Islamic University of Technology (IUT)

Department of Computer Science and Engineering (CSE)

Binary Function Clone Detection using MBA Simplification

Authors

Md Abdullahil Kafi - 190041224

Sumit Alam Khan - 190041207

Tasnimul Hasnat - 190041113

Supervisor

Dr. Md. Moniruzzaman

Assistant Professor, Department of CSE

Islamic University of Technology

Co-Supervisor

Imtiaj Ahmed Chowdhury

Lecturer, Department of CSE

Islamic University of Technology

Declaration of Authorship

This is to certify that the research done by **Md Abdullahil Kafi, Sumit Alam Khan**, and **Tasnimul Hasnat** under the supervision of **Dr. Md. Moniruzzaman**, Professor, Department of Computer Science and Engineering (CSE), Islamic University of Technology (IUT), and **Imtiaj Ahmed Chowdhury**, Lecturer, Department of Computer Science and Engineering (CSE), Islamic University of Technology (IUT), resulted in the work presented in this thesis. It is further declared that no part of this thesis or the entire thesis has ever been submitted to another institution for a degree or diploma. A list of references is provided, and information taken from both published and unpublished works of others is recognized in the text.

Authors:

Md Abdullahil Kafi
Student ID- 190041224

Sumit Alam Khan
Student ID- 190041207

Tasnimul Hasnat
Student ID- 190041113

Supervisor:

Dr. Md. Moniruzzaman
Assistant Professor
Department of Computer Science and Engineering
Islamic University of Technology (IUT)

Co-supervisor:

Imtiaj Ahmed Chowdhury
Lecturer
Department of Computer Science and Engineering
Islamic University of Technology (IUT)

Acknowledgement

We would like to express our deepest gratitude to **Dr. Md. Moniruzzaman** for his exceptional guidance, mentorship, and invaluable contributions throughout our thesis work. As a professor in the department of computer science and engineering at IUT, his extensive knowledge and expertise have been instrumental in shaping our research and academic journey.

Furthermore, we would like to extend our thanks to **Imtiaj Ahmed Chowdhury**, Lecturer in the Department of Computer Science and Engineering at IUT. His insightful review of our proposal and simulation techniques played a vital role in shaping our research approach. His encouragement and support motivated us to persevere and continue our hard work.

We acknowledge the significant contributions of Dr. Md. Moniruzzaman and Imtiaj Ahmed Chowdhury to our thesis work. Their guidance, advice, and input have been invaluable in ensuring the successful completion of our research. We are truly grateful for their mentorship and are honored to have had the opportunity to learn from them.

Contents

List of Figures

Abstract

Detecting function clones in closed-source binaries remains a challenging problem, especially when compiler optimizations or obfuscation techniques reshape the code without altering its behavior. These transformations often break the structural patterns that traditional similarity-based or machine learning–based approaches rely on, making it difficult to obtain reliable or interpretable results. In this work, we introduce a method for binary function clone detection that focuses on the semantic behavior of functions by extracting Mixed Boolean–Arithmetic (MBA) expressions through dynamic symbolic execution using the angr framework. The resulting expressions capture the combined arithmetic and logical effects of the assembly instructions, and by simplifying them with off-the-shelf MBA simplifiers and comparing them across functions, paths, and constraints using a structured matching process, we can detect equivalence even under moderate obfuscation and across different compilers and optimization levels. Our evaluation on gcc- and clang-compiled binaries with varying optimization levels, along with several obfuscated samples, yields an overall accuracy of 87.87%, showing that simplified MBA expressions reveal stable behavioral signatures despite significant code transformations. This approach provides a transparent and interpretable alternative to ML-based techniques and offers practical utility for code reuse detection, software integrity verification, and malware analysis in closed-source environments.

Chapter 1

Introduction

In the contemporary software landscape, the prevalence of closed-source applications has become ubiquitous, leading to a critical challenge in ensuring the integrity and authenticity of software systems. The inherent opacity of closed-source code hinders the straightforward verification of potential code reuse and raises concerns about the existence of unauthorized or plagiarized content. This challenge is further compounded by the deployment of various optimizations and obfuscations during the compilation process, rendering traditional methods of plagiarism detection ineffective.

Additionally, the field of cybersecurity grapples with a rising tide of malicious activities, wherein attackers leverage pre-existing malware, making subtle modifications and applying sophisticated obfuscation techniques to outsmart conventional antivirus software. This practice not only underscores the need for advanced detection methods but also highlights the critical role of binary-level analysis in identifying and mitigating potential security threats.

This thesis addresses these pressing issues by proposing a solution to binary function clone detection through the application of Mixed Boolean Arithmetic (MBA) simplification. By focusing on the equivalency of source and target functions, we aim to overcome the limitations posed by closed-source environments and intricate obfuscation techniques. Leveraging dynamic symbolic execution [1], we transform assembly instructions into mathematical expressions, subsequently employing MBA simplification algorithms to facilitate a precise and efficient comparison of function equivalency.

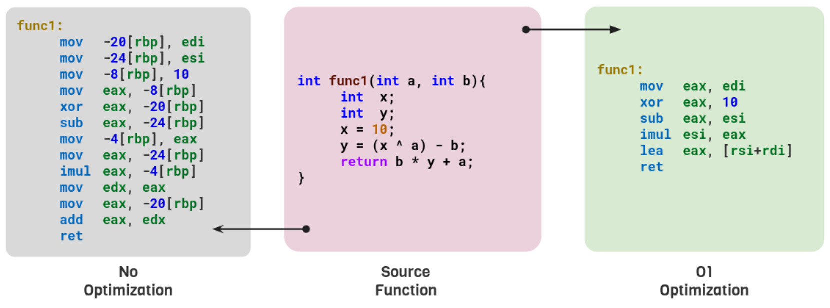
The paper is structured as follows. In Section 2, we introduce fundamental concepts relevant to binary function clone detection, including control flow graphs, symbolic execution, and Mixed Boolean Arithmetic (MBA) simplification. Section 3 reviews related works and highlights existing approaches for binary similarity detection and obfuscation techniques. In Section 4, we present the motivation behind our research, focusing on challenges in detecting function clones within obfuscated binaries. Section 5 defines the problem statement and objectives of our approach. Section 6 details the methodology, including the steps for extracting and comparing function clones through symbolic execution and MBA simplification. Section 7 discusses the results obtained from testing our method on various binaries, compiled with different optimizations and obfuscations. Section 8 outlines future work to improve the robustness and scalability of our technique. Finally, in Section 9, we conclude and provide an overview of the contributions and implications of our research.

Chapter 2

Preliminaries

2.1 Function Cloning

Function cloning is a method of modifying a function in such a way that the changes are limited to the internal code structure, without altering the function's overall behavior. This can be achieved either through obfuscation or through compiler optimizations. The goal of optimization is to produce an executable that executes more efficiently in terms of speed, memory usage, or both, without changing the program's intended behavior. When code is being compiled, developers have the option to specify different optimization levels based on their priorities and trade-offs. The optimization levels, often denoted by flags such as -Og, -O1, -O2, -O3, etc., represent different sets of compiler optimizations. In each step, the machine instructions become more optimized and efficient. Notice the reduced number of assembly instructions in Fig. 2.1.



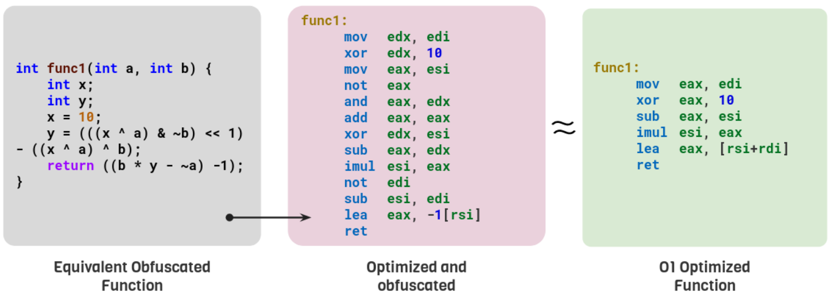
```
func1:
      mov   -20[rbp], edi
      mov   -24[rbp], esi
      mov   -8[rbp], 10
      mov   eax, -8[rbp]
      xor   eax, -20[rbp]
      sub   eax, -24[rbp]
      mov   -4[rbp], eax
      mov   eax, -24[rbp]
      imul  eax, -4[rbp]
      mov   edx, eax
      mov   eax, -20[rbp]
      add   eax, edx
      ret
```

```
int func1(int a, int b){
      int  x;
      int  y;
      x = 10;
      y = (x ^ a) - b;
      return b * y + a;
}
```

```
func1:
      mov   eax, edi
      xor   eax, 10
      sub   eax, esi
      imul  esi, eax
      lea   eax, [rsi+rdi]
      ret
```

Figure 2.1: Different Levels of Optimizations

Similarly, obfuscation in binary level, is a technique used in computer security to make the analysis and understanding of software more challenging. Obfuscation involves intentionally introducing complexity and ambiguity into the binary code to thwart reverse engineering attempts. It can be done through either code obfuscation, data obfuscation or control flow obfuscation. Obfuscators can introduce new data computation into the assembly instructions which at the end of the function does not contribute anything to the return value. Or the function might have a junk branch or loop/jump instruction that has no impact on the behavior of the function. For example, in Fig. 2.2 the source function of the Fig. 2.1 is obfuscated and converted into assembly instructions. Although when converted to O1 optimization, the compiler discards the redundant assembly instructions.



```
int func1(int a, int b) {
    int x;
    int y;
    x = 10;
    y = (((x ^ a) & ~b) << 1)
- ((x ^ a) ^ b);
    return ((b * y - ~a) -1);
}
```

```
func1:
        mov   edx, edi
        xor   edx, 10
        mov   eax, esi
        not   eax
        and   eax, edx
        add   eax, eax
        xor   edx, esi
        sub   eax, edx
        imul  esi, eax
        not   edi
        sub   esi, edi
        lea   eax, -1[rsi]
        ret
```

```
func1:
        mov   eax, edi
        xor   eax, 10
        sub   eax, esi
        imul  esi, eax
        lea   eax, [rsi+rdi]
        ret
```

Figure 2.2: Function Obfuscated and Optimized.

2.2 Control Flow Graph

A Control Flow Graph (CFG) is a graphical representation of a program's control flow, showing the flow of execution between various parts of the code. It provides a visual depiction of how the program's control is transferred among different statements, functions, and branches during its execution. Each node in the graph encapsulates a basic block, which is a sequence of instructions without any internal branches, and the edges indicate the flow of control between these blocks. The graph usually has a special entry node representing the starting point of the program and an exit node representing the end or exit point. Conditional branches (e.g., if statements) create edges that lead to different basic blocks based on a

```
int f() {
  ...
  y = read();
  z = y * 2;
  if (z == 12) {
    return;
  } else {
    printf("OK");
  }
}
```

Figure 2.3: Example Source Code

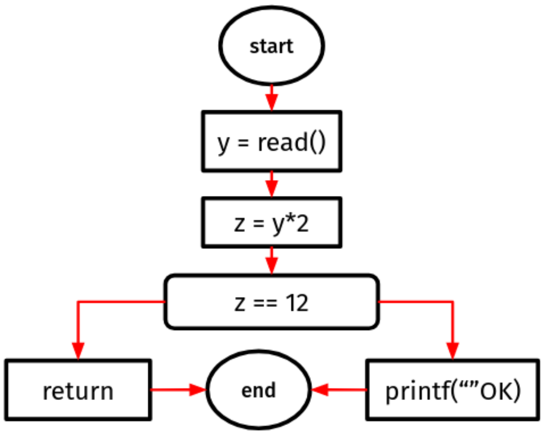


Figure 2.4: Control Flow Graph

condition. Unconditional branches (e.g., loops or goto statements) create edges that unconditionally transfer control to another basic block. Lastly, Decision nodes represent points in the control flow where a decision is made, typically corresponding to conditional statements. The graph branches into multiple paths based on the conditions. See for the source code in Fig. 2.3, the control flow graph will look like Fig. 2.4.

By mapping the paths of control through a program, the graph becomes useful for analyzing complex programs, enabling developers and analysts to grasp the logical structure of the code, identify potential execution paths, and pinpoint areas prone to bugs or vulnerabilities. CFGs play a crucial role in static analysis tools, compiler optimizations, and security assessments, offering a visual road map that aids in comprehension, optimization, and the detection of control-flow-related issues within a software system.

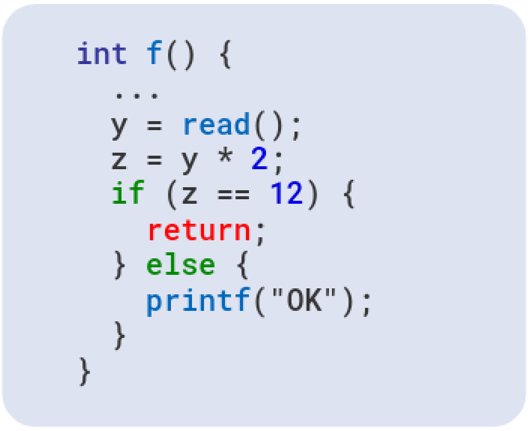
2.3 Symbolic Execution

Symbolic execution is a program analysis technique that explores the potential execution paths of a program by using symbolic values instead of concrete input values. It operates on a single path within a control flow graph, and using symbolic values computes the mathematical representative equation of the the control flow part. Unlike traditional testing methods that use specific input values to explore the code's behavior, symbolic execution allows variables to take on symbolic values, representing entire sets of possible values. As the program executes, these symbolic values are manipulated symbolically through the code, and constraints are collected based on the conditions encountered. The symbolic execution engine reasons about different paths in the program by solving these constraints to determine feasible and infeasible paths. This approach enables the exploration of various code paths, including those that may be challenging to reach with specific concrete inputs.

Symbolic execution is particularly valuable in areas such as software testing, bug detection, and security analysis, as it can systematically uncover vulnerabilities, identify corner cases, and provide insights into the program's behavior under different conditions. However, it also faces challenges, such as the path explosion problem, which arises when the number of possible paths becomes impractical to explore exhaustively.

This is where dynamic symbolic execution comes in. **Dynamic Symbolic Execution** is an advancement of symbolic execution technique combining elements of both symbolic execution and dynamic testing. It aims to explore the execution paths of a program by using symbolic values, like in symbolic execution, but it also incorporates concrete values obtained during the actual program execution. This hybrid approach helps address some of the limitations associated with purely symbolic execution, such as the path explosion problem.

In dynamic symbolic execution, the analysis starts with symbolic values for the program's inputs. As the program runs, the symbolic values are used to explore different paths through the code, similar to symbolic execution. However, when the execution encounters branches or conditions, the symbolic execution engine collects constraints based on these conditions. Rather than solving these constraints symbolically, dynamic symbolic execution takes a dynamic testing approach: it uses concrete input values to explore one branch of the execution. This process is repeated iteratively, exploring different paths through the program dynamically. For example, for the source code of Fig. 2.5, y is the input variable. Consider the



```c
int f() {
  ...
  y = read();
  z = y * 2;
  if (z == 12) {
    return;
  } else {
    printf("OK");
  }
}
```

Figure 2.5: Example Source Code

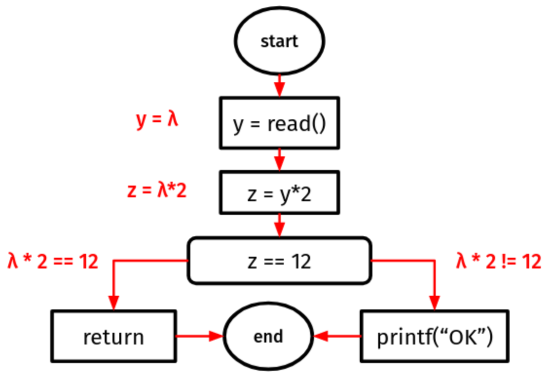


Figure 2.6: Control Flow Graph with Symbolic Execution

value of y as some symbol λ and traverse the CFG like the following Fig. 2.6.

Dynamic symbolic execution has several advantages. It can efficiently explore paths that are more likely to be executed in real-world scenarios, and it can handle complex programs with loops and conditionals more effectively than purely symbolic execution. This technique is particularly valuable for identifying specific inputs that lead to the execution of rare or hard-to-reach code paths, making it a powerful tool for software testing, bug detection, and vulnerability analysis.

2.4 Mixed Boolean Arithmetic

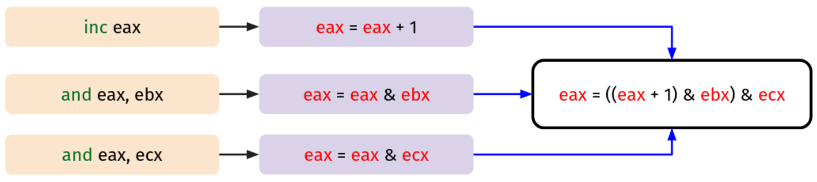


Figure 2.7: Assembly Instructions to MBA Expressions

Mixed Boolean Arithmetic (MBA) is a mathematical representation and manipulation technique that combines Boolean algebra with integer arithmetic. It allows for the seamless integration of logical operations and arithmetic operations within a unified algebraic framework. MBA provides a flexible and expressive way to represent complex computations involving both logical and arithmetic components.

In the context of program analysis, MBA is often used to represent and manipulate binary code, particularly in the symbolic execution of assembly instructions. Each assembly instruction is translated into an MBA expression, where Boolean variables represent the status of flags (e.g., zero, carry) and integer variables represent data values. For example, an arithmetic operation like addition can be represented as an MBA expression that includes both Boolean conditions (e.g., carry flag) and arithmetic operations (e.g., addition of integer variables).

The process of converting assembly instructions into MBA expressions involves mapping each instruction to its corresponding MBA representation based on the operation performed and the state of the processor flags. This allows for a unified treatment of logical and arithmetic operations within the analysis.

Furthermore, the resulting MBA expressions can be coalesced or simplified [2] to form a more compact and efficient representation [3]. This coalescing process involves combining redundant or equivalent sub-expressions, optimizing arithmetic operations, and simplifying Boolean conditions. The goal is to create a concise MBA expression that captures the essential behavior of the original assembly instructions.

2.5 SAT SOLVER

Boolean satisfiability problem (SAT) solving tackles a fundamental question: can a given formula be satisfied by assigning truth values (true or false) to its variables in a way that makes the entire formula true?

SAT solvers are specialized algorithms designed to efficiently solve this challenge. These solvers have become crucial tools in various domains, including hardware and software verification, artificial intelligence, and optimization. Over the years, SAT solvers have incorporated advanced techniques to handle increasingly complex problems.

One such solver is Z3 [4], a high-performance theorem prover developed by Microsoft Research. In our research, we leverage Z3's capabilities within the angr framework for binary analysis and function clone detection. Angr harnesses

Z3's symbolic execution abilities to uncover hidden properties within programs. Symbolic execution allows angr to analyze all potential execution paths of a program by treating inputs as symbolic variables. As the program runs symbolically, Z3 constructs and manages symbolic expressions representing the program state at various points. During this execution, constraints are generated based on conditions encountered in the program, such as branching decisions or loops. These constraints form a Boolean formula that encodes the conditions required for a specific program path to be taken.

Z3 systematically analyzes these constraints to determine their satisfiability. If Z3 identifies an assignment of values to the symbolic variables that satisfies all constraints, it confirms that the corresponding path is a valid execution path within the program. Conversely, if no such assignment exists, the path is considered infeasible, and the solver proceeds to explore alternative paths. By efficiently resolving constraints and identifying feasible paths, Z3 enables angr to eliminate irrelevant paths, thereby narrowing its analysis to the most promising execution paths. This approach significantly reduces computational overhead and enhances overall analysis efficiency.

Our function clone detection approach takes advantage of both symbolic execution and SAT solving. We use them to extract and simplify mathematical expressions from binary functions. Z3 plays a critical role in comparing these expressions by solving the constraints derived from their symbolic representations. This allows us to identify functionally identical pieces of code (function clones) even with moderate obfuscations designed to impede analysis. The combined power of Z3 and angr provides a valuable toolkit for performing in-depth binary analysis and ensuring the accuracy and efficiency of our function clone detection methodology.

Chapter 3

Related Works

3.1 BinFinder

BinFinder [5] is an end-to-end learning model that uses a customized Multi-layer Perceptron Neural Network within a Siamese neural network to learn binary function representations. It is trained on a set of manually engineered interpretable features selected at the binary function level. These features are robust, CPU architecture independent, and resilient to both compiler optimization and code obfuscation techniques. BinFinder aims to accurately identify similar binary functions even in the presence of obfuscation and optimization, providing a valuable tool for software reverse engineering and security analysis.

Existing binary function clone search tools face limitations such as struggling with code obfuscation, including techniques like control flow flattening. These tools may also struggle with varying compiler optimization levels, leading to reduced accuracy in detecting function clones. Some tools are restricted to specific CPU architectures, like x86, limiting their applicability in modern software systems with diverse architectures. Performance issues, particularly in terms of efficiency and speed, can be significant drawbacks. Additionally, there's a lack of comprehensive support for advanced obfuscation techniques, including those introduced by open-source tools like Tigress, posing challenges in handling sophisticated code transformations.

The architecture here consists of four phases:

1. Identification of resilient features
2. Preprocessing and representation of the features
3. Train the Siamese neural network architecture
4. Given a new binary, generate it's embedding vector using the trained model and compare it against existing vectors

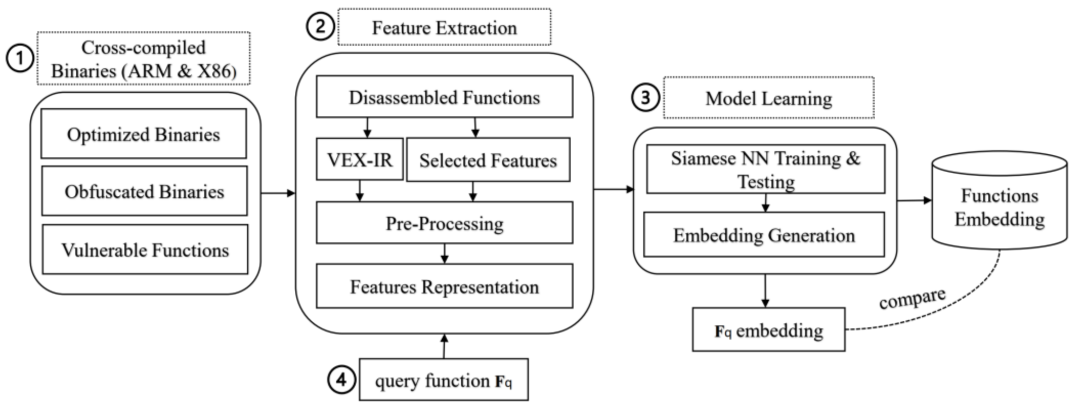


Figure 3.1: BinFinder working mechanism

The paper utilizes multiple datasets to evaluate the performance of the proposed Siamese neural network for binary function similarity detection. Dataset-I comprises 10,000 binary functions from diverse open-source projects, compiled using various compilers and optimization levels. Dataset-III involves packages cross-compiled to x86 and ARM architectures, totaling 60,395 functions. Dataset-IV is generated by obfuscating specific packages with five techniques and compiling them for the x86 architecture. Dataset-V, sourced from an online dataset, includes 164,700 optimized and obfuscated functions for x86, compiled using the clang compiler and O-LLVM obfuscator. These datasets are strategically split into training, validation, and testing sets to prevent over-fitting, ensuring a robust evaluation of the Siamese neural network's generalization capability. The datasets encompass numerical features representing functions and lists of libc calls and Constants, contributing to a comprehensive evaluation of the proposed approach across various code characteristics and transformations.

Through repeated testing, the authors have determined some robust functions that can resist the impact of compiler optimization or any kind of obfuscation. Some of them are the number of caller functions, how many times the target function has been called, the number of unique callees, etc.

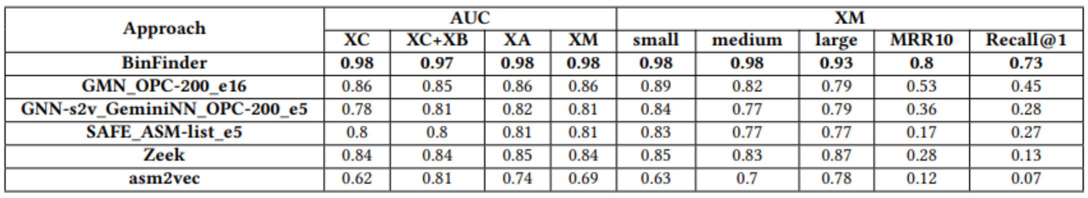
The reason for using a Siamese neural network in the study was that binary function similarity cannot be addressed with traditional classification techniques. Since the final number of binary functions cannot be determined, a different approach was needed. The Siamese neural network is an end-to-end machine learning technique that is suitable for similarity problems. It allows for the comparison of two inputs and produces a similarity score as the output. In the study, the Siamese neural network was used to learn binary function representations and perform binary function clone search.

When a new target binary is provided for clone detection, the embedding vector of the binary is generated using the Siamese model. That embedding is checked against all the existing vector embeddings. The metric used here is the cosine-similarity. For each pair of functions, the cosine similarity is calculated, and the minimum summation is taken:

$$\min \sum_{i=1}^{n} (\cos(\frac{emd_1 * emd_2}{||emd_1|| * ||emd_2||}) - y_i)^2 Enter$$

The metrics for evaluation were Precision(P), Recall(R), normalized Discounted Cumulative Gain(nDCG).

BinFinder exhibits limitations in dealing with unseen system calls, where new LibcCalls or VEX tokens introduced in the dataset may not be accurately identified due to the model's lack of specific handling for unseen system calls. Additionally, BinFinder faces challenges when obfuscation techniques, particularly



Approach	AUC				XM				
	XC	XC+XB	XA	XM	small	medium	large	MRR10	Recall@1
BinFinder	**0.98**	**0.97**	**0.98**	**0.98**	**0.98**	**0.98**	**0.93**	**0.8**	**0.73**
GMN_OPC-200_e16	0.86	0.85	0.86	0.86	0.89	0.82	0.79	0.53	0.45
GNN-s2v_GeminiNN_OPC-200_e5	0.78	0.81	0.82	0.81	0.84	0.77	0.79	0.36	0.28
SAFE_ASM-list_e5	0.8	0.8	0.81	0.81	0.83	0.77	0.77	0.17	0.27
Zeek	0.84	0.84	0.85	0.84	0.85	0.83	0.87	0.28	0.13
asm2vec	0.62	0.81	0.74	0.69	0.63	0.7	0.78	0.12	0.07

Figure 3.2: Evalutaion of BinFinder against existing solutions

control flow graph modifications like flattening (FLA), are applied. While the tool can maintain performance in the presence of certain obfuscation methods, its practicality diminishes when confronted with specific types of obfuscation, impacting its overall effectiveness.

3.2 Qsynth

QSynth [6] presents a deobfuscation approach that aims to unravel obfuscated programs by combining Dynamic Symbolic Execution (DSE) and program synthesis. The working mechanism involves several key steps. First, the program is traced using Dynamic Binary Instrumentation (DBI) to collect all instructions and their concrete side-effects on registers and memory, forming an execution trace. Then, DSE is performed as a separate step after program execution, considering the symbolic values of program inputs and tracking them during the execution to solve constraints. A synthesis oracle function is defined to return a new expression formula satisfying high-level specifications. An offline enumerative search is performed on a context-free grammar to generate terminal expressions up to a defined number of derivations, ensuring an optimal solution by keeping only the first expression corresponding to the given output vector. These mechanisms work together to achieve the goal of deobfuscating obfuscated programs by combining DSE and program synthesis.

Existing deobfuscation tools encounter significant challenges when tackling obfuscated programs, especially in the realms of program synthesis and deobfuscation.

These challenges include grappling with an expansive search space, particularly in cases involving complex obfuscation techniques like Mixed-Boolean-Arithmetic (MBA) or data encoding. Additionally, semantic complexity arises, where obfuscation introduces intricate and excessively large expressions, posing difficulties for tools to find semantically equivalent alternatives. Syntactic complexity presents another hurdle, as certain synthesis-based approaches, despite being generally resilient, may struggle when analyzing program structure. The compounded challenge of composite obfuscation, combining various techniques, necessitates resilience to both syntactic and semantic complexities. Furthermore, scalability becomes a concern, with the increasing size of programs intensifying the complexity of obfuscation techniques, making it harder for existing tools to maintain effectiveness at scale.

Control Flow Obfuscation can be handled by Dynamic Symbolic Execution(DSE). But when data flow obfuscation is used, DSE becomes useless. For such cases, we can use the program synthesis approach. The architecture of synthesis takes place in five steps -

1. Program tracing
2. Dynamic symbolic execution
3. Expression abstract syntax tree computation
4. Synthesis oracle
5. Expression simplification

For the experiments, four datasets, each comprising 500 obfuscated functions, were utilized. The first dataset, employed by Syntia authors, was challenging to replicate due to difficulties with Syntia's functionality, prompting reliance on reported results. Syntia's efficacy was inconsistent on other datasets, either yielding uninteresting results or failing to terminate within a reasonable time frame. The three additional datasets involved expressions obfuscated with various techniques, aiming to compare simplification strategies. Original expressions, generated with

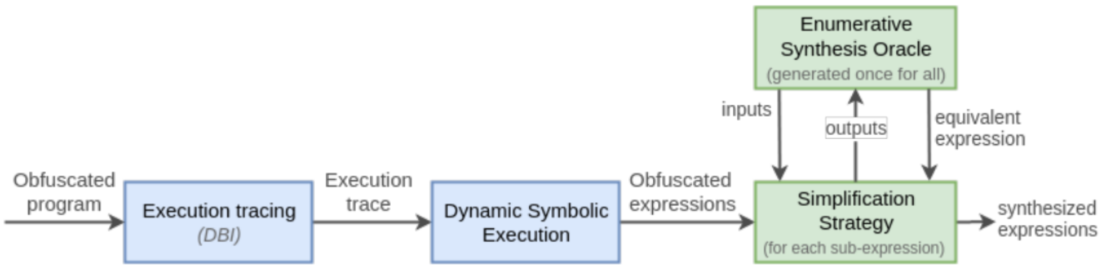


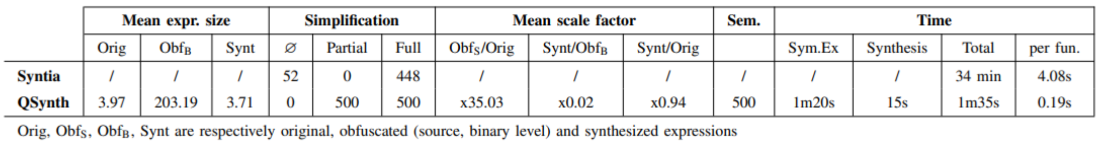
Figure 3.3: Qsynth working procedure

Syntia's grammar, incorporated up to 3 variables and 8 operators. Dataset details are as follows: Dataset 1 employed EncodeArithmetic and EncodeData transformations; Dataset 2 was exclusively obfuscated with EncodeArithmetic; Dataset 3 used expressions from Dataset 2 and applied Virtualize and EncodeArithmetic transformations; Dataset 4 utilized the original expressions of Dataset 2, obfuscated with EncodeArithmetic and EncodeData passes. The complexity of these datasets varied, incorporating techniques like MBA equivalence, parameter encoding, virtualization, and complex obfuscation, with expression sizes ranging from 3.97 to extremes.

The performance measure are these three things -

1. Success rate
2. Correctness
3. Execution time
4. Understandability

Based on the above criterion, the following performance was documented -



	Mean expr. size			Simplification			Mean scale factor			Sem.	Time			
	Orig	Obf_B	Synt	∅	Partial	Full	Obf_S/Orig	Synt/Obf_B	Synt/Orig		Sym.Ex	Synthesis	Total	per fun.
Syntia	/	/	/	52	0	448	/	/	/	/	/	/	34 min	4.08s
QSynth	3.97	203.19	3.71	0	500	500	x35.03	x0.02	x0.94	500	1m20s	15s	1m35s	0.19s

Orig, Obf_S, Obf_B, Synt are respectively original, obfuscated (source, binary level) and synthesized expressions

Figure 3.4: Evaluation of Qsynth against Syntia

The paper acknowledges several limitations in its proposed approach. These include challenges posed by the vast search space for program expressions, particularly in the presence of complex obfuscation techniques, hindering the efficient exploration of potential solutions. The success of the synthesis oracle is contingent on the node processing order, potentially limiting its ability to fully synthesize the Abstract Syntax Tree (AST). While the worst-case time complexity of the synthesis algorithm is $O(n^2)$, efforts are made to mitigate this through iterative reductions in AST size and early termination of synthesis steps. Scalability concerns arise, with the approach's effectiveness dependent on the size and complexity of original and obfuscated expressions in datasets. The reliance on input/output behavior rather than internal structure exposes the approach to syntactic complexity, particularly in the face of composite obfuscation. Despite these limitations, the paper underscores the demonstrated effectiveness of the proposed approach in handling complex obfuscation techniques and its comparative scalability against similar synthesis-based approaches.

3.3 ProMBA

The paper [7] proposes a versatile method for deobfuscating Mixed Boolean Arithmetic (MBA) expressions, which are commonly used to protect programs from reverse engineering but can also be used for malicious purposes. The method combines program synthesis, term rewriting, and an algebraic simplification method to overcome the limitations of existing deobfuscation techniques. The key novelty is the on-the-fly learning of transformation rules for deobfuscation, allowing the handling of diverse MBA obfuscation rules. The proposed method, implemented in a tool called ProMBA, outperforms state-of-the-art MBA deobfuscation tools, successfully simplifying a large majority of obfuscated expressions. The paper provides a detailed overview of the approach, including the theoretical foundations, limitations of existing techniques, and the proposed solution.

Some algebraic methods for deobfuscation, as highlighted in the literature, exhibit specific limitations. First, certain techniques are constrained to particular classes of Mixed-Boolean-Arithmetic (MBA) expressions, limiting their applicability to state-of-the-art MBA obfuscation techniques that go beyond linear or polynomial MBA expressions. These methods may only handle expressions involving logical operators ($\wedge, \vee, \neg, \oplus$) and arithmetic operators (+, -, $\times$). Second, a lack of flexibility is noted, as some techniques are confined to specific MBA transformations defined in the literature or observed in a limited set of samples, restricting their adaptability to a broader range of obfuscation scenarios. Third, scalability issues arise, with certain techniques unable to effectively handle large MBA expressions, despite the significant increase in expression size caused by MBA obfuscation. Lastly, the absence of a guarantee of correctness is highlighted as a concern, particularly for techniques used in conjunction with program analysis to detect malicious behaviors of malware. Some methods lack soundness, raising doubts about the reliability of the deobfuscation results they produce.

The working mechanism involves the following steps -

1. Simplify linear MBA expressions using existing tools
2. Recursively simplify non-linear sub-expressions by synthesizing simpler sub-expressions
3. Keep applying the simplification rules to other sub-expressions until no further simplification is possible

The evaluation of ProMBA focuses on non-linear Mixed-Boolean Arithmetic (MBA) expressions, excluding linear ones. Linear MBA sub-expressions are addressed separately using the linear MBA deobfuscation tool SimbaD. Existing datasets, such as Neureduce, containing only linear MBA expressions are excluded from consideration. The dataset comprises 4011 non-linear MBA expressions from various sources, including the MBA-Solver dataset (2011 expressions), QSynth dataset (500 expressions obfuscated with EncodeArithmetic scheme), and Loki dataset (1500 expressions obfuscated using Loki's recursive and randomized ex-

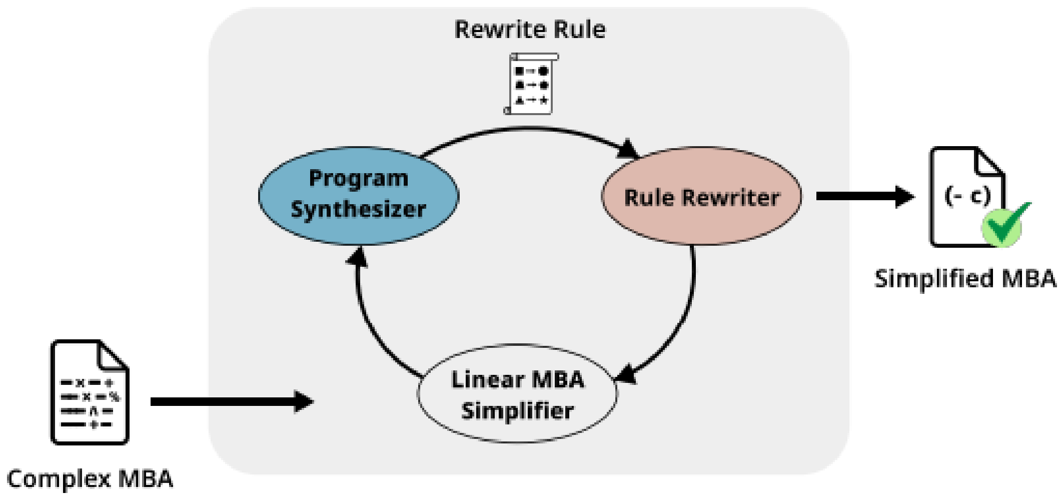


Figure 3.5: ProMBA working mechanism

pression rewriting method). Among the Loki expressions, 1000 were utilized to evaluate obfuscation resilience against existing tools, while an additional 500 were generated to assess ProMBA's effectiveness against full-fledged MBA obfuscation, featuring constants and all arithmetic and logical operations.

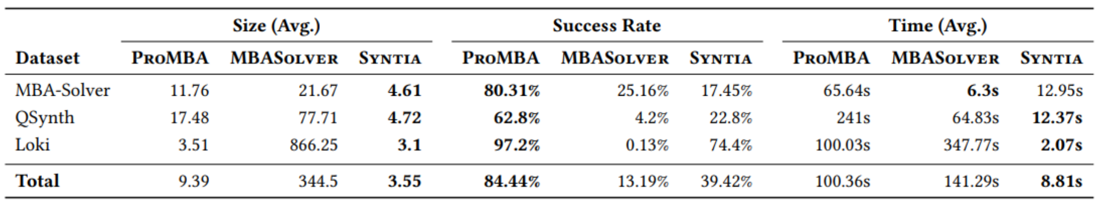
The evaluation metrics are as follows-

1. Size of the deobfuscated expression

2. Success rate

3. Time

Based on the said criterion, the performance of ProMBA has been evaluated [7],



	Size (Avg.)			Success Rate			Time (Avg.)		
Dataset	**ProMBA**	**MBASolver**	**Syntia**	**ProMBA**	**MBASolver**	**Syntia**	**ProMBA**	**MBASolver**	**Syntia**
MBA-Solver	11.76	21.67	**4.61**	**80.31%**	25.16%	17.45%	65.64s	**6.3s**	12.95s
QSynth	17.48	77.71	**4.72**	**62.8%**	4.2%	22.8%	241s	64.83s	**12.37s**
Loki	3.51	866.25	**3.1**	**97.2%**	0.13%	74.4%	100.03s	347.77s	**2.07s**
Total	9.39	344.5	**3.55**	**84.44%**	13.19%	39.42%	100.36s	141.29s	**8.81s**

Figure 3.6: Performance comparison of ProMBA against other solutions

Chapter 4

Motivation

In the modern software ecosystem, the ubiquity of closed-source applications presents a significant challenge for ensuring the integrity and authenticity of software systems. The closed-source nature of these applications inherently limits transparency, making it difficult to verify the originality of code and detect unauthorized or plagiarized content. This issue is exacerbated by the use of various compiler optimizations and obfuscation techniques, which further obscure the original code and render traditional plagiarism detection methods less effective.

In parallel, the cybersecurity domain faces an escalating threat from increasingly sophisticated malicious activities. Attackers often reuse existing malware, making slight modifications and employing advanced obfuscation strategies to evade detection by conventional antivirus software. This growing trend underscores the necessity for advanced methods capable of analyzing and identifying threats at the binary level.

Given these challenges, there is a pressing need for innovative approaches to detect code reuse and unauthorized modifications in closed-source binaries. While machine learning (ML) techniques have been explored for this purpose, they come with their own set of limitations, including the need for extensive labeled training data, potential overfitting, and difficulties in interpreting the results.

As an alternative, this research proposes the use of Mixed Boolean-Arithmetic (MBA) extraction and simplification for detecting function clones from binary files. MBA expressions, which are commonly used in obfuscation techniques, can be highly indicative of code reuse when identified and simplified correctly. By focusing on MBA extraction and simplification, this approach aims to overcome the limitations of ML-based methods, providing a more transparent and interpretable means of detecting function clones. This technique leverages the mathematical properties of MBAs to reveal underlying similarities between obfuscated functions, facilitating the detection of reused or plagiarized code within closed-source binaries.

The potential impact of this research is significant. It promises to enhance the ability of software analysts to verify the integrity of closed-source applications, improve the detection of unauthorized code reuse, and bolster defenses against sophisticated malware that employs obfuscation techniques. Ultimately, this approach aims to contribute to the development of more robust tools for ensuring software authenticity and cybersecurity in an increasingly opaque software landscape.

Chapter 5

Problem Statement

Ensuring the integrity and authenticity of software systems, particularly closed-source applications, has become increasingly challenging. Closed-source software is inherently opaque, making it difficult to verify code originality and detect unauthorized reuse or plagiarism. This issue is exacerbated by various compiler optimizations and obfuscation techniques employed during the compilation process, which further obscure the original code and render traditional plagiarism detection methods ineffective. Additionally, the cybersecurity domain faces growing threats from sophisticated malicious activities, where attackers reuse existing malware, making minor modifications and applying advanced obfuscation strategies to evade detection by conventional antivirus software.

Our problem is to find a new method for binary function clone detection that:

- **Overcomes the inherent limitations of closed-source applications**: Provides a transparent and interpretable means of detecting function clones despite the opacity of closed-source software.

- **Mitigates the impact of compiler optimizations and obfuscation**: Effectively handles various compiler optimizations and obfuscation techniques that obscure original code.

- **Enhances cybersecurity defenses**: Improves the detection of unauthorized code reuse and bolsters defenses against sophisticated malware that employs obfuscation techniques.

- **Offers an alternative to traditional methods**: Surpasses the limitations of machine learning-based methods, including the need for extensive labeled training data, heavy resource usage, potential overfitting, and difficulties in interpreting results.

- **Utilizes Mixed Boolean-Arithmetic (MBA) expressions**: Leverages the mathematical properties of MBAs to reveal underlying similarities between analogous functions, facilitating the detection of reused or plagiarized code within closed-source binaries.

Chapter 6

Methodology

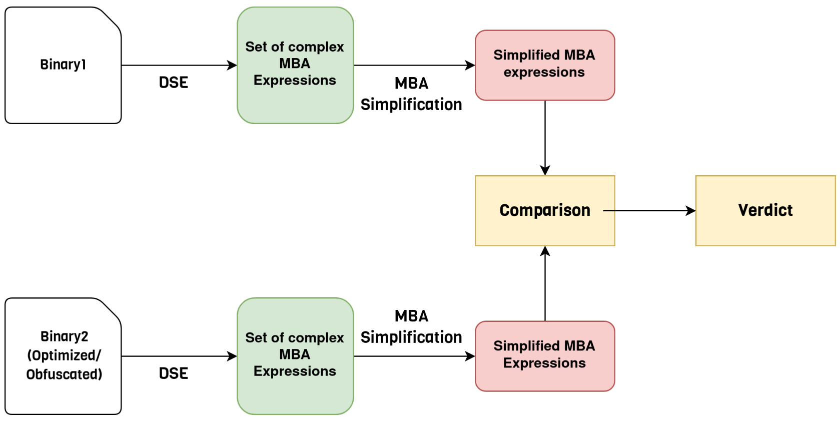

Binary1
DSE
Set of complex MBA Expressions
MBA Simplification
Simplified MBA expressions
Comparison
Verdict
Binary2 (Optimized/ Obfuscated)
DSE
Set of complex MBA Expressions
MBA Simplification
Simplified MBA Expressions

Figure 6.1: Outline of the Proposed Methodology

Our function clone detection approach involves various steps for which the global overview is shown in Fig. 6.1. First we select a candidate function from all functions in $Binary_1$. Then we extract a set of complex MBA expressions with the help of dynamic symbolic execution [8] on the candidate function.

We then simplify the complex MBA expression using angr [9], which we employ to reduce and simplify boolean expressions, to obtain a set of simplified MBA expressions. We follow a similar process to obtain the simplified MBA expressions of the candidate function from $Binary_2$. In the final step, we compare the resulting sets of simplified MBA expressions for equivalency. The different steps are defined in detail hereafter.

6.1 Extraction

To conduct our binary analysis, we employed the widely-used binary analysis framework called angr [10]. Our choice of angr is based on several compelling factors: it is written in Python, actively maintained, well-documented, architecture-independent, and capable of handling binaries from multiple architectures. Furthermore, it supports both static and dynamic analysis, has its own symbolic execution and constraint solver engines, and is highly extensible. Angr is also well-regarded in the reverse engineering community.

Our analysis follows a systematic approach. We begin by creating an angr project with the path to the binary executable that we want to analyze. Using angr's built-in CFG (Control Flow Graph) analysis engine, we extract the control flow graph of the binary, which consists of nodes representing basic blocks of functions and edges representing their interconnections. Each function has a starting address, which serves as an offset from the binary file's beginning.

With this starting address, we create a new simulation manager with an empty state. This manager simulates the function's execution in a controlled environment. Depending on the availability of concrete values, the simulation manager selects the appropriate execution engines. For concrete values, it uses the Unicorn engine for execution simulation; for symbolic values, it employs angr's symbolic execution engine. We initialize the state with an empty state to ensure that all values are symbolic, allowing the symbolic execution engine to handle the dynamic symbolic execution of the function.

During dynamic symbolic execution, each execution step generates or stores a symbolic expression in a bit-vector format. These expressions are already simplified using angr's built-in solver. To capture and store these expressions, we create a hook that triggers whenever a new expression is generated. This hook collects the symbolic expressions, and we keep only unique expressions to simplify later comparisons. We also add a second hook to capture the constraints associated with each path. Consequently, each function path is represented by a list of symbolic expressions and constraints, which we extract using angr's depth-first search (DFS) exploration technique.

After exploring all functions in a binary, the result is a list of function representations, with each function comprising multiple paths, each path containing symbolic expressions and constraints. We then analyze another binary in the same manner, generating a comparable set of symbolic representations to detect similarities or clones.

To ensure that each function is analyzed independently, even when it calls other functions, we implemented a hook to step over function call instructions, adding a symbolic return value instead. This allows the caller function to be analyzed without dependency on the called function, while still incorporating the symbolic effect of the called function's return value.

This methodology enables us to comprehensively analyze and compare binary functions for similarities, even in the presence of optimizations and obfuscations, providing a robust solution for detecting function clones and ensuring software integrity.

6.2 Comparison

With all the functions extracted from the last step, we are ready to focus on the crux of our methodology : the comparison algorithm that tells how similar the given binaries are. As seen from the Fig. 6.2 below, a binary can be dissected into multiple layers, best understood when compared to an onion. Just as an onion has multiple layers, so does a binary, with the functions being the top-most layer, and the lower layers being paths and constraints respectively.

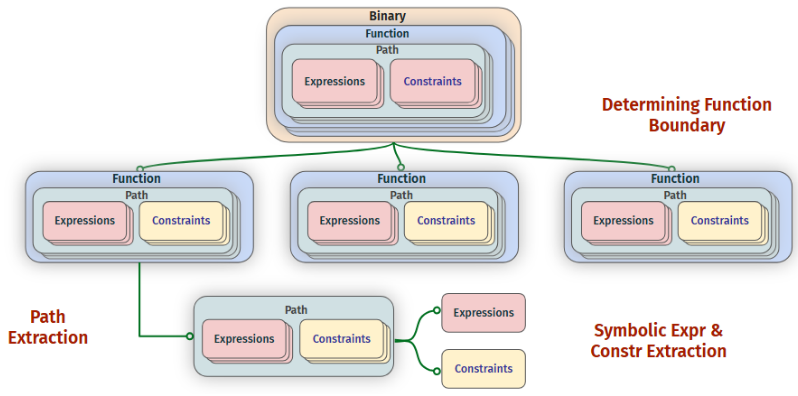


Figure 6.2: Layers of a binary file

We follow a top down approach here. To check if two binaries are similar, our goal is to match as many functions as possible between the two binaries. But how do we "match" two functions exactly? There comes the second layer, the layer of paths. Each function can be divided into multiple paths, and so in order to check if two functions are similar, we try to match as many paths as possible. Consequently, we reach the bottom layer, constraints and expressions. Because paths are represented by constraints and expressions. In order to match two paths, we try to group together as many constraints and expressions possible.

A score is propagated up through each layers back to the topmost layer. The score here is represented using a similarity matrix that is calculated at each layer.

Another important thing to understand is the greedy matching function. It takes a weighted graph as input and returns the similarity score between the entities. What the graph contains is tuples of the following type: $(weight, node1, node2)$. We sort the graph based on the *weight* parameter, hence the greedy part. The total matching is divided by the maximum number of nodes. The rationale behind this is that, we are checking the percentage of functions that have matched, in comparison to the maximum possible matching. With the scores received, we can build a similarity matrix.

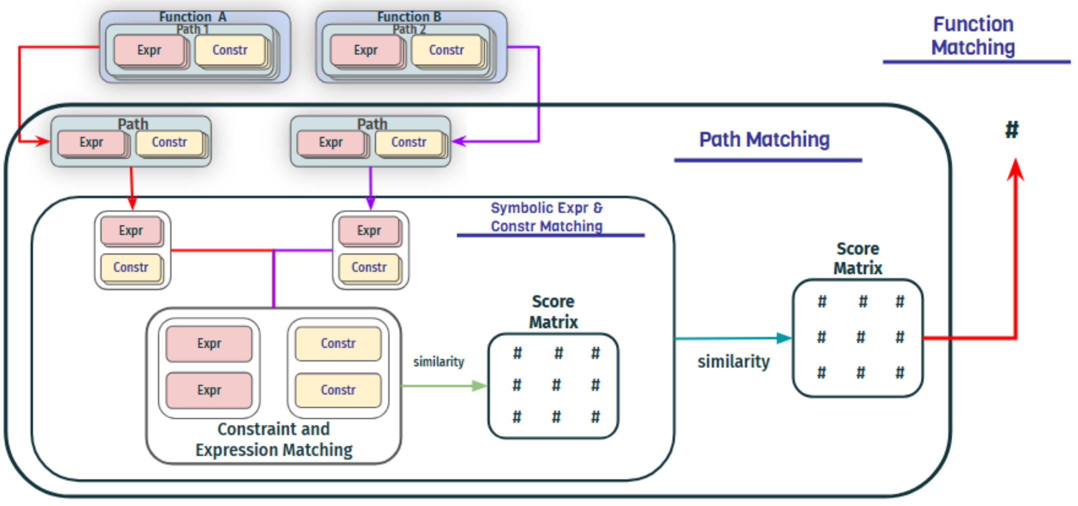


Figure 6.3: Top down similarity matching

The process explained above in figure 6.3 can be represented by the algorithm below:

Algorithm 1: Similarity matching between two binaries

```
Input: Two binaries $binary1$, $binary2$
Output: Similarity matrix
Function path_similarity($path1$, $path2$):
    $edgeList \leftarrow \{\emptyset\}$
    for $id1 \leftarrow 1$ to $length(path1)$ do
        $elem1 \leftarrow path1[id1]$
        for $id2 \leftarrow 1$ to $length(path2)$ do
            $elem2 \leftarrow path2[id2]$
            // make sure both elements are of the same type(expression or constraint)
            if $type(elem1) \neq type(elem2)$ then
                continue
            $isSimilar \leftarrow constraint\_similarity(elem1, elem2)$ // this function is provided by angr
            if $isSimilar$ then
                $edgeList \leftarrow edgeList \cup \{(1, id1, id2)\}$
    return greedy_match(edgeList, $length$(path1), $length$(path2)) // a greedy mathcing algorithm explained later

/* function1 belongs to the first binary, and function2 to the second binary */
Function function_similarity($function1$, $function2$):
    $paths1 \leftarrow \{\emptyset\}$
    $paths2 \leftarrow \{\emptyset\}$
    for $path$ in $function1$ do
        if $path$ is not empty then
            $paths1 \leftarrow paths1 \cup \{path\}$
    for $path$ in $function2$ do
        if $path$ is not empty then
            $paths2 \leftarrow paths2 \cup \{path\}$
    return array_similarity($paths1$, $paths2$, path_similarity) // an abstraction defined to group the most similar elements(path or function) together

/* this is the principal function here */
Function binary_similarity($binary1$, $binary2$):
    $similarity\_matrix \leftarrow [\,]$
    for $function1$ in $binary1$ do
        for $function2$ in $binary2$ do
            $score \leftarrow function\_similarity$(function1, function2)
            $similarity\_matrix$[function1][function2] $\leftarrow score$
    return $similarity\_matrix$
```

Algorithm 2: Greedy Matching Algorithm

```
Input: A weighted bipartite graph $graph$, number of nodes $n1$, $n2$
Output: Maximum match score
Function greedy_match($graph$, $n1$, $n2$):
    if $n1 = 0$ or $n2 = 0$ then
        return 0
    $graph \leftarrow sort_{desc}(graph)$// sorted in decreasing order based on the
        edge weight
    $vis1 \leftarrow \{\emptyset\}$
    $vis2 \leftarrow \{\emptyset\}$
    $matchWeight \leftarrow 0$
    for $\{w, v1, v2\}$ in $graph$ do
        if $v1$ in $vis1$ or $v2$ in $vis2$ then
            continue
        $matchWeight$ += $w$
        $vis1 \leftarrow vis1 \cup \{v1\}$
        $vis2 \leftarrow vis2 \cup \{v2\}$
    $score \leftarrow matchWeight/max(n1, n2)$
    return $score$
```

Chapter 7

Obtained Results

7.1 Experimental Setup

The experiments conducted to evaluate the proposed binary function clone detection method involved the following steps:

- **Source Code Collection**: Collected source codes of 102 binaries from the `Coreutils` library.
- **Compilation**: Compiled binaries using both `gcc` and `clang` compilers.
- **Optimization Levels**: Compiled the binaries with multiple optimization levels (`-Og`, `-O1`, `-O3`) for each compiler.
- **Obfuscated Binaries**: Collected 8 obfuscated binaries from various sources, including custom codes and CTF challenges.
- **Experimental Environment**: The experiments were conducted on a computer with 128GB RAM and an Intel Core i9 12900K CPU.

7.2 Results

The performance of our proposed binary function clone detection method was evaluated using the following metrics: similarity detection accuracy and dissimilarity detection accuracy for different compilers and optimization levels. The overall accuracy of our method was found to be 87.87%.

7.2.1 Similarity Detection

The accuracy of similarity detection for different compilers and optimization levels is shown in Fig. 7.1.

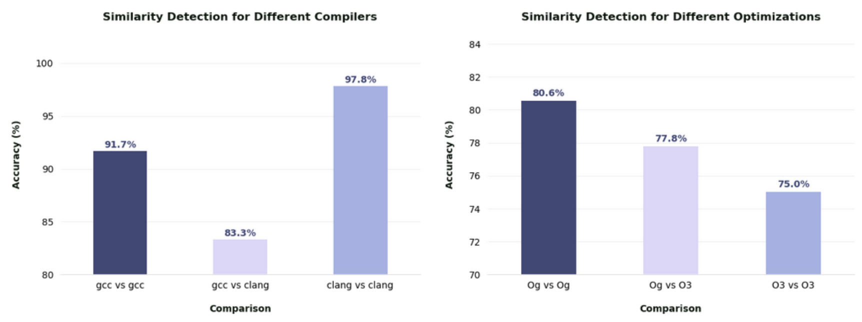


Figure 7.1: Similarity Detection for Different Compilers and Optimization Levels

- **Different Compilers**: The similarity detection accuracy for binaries compiled with `gcc` and `clang` compilers is 91.7%, 83.3%, and 97.8% for `gcc` vs `gcc`, `gcc` vs `clang`, and `clang` vs `clang` respectively.
- **Different Optimization Levels**: The similarity detection accuracy for binaries compiled with `gcc` compiler with different optimization levels is 80.6%, 77.8%, and 75.0% for `-Og` vs `-Og`, `-Og` vs `-O3`, and `-O3` vs `-O3` respectively.

7.2.2 Dissimilarity Detection

The accuracy of dissimilarity detection for different compilers and optimization levels is shown in Fig. 7.2.

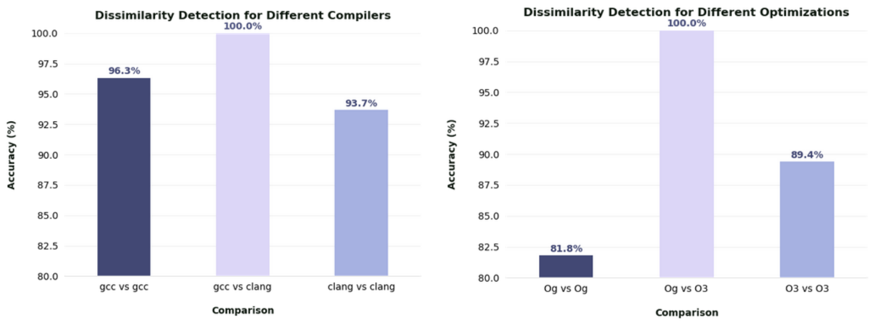


Figure 7.2: Dissimilarity Detection for Different Compilers and Optimization Levels

- **Different Compilers**: The dissimilarity detection accuracy for binaries compiled with `gcc` and `clang` compilers is 96.3%, 100.0%, and 93.7% for `gcc` vs `gcc`, `gcc` vs `clang`, and `clang` vs `clang` respectively.
- **Different Optimization Levels**: The dissimilarity detection accuracy for binaries compiled with different optimization levels is 81.8%, 100.0%, and 89.4% for `-Og` vs `-Og`, `-Og` vs `-O3`, and `-O3` vs `-O3` respectively.

7.3 Discussion

The results demonstrate that our method achieves high accuracy in detecting both similarities and dissimilarities across different compilers and optimization levels. The highest accuracy was observed in the detection of similarities for binaries compiled with the `clang` compiler (97.8%), and the highest dissimilarity detection accuracy was observed for `gcc` vs `clang` (100.0%). The variation in accuracy across different optimization levels suggests that optimization can have a significant impact on binary function similarity and dissimilarity detection, which is an important consideration for practical applications of our method.

Chapter 8

Future Work

The proposed approach of using Mixed Boolean-Arithmetic (MBA) extraction and simplification for detecting function clones in binary files has shown promising results. However, several avenues for future research and improvements remain.

- **Extending the Dataset**: Increase the size and diversity of the dataset used for evaluation to include more varied and complex obfuscation techniques. This will help in assessing the robustness and generalizability of the proposed method.

- **Conditional Branching with Side Effects**: Future work will focus on developing techniques to handle and accurately detect clones in scenarios involving conditional branching with side effects.

- **Decompiling Optimized Binaries**: Improve the decompilation process for `-O3` optimized binaries, as this process proved to be very error-prone and significantly reduced accuracy.

- **Improving Simplification Algorithms**: Develop and integrate more advanced SAT solving algorithms to enhance the accuracy and efficiency of detecting function clones, reducing the time spent proving satisfiability between expressions.

- **Integration with Other Analysis Techniques**: Combine the proposed MBA-based method with other binary analysis techniques, such as control flow analysis and data flow analysis, to improve detection rates and reduce false positives.

- **Handling Dynamic Obfuscation**: Explore methods to effectively handle dynamically obfuscated code, which presents additional challenges compared to statically obfuscated binaries.

- **Real-time Detection Capabilities**: Adapt the approach for real-time binary function clone detection, enabling its use in live environments such as network intrusion detection systems.

- **Cross-Architecture Support**: Extend the approach to support function clone detection across different CPU architectures, increasing its applicability in diverse computing environments.

By addressing these future research directions, the proposed method can be further refined and its applicability in various cybersecurity and software integrity verification scenarios can be significantly enhanced.

Chapter 9

Conclusion

This thesis addressed the challenge of ensuring the integrity and authenticity of closed-source software systems through the detection of binary function clones. The opacity of closed-source applications, combined with compiler optimizations and obfuscation techniques, complicates traditional plagiarism detection methods and necessitates advanced analytical approaches.

Our research introduced the application of Mixed Boolean-Arithmetic (MBA) extraction and simplification for binary function clone detection. Leveraging the mathematical properties of MBA expressions, we developed a transparent and interpretable method to identify code reuse and unauthorized modifications in obfuscated binaries. This approach showed significant promise, overcoming limitations of machine learning-based methods that require extensive labeled data and are prone to overfitting.

Dynamic symbolic execution and the transformation of assembly instructions into MBA representations streamlined the comparison process of source and target functions. Our evaluations demonstrated the robustness and efficiency of this method in detecting function clones, even amidst sophisticated obfuscation techniques.

We achieved an overall accuracy of 87.87% in identifying binary similarities. However, conditional branching with side effects remains unresolved, representing a scope for future work. Decompiling `-O3` optimized binaries proved error-prone, reducing accuracy. Additionally, significant time was spent proving satisfiability between expressions, suggesting that improved SAT solvers could reduce operation time in the future.

Our findings extend beyond software plagiarism detection to enhancing cybersecurity defenses. By improving the detection of reused or plagiarized code, our method can bolster defenses against malware employing advanced obfuscation strategies. This research contributes to developing more robust tools for ensuring software authenticity and security in an increasingly opaque digital landscape.

In conclusion, the proposed MBA-based binary function clone detection method offers a promising solution for closed-source software environments. Future research should focus on extending the dataset, enhancing simplification algorithms, integrating with other analysis techniques, handling dynamic obfuscation, adapting for real-time detection, and improving the accuracy of decompiling optimized binaries. Addressing these areas will further refine the method and expand its applicability in various cybersecurity and software integrity verification scenarios.

Bibliography

[1] T. Ball and J. Daniel, "Deconstructing dynamic symbolic execution," in *Dependable Software Systems Engineering.* IOS Press, 2015, pp. 26–41.

[2] N. Eyrolles, "Obfuscation with Mixed Boolean-Arithmetic Expressions : reconstruction, analysis and simplification tools," Theses, Université Paris Saclay (COmUE), Jun. 2017. [Online]. Available: https://theses.hal.science/tel-01623849

[3] B. Reichenwallner and P. Meerwald-Stadler, "Efficient deobfuscation of linear mixed boolean-arithmetic expressions," in *Proceedings of the 2022 ACM Workshop on Research on Offensive and Defensive Techniques in the Context of Man At The End (MATE) Attacks*, ser. Checkmate '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 19–28. [Online]. Available: https://doi.org/10.1145/3560831.3564256

[4] L. de Moura and N. Bjørner, "Z3: an efficient smt solver," in *2008 Tools and Algorithms for Construction and Analysis of Systems.* Springer, Berlin, Heidelberg, March 2008, pp. 337–340. [Online]. Available: https://www.microsoft.com/en-us/research/publication/z3-an-efficient-smt-solver/

[5] 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 2023 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 443–456. [Online]. Available: https://doi.org/10.1145/3579856.3582818

[6] R. David, L. Coniglio, and M. Ceccato, “Qsynth-a program synthesis based approach for binary code deobfuscation,” in *BAR 2020 Workshop*, 2020.

[7] J. Lee and W. Lee, “Simplifying mixed boolean-arithmetic obfuscation by program synthesis and term rewriting,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 2351–2365. [Online]. Available: https://doi.org/10.1145/3576915.3623186

[8] E. Cheng, “Binary analysis and symbolic execution with angr,” 2016.

[9] F. Wang and Y. Shoshitaishvili, “Angr - the next generation of binary analysis,” in *2017 IEEE Cybersecurity Development (SecDev)*, 2017, pp. 8–9.

[10] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, “SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis,” in *IEEE Symposium on Security and Privacy*, 2016.