ptr::drop_in_place<>(local_d0);
    return auVar1;
  }
                    /* try { // try from 00111b67 to 00111b89 has its CatchHandler @ 00111b99 */
                    /* WARNING: Subroutine does not return */
  core::result::unwrap_failed();
}
```

Listing 1: Macro expanded at compile-time

```rust
use std::error::Error;

#[tokio::main]
pub async fn main() -> Result<(), Box<dyn Error>> {
    Ok(())
}
```

Listing 2: Original rust code

The following chapters explore Rust reverse engineering and string analysis through four research questions.

1. **How can the recovery of strings in Rust binaries be improved?** The goal is to recover as many meaningful strings as possible from Rust binaries, aiming to outperform existing analysis tools like Ghidra's RustStringsAnalyzer.

2. **Can external factors influence script performance?** The accuracy of the scripts should not vary depending on the underlying architecture. As the length

of a string might be encoded in the instructions of a program directly, trying to recover all strings might involve architecture specific information. It is therefore necessary to ensure that scripts perform the same way on different architectures.

3. **How can common macros be reconstructed?** The script should be able to reconstruct commonly used macros like `format!`, `print!`, and `println!`. The underlying rules for these macros are concise, but not trivial to understand in decompiled code. This makes them interesting targets for attempting reconstruction.
4. **How many macros can the script detect/reconstruct?** Although the print-macros are commonly used, some real-world binaries might not use them, resulting in less detected macros. Reconstruction should detect the presence and attempt to reconstruct macros in real-world Rust binaries.

This thesis aims to answer the research questions by analyzing language specific artifacts in Rust programs. The aim is to gain a better understanding of the underlying structures and behaviors of Rust codebases. This is achieved by developing scripts to aid in Rust reverse engineering.

The thesis starts off with chapter 2 which provides the necessary background information for understanding the context of reverse engineering and the pain points of the Rust programming language. In chapter 3 existing research and tools are discussed to find ideas and opportunities for improving the state-of-the-art in reverse engineering Rust programs. In chapter 4 **Rusteronies**, a set of scripts were specifically programmed for reverse engineering Rust programs. These methods include string representation analysis, applying function ID analysis, and macro analysis with tools such as *ANGR*. In chapter 5 evaluates the developed scripts using a series of case studies and comparisons. The setup of the experiments, including the datasets and tools used is discussed followed by the presentation of the results. Finally, chapter 6 wraps up the thesis by summarizing the key findings and identifies potential areas for future work.

# 2. Background

This chapter provides an overview of the needed background material necessary to understand the thesis. First, this text introduces Rust as a programming language and highlights its key features relevant for chapter 4. Following that, Ghidra, a reverse engineering tool, is introduced including an in-depth analysis of its various functionalities of how it aids the process of binary analysis. Finally, *ANGR*, a Python tool for symbolic execution, is presented.

## 2.1. Rust

Rust [1], released in 2015, is a modern programming language developed by Mozilla Research which focuses on safety, performance, and concurrency. The programming language mainly wanted to enforce strict memory management rules to prevent daily bugs in a programmers life which occur in other languages such as `C` and `C++`. For instance, common issues are bugs due to dangling references, use after frees, double frees, and buffer overflows which are all checked by the Rust compiler.

Rust's ownership and borrowing model is intended to stop data races and memory problems which complicates reverse engineering. It is difficult to monitor and comprehend memory allocations and deallocations during binary analysis due to the concept of ownership, where variables have exclusive access to their data. Listing 3 is a simple program which creates a string and borrows the value and prints Hello World in the end. Although Rust promotes stack allocation, heap allocation is required. Therefore, Rust allocates memory for variables. For example, `s1` owns the string `Hello`. When `s2` is assigned a reference (`&`) to `s1`, it borrows the ownership without actually resulting in `s1` being dropped with the drop trait. This ensures that `s1` remains valid. However, removing the reference from `s2` would cause `s2` to take ownership of `s1`, which would violate Rust's borrowing rules and cause a compilation error by attempting to use `s1` after it has been moved. This is very different from C, where such code would result in no compilation errors but cause problems which lead to undefined behavior. Rust also has string slices which provide a way to reference parts of a string without taking ownership of the entire string. Listing 4 shows how a string slice gets defined.

In Rust, there are two types of macros: `declarative` macros and `procedural` macros. Declarative macros are executed during compilation and generate code based on predefined patterns and rules. An example of a declarative macro is the one from the standard library `println!`, which is identical to the `printf` function in `C`. The other feature is

```
fn main() {
    let s1 = String::from("Hello");
    let s2 = &s1;

    println!("{} World!", s1);

}
```

Listing 3: Hello World program in Rust with ownership and borrowing models

```
let s = String::from("Hello, world!");
let slice = &s[0..5]; // String slice referencing the first 5 characters of s
```

Listing 4: String slices in Rust

called procedural macros. These macros allow code to be generated during compile time based on custom annotations or attributes. Procedural macros enable developers to add new syntax and constructs to the language that would not be possible with declarative macros alone. Unlike declarative macros, which perform pattern-based transformations on source code, procedural macros work with abstract syntax trees (ASTs) and can generate entirely new code [2]. For example, [3] provides a tutorial for a practical example of using a procedural macro in Rust for asynchronous programming.

## 2.2. Ghidra

Ghidra [4] is an open-source reverse engineering tool that was developed by the National Security Agency (NSA). With Ghidra one can analyze binaries, making it possible to understand the underlying structure when no source code is available. One of the main features is the decompiler interface where the binary code is displayed as human-readable C code. Compiling Listing 5 results in a binary which can be imported into Ghidra. After letting Ghidra auto-analyze the binary, it manages to find the `main` function and shows the decompiled code as shown in Listing 6.

When analyzed in Ghidra, the decompiled code has almost an identical structure as the source code, with the `printf` function call for printing the message. There is only one noticeable difference between the two. In the decompiled output from Listing 6, the `main` function is declared with `void` as parameter and returns an `undefined8` type, which in other words is in Ghidra a placeholder for an unknown data type. These types have a common occurrence in the decompilation process because Ghidra is not capable of accurately reconstructing the exact return type of a function.

```c
#include <stdio.h>

int main() {
   printf("Hello, World!");

   return 0;
}
```

Listing 5: Simple Hello World program in C

```c
undefined8 main(void)
{
    printf("Hello, World!");
    return 0;
}
```

Listing 6: Decompiled C Hello World program in Ghidra

Looking at the decompiled code of a Rust Hello World program is more challenging. The Rust language is well known for its memory safety, concurrency, lifetime and borrowing models. However, these factors add layers of complexity when attempting to reverse engineer the binary. For example Listing 7 shows a simple Hello World program written in Rust. Listing 8 shows the decompiled code of the same program, revealing the additional lines of decompiled code generated by Ghidra. One feature that draws attention is the extra lines of code for variable declarations. At the end of the decompiled code the final function call is to *std::io::stdio::_print* with the string pointer saved in the *local_30* variable to print the output. It is also visible how Rust approaches the memory management and frequent use of pointers. Ghidra v11 supports demangling Rust symbols, analyzing strings, and also redefining them for Rust binaries.

```rust
fn main() {
    println!("Hello World!");
}
```

Listing 7: Simple Hello World program in Rust

```
void hw::main(void) {
    undefined8 local_30 [2];
    undefined **local_20;
    undefined8 local_18;
    char *local_10;
    undefined8 local_8;

    local_20 = &PTR_s_Hello,_world!_0014b350;
    local_18 = 1;
    local_30[0] = 0;
    local_10 = "Hello, world! ";
    local_8 = 0;
    std::io::stdio::_print(local_30);
    return;
}
```

Listing 8: Decompiled Rust Hello World program in Ghidra

## 2.3. ANGR

*ANGR* [5] is a Python library based on symbolic execution, which is used for binary analysis. Symbolic execution executes the binary symbolically (as the name implies) rather than with specific input values. During execution, *ANGR* saves the program's state and propagates symbolic values, generating a symbolic execution tree that represents all possible paths through the program and allows users to explore different execution paths under certain conditions. For example, by using constraint solving one can specifically search for a value in the binary. Every other value would not satisfy the constraint and therefore will not be considered. It is also possible to check which value a specific variable can hold at a certain part of the program. Overall, with the power of *ANGR* it is possible to inspect and analyze certain memory values without needing to start the program.

# 3. Related Work

As Rust grows in popularity as a systems programming language, there is a growing demand for effective tools and approaches for reverse engineering Rust programs. This goes hand in hand with a growing body of research into how Rust can be effectively reverse engineered, together with lessons learned from other modern programming languages, e.g. Go. This section provides a brief overview of the prior work in these areas. The conducted research includes non-academic sources due to the novelty of the topic in addition to academic references and tools available.

## 3.1. Rust Analysis

Due to the sophisticated language features and robust memory safety guarantees, Rust binaries provide particular difficulties and pain points making it harder to reverse engineer. The blog “Rust Binary Analysis: Feature by Feature” [6] provides an in-depth discussion of the significant challenges of reverse engineering Rust binaries due to their complex language features and powerful memory safety guarantees. This examination goes into numerous areas of reverse engineering by studying sample programs and closely examining disassembly, with a special emphasis on discovering Rust-specific features. The analysis highlights the complexity required to understand and analyze Rust binaries, such as compiler optimizations, string representations, and memory management.

While Giordano’s blog post [7] presents a brief analysis that focuses on important topics such as destructuring, loop unrolling, and arrays by looking at the disassembly, Zeropio’s post [8] analyzes the features by looking at the same program written in C and the Rust programming language, and explains the differences while reversing them. All of these blog posts have in common that they explore Rust features by writing their own small sample programs and reverse engineer them using a reverse engineering tool. Unlike the rest, this blog “Digging through Rust to find Gold: Extracting Secrets from Rust Malware” [9] looks at the pain points from a different angle. It starts with a basic analysis of the calling convention and strings. It then applies these steps to an unknown Rust malware. To do this it also uses a Ghidra script called RustDependencyStrings.py [10], to find the crates being used in the binary making it possible to find the dependencies specified by the malware developer.

In contrast, GhidRust [11, 12], an open-source project available on GitHub, provides a Ghidra plugin to analyze Rust binaries. GhidRust determines whether a binary was written in the Rust programming language or not and translates Ghidra’s C-style decompiled code into Rust code. The author decided to not add a Demangler to the plugin since Ghidra already performs demangling on Rust binaries. Nevertheless, the integrated Demangler does not always work and could potentially make it more difficult and time-consuming to reverse. As a result, another open-source tool published on GitHub called DemangleRust [13] focuses on demangling the symbols by using the Rust v0 mangling scheme [14]. By combining GhidRust and DemangleRust, analysts may produce a more precise and understandable representation of Rust symbols, improving the process of reverse engineering.

During development of this thesis project, the NSA released their own Rust String support [15] and their own Rust Demangler [16] with Ghidra 11. The problem was addressed using a recursive approach. The main function of the script `recurseString` first determines the maximum string length up to a specified reference or the maximum length allowed. If the new length is greater than zero, it creates a new char array of that length and the string. If the new length is less than the maximum length, the function recursively calls itself using the remaining string.

In comparison to reverse engineering or binary analysis tools where the source code is not available, there is a lot of research and tools which focus on the static and dynamic analysis when the source code is available. The approaches these tools use could potentially be used to simplify reverse engineering as well. Rust has an `unsafe` tag [17] that allows code execution that is not allowed by Rust's safety rules such as dereferencing a raw pointer. Unsafe code leads to bugs, security vulnerabilities, and memory safety issues that are difficult to detect by looking at the code or reverse engineering the binary. Clippy [18] is a resource for finding simple programming problems and improving code quality by printing out different levels of warnings to ensure code safety. Another approach to ensure safe code uses Miri [19], an experimental Rust interpreter, that compiles source code and runs the generated intermediate representation in an interpreter to find undefined behavior and memory leaks. While Miri uses an interpreter to run the MIR, making it a dynamic analysis tool, Clippy only analyzes the source code statically. Finally, another tool called Rudra [20] is a system for finding memory safety bugs in Rust code. Unsafe code blocks are usually checked, and the goal of Rudra is to eliminate them and find security problems. While Mira and Rudra are tools for specifically analyzing unsafe code, Clippy can be useful for new Rust developers because of the feedback it provides to help users to learn Rust's idiomatic standards.

## 3.2. Go Analysis

While the analysis of Rust binaries is the main focus of this thesis, it is also worthwhile to examine and contrast similar techniques used with Go. Many of the difficulties encountered in Go binary analysis are also present in Rust binaries such as binary size and unusual string handling. Go and Rust binaries have large sizes due to code duplication and static compilation. Go-Clone [21] is a tool that detects copy-pasted or duplicate code in Go. To achieve this the authors use source code analysis, creating labeled semantic flow graphs (LSFGs), neural network training, and user interaction. The important step is that the source code is converted to LLVM Intermediate Representation (IR) [22] and for each function an LSFG will be created showing the structure and the flow behind the function. It then compares multiple iterations of the same project and detects clone pairs. Using a deep neural network model Go-Clone can find repeating patterns and generate a list of cloned code which helps to reduce binary size.

Moreover, there is a different set of Go reversing tools [23] published on GitHub that make the process easier. The first tool recovers the function names from stripped binaries. The other two scripts identify string structures (static and dynamic) because Go and Rust both have non null-terminated strings. Instead, a pointer is stored which points to the actual string, in addition to the string length. As a result, there is no clear sign that a string has ended or started. Thus, recovering strings from Go/Rust binaries is harder.

## 3.3. Other Analysis Methods

In addition to the Go reversing tools, researchers have also explored potential applications of LLVM, such as developing code property graphs and how metamorphic malware developers evade signature-based detection. Furthermore, LLVM is widely used for optimization in RustC.

Kuechler et al. [24] investigated the use of LLVM in the creation of code property graphs using LLVM intermediate code. While the study focuses on software dependencies, the methodology and insights gained from generating code property graphs could be applied to the analysis of Rust binaries and their dependencies providing reverse engineers with a complete picture of Rust programs and their interactions with external libraries or modules. On the other hand, Dube et al. [25] conducted research on malware obfuscation tactics with the primary goal of protecting software intellectual property. The authors discovered that after obfuscation, the binaries are difficult to reverse using popular tools such as IDA (Interactive Disassembler) [26]. While their research was not directly related to Rust binaries, it provided useful insights into obfuscation techniques that could potentially be adopted and used for Rust-based malware, making reverse engineering more difficult and requiring complex analysis techniques.

The size of Rust binaries is remarkably larger compared to C binaries, potentially resulting in a large amount of instructions that introduce security risks which could be exploited by attackers and increases the amount of code that needs to be reverse engineered. To address this concern, researchers have developed different tools aiming at reducing the binary size. One such tool is TRIMMER [27] which focuses on code debloating by analyzing user-provided data to figure out unused functionality, thereby decreasing the final binary size. In comparison, Chisel [28] uses a different approach by performing reinforcement learning to debloat programs. It takes two input parameters, the original binary and a specification of its functionalities and outputs a debloated program including the specification. Whereas, Nibbler [29], another noteworthy tool designed to optimize shared libraries by spotting and deleting unused functions within shared libraries.

Similar to Go-clone, there is existing research in the area of clone detection and binary similarity not limited to Go binaries. Luo et al. [30] introduce VulHawk [31], a cross-architecture binary code search approach based on an intermediate representation function model, natural language processing, and graph convolutional networks. The goal is to identify vulnerabilities caused by code reuse in IoT firmware images. Another tool BinFinder developed by Qasem et al. [32] is based on a neural network to find differences and similarities in disassembled code. For comparison, BinDiff [33] is an open-source tool that also provides the same functionality without using machine learning. According to Marcelli et al. [34] machine learning is a critical aspect to find binary similarities. As a matter of fact the strength of machine learning lies in analyzing and comparing complex patterns within code. In contrast to machine learning methods, reverse engineering tools usually implement simpler and more performant approaches to detect function clones. Ghidra uses the `.fidb` database [35] to store function signatures, while Radare2 uses the `.sdb` database. The way function ID works in Ghidra is that if the function hashes of for example the OpenSSL library are known, the program can recognize those functions. On the other hand, IDA Pro uses F.L.I.R.T. [36] function signatures. FLIRT stands for Fast Library Identification and Recognition Technology which allows IDA to detect standard library functions. Overall, function clone- and binary similarity detection increase the usefulness and readability of the generated disassemblies.

# 4. Rusteronies

As stated in chapter 3, prior research has uncovered many pain points and difficulties related to Rust reverse engineering. While this work offers insight and techniques for analyzing Rust binaries manually, there is still room for improvement with reference to development of automatic tools.

The calling convention in Rust is noticeably different from C binaries. When loading a C binary into a reverse engineering tool the `main` function is located by checking the entry point which is `_start` in Ghidra. From there, the `__libc_start_main` function is called, which in turn calls the actual `main` function with the required arguments. However, this process differs in Rust. Even when following the exact same steps, the real `main` function cannot be found in Rust. The `__libc_start_main` function calls Rust entry point `std::rt::lang_start_internal` function which then proceeds to invoke the actual `main`. Rust uses the default calling convention on `x86_64` Linux (`SYSTEM-V`) which already passes most of the arguments in the registers. In some cases Rust creates structs on the stack and passes their addresses in the registers. However, a stable internal Rust ABI does not exist since Rust programs are compiled statically and thus do not require a stable ABI.

Rust utilizes LLVM for compiler optimization. If the binary is built with the `-release` flag, the debugging symbols are removed, resulting in a faster program runtime and smaller binary size. The `.toml` file allows adding customized compile-profiles i.e. the strip option [37] can be used to strip symbols from the binary as shown in Listing 9.

```
[profile.release]
strip = "symbols"
```

Listing 9: Stripping symbols from binary

In the debug build, the operation that significantly decreases performance is frequently moving memory between registers and stack. Furthermore, more checks are performed on the code during the debugging phase, including the detection of integer overflows. Upon inspection of the same binary, it is apparent that different functions are called when it is compiled in debugging mode versus in release mode. For example, in the release build the function `lang_start_internal` is responsible for calling the main program function, while in debug mode the function name is slightly different: `lang_start<()>`.

## 4.1. String Representation

In C, strings are represented using null-terminated character arrays. A null-byte denotes the end of the string making it simple for reversing programs like Ghidra to find the string's start address and reading characters up until the null-byte. Rust, on the other hand, has a more intricate string representation because of its ownership system and UTF-8 encoding. In Rust there are two different types of strings: string slices which do not have ownership and owned strings. String slices (`&str`) are references to an existing string, and it does not possess ownership of the data to which it refers. They have a defined size and are stored in memory as a reference to a range of bytes. Owned Strings (`String`) are growable strings that own their data and are represented by the String type [38]. Unlike C strings, Rust explicitly records the string's length, making length retrieval possible in constant time. The string representation in Rust has benefits for memory safety and language features, but it makes reverse engineering more difficult. Reverse engineers must rely on the stored length information to determine the string's boundaries as there is no null-terminated byte to indicate the string's end.

Ghidra is an open-source tool for reverse engineering developed in Java that offers an extensive API [39]. This allows users to extend the functionality of Ghidra with custom scripts which can be written in either Java or Python. To start writing a Ghidra script, the first step is to set up a development environment. The setup used in this thesis was taken from the following blog post [40]. Also, Java 17 was used as the programming language for the scripts, which were first written for Ghidra v10.3.2. In December 2024 with v11, Ghidra released their own RustStringsAnalyzer [15] and RustDemangler [16]. The RustStringsAnalyzer works similarly to the dynamic script. Later in chapter 5 the results of the RustStringsAnalyzer will be compared with the developed scripts.

Among the problems mentioned above, the easiest and most straightforward one to approach was improving Ghidra's string representation of Rust strings. The first step consisted of creating tailored Ghidra scripts that analyze strings in Rust binary executables. The problem was approached by treating the string occurrences as "static" and "dynamic", similar to scripts for executables of the Go programming language [23], which the scripts of this thesis were based on. In order to discover "static" strings, string structures are utilized. These structures consist of a `char*` pointer and a length, and they are stored in a specific data section (`.rodata`) during the compilation process shown in Listing 10. In contrast, the "dynamic" strings does not necessitate the existence of specific string structures. Instead, the pointer and length of the string are directly stored in the instructions within the `.text` section like in Listing 11. These encoded instructions then reference the string contents located in the `.rodata` data section.

Simply porting the Python scripts for analyzing strings in Go binaries was not sufficient to make them work for Rust. Additionally, these scripts disregard a number of edge cases which were found through manual analysis of the annotations generated by the script. The first improvement implemented was getting the string address. The Go script obtains the string's address by directly converting the integer offset from the string address pointer

```
    PTR_s_/rustc/90c541806f23a127002de5b40_0

00151f50 7d 42 14   addr   s_/rustc/90c541806f23a127002de5b40_001442
        00 00 00           = "/rustc/90c541806f23a12700
        00 00
00151f58 4f 00 00   int    4Fh
        00
```

Listing 10: Static strings in Ghidra

```
00119f2a 48 8d 15   LEA   RDX,[PTR_s_/cargo/registry/src/index.crat
        1f 83 03          = 00144625
        00
00119f31 be 2b 00   MOV   ESI,0x2b
        00 00
```

Listing 11: Dynamic strings in Ghidra

to hexadecimal, without utilizing the address space which refers to the total amount of memory allocated for all possible addresses [41]. This approach increases the number of operations, while on the other hand the `getAddress` function combined with the address space and the identifier minimizes the computational overhead. The next problem was the printable check. The Python script checks for printable ASCII characters in a hardcoded predefined range. This check can fail for Rust strings because they are UTF-8 encoded. The approach used in this thesis was to decode the entire sequence of bytes as characters using UTF-8 encoding like in Listing 12.

In the next step, the identified strings should be declared as the string data type in Ghidra. According to the Go script, it is necessary to delete any previously defined data and then redefine it. The FlatProgramAPI offers a function called `removeData` to accomplish this. Despite implementing these changes, the script still threw a CodeUnitException (Listing 13) because of conflicts created by data being defined in an already-allocated memory space. This exception additionally revealed multiple other edge cases.

One of the edge cases was observed through manual analysis about strings being truncated or disregarded. There were two causes for these issues. If the pointer pointed to the middle of a string, shown in Figure 4.1, the allocated memory for the full string length needed to be cleared. This was achieved by replacing `getData` with `getDataContaining`. The difference between these two is that the first one gets the AddressSpace data at the specified address and the second one returns the data containing the specified address.

```
public boolean isPrintable(final Address strAddr, final int length) throws
↪   MemoryAccessException {
    final CharsetDecoder decoder = StandardCharsets.UTF_8.newDecoder();
    final byte[] value = getBytes(strAddr, length);
    final ByteBuffer buf = ByteBuffer.wrap(value);

    try {
        decoder.decode(buf);
    }
    catch(CharacterCodingException e) {
        return false;
    }

    return true;
}
```

Listing 12: Check for UTF-8 printable characters

```
Conflicting data exists at address 0014827b to 0014834c
```

Listing 13: CodeUnitException due to data conflicts

If the string was divided, the entire block had to be cleared before creating any data shown in Figure 4.2. Therefore, the length of the string was checked, whether its bigger than one and then iterating over the full length of the string to be defined (represented as `AddressSet`) and deleting the data.

Fixing these two edge cases also solved the data conflicts exceptions. In addition, Rust uses UTF-8 encoded strings, hence creating the final string follows a different approach. Unlike the Go script which uses `createAsciiString` to create an `ASCII` string and unlike Ghidra's own String Analyzer [15] which uses a `char` array, `createData` is utilized to create a `StringDataType` struct containing the string address, the length with the correct Rust string encoding.

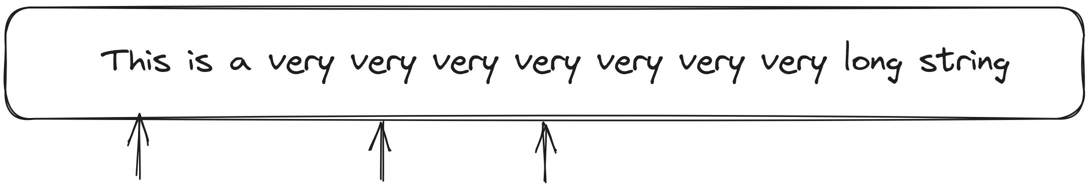


Figure 4.1.: Edge case 1: Multiple string pointers pointing to different locations

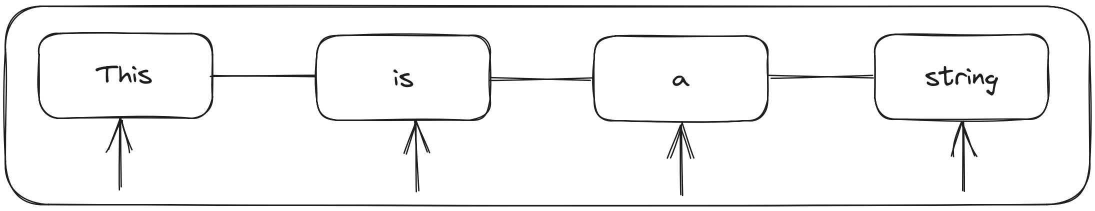


Figure 4.2.: Edge case 2: String is split into smaller blocks

The methodology for searching strings "statically", i.e. by searching for string structures in `.rodata`, leaves many strings undiscovered that are not part of such string structures. A large portion of strings in the binary does not have their associated length stored in memory, but directly loaded into registers as immediate values, at the time when the string is used. Go and Rust exhibit distinct patterns in disassembly to identify strings. In the initial version of the script, only the `x86_64` architecture was relevant. In Rust the pattern to follow in assembly can be found in Listing 14.

```
LEA REG [addr] # String address
MOV REG 0xXX   # String length
```

Listing 14: Pattern for dynamic strings in x86_64 in Rust

Many strings were recovered simply by following this pattern. However, there were again edge cases that must be taken into account. After analysis it was revealed that, in some cases the string length is stored one instruction before the string address. In another instance, it was observed that the length of the string was occasionally copied after a brief sequence of other instructions. In some cases just checking if there was any memory address put into the destination was not enough. Additional checks were needed such as checking if the `LEA` instruction was followed by a pointer or if it loads a function address or an address of executable code. To determine the length, several checks are needed. First a `MOV` command followed by a register and a scalar is required. If these conditions are met, a printable check is needed, just like for static strings, with the slight difference that null bytes are ignored here. After finding the longest length, it gets stored in a variable. If the string address found is undefined, the same steps are followed to create the string structure as in the previous script, otherwise the address is skipped.

The end goal was to make the script architecture independent. Therefore, to accomplish this task several changes were made. Instruction names cannot be used as a guide, as they differ across architectures. The algorithm in Listing 15 checks each instruction and the addresses of the references instead of searching for a `LEA` instruction. While manually analyzing the code in Ghidra, there were a lot of pointers found that start with valid UTF-8 printable characters. The dynamic string detected those as valid strings and redefined them, which caused some false positives. The solution to this problem was to skip all pointers where Ghidra had previously defined data during its analysis steps. Afterwards the same checks mentioned above can be performed to determine if a string has been found.

```
final Address strAddr = instruction.getReferencesFrom()[0].getToAddress();

// If Ghidra has already defined a pointer there we skip
final Data data = getDataContaining(strAddr);
if (data != null && data.isPointer()) {
    continue;
}

// Make sure that we don't load a function address/address of executable code
if (getInstructionContaining(strAddr) != null) {
    continue;
}
```

Listing 15: Algorithm for searching for string address

Furthermore, it was necessary to adjust the algorithm to find the length. Firstly, all operands needed to be either registers or scalars and registers needed to be outputs. Secondly, for each string address, the maximum length must be determined and stored. The most convenient approach was to use a map that takes the string address as a key and a list of possible lengths as a value. Lastly, the missing step is to iterate over the map and pick the longest length for each string address and define the data.

The final problem encountered was losing the old string data when removing the already fully defined data and replacing it with the new string data shown in the first drawing in Figure 4.3. In this case, before deleting any data, the old data is saved and checked if the data had any string value and where the original string was located. At the end the previous string data is once again redefined, as shown in the bottom drawing in Figure 4.3. To avoid false positives, the data is checked for non-UTF-8 encoded characters prior to creating the final string and any strings shorter than length 1 are being skipped. This fix has been implemented in both string scripts.

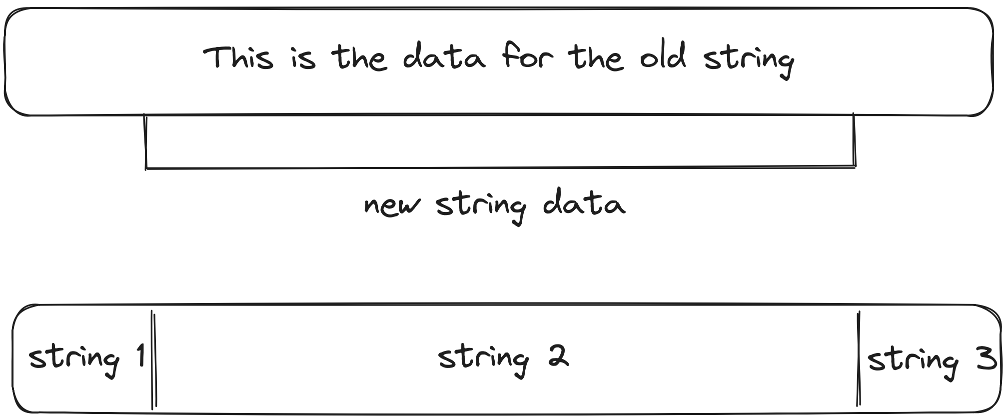


Figure 4.3.: Redefining every split string

## 4.2. Function ID Analysis

When stripping a binary of symbols or function signatures, reverse engineers encounter difficulties in detecting fundamental Rust functions from the standard library. The concept of integrating a function ID database is to compare the function hashes in the binary with those of the database while considering mnemonic and operand type. If matches are found, the corresponding function signature is added to the binary.

At the time of writing there is no database that fully includes every standard library function ID. This database is hard to obtain since to include every function ID a program must use every single function. Rust compilers optimize functions out but they also clone (monomorphize) trait generics and macros, so it is not possible to create a full database at all. To create a custom function ID database, the thesis generates it from a non-stripped binary rather than relying on user-generated function ID databases. During the analysis, an issue was encountered where Ghidra was unable to recognize the loaded database. The process of debugging this error proved to be time-consuming as Ghidra's visual function ID debugger listed the correct functions. Interestingly, other databases were successfully added without any issues. After manually detaching the database and reattaching it, the user-generated function ID databases functioned properly.

Function ID analysis offers multiple benefits. On one hand, it allows reverse engineers to ignore library code, which they are not concerned with. On the other hand, it provides additional insights into program functionality, enabling macro analysis. For instance, it is possible to reconstruct the `print` and `format` macros by recognizing the underlying functions. However, the mangled function name should incorporate typing information, which is not extracted. Thus, it is necessary to manually analyze the typing information.

## 4.3. Macro Analysis - ANGR

Since a human reverse engineer can reconstruct macros by comparing the macro rules with the expanded, compiled code, it must be possible to reconstruct macros when their macro expansion rules are known. However, there is no general solution for this problem yet. As proof-of-concept this thesis demonstrates one possible solution with the `print` and `format` macros.

*ANGR* [5] is an open-source platform primarily used for symbolic execution and binary analysis. Symbolic execution involves code evaluation using symbolic variables, instead of specific values, to represent program inputs and states. This technique allows reverse engineers to explore different program execution paths without actually running the program. *ANGR* [5] is only compatible with Python 3. Jython, which is used as a Python interpreter in Ghidra, only supports Python 2 and installing packages is generally not possible. In order to run Python 3 and additional packages, projects like Pyhidra [42], Ghidrathon [43], or Ghidra Bridge [44] are required. First, Pyhidra seemed promising but it was not possible to link it with the current Ghidra installation. The second option Ghidrathon is not able to work with multiple threads, despite citing *ANGR* as a motivating case for using Ghidrathon. After installing and attempting to import *ANGR*, the import process fails and displays the following error message shown in Listing 16.

```
 line 48, in <module>
    signal.signal(signal.SIGINT, handle\_sigint)
  File "/usr/lib/python3.11/signal.py", line 56, in signal
    handler = _signal.signal(_enum_to_int(signalnum), _enum_to_int(handler))

ValueError: signal only works in main thread of the main interpreter
```

Listing 16: Error shown when trying to import `angr`

Upon further research there are many well-known issues listed on GitHub [45] that explain the problems with using threads in Ghidrathon which leads to the same situation as the built-in Jython since both are limited to what they can achieve. Thus, the last option Ghidra Bridge [44] seemed to work after following the setup instructions in the readme file.

In order to develop a methodology for recreating Rust macros, this thesis chose to work with the format macros found in the Rust standard library. These are available to users by calling `print!`, `println!` or `format!` and use the same functions internally. For this thesis, the approach to reconstruct these macros therefore relies on first identifying the main functions invoked by these macros. The first step is to iterate over each function used in the program, looking for the cross-references of the `print` and `format_inner` function. The next step contains finding out where the cross-references are located.

Lastly, the entry points of those containing functions need to be stored. The steps have been outlined in a pseudocode in Algorithm 1.

**Algorithm 1** Iterate over Functions

```
function ITERATE_FUNCTIONS(program):
    for all function in program do
        if function uses print or format_inner then
            cross_references ← []
            for all reference in function do
                if reference is a call to print or format_inner then
                    cross_references.append(reference)
                end if
            end for
            if cross_references is not empty then
                entry_points.append(function)
            end if
        end if
    end for
end function
```

This process can be time-consuming, particularly when sending many messages back and forth between the client and Ghidra Bridge server. One solution is to use Ghidra Bridge's `bridge.remote_eval` to gather all the data needed and then return it all at once resulting in only one message sent. However, Ghidra Bridge seemed to return `None` for the cross-references used. This led to a creation of a Java script that gathers the information and writes them into a text document which then gets parsed by the *ANGR* Python script. To inspect the program's state, *ANGR* requires the function's entry point and follows the execution path until reaching the state at the function call's address. *ANGR* uses different offsets by default than Ghidra. In order for *ANGR* to work with the Ghidra addresses, it is necessary to use the image base found in Ghidra as `base_address` offset in *ANGR*.

While exploring the execution paths *ANGR* executes the existing function code which is also very time-consuming. To avoid this, hooks can be used. Instead of following every single function call it is possible to hook the function and run custom written code. This has been used for the `print` to skip every function call and the `__rdl_alloc` function to return a symbolic representation of the `rax` register.

For macro reconstruction the variable names from the stack are needed. Initially, this requires retrieving the variable names from the decompiler and applying the stack offsets. The corresponding function is shown in Listing 17.

```python
def get_variables_remote(func):
    ifc = ghidra.app.decompiler.DecompInterface()
    options = ghidra.app.decompiler.DecompileOptions()
    ifc.setOptions(options)
    ifc.openProgram(func.getProgram())
    monitor = ghidra.util.task.ConsoleTaskMonitor()
    res = ifc.decompileFunction(func, 60, monitor)

    high_func = res.getHighFunction()
    lsm = high_func.getLocalSymbolMap()
    symbols = lsm.getSymbols()

    offset_mapping = dict()

    # NOTE high variables offset could overlap
    for symbol in symbols:
        hs = symbol.getHighVariable()
        if hs == None:
            continue
        instances = hs.getInstances()
        for instance in instances:
            name = instance.getAddress().getAddressSpace().getName()
            if not name == "stack":
                continue
            offset_mapping[instance.getOffset()] = symbol.name

    return offset_mapping
```

Listing 17: Getting stack addresses from the variables in the decompiler

Following this, the register containing the macro pointer needs to be found. After manually analyzing the decompiled Ghidra code `print` only uses one single argument as a parameter. Thus, it stores the pointer to the macro in `rdi`. Meanwhile, `format_inner` requires two arguments, storing the macro pointer in `rsi` as the second argument and as the first argument it stores the pointer to the string. After determining the parameters for the call, it became necessary to analyze the memory layout for the passed structure. In order to extract the memory information needed `ANGR` offers this functionality by reading `uint64_t` (unsigned integer) values stored at `calculated_offset` location in memory. Since the values stored are concrete and not symbolic, the `res` variable gets assigned with the actual values and finally saved in the `.csv` file as shown in Listing 18.

```
with open('memory_dump.csv', 'w') as f:
    stack_offset = s.regs.rdi
    for i in range(0, 0x1000 // pointer_size):
        calculated_offset = stack_offset + pointer_size * i
        calculated_offset = calculated_offset._model_concrete.value
        res = s.mem[calculated_offset].uint64_t.concrete
        print(f'0x{calculated_offset:016x},0x{res:016x}', file=f)
```

Listing 18: Memory dump layout

The start of the stack is shown in Table 4.1. Since there was no way of mapping the memory dump to the Rust source code in Listing 19 at first glance, a couple of assumptions were made. Proving these assumptions was done by trial and error. A self-written binary was used to try out different ways to call the `println!` macro with various parameters and watching which dump values change accordingly to the code. By following these assumptions and the stack memory addresses in the concrete memory calculated by *ANGR*, the macro input can be derived from stack memory. Other than just deriving the string, format macros can take in more parameters like the precision which also can be detected by looking into the stack memory.

| | | |
|---|---|---|
| 0x07fffffffffefe80 | 0x0000000000000001 | num_format_info |
| 0x07fffffffffefe78 | 0x07fffffffffeff20 | format_info |
| 0x07fffffffffefe70 | 0x0000000000000001 | num_formatters |
| 0x07fffffffffefe68 | 0x07fffffffffeffc0 | fmt_ptr |
| 0x07fffffffffefe60 | 0x0000000000000002 | [&str].len() |
| 0x07fffffffffefe50 | 0x0000000000154c90 | [&str] |

Table 4.1.: Start of memory dump

The figure illustrates a memory layout with memory addresses on the left and their corresponding contents in hexadecimal format on the right. The color-coding is used to group related pairs of memory addresses and contents together, providing a visual aid for explanation. The first memory address (starting from the bottom) contains the content of a string pointer address, followed by a number representing the number of constant string parts. The first value of the next pair points to the `Fmt<f32>` function, followed by a count of formatters. Lastly, the final value points to a slice containing non-default format specifiers.

In the context of the Fmt<f32> function, there is a pointer to its parameters and the function address itself shown in Table 4.2. This data was collected by double checking the address in Ghidra and comparing them to the memory dump generated.

| | | |
|---|---|---|
| 0x07ffffffffeffc8 | 0x0000000000143ab0 | function_addr |
| 0x07ffffffffeffc0 | 0x07ffffffffeff18 | parameter_ptr |

Table 4.2.: *fmt* function in memory

The Fmt<f32> function is associated with a set of parameters. These parameters, namely precision, width, position, fill, and flags, each occupy 4 bytes of memory space, along with an alignment parameter. The Rust library's source code [46, 47] in Listing 19, makes use of a placeholder struct to hold these six relevant parameters. Furthermore, the code utilizes enumerations - such as the `Alignment` enum for alignment, and `Count` enum for precision and width - to store values. If a value for width or precision is specified, they are displayed as `usize`. Otherwise, they are `Implied` values, which means they can contain random old data.

```
pub struct Placeholder {
    pub position: usize,
    pub fill: char,
    pub align: Alignment,
    pub flags: u32,
    pub precision: Count,
    pub width: Count,
}

pub enum Alignment {
    Left,
    Right,
    Center,
    Unknown,
}

pub enum Count {
    Is(usize),
    Param(usize),
    Implied,
}
```

Listing 19: Source code of the placeholder parameters

For *ANGR* to extract and make use of these values the following struct in Listing 20 stores them. Different from the Rust library source code [46, 47] there are the `flags` stored and the extra `padding` variable. This variable is needed because the alignment itself is only 1 byte long and 7 additional bytes are needed because structs are generally aligned to 8 bytes.

```
struct placeholder {
    uint64_t precision_flag;
    uint64_t precision;
    uint64_t width_flag;
    uint64_t width;
    uint64_t position;
    uint32_t fill;
    uint32_t flags;
    uint8_t alignment;
    uint8_t padding[7];
};
```

Listing 20: Memory struct in *ANGR*

Now all the necessary values are stored inside the struct and in order to gain access to the values the memory state needs to be dereferenced. After obtaining these locally stored values are utilized to recreate the call of the `format!` and `println!` macros in Rust. As an example a small program was written in Rust which invokes both function calls as shown in Listing 22. Running the script produces the following output in Listing 21. Not only was the script successful to reconstruct the format strings but also additional parameters like precision and width could be fully restored.

```
println!("Employee ID: {}", local_128)
format_inner!("Name: {} Age: {} Workplace: {}", local_158, local_120, local_140)
println!("Salary: ${:.2}", local_120)
format_inner!("Height: {:13.5} Weight: {:.9}", local_118, local_118)
```

Listing 21: Proof-of-Concept of recreating `format!` and `println!` macros

```
println!("Employee ID: {}", self.id);
format!("Name: {} Age: {} Workplace: {}", self.name, self.age, self.workplace);
println!("Salary: ${:.2}", self.salary);
format!("Height: {:13.5} Weight: {:.9}", self.height, self.weight);
```

Listing 22: Source code of `format` and `println!` macro usage

# 5. Evaluation

This chapter covers the evaluation of real-world Rust binaries. A combination of ten widely-used Rust tools was selected to test the scripts. As described in chapter 4 the developed Ghidra scripts try to simplify the reverse engineering process of Rust binaries. In order to evaluate the string detection scripts, the number and length of detected strings is compared against the strings found by Ghidra and by Ghidra's own RustStringsAnalyzer versus the number of strings detected after running the scripts. To ensure that the string scripts are architecture-independent they must run on the same binary across different architectures. For recreating macros the script currently supports `print!`, `println!`, and `format!`. To validate and evaluate the script, it will be executed on real-world binaries and the results will be thoroughly documented.

## 5.1. Setup

In order to evaluate the string analysis techniques in Ghidra two different versions were used. The main focus was to test the scripts on Ghidra v10.3.2, the version they were developed on. Additionally v11.0.1 was also used which introduces Ghidra's own Rust-StringsAnalyzer and RustDemangler. The setup process began with manually importing the needed binaries into a new Ghidra project for both versions. The RustStringsAnalyzer and RustDemangler in the recent Ghidra version were disabled to create a baseline whether more or less strings get detected. Ghidra also has a headless analyzer [48] which allows it to analyze binaries with command-line commands. It can create projects, import binaries, and execute scripts, which is particularly useful for repetitive tasks. Therefore, basic shell scripts were developed accepting the project path, name, and script path as arguments. Additionally, the headless analyzer allows keeping the project read-only, allowing analysis without modifying the underlying project. The data extracted from the binaries included the number of strings before and after script analysis, as well as their lengths. This information is saved as a `.csv` file and processed using Python scripts to generate histograms and tables that illustrate the impact of the scripts. The baseline values from the new Ghidra version without the RustStringsAnalyzer and RustDemangler matched the values from the old version with no difference being visible in the graphs. Afterwards, in the new Ghidra version the new analysis scripts were enabled to get the new baseline values to compare against in the tables.

## 5.2. Comparison of Script Performance

To validate the effectiveness of the scripts finding strings within binaries, ten widely-used command-line executables were chosen. These included *bat*, *btm*, *dust*, *exa*, *fd*, *procs*, *rg*, *starship*, *tldr* and *tokei* (linked in Appendix A). By using the `file` command it was possible to check if the binary was stripped of symbol and debug information. Since most real-world binaries are typically stripped to reduce file size and make it harder to analyze or exploit vulnerabilities, this thesis focused on stripped binaries to test the scripts against. Five binaries from the collection were stripped: *bat*, *btm*, *starship*, *ripgrep*, and *dust*. Additionally, the difference between the stripped and non-stripped binaries is evaluated in regards to the analysis scripts of this thesis.

Figure 5.1 shows the average string count calculated by our scripts relative to the baseline for all binaries. The process involved running both dynamic and static scripts on each binary, extracting the string count, and then dividing the result by the baseline of strings obtained solely by Ghidra's auto-analysis. The resulting values were then plotted on a graph. The goal is to find a representative example that clearly shows the script's functionality and effectiveness. Two significant binaries score higher than the rest. *Dust* with having around 19 times more strings than Ghidra's auto-analysis alone and *ripgrep* by approximately 17 times. In contrast, *bat*, *exa*, *fd*, *procs*, *tldr*, and *tokei* show only minor improvements. *Starship* and *btm* experienced a noticeable boost in the total string count.

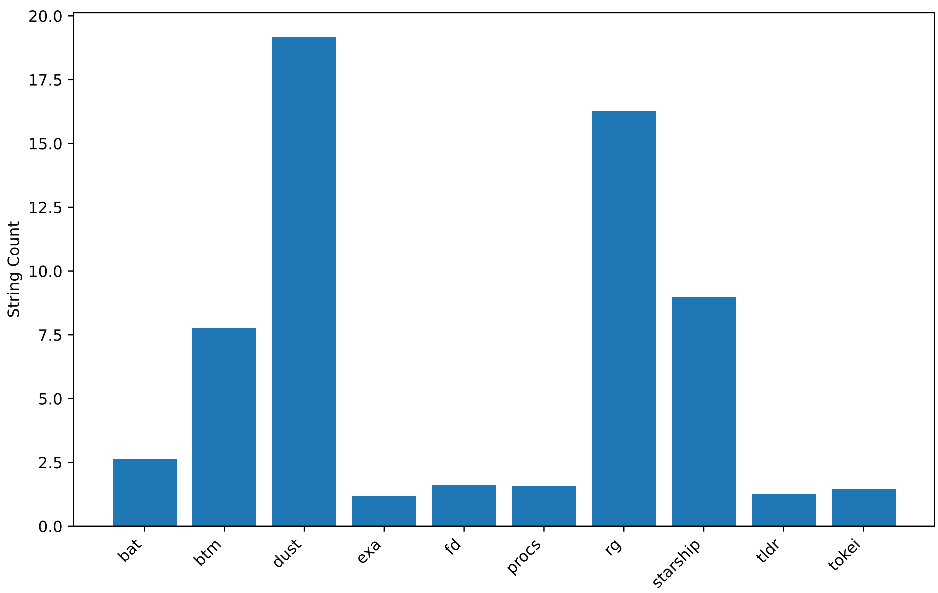


Figure 5.1.: Average string count relative to baseline for all binaries

## 5.3. Comparison of String Count

After reviewing the graph, *ripgrep* seemed the most appropriate due to its popularity and the fact that it recovers a lot of strings. The following graph in Figure 5.2 presents the number of strings found in the binary based on the script being run. Ghidra by default identifies approximately 400 strings, which is relatively low compared to other available scripts. Ghidra's own RustStringsAnalyzer finds about 2500 strings whereas the static strings script locates around 5500 strings. The dynamic script alone found about 2000 strings. An important point is that combining both static and dynamic string analysis resulted in the detection of more strings than the RustStringsAnalyzer. However, the dynamic script generates a greater number of false positives due to checking the instructions directly and possibly finding useless data that still matches the pattern. Additionally, the dynamic string code is designed to avoid replacing strings that were already defined. Therefore, using the static string scripts to clean up the defined data after the dynamic script should in theory lead to better results. Nevertheless, the graph shows that the order of script execution does not have a big impact on the overall number of strings detected.

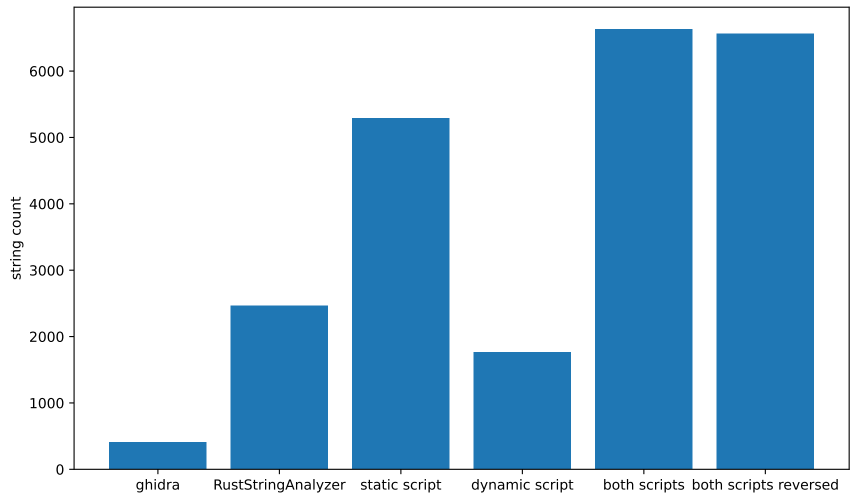


Figure 5.2.: Number of strings found on stripped binary

Figure 5.3 illustrates the results when performing an identical analysis to Figure 5.2 on non-stripped binaries. It shows that the non-stripped version had a higher overall string count due to the inclusion of symbol and debug information. What is interesting here is that the RustStringsAnalyzer discovers more strings than the dynamic script. One reason for this could be that the RustStringsAnalyzer script does not have an instruction interval of where the length of the string needs to be encoded. The dynamic script looks one instruction before the string pointer and five afterwards. However, the

RustStringsAnalyzer continues searching until it can find a length which is not negative, zero or smaller than the max length. When trying to implement it this way in the dynamic script, the dynamic script detection resulted in many unrelated data fields getting detected as strings. Most of the strings were found by the static script and by running both scripts the total count got slightly increased. Despite the difference of stripped and non-stripped binaries, the execution order of the scripts also has an effect on the number of identified strings. This is due to the fact that the dynamic script is designed to avoid the replacement of strings that have already been defined.

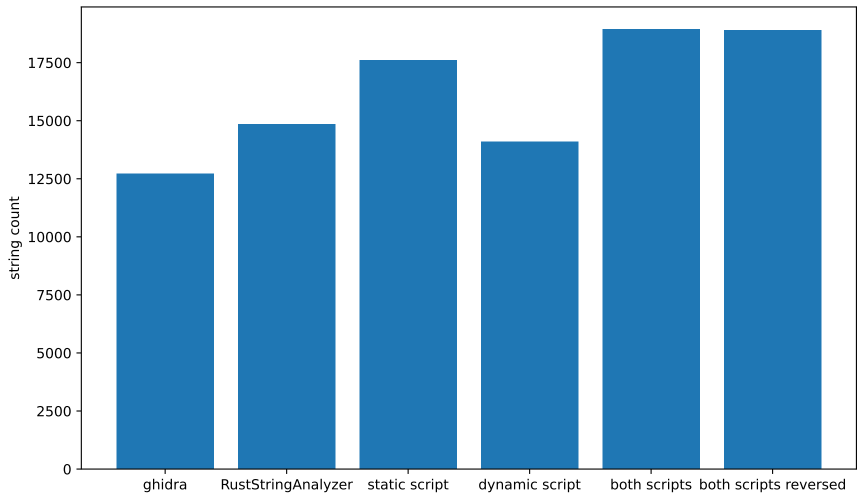


Figure 5.3.: Number on strings found on non-stripped binary

The length of the undetected string from Figure 5.4 is not explicitly encoded in the instruction sequence of the program. This information for this specific string is only found in the `.rodata` section. Therefore, when running Ghidra's RustStringsAnalyzer which takes a similar approach to the dynamic script by only looking for strings with length and pointer in the instruction operands, the string cannot be detected. However, the static string script inspects the `.rodata` section directly to search for the string's content and associated length, allowing detection and string definition as shown in Figure 5.5.

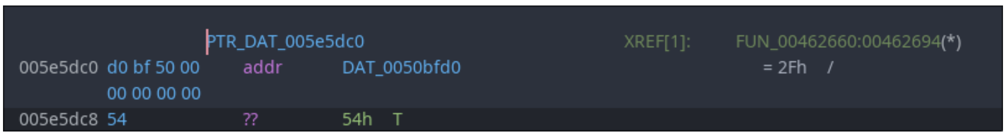


Figure 5.4.: Undetected string from Ghidra's RustStringsAnalyzer

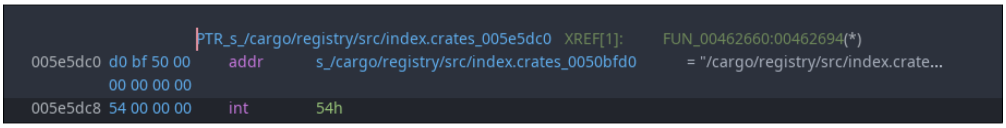


Figure 5.5.: Detected string with static string script

## 5.4. Comparison Across Architectures

The scripts were initially developed and tested on `x86_64` architecture to develop a first prototype. The next step was to adapt the architecture specific script making it architecture-independent to check the script's adaptability and effectiveness in handling binaries with different instruction sets and hardware platforms.

Figure 5.6 compares the architecture-specific script against the refactored independent script on a stripped *ripgrep* binary. Both scripts were tested on Ghidra v11 and with the RustStringsAnalyzer and RustDemangler turned off. It is noteworthy that the architecture independent script identified more strings in total than the architecture-specific script. This could be due to the fact that the architecture-specific script hardcoded the instruction names instead of searching for the according pattern (string pointer with a close length encoding) in the disassembly.

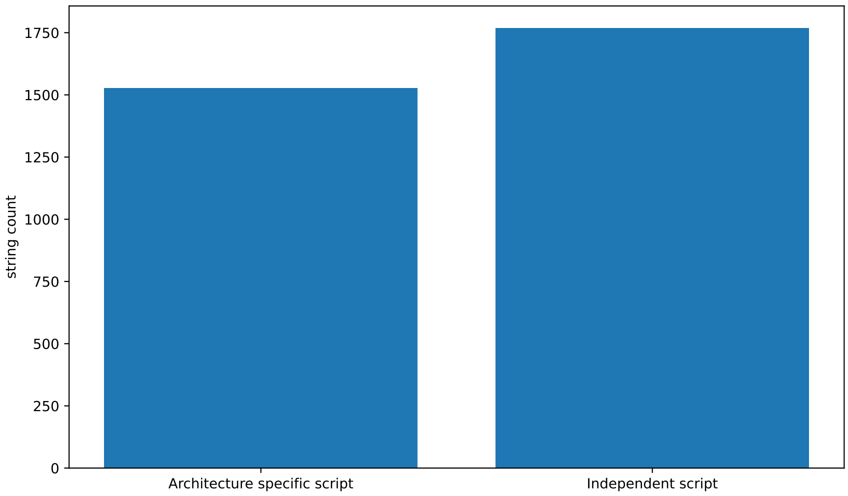


Figure 5.6.: Different versions of dynamic script on `x86_64` architecture

*Ripgrep* [49] has its source code publicly available on GitHub. By cloning the repository it was possible to compile the code across multiple architectures, including `armv7`, and `aarch64` for evaluating the scripts. The binaries were imported in Ghidra with the RustStringsAnalyzer and RustDemangler disabled. Afterwards, the shell scripts for running the static string script and dynamic script were performed on all three binaries. Lastly, the three binaries were reimported with Ghidra's full analysis enabled and the data was extracted for testing how well the RustStringsAnalyzer script performs across architectures.

The three diagrams below visualize the number of strings found across different architectures. An evaluation of the data in Figure 5.7 provides evidence that the number of strings returned by Ghidra's RustStringsAnalyzer varies by architecture. The most strings were discovered on the `x86_64` architecture, followed by `armv7`, and lastly `aarch64`. One reason for this could be that Ghidra's RustStringsAnalyzer follows a similar approach like the dynamic string script. The script searches for string structures in the binary by creating an iterator that traverses the program address space and returns instances of string data. For each string, it calls `recurseString` to recursively split the string into slices with the correct length.

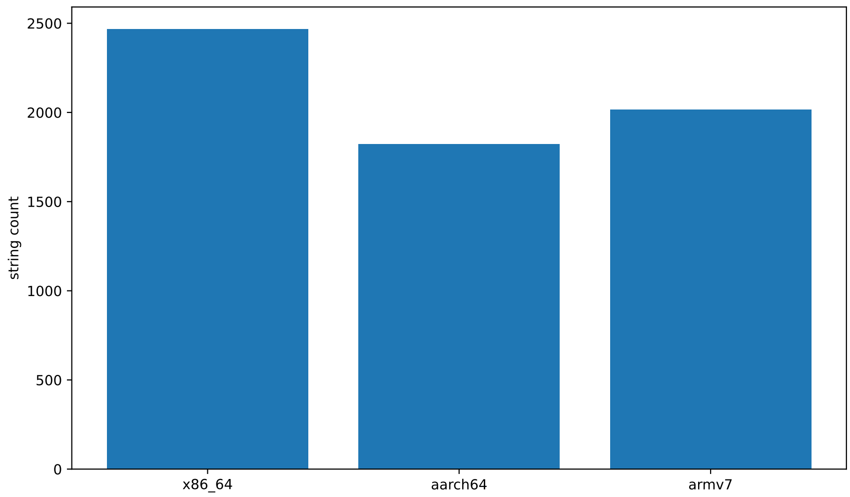


Figure 5.7.: Architecture-dependent Ghidra RustStringsAnalyzer

Additionally, the dynamic script also identifies varying numbers of strings across different architectures as shown in Figure 5.8. This may be due to the fact that "dynamic" strings store the string pointer and length directly in the instructions, which can vary across different architectures. In conclusion, since both scripts rely on how the string data is encoded in the instruction, their performance may not be entirely reliable. Instead, the static string scripts only search for string structures stored in the `.rodata` section which stays similar across different architectures. Thus, resulting in a more reliable script across architectures as shown in Figure 5.9.

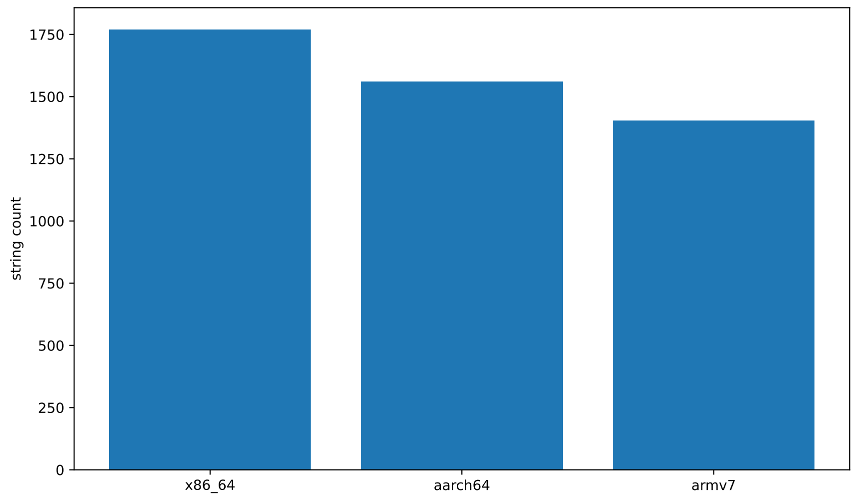


Figure 5.8.: Architecture-dependent dynamic string script

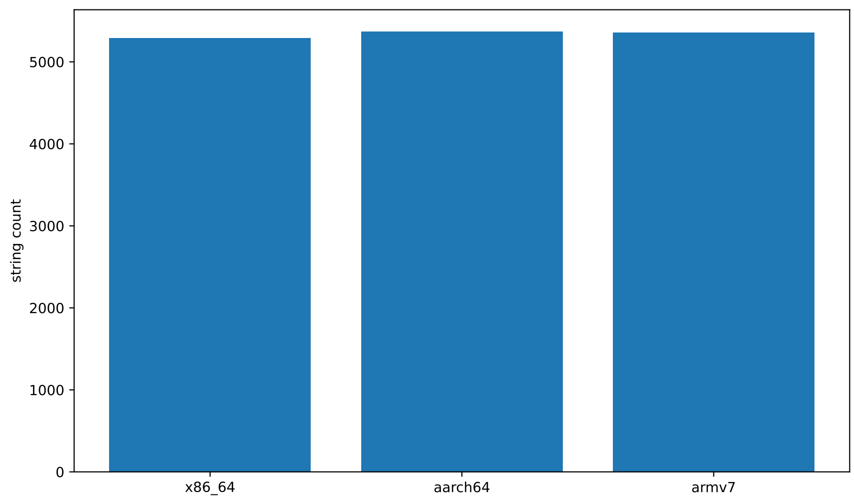


Figure 5.9.: Architecture-independent static string script

## 5.5. Comparison of Length Distribution

Simply evaluating the total number of strings is not enough to show the validity of the script's results. For a more detailed analysis, the length of each string found in the stripped *ripgrep* binary was extracted. The goal is to analyze the distribution of string lengths in the binary, evaluating whether the majority of strings are shorter or longer. This data was collected for Ghidra's analysis both with and without the Rust analyzers enabled. The developed scripts were run without Ghidra's Rust analysis activated. The reason for this evaluation is that if the binary contains unusually long strings, this may indicate that the scripts failed to correctly split existing strings into shorter single strings.

The boxplot from Figure 5.10 presents the distribution of string lengths within the *ripgrep* binary. Each boxplot displays the range and distribution of the string lengths, with outliers marked separately. Ghidra's auto-analysis shows a broader distribution of string lengths with many extremely long strings. For example, the third quartile shows that 25% of the strings exceed a length of roughly 270, which contrasts to the static string representation, where 25% only exceed a length of 70. This suggests that in the default analysis of Ghidra, 75% of the strings are shorter than 270, while in the static representation, the majority are shorter than 70.

In contrast, the second plot enables Ghidra's own RustStringsAnalyzer and shows a shift towards shorter strings, with 75% of the strings below approximately 100. This indicates a bias towards shorter strings over longer ones. Despite this distribution, the plot still reveals the presence of numerous outliers, which represent a significant number of long strings that have not been split.

The static string effectively splits extremely long strings, eliminating those above 700. In contrast, the dynamic string script mirrors Ghidra's default string analysis to some extent, albeit producing slightly shorter strings than Ghidra's default output. Although this script is effective in splitting some very long strings, it does not address all outliers to the same extent as the static script. Nevertheless, its ability to manage and split longer strings represents an improvement over Ghidra's default functionality.

Both static and dynamic script representations show a high count of short strings but they are also significantly higher in count compared to Ghidra. The static string script identifies the shortest string length. Overall, these analyses demonstrate a shift towards shorter string lengths in subsequent representations, with varying degrees of success in dealing with outliers and splitting up extremely long strings.

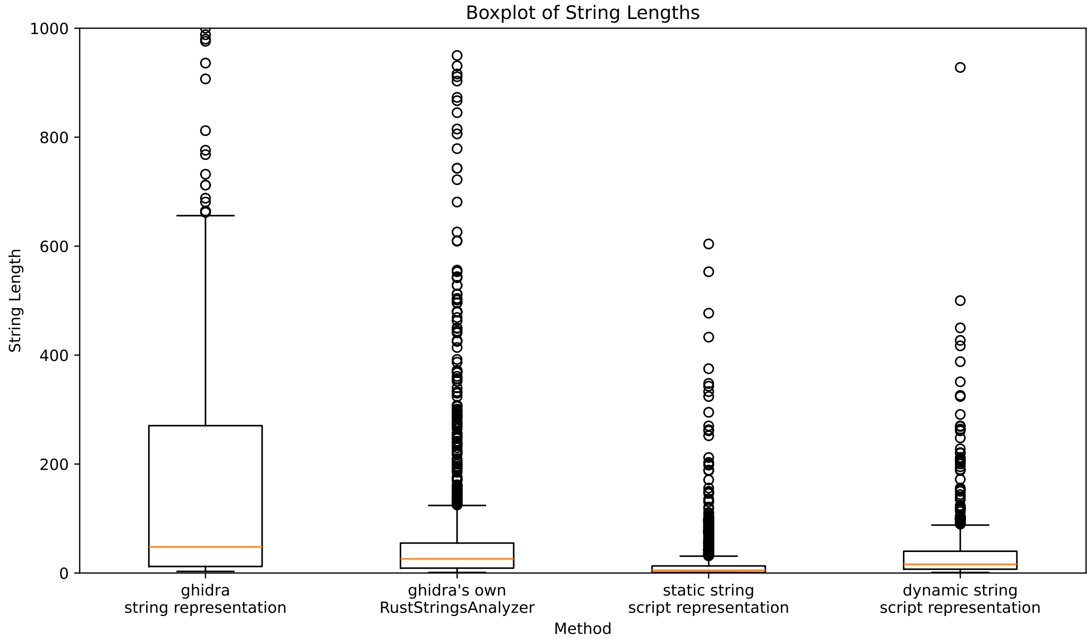


Figure 5.10.: String length distribution of scripts in comparison with Ghidra

The string search algorithm was designed to skip strings that were already defined at an address. Running one of the two scripts could result in more or fewer strings being found in total due to the already defined strings from the first script being run. Additionally, the dynamic script could sometimes create strings that were longer than they actually are because of how the information was encoded in the instruction operands. If the string was also stored in the `.rodata` section the static string would redefine the string with the new length. This sequence could potentially result in having more shorter strings than the other way around. To determine whether the number of strings found as well as if the string lengths would change based on the order in which the scripts are executed, both scripts were tested on the same binary in different orders.

The evaluation of the data results in Figure 5.11. Overall, there is a slight difference in the outcome depending on the execution order. However, the assumption that running the scripts in the following order: dynamic and then static produces more shorter strings, does not fully hold since reversing the order identified more strings with lengths around 200. It is important to mention that there are no outliers when using the static string script to clean up. Nevertheless, which order is better depends on the use-case as well as the binary.

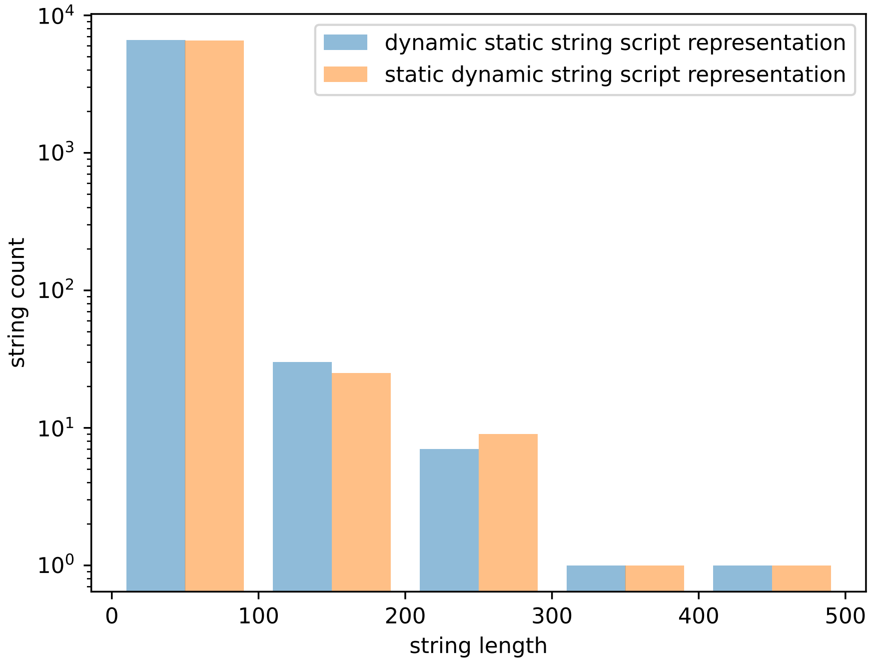


Figure 5.11.: Histogram of string length distribution depending on the order of script execution, x-axis marks the histogram bin boundaries

## 5.6. Comparison of Baselines

As mentioned before, real-world binaries are often stripped of symbol and debug information, so this thesis selected five stripped binaries for the following analysis. Each stripped binary was manually imported into Ghidra with the Rust analysis feature disabled. Afterwards, the headless analysis scripts were executed to run the scripts and extract the string data from the binaries. The process was then repeated, this time enabling Ghidra's Rust analysis before running the headless scripts again.

To establish a baseline for evaluation Ghidra's auto-analysis without the use of additional scripts was used. This baseline provided a reference point for evaluating the effectiveness of the scripts. To measure the performance of the scripts against the baseline, the outcome of the script was divided by the baseline. This calculated factor represents the performance of the scripts in increasing the number of strings identified by Ghidra's auto-analysis alone. For instance, a factor of 1.5 indicates a 50% increase in string discovery compared to the baseline. This analysis was conducted twice, once with Ghidra's Rust analysis feature disabled and once with it enabled.

The first sub-table given in Table 5.1 compares the baseline string counts obtained solely through Ghidra's auto-analysis to the string counts increased by the use of the static script. The factor column measures the increase in string detection by the static script compared to Ghidra's baseline. Notably, *dust* and *ripgrep* have significant improvements, with factors of 16.73 and 12.96, respectively, proving a clear improvement in string detection with the use of static scripts. Compared to other binaries, *bat* showed a slight improvement in string detection by a factor of 2.32. Nonetheless, *btm* and *starship* displayed a significant increase in the number of strings, approximately six times more than Ghidra's auto-analysis alone. In short, the effectiveness of the static scripts is visible in the significant improvements observed across different binaries.

Similarly, the second sub-table compares the baseline string counts obtained via Ghidra's Rust analysis to the total number of strings gained through the use of static scripts. Enabling Ghidra's Rust features results in the discovery of a large number of the strings available. Therefore, when combining it with the static script, the factor which measures the improvement in string detection relative to Ghidra's baseline alone seems to be lower. This implies that Ghidra's Rust features already boost string detection, reducing the relative improvement achieved by the static string scripts alone. Thus, although the static scripts remain effective in general, their impact is reduced when Ghidra's Rust features are enabled.

| Name | Baseline | Static script + baseline | Factor |
|---|---|---|---|
| bat | 2601 | 6022 | 2.32 |
| btm | 662 | 4236 | 6.40 |
| dust | 198 | 3313 | 16.73 |
| ripgrep | 408 | 5289 | 12.96 |
| starship | 710 | 4265 | 6.00 |

(a) Ghidra baseline comparison

| Name | Baseline | Static scripts + baseline | Factor |
|---|---|---|---|
| bat | 3734 | 6832 | 1.83 |
| btm | 2215 | 5368 | 2.42 |
| dust | 925 | 3764 | 4.07 |
| ripgrep | 2468 | 6745 | 2.73 |
| starship | 3955 | 6426 | 1.62 |

(b) Ghidra with RustStringsAnalyzer + RustDemangler baseline comparison

Table 5.1.: Comparison of static string baseline

The second table in Table 5.2 compares the baseline string counts collected from Ghidra's auto-analysis alone to those extended by the developed dynamic scripts. Here *ripgrep* and *starship* with factors of 4.34 and 4.35 experience almost the same improvement. Similarly *dust* shows a significant increase in the number of strings, roughly three to four times more than Ghidra's auto-analysis. In contrast, *bat* shows only minor improvements here.

The sub-table compares Ghidra's baseline with RustStringsAnalyzer and RustDemangler to the dynamic scripts. Compared to the first sub-table with only the dynamic scripts executed, enabling Ghidra's Rust features only results in a slight increase in string detection, as indicated by the factors in this sub-table remaining close to one. While some binaries, such as *dust* and *starship*, show slightly higher factors, suggesting a minor improvement, the overall impact of the scripts in addition to Ghidra's Rust features appears limited.

| Name | Baseline | Dynamic scripts + baseline | Factor |
|---|---|---|---|
| bat | 2601 | 3487 | 1.34 |
| btm | 662 | 1611 | 2.43 |
| dust | 198 | 726 | 3.67 |
| ripgrep | 408 | 1769 | 4.34 |
| starship | 710 | 3085 | 4.35 |

(a) Ghidra baseline comparison

| Name | Baseline | Dynamic scripts + baseline | Factor |
|---|---|---|---|
| bat | 3734 | 4029 | 1.08 |
| btm | 2215 | 2371 | 1.07 |
| dust | 925 | 1085 | 1.17 |
| ripgrep | 2468 | 2710 | 1.10 |
| starship | 3955 | 4520 | 1.14 |

(b) Ghidra with RustStringsAnalyzer + RustDemangler baseline comparison

Table 5.2.: Comparison of dynamic string baseline

The last table in Table 5.3 compares the baseline string counts collected from Ghidra's auto-analysis to those extended by running the dynamic script first and then the static script. Comparing the factors to those from Table 5.1 and Table 5.2, it appears that the factors from the combined script application tend to be higher than those obtained from single-script applications. In Table 5.1 the factor for *dust* is 16.73 and in Table 5.2 it is 3.67. Combining these values results in a slightly higher number than the one in Table 5.3 but it still seems like the addition of both values. This pattern is also consistent with the other binaries. It is also clearly visible that enabling Ghidra's Rust analysis limits the effectiveness of the scripts keeping the factors low compared to the first sub-table.

| Name | Baseline | Both scripts + baseline | Factor |
|---|---|---|---|
| bat | 2601 | 6860 | 2.64 |
| btm | 662 | 5130 | 7.75 |
| dust | 198 | 3795 | 19.17 |
| ripgrep | 408 | 6631 | 16.25 |
| starship | 710 | 6374 | 8.98 |

(a) Ghidra baseline comparison

| Name | Baseline | Both scripts + baseline | Factor |
|---|---|---|---|
| bat | 3734 | 7140 | 1.91 |
| btm | 2215 | 5558 | 2.51 |
| dust | 925 | 3912 | 4.23 |
| ripgrep | 2468 | 6979 | 2.83 |
| starship | 3955 | 6987 | 1.77 |

(b) Ghidra with RustStringsAnalyzer + RustDemangler baseline comparison

Table 5.3.: Comparison of both string baseline

## 5.7. Format Macros

During the development phase, a short test binary was written and used to validate and verify that the code works. The transition to real-world binaries was intended but the script was not specifically designed for such binaries. Nevertheless, the script was tested on *exa* which was randomly selected from the available set of binaries. For the evaluation, both the dynamic and static script were executed on the binary. After running the Java script which whole purpose is to identify `println!`, `print!`, and `format_inner!` calls, the code managed to obtain a total of two `println!` calls and six `format_inner` function calls as shown in Listing 23. However, the execution of the Python MacroExplorer script came across many difficulties while trying to reconstruct the macros.

```
format_inner 001ba2e4
format_inner 0014840a
_print 0014fe38
_print 00110e4d
format_inner 0014a47f
format_inner 00147f7d
format_inner 0014a402
format_inner 00110b63
```

Listing 23: List of macro calls found in the *exa* binary

One problem encountered was that although the script managed to run through the first call of `format_inner`, the printed outcome, shown in Listing 24, has the string duplicated.

```
format!("invalid magic number{}invalid magic number{}", local_60, local_78)
```

Listing 24: Wrong function call due to duplicated string

Upon examining the decompiled code in Ghidra, it was found that the function utilizing the `format_inner` call is called `pad_string` [50]. Further research into the original source code of this function, revealed that the `format!` call did indeed generate an empty "duplicate" string as shown in Listing 25.

```
fn pad_string(string: &str, padding: usize, alignment: Alignment) -> String {
    if alignment == Alignment::Left {
        format!("{}{}", string, spaces(padding))
    }
    else {
        format!("{}{}", spaces(padding), string)
    }
}
```

Listing 25: Source code of `pad_string` function [50]

Upon inspecting the address where Ghidra defined the `invalid magic number` string, it was determined that the length of the actual string should have been zero. One possible reason why Ghidra did not find or define this string is that it may have been invalid. Ghidra marks strings only if they have been referenced somewhere and ignores if the string has the necessary length information to do so. The static and dynamic scripts do not delete already defined strings, but only split them up if needed. Therefore, it was not possible for the developed scripts to recover the empty string. This was also tested with Ghidra's RustStringsAnalyzer which also resulted in Ghidra not defining the empty string. To fix this, the current Python script was rewritten to check if the string length was zero shown in Listing 26 . Otherwise, the corresponding string defined from Ghidra at the address is returned.

The modified script was rerun, resulting in the correct output shown in Listing 27. This is identical to the `format!` call from the `pad_string` source code.

```
final_str = []
for i in range(num_strings):
    if str_array[i].len.uint64_t.concrete == 0:
        final_str.append('')
    else:
        string_address = toAddr(str_array[i].str.concrete)
        string_value = getDataAt(string_address).getValue()
        final_str.append(string_value)
```

Listing 26: Fix to append an empty string if the length is zero

```
format!("{}{}", local_60, local_78)
```

Listing 27: Correct function call after implementing the fix

Moving on to the next function call the script failed to terminate. After leaving the script running for approximately two minutes it was terminated manually. The reason for the script's failure is unclear. The delay in reaching the desired function call may be due to the large number of function calls that *ANGR* follows. Another reason could be that the concrete return value destroys *ANGRs* exploration path, leading to the script running for a long time.

The third and fourth call from the binary was `println!` which resulted in the script throwing an exception due to failing to find a valid state. This error may have been caused by Ghidra's incorrect decompilation of the caller function, which was labeled as an `UndefinedFunction` as shown in Listing 28. Furthermore, the `print` function is missing parameters, possibly due to Ghidra's failure to decompile the function correctly.

```
void UndefinedFunction_0014fe30(void) {
    void *unaff_RBX;
    long unaff_R14;
    long *plVar1;
    bool bVar2;
    long in_stack_00000098;

    std::io::stdio::_print();
}
```

Listing 28: Ghidra fails to mark caller function correctly

The next call turned out to be another empty string which was correctly identified, as shown in Listing 29. When compared to the source code from *exa* Listing 30 there are two `format!` calls. The second one was reconstructed from the script but since the string was invalid and empty, the Python script only appended the curly braces since these were hardcoded. However, the first `format!` call resulted in the script crashing with the error, that no valid state was found.

```
format!("<{}>", local_c0)
```

Listing 29: Another correctly identified macro

```
let error_message = if let Some(path) = path {
    format!("<{}: {}>", path.display(), error)
} else {
    format!("<{}>", error)
};
```

Listing 30: Source code from exa

During the development phase of the *ANGR* script, when creating the state, there is an option called `ZERO_FILL_UNCONSTRAINED_MEMORY`. This option replaces the unknown memory from an uninitialized address to zero instead of an unconstrained symbol [51]. However, this option caused the script to fail to find a possible branch to get into the `if` branch of the code. Removing this option from the script led to correctly extracting the macro and string from the binary shown in Listing 31.

```
format!("<{}: {}>", local_40, local_c0)
```

Listing 31: Correctly identified macro after removing the zero-initialized option

The recent changes to the script have improved its ability to handle different calls within the binary. For example, the last call previously terminated with an error showing an invalid state. However, this issue was resolved by removing the zero-initialized option. This fix led to the discovery of the `format!` call in the list which contained a non-empty string. A comparison between the output of the script in Listing 32 and the source code Listing 33 confirms that the script can successfully reconstruct macros.

```
format!("/usr/share/zoneinfo/{}", local_368)
```

Listing 32: Correctly identified macro after removing the zero-initialized option

```
format!("/usr/share/zoneinfo/{}", {
    if file.starts_with(':') {
        file.replacen(':', "", 1)
    } else {
        file
    }
})
```

Listing 33: `format!` macro call in the *exa* binary for comparison

After the implementation of the above changes, the script was run one more time to see how well it performed now, and if there were any changes to the old results. The modifications did not impact the outcomes of previous successful calls. However, calls where the script crashed or failed to terminate now show different results. For instance, the two `format_inner` calls which needed to be terminated manually crash with the following exception in Listing 34.

```
ValueError: Exceeds the limit (4300 digits) for integer string conversion:
value has 9864 digits; use sys.set_int_max_str_digits() to increase the limit
```

Listing 34: *ANGR* error if script fails to terminate

Furthermore, there were inconsistencies of the script handling both `print` calls. The first print call terminated without producing an error, but no output was printed. This may have been due to the missing arguments in the `print` function within Ghidra. In contrast, the second print call threw the same exception as the previous two discussed `format_inner` calls, shown in Listing 34.

# 6. Conclusion

To conclude, this thesis tried to close the gap in reverse engineering Rust binaries by developing and evaluating string recovery and macro reconstruction scripts and methods. The final results of chapter 5 show that the scripts provide useful information about the functionality of Rust binaries as well as trying to make the reverse engineering process easier. By developing scripts to reconstruct both static and dynamic strings, it was possible to outperform Ghidra's RustStringsAnalyzer which only tries to recover strings which are encoded in the instructions of the Rust binary. Additionally, `format!`, `print!`, and `println!` macros can be detected and reconstructed with the help of *ANGR*. While it is possible to recover macros in general, the evaluation has also shown that this is limited by two things: In order for *ANGR* to work properly, Ghidra has to produce correct decompiled code; if the decompiled code is not correct, then *ANGR* can also not correctly perform symbolic execution. Secondly, *ANGR* itself is not free of errors and due to symbolic execution facing issues with path explosion, it might not terminate for complicated parts in the code.

## 6.1. Future Work

Concerning future work, there are a large number of different areas in reverse engineering Rust binaries that need further improvement. For example, developing scripts that can handle more advanced macro structures and are robust against edge cases. Rather than simply printing macro structures to the console, the ultimate goal is to display every reconstructed macro structure directly in Ghidra's decompiler interface. This can be done by replacing the existing decompiled code by the correct macro calls or by including them as code comments throughout the binary. By achieving this goal, the reverse engineering process with tools such as Ghidra will become easier. In summary, it is important to continue research in this field to overcome challenges related to reverse engineering Rust binaries.

# List of Tables


4.1. Start of memory dump . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.2. *fmt* function in memory . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

5.1. Comparison of static string baseline . . . . . . . . . . . . . . . . . . . . . . . . 37
5.2. Comparison of dynamic string baseline . . . . . . . . . . . . . . . . . . . . . . . 38
5.3. Comparison of both string baseline . . . . . . . . . . . . . . . . . . . . . . . . 39

# List of Figures


4.1. Edge case 1: Multiple string pointers pointing to different locations . . . . 16
4.2. Edge case 2: String is split into smaller blocks . . . . . . . . . . . . . . . . 17
4.3. Redefining every split string . . . . . . . . . . . . . . . . . . . . . . . . . 19

5.1. Average string count relative to baseline for all binaries . . . . . . . . . . . 28
5.2. Number of strings found on stripped binary . . . . . . . . . . . . . . . . . 29
5.3. Number on strings found on non-stripped binary . . . . . . . . . . . . . . 30
5.4. Undetected string from Ghidra's RustStringsAnalyzer . . . . . . . . . . . 30
5.5. Detected string with static string script . . . . . . . . . . . . . . . . . . . 31
5.6. Different versions of dynamic script on `x86_64` architecture . . . . . . . . 31
5.7. Architecture-dependent Ghidra RustStringsAnalyzer . . . . . . . . . . . . 32
5.8. Architecture-dependent dynamic string script . . . . . . . . . . . . . . . . 33
5.9. Architecture-independent static string script . . . . . . . . . . . . . . . . . 33
5.10. String length distribution of scripts in comparison with Ghidra . . . . . . 35
5.11. Histogram of string length distribution depending on the order of script execution, x-axis marks the histogram bin boundaries . . . . . . . . . . . . 36

# List of Listings


1. Macro expanded at compile-time . . . . . . . . . . . . . . . . . . . . . . . 2
2. Original rust code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
3. Hello World program in Rust with ownership and borrowing models . . . 6
4. String slices in Rust . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
5. Simple Hello World program in C . . . . . . . . . . . . . . . . . . . . . . 7
6. Decompiled C Hello World program in Ghidra . . . . . . . . . . . . . . . . 7
7. Simple Hello World program in Rust . . . . . . . . . . . . . . . . . . . . 7
8. Decompiled Rust Hello World program in Ghidra . . . . . . . . . . . . . . . 8
9. Stripping symbols from binary . . . . . . . . . . . . . . . . . . . . . . . 13
10. Static strings in Ghidra . . . . . . . . . . . . . . . . . . . . . . . . . 15
11. Dynamic strings in Ghidra . . . . . . . . . . . . . . . . . . . . . . . . 15
12. Check for UTF-8 printable characters . . . . . . . . . . . . . . . . . . . 16
13. CodeUnitException due to data conflicts . . . . . . . . . . . . . . . . . 16
14. Pattern for dynamic strings in x86_64 in Rust . . . . . . . . . . . . . . 17
15. Algorithm for searching for string address . . . . . . . . . . . . . . . . 18
16. Error shown when trying to import `angr` . . . . . . . . . . . . . . . . . 20
17. Getting stack addresses from the variables in the decompiler . . . . . . . 22
18. Memory dump layout . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
19. Source code of the placeholder parameters . . . . . . . . . . . . . . . . . 24
20. Memory struct in *ANGR* . . . . . . . . . . . . . . . . . . . . . . . . . 25
21. Proof-of-Concept of recreating `format!` and `println!` macros . . . . . . . 25
22. Source code of `format` and `println!` macro usage . . . . . . . . . . . . . 25
23. List of macro calls found in the *exa* binary . . . . . . . . . . . . . . . 39
24. Wrong function call due to duplicated string . . . . . . . . . . . . . . . 40
25. Source code of `pad_string` function [50] . . . . . . . . . . . . . . . . . 40
26. Fix to append an empty string if the length is zero . . . . . . . . . . . . 41
27. Correct function call after implementing the fix . . . . . . . . . . . . . 41
28. Ghidra fails to mark caller function correctly . . . . . . . . . . . . . . 41
29. Another correctly identified macro . . . . . . . . . . . . . . . . . . . . 42
30. Source code from exa . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
31. Correctly identified macro after removing the zero-initialized option . . . . 42
32. Correctly identified macro after removing the zero-initialized option . . . . 43
33. `format!` macro call in the *exa* binary for comparison . . . . . . . . . . 43

34. *ANGR* error if script fails to terminate . . . . . . . . . . . . . . . . . . . . . 43

# Bibliography


[1] M. Research, "Rust Programming Language." https://www.rust-lang.org/. [Accessed 21-02-2024].

[2] "Macros - The Rust Programming Language — doc.rust-lang.org." https://doc.rust-lang.org/book/ch19-06-macros.html. [Accessed 20-Jul-2023].

[3] "Hello Tokio | Tokio - An asynchronous Rust runtime." https://tokio.rs/tokio/tutorial/hello-tokio. [Accessed 22-02-2024].

[4] N. S. Agency, "Ghidra." https://ghidra-sre.org/. [Accessed 20-02-2024].

[5] "angr." https://angr.io/. [Accessed 09-10-2023].

[6] benhe, "Rust Binary Analysis, Feature by Feature - Check Point Research — research.checkpoint.com." https://research.checkpoint.com/2023/rust-binary-analysis-feature-by-feature/, June 2023. [Accessed 19-Jul-2023].

[7] M. Giordano, "Rust Disassembly: part 1 — giordi91.github.io." https://giordi91.github.io/post/disassemlbyrust1/. [Accessed 21-Jul-2023].

[8] Zeropio, "Reversing Rust 1 Research — zeropio.ninja." https://zeropio.ninja/research/rustversing/part-1. [Accessed 21-Jul-2023].

[9] B. Defense, "Digging through Rust to find Gold: Extracting Secrets from Rust Malware | Binary Defense — binarydefense.com." https://www.binarydefense.com/resources/blog/digging-through-rust-to-find-gold-extracting-secrets-from-rust-malware/. [Accessed 21-Jul-2023].

[10] Binary Defense, "GhidraRustDependenciesExtractor." https://github.com/BinaryDefense/GhidraRustDependenciesExtractor/blob/main/RustDependencyStrings.py. [Accessed 21-Jul-2023].

[11] D. Maroo, "GhidRust: Rust decompiler plugin for Ghidra." https://github.com/DMaroo/GhidRust/blob/master/media/report.pdf. [Accessed 20-Jul-2023].

[12] D. Maroo, "GhidRust: Rust decompiler plugin for Ghidra." https://github.com/DMaroo/GhidRust. [Accessed 20-Jul-2023].

[13] str4d, "Ghidra script for demangling Rust symbols." https://gist.github.com/str4d/e541f4c28e2bca80d222434ac1a204f4. [Accessed 20-Jul-2023].

[14] "2603-rust-symbol-name-mangling-v0 - The Rust RFC Book — rust-lang.github.io." `https://rust-lang.github.io/rfcs/2603-rust-symbol-name-mangling-v0.html`. [Accessed 21-Jul-2023].

[15] NationalSecurityAgency, "RustStringAnalyzer." `https://github.com/NationalSecurityAgency/ghidra/blob/921247f640c9a313bce80dc26e0f99939ddae4ad/Ghidra/Features/Base/src/main/java/ghidra/app/plugin/core/analysis/rust/RustStringAnalyzer.java`. [Accessed 30-01-2024].

[16] NationalSecurityAgency, "RustDemangler." `https://github.com/NationalSecurityAgency/ghidra/blob/921247f640c9a313bce80dc26e0f99939ddae4ad/Ghidra/Features/Base/src/main/java/ghidra/app/plugin/core/analysis/rust/demangler/RustDemangler.java`. [Accessed 30-01-2024].

[17] rust lang, "Unsafe Rust - The Rust Programming Language." `https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html`. [Accessed 23-Jul-2023].

[18] rust lang, "Clippy: A collection of lints to catch common mistakes and improve your Rust code." `https://github.com/rust-lang/rust-clippy`. [Accessed 23-Jul-2023].

[19] rust lang, "Miri: An interpreter for Rust's mid-level intermediate representation." `https://github.com/rust-lang/miri`. [Accessed 23-Jul-2023].

[20] Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim, "Rudra," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ACM, oct 2021.

[21] C. Wang, J. Gao, Y. Jiang, Z. Xing, H. Zhang, W. Yin, M. Gu, and J. Sun, "Go-clone: graph-embedding based clone detector for Golang," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ACM, jul 2019.

[22] "The LLVM Compiler Infrastructure Project — llvm.org." `https://llvm.org/`. [Accessed 21-Jul-2023].

[23] "ThreatIntel/Scripts/Ghidra at master · getCUJO/ThreatIntel — github.com." `https://github.com/getCUJO/ThreatIntel/tree/master/Scripts/Ghidra`. [Accessed 20-Jul-2023].

[24] A. Küchler and C. Banse, "Representing LLVM-IR in a Code Property Graph," in *Information Security* (W. Susilo, X. Chen, F. Guo, Y. Zhang, and R. Intan, eds.), (Cham), pp. 360–380, Springer International Publishing, 2022.

[25] T. E. Dube, B. D. Birrer, R. A. Raines, R. O. Baldwin, B. E. Mullins, R. W. Bennington, and C. E. Reuter, "Hindering Reverse Engineering: Thinking Outside the Box," *IEEE Security & Privacy*, vol. 6, no. 2, pp. 58–65, 2008.

[26] "Hex Rays - State-of-the-art binary code analysis solutions." `https://hex-rays.com/ida-pro/`. [Accessed 20-Jul-2023].

[27] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, "TRIMMER: application specialization for code debloating," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ACM, Sept. 2018.

[28] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective Program Debloating via Reinforcement Learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ACM, Oct. 2018.

[29] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ACM, Dec. 2019.

[30] Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu, and K. Lu, "VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search," in *Proceedings 2023 Network and Distributed System Security Symposium*, Internet Society, 2023.

[31] RazorMegrez, "VulHawk." `https://github.com/RazorMegrez/VulHawk`. [Accessed 24-Jul-2023].

[32] A. Qasem, M. Debbabi, B. Lebel, and M. Kassouf, "Binary Function Clone Search in the Presence of Code Obfuscation and Optimization over Multi-CPU Architectures," in *Proceedings of the ACM Asia Conference on Computer and Communications Security*, ACM, July 2023.

[33] "BinDiff." `https://www.zynamics.com/bindiff.html`. [Accessed 24-Jul-2023].

[34] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How Machine Learning Is Solving the Binary Function Similarity Problem," in *31st USENIX Security Symposium (USENIX Security 22)*, (Boston, MA), pp. 2099–2116, USENIX Association, Aug. 2022.

[35] threatrack, "ghidra-fidb-repo: Ghidra Function ID dataset repository." `https://github.com/threatrack/ghidra-fidb-repo`. [Accessed 24-Jul-2023].

[36] "F.L.I.R.T. 2013; Hex Rays." `https://hex-rays.com/products/ida/tech/flirt/`. [Accessed 24-Jul-2023].

[37] rust lang, "Profiles - The Cargo Book." `https://doc.rust-lang.org/cargo/reference/profiles.html#strip`. [Accessed 08-08-2023].

[38] rust lang, "The Slice Type - The Rust Programming Language." `https://doc.rust-lang.org/book/ch04-03-slices.html`. [Accessed 08-08-2023].

[39] "FlatProgramAPI." `https://ghidra.re/ghidra_docs/api/ghidra/program/flatapi/FlatProgramAPI.html`. [Accessed 07-08-2023].

[40] M. Pucher, "Ghidra Extension Development." `https://appsec.at/blog/2023/07/31/ghidra-extension-development/`, July 2023. [Accessed 04-08-2023].

[41] "AddressSpace." https://ghidra.re/ghidra_docs/api/ghidra/program/model/address/AddressSpace.html. [Accessed 09-08-2023].

[42] dod cyber crime center, "pyhidra: Pyhidra is a Python library that provides direct access to the Ghidra API within a native CPython interpreter using jpype." https://github.com/dod-cyber-crime-center/pyhidra. [Accessed 09-10-2023].

[43] mandiant, "Ghidrathon: The FLARE team's open-source extension to add Python 3 scripting to Ghidra." https://github.com/mandiant/Ghidrathon. [Accessed 10-10-2023].

[44] Justfoxing, "GitHub - Ghidra Bridge: Python 3 bridge to Ghidra's Python scripting." https://github.com/justfoxing/ghidra_bridge. [Accessed 09-10-2023].

[45] "Ghidrathon Incompatible with Python Threads." https://github.com/mandiant/Ghidrathon/issues/7. [Accessed 10-10-2023].

[46] rust lang, "rust/library/core/src/fmt/rt.rs at master · rust-lang/rust." https://github.com/rust-lang/rust/blob/master/library/core/src/fmt/rt.rs#L10. [Accessed 16-10-2023].

[47] rust lang, "rust/library/core/src/fmt/rt.rs at master · rust-lang/rust." https://github.com/rust-lang/rust/blob/master/library/core/src/fmt/rt.rs#L35. [Accessed 16-10-2023].

[48] NationalSecurityAgency, "Headless Analyzer README." https://static.grumpycoder.net/pixel/support/analyzeHeadlessREADME.html. [Accessed 02-02-2024].

[49] BurntSushi, "ripgrep: ripgrep recursively searches directories for a regex pattern while respecting your gitignore." https://github.com/BurntSushi/ripgrep. [Accessed 01-02-2024].

[50] ogham, "rust-term-grid/src/lib.rs." https://github.com/ogham/rust-term-grid/blob/master/src/lib.rs. [Accessed 11-02-2024].

[51] The angr Project contributors, "List of State Options - angr documentation." https://docs.angr.io/en/latest/appendix/options.html. [Accessed 12-02-2024].

# A. Appendix

List of tools used for evaluating the effectiveness of the developed scripts:

*bat*: `https://github.com/sharkdp/bat`
*btm*: `https://github.com/ClementTsang/bottom`
*dust*: `https://github.com/bootandy/dust`
*exa*: `https://github.com/ogham/exa`
*fd*: `https://github.com/sharkdp/fd`
*procs*: `https://github.com/dalance/procs`
*ripgrep*: `https://github.com/BurntSushi/ripgrep`
*starship*: `https://github.com/starship/starship`
*tldr*: `https://github.com/dbrgn/tealdeer`
*tokei*: `https://github.com/XAMPPRocky/tokei`