Computers and Information Technology
2023

DEMYSTIFYING BINARY CODE: A LLVM-ENABLED DECOMPILER

Candidate: Dragoș-Andrei SURUGIU

Scientific coordinator: Prof. Alexandru IOVANOVICI

Session: June 2023

ABSTRACT

The abstract is intended to inform about the content of the paper through a brief description of the research of up to one page, the procedures/methods, as well as its results or conclusions. The abstract in Romanian becomes mandatory for works edited in languages other than Romanian and will be written in 12 pt Nimbus Sans fonts. It will start two blank lines after the heading "ABSTRACT". Before the title, there will be three blank 12 pt. lines.

ABSTRACT

The reverse engineering tool developed in this project is a sophisticated software suite designed to decompile binaries in multiple formats and from different processor architectures into a high-level, domain-specific language that can be easily analyzed and understood by humans. With the ability to detect bugs, analyze malicious executables, and perform static analysis of code for optimization purposes, this tool has broad applications for software developers, security researchers, and computer engineers. It is important to note that the tool is intended for ethical and lawful uses only and will not be utilized for any malicious activities. By leveraging state-of-the-art decompiling techniques, this tool has the potential to unlock insights into the workings of complex software systems that were previously difficult to obtain.

This tool comprises three main modules, each contributing to the overall high-level design. The first module, the "lifter," is responsible for converting raw binary code in formats like ELF, PE, or Mach-O, which have been compiled on different architectures such as x86 or ARM, to LLVM intermediate representation (IR), generating a low-level control flow graph (CFG). The second module, the "universal decompiling machine," performs both data flow analysis and control flow analysis on the CFG generated by the lifter. During data flow analysis, the tool applies optimizations such as dead code elimination, loop simplification, Sparse Conditional Constant Propagation (SCCP), and instruction combination. Additionally, the tool uses interval analysis to detect loops, enabling the identification of opportunities for further optimization. The output of this stage includes the CFG as well as information on loops, loop types, conditionals, variable types, functions, and more. Finally, the "code generation" module combines the CFG with the information gathered from the previous stages to produce an equivalent version of the code in the project's domain-specific language, which is designed to be easily understandable by humans.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

FRAGMENT LIST

1. INTRODUCTION

As computers have become increasingly ubiquitous, software has become more complex to meet the demands of a growing user base.This complexity has led to a greater risk of software bugs and vulnerabilities, which can be exploited by attackers to compromise systems and steal sensitive information. According to a recent study by the Ponemon Institute[1], the average cost of a data breach is now over $4.35 million, highlighting the need for effective cybersecurity measures to protect software systems from attack.

The growing threat of cybersecurity attacks has made software engineering more challenging than ever. Developers must not only create software that meets the functional requirements of users but also ensure that the software is secure and resistant to attack. This requires a deep understanding of software design principles, programming languages, and security best practices.

Another issue that is perfectly described in [2] is that software is often distributed in binary form and even if we have access to the source code it is not always possible to compile it. Therefore, the following phenomenom occurs: the source code doesn't match the binary code, it could be that the intent of the binary is malicious or maybe the optimizations brought by the compiler are not the ones that the developer desired.

To address this challenges, decompilation has emerged as a critical tool for software engineers. Decompilation allows developers to analyze and translate the machine code back into a high-level domain-specific language(DSL) such as C or Java. This DSL is way easier for the developer to understand, enabling the developer to efficiently analyze the code and identify potential security vulnerabilities with a greater degree of confidence. By using decompilation, developers can get a better sense of how their code will behave when it's executed and can make informed decisions about how to optimize it or fix certain undesired behaviours.

Overall, decompilation is an essential tool for modern software engineering. By enabling developers to better understand and modify software, decompilation plays a critical role in improving the security, reliability, and performance of the software that powers our digital lives.

1.1 FIELDS AND OBJECTIVES TARGETED

1.1.1 REVERSE ENGINEERING

Reverse engineering is the process of analyzing existing software to understand how it works. This process can be used for a variety of purposes, including identifying security vulnerabilities, understanding how already existing software works, or simply learning about new programming techniques.

The idea that reverse engineering is reverse forward engineering introduced in [3] is an intriguing concept that highlights the importance of understanding existing software in order to create new and innovative solutions.In traditional software development, engineers start with a set of requirements and then design and implement a solution from scratch. However, as software systems have become more complex, this approach is no longer always practical or efficient. Instead, many developers now use reverse engineering techniques to analyze and understand existing software systems. By reverse engineering a system, engineers can identify its underlying components, dependencies, and design patterns. This information can then be used to create new software systems that are more efficient, secure, and reliable.

In this way, reverse engineering can be seen as a form of forward engineering. By understanding and improving upon existing software systems, developers can create new and innovative solutions that build upon the foundations of what has come before.

1.1.2 BINARY ANALYSIS AND SOURCE CODE ANALYSIS

Binary analysis is a key aspect of reverse engineering, which is used to gain a deep understanding of compiled executable code. This process involves analyzing binary files to identify the underlying logic and structure of software systems.

One of the most important aspects of binary analysis while comparing it to source code analysis is the fact that often the semantics of programming language leave certain aspects not specificed and the compiler is free to choose the implementation. Therefore, in order to do a proper source code analysis one need to take into consideration all possible behaviours of the given code, which is tremendously difficult. On the other hand, binary analysis is more precise because the compiler has already chosen the implementation, and we are left with only one case to consider. This idea is underlined in [2].

A good example of this is shown in the following code snippet:

```c
1  int function(int32_t x)
2  {
3    return x + 1 > x;
4  }
5
6  int main()
7  {
8    int32_t  x = 0x7FFFFFFF;
9    printf("%d\n", function(x));
10   return 0;
11 }
```

Fragment 1.1: Undefined behaviour(UB) in C

The code snippet provided checks whether adding 1 to an integer results in a value greater than the integer itself, which can lead to undefined behavior due to integer overflow. From a binary analysis perspective, the information on line (8) is enough to determine that the adding operation in `function` will always result in an integer overflow. On the other hand, source code analysis would require a deeper understanding of the program semantics, as it needs to consider different possible values of variable x that can lead to undefined behavior.

1.2 THESIS STRUCTURE

In the following chapters of the decompiler thesis, the topic is explored in a structured manner to provide a comprehensive understanding of the decompilation process, namely: lifting a binary to an intermediate representation(IR), conducting a control-flow graph(CFG) analysis, and finally generating the high-level code in the project's target domain-specific language(DSL).

The introduction chapter provides an overview of the thesis and highlights the significance of the decompilation process in software engineering, presenting the concepts of reverse engineering and binary analysis and their importance in creating innovative solutions and high quality software.

The state of the art chapter discusses the current state of decompilation technology and highlights the strengths and weaknesses of existing decompilers, with a focus on two popular tools, Hex-Rays and RetDec and their intermediate representations, specifically Hex-Rays' microcode and RetDec's LLVM IR.

The theoretical foundations chapter delves into the underlying principles of decompilation and the LLVM compiler infrastructure, which is used in the design and implementation of the decompiler. Things like the LLVM functions, basic blocks, modules, optimizations will be discussed in detail, as well as control and data flow analysis techniques presented in [4].

The chapter on design and implementation in the decompiler thesis outlines the methodology employed to construct the decompiler. This includes the selection of programming language, libraries, and algorithms utilized in the process. Additionally, it covers the choice of design patterns and build system, as well as the build generator (CMake) and the server for the lifter service, which is written in Typescript.

The experimental results chapter provides a detailed analysis of the decompiler's performance, accuracy, and scalability, with examples of decompiled code and comparison with existing decompilers.

The last section of this paper summarizes the key findings of the thesis and presents ideas for future work on the decompiler.

2. STATE OF THE ART

2.1 STATE OF THE ART

2.1.1 RETARGETABLE DECOMPILIER(RETDEC)

Each file belonging to the document can be found in `bachelors_en/`. The main file of the template is `main.tex`, which must be compiled using pdflatex (usually the default option). Files in `components/` define the template and **should not be modified**.

In `customs.tex` reside the personal data which appear in places not directly accessible by the user and should be modified. Every file in `chapters/` represents a chapter where the content of the thesis is written. Under `images/` will be placed all the images used in the document.

2.1.2 HEXRAYS(IDA)

3. THEORETICAL FOUNDATIONS

3.1 THEORETICAL FOUNDATIONS

3.1.1 STRUCTURE

3.1.2 AUTHENTICITY DECLARATION

3.2 GENERAL INFORMATION

4. DESIGN AND IMPLEMENTATION

4.1 THEORETICAL ASPECTS

4.1.1 STRUCTURE

4.1.2 AUTHENTICITY DECLARATION

4.2 GENERAL INFORMATION

5. EXPERIMENTAL RESULTS

5.1 EXPERIMENTAL RESULTS

5.1.1 STRUCTURE

5.1.2 AUTHENTICITY DECLARATION

5.2 GENERAL INFORMATION

6. CONCLUSIONS

6.1 BIBLIOGRAPHY

6.2 AUTHENTICITY DECLARATION

7. BIBLIOGRAPHY

[1] *Cost of a data breach 2022 — ibm.com*, https://www.ibm.com/reports/data-breach, 2022.

[2] G. Balakrishnan and T. Reps, “Wysinwyx: What you see is not what you execute,” *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, 2010, ISSN: 0164-0925. DOI: 10.1145/1749608.1749612. [Online]. Available: https://doi.org/10.1145/1749608.1749612.

[3] I. D. Baxter and M. Mehlich, “Reverse engineering is reverse forward engineering,” *Science of Computer Programming*, vol. 36, no. 2, pp. 131–147, 2000, ISSN: 0167-6423. DOI: https://doi.org/10.1016/S0167-6423(99)00034-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642399000349.

[4] C. Cifuentes and K. J. Gough, “Decompilation of binary programs,” *Software: Practice and Experience*, vol. 25, no. 7, pp. 811–829, 1995. DOI: https://doi.org/10.1002/spe.4380250706. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380250706. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380250706.