Binary Function Clone Detection Using MBA Simplification

Tasnimul Hasnat
ID: 190041113
tasnimulhasnat@iut-dhaka.edu

Sumit Alam Khan
ID: 190041207
sumitalam@iut-dhaka.edu

Md Abdullahil Kafi
ID: 190041224
abdullahilkafi@iut-dhaka.edu

Abstract

The prevalence of closed-source applications poses a significant hurdle in confirming the legitimacy of software systems and identifying potential code reuse. Detecting plagiarism becomes especially challenging when access to the source code is unavailable, compounded by the various optimizations and obfuscations applied during compilation. This difficulty extends into the realm of cybersecurity, where attackers utilize existing malware, modifying and obfuscating it to bypass antivirus detection. In response to these challenges, this thesis proposes a novel approach to binary function clone detection utilizing MBA (Mathematical Binary Analysis) simplification. By focusing on the equivalency of source and target functions, we employ dynamic symbolic execution to coalesce assembly instructions into mathematical expressions. Subsequently, these expressions are transformed into MBA representations. Leveraging MBA simplification algorithms, we streamline the return values of both the source and target functions, enabling a precise and efficient comparison for equivalency.

Contents

1 Introduction **2**

2 Relevant concepts **2**
2.1 Function Cloning 2
2.2 Control Flow Graph 3
2.3 Symbolic Execution 3
2.4 Mixed Boolean Arithmetic 4

3 Related Work **5**
3.1 Binary Function Clone Search in the Presence of Code Obfuscation and Optimization over Multi-CPU Architectures 5
3.2 Qsynth- A Program Synthesis based Approach for Binary Code Deobfuscation 7
3.3 Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Term Rewriting 9

4 Problem Statement **11**

5 Proposed Methodology **11**
5.1 Dynamic Symbolic Execution 11
5.2 MBA Simplification 12
5.3 Comparison 12

6 Challenges **12**

7 Conclusion **13**

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 groundbreaking solution to binary function clone detection through the application of Mathematical Binary Analysis (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, we transform assembly instructions into mathematical expressions, subsequently employing MBA simplification algorithms to facilitate a precise and efficient comparison of function equivalency.

2 Relevant concepts

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.

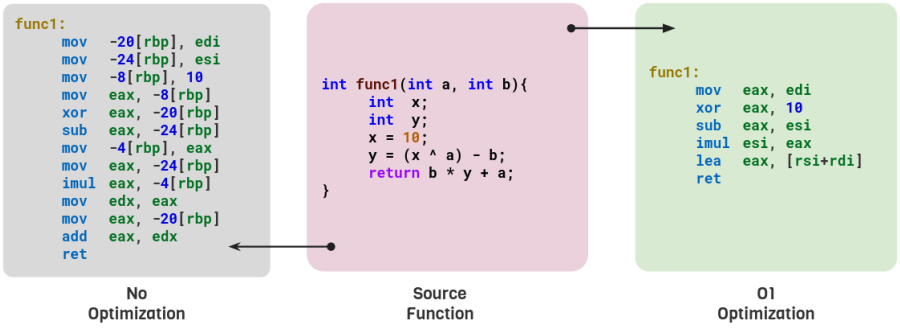

Figure 1: Different Levels of Optimizations

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. For example, 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.

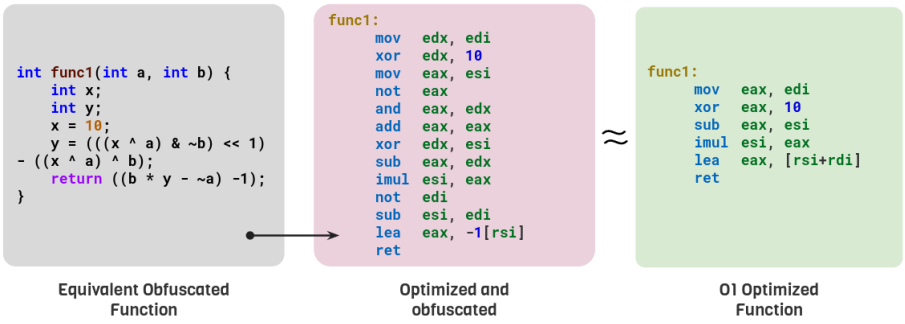

Figure 2: Function Obfuscated and Optimized.

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 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.

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.

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

Figure 3: Example Source Code

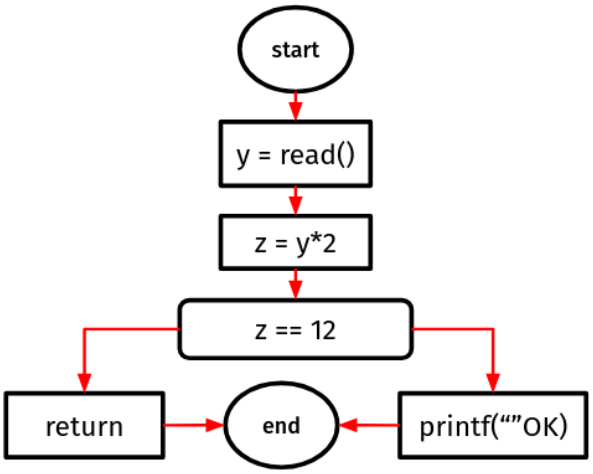

Figure 4: Control Flow Graph

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.

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

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

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

Figure 5: Example Source Code

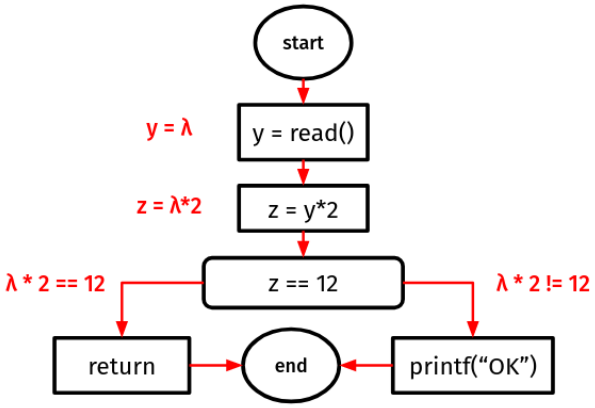

Figure 6: Control Flow Graph with Symbolic Execution

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 to form a more compact and efficient representation. 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.

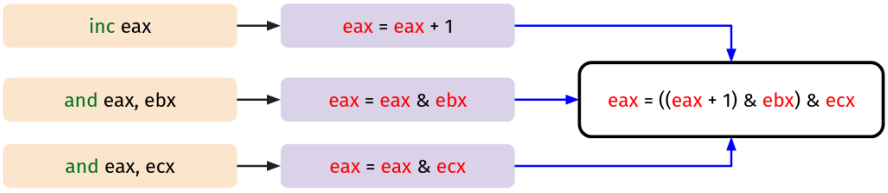

Figure 7: Assembly Instructions to MBA Expressions

3 Related Work

3.1 Binary Function Clone Search in the Presence of Code Obfuscation and Optimization over Multi-CPU Architectures

BinFinder (1) 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

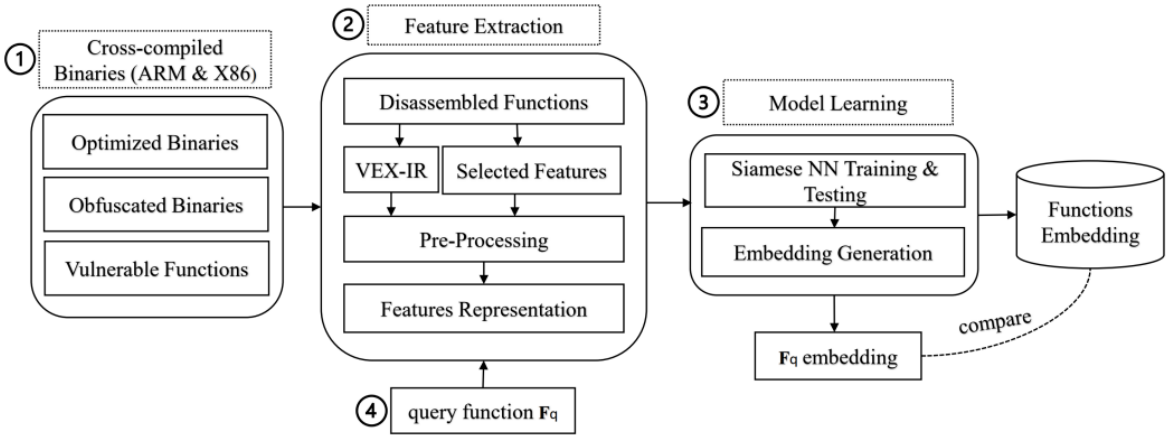

Figure 8: BinFinder working mechanism

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).

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 9: Evalutaion of BinFinder against existing solutions

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 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- A Program Synthesis based Approach for Binary Code Deobfuscation

QSynth (2) 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

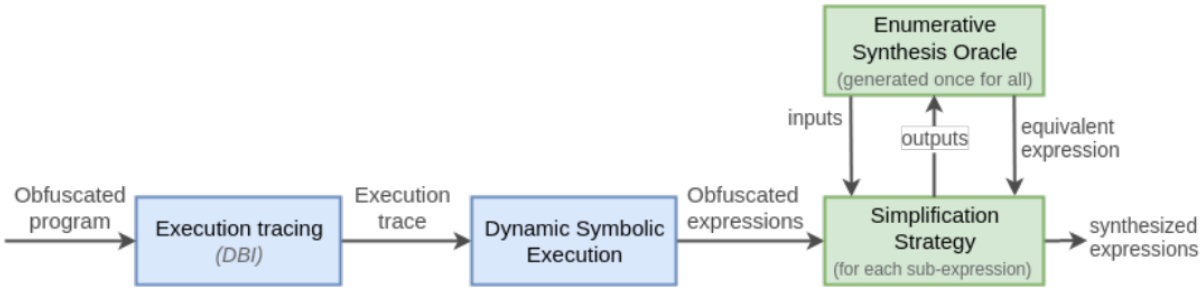

Figure 10: Qsynth working procedure

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 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 11: 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 Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Term Rewriting

The paper (3) 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 (+, -, ×). 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

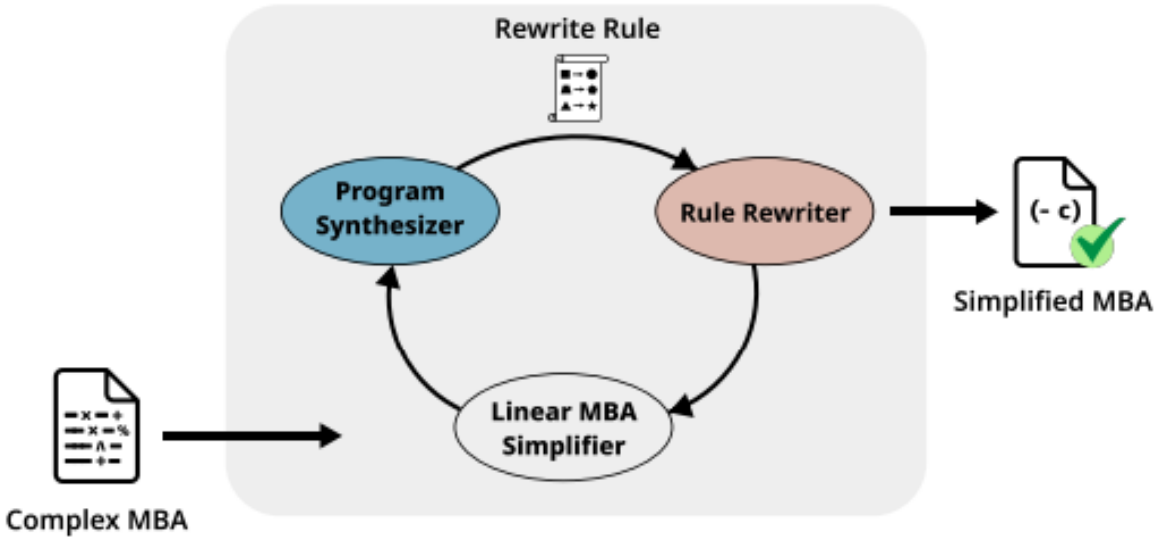

Figure 12: ProMBA working mechanism

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 expression 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(3),

	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 13: Performance comparison of ProMBA against other solutions

4 Problem Statement

Given two binary files identify any function equivalency through extraction and simplification of MBA expressions.

5 Proposed Methodology

Our function clone detection approach involves various steps for which the global overview is shown in Figure 14. 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 on the candidate function. We then simplify the complex MBA expression using an off-the-shelf MBA simplifier to get a set of simplified MBA expressions. We follow the similar process to obtain the simplified MBA expressions of the candidate function from $Binary_2$. For the last step, we compare the set of simplified MBA expressions for equivalency. The different steps are defined in details hereafter.

5.1 Dynamic Symbolic Execution

Dynamic Symbolic Execution (DSE) combines symbolic execution with concrete execution. It executes the program with concrete inputs while tracking symbolic values and uses these symbolic values to guide the exploration of different paths. DSE starts with concrete input values, observes the execution, and dynamically identifies interesting paths or conditions in the code. It then explores these paths symbolically to generate new test cases or analyze the behavior of the program under different conditions. It leverages both concrete and symbolic execution, allowing for a more realistic exploration of program paths. It can be more efficient than pure symbolic

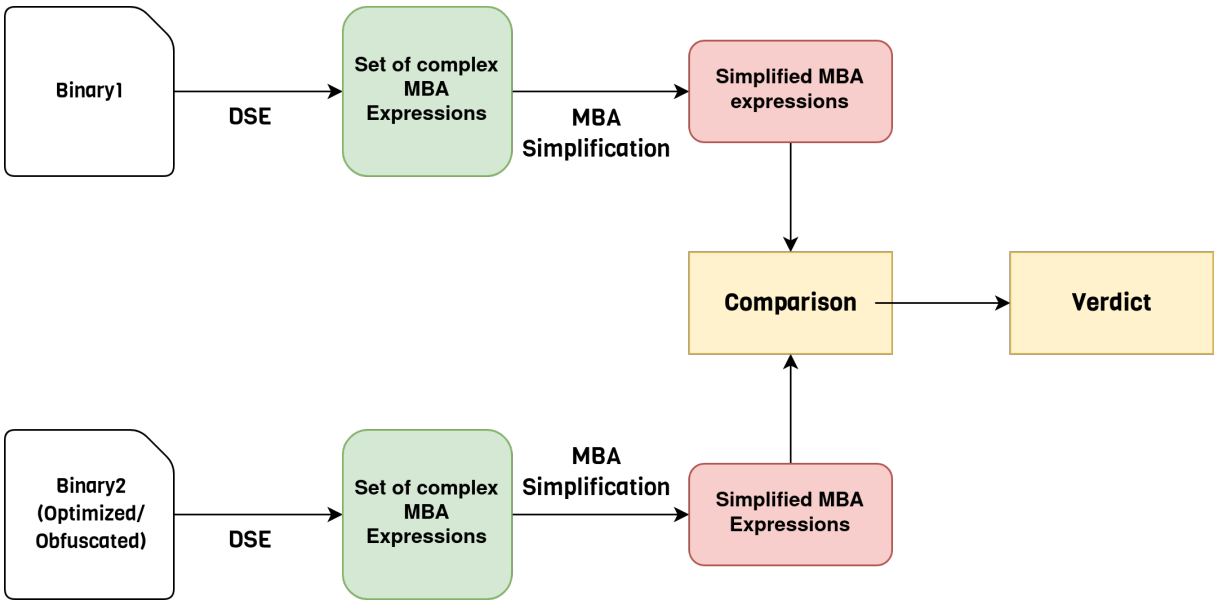

Figure 14: Identifying function clones from two binary files.

execution since it only explores paths that are relevant to the specific execution. Let a set of expressions S_0 represent the state of our CPU before execution. This set holds the current state of the CPU registers and variables. The CPU executes an instruction ins_0 and goes to an update state S_1. We track the changes and update our symbols symbolically. This way, we can find an execution trace and obtain our MBA expressions after analyzing the traces.

5.2 MBA Simplification

The working mechanism of simplifying the MBA expressions obtained in the previous step is explained in figure 12. After simplification of the set of complex MBA expressions, we will obtain a set of simplified MBA expressions.

5.3 Comparison

Currently we are struggling to find a good solution for comparison of two set of MBA expressions to complete our approach. We looked into some possible solutions.
One way is to use SMT solvers to find out if any two expressions from each set are equivalent. If equivalent, we remove them from the set. We repeat the step until both sets are empty or there is no equivalency check left. If the two functions are a clone of each other, all similar expressions should be removed from the set. Based on the number of expressions that remains, we can also come to a conclusion that one function was to a certain fraction similar to the other function in question.
Another solution is to convert these simplified expressions into vector embedding and use machine learning to learn the similarity of two set of MBA expressions.
Another way to approach this problem is using fuzzing on the simplified expressions and match their domain and co-domain with the help of machine learning.

6 Challenges

While proposing our solution to finding function clone from binary executables, we are facing some challenges that needs to be addressed:

1. Extracting MBA expressions from function containing conditional branching, loops and function calls. So far we could find methods to extract MBA expressions from only block

of binary code without branches and looping.

2. Comparing equivalence of two sets of MBA expressions is a problem we find difficult to reach a solution.

3. We are facing difficulty in reaching a conclusion on how to measure our finding in terms of function clone. Whether our verdict will be binary or similarity matrix is still a question we need to solve by reaching a solution to comparison of two sets of MBA expressions.

7 Conclusion

In conclusion, our research aims to provide a pioneering solution to the complex issues surrounding binary function clone detection in closed-source applications plagued by optimization and obfuscation challenges. The prevalence of closed-source software has created a daunting barrier to verifying software integrity, necessitating advanced techniques for plagiarism detection. Our approach, rooted in Mixed Boolean Arithmetic (MBA) simplification, demonstrates a promising avenue for addressing this challenge.

By centering our methodology on the equivalency of source and target functions and employing dynamic symbolic execution, we successfully bridge the gap imposed by the opacity of closed-source environments. The transformation of assembly instructions into mathematical expressions, coupled with MBA simplification algorithms, enables a precise and efficient comparison of function equivalency, even in the presence of intricate obfuscation techniques.

Our proposed solution not only responds to the immediate need for effective plagiarism detection in closed-source applications but also stands as a robust tool for identifying modified and obfuscated malware in the cybersecurity landscape. In the face of rising malicious activities, where attackers exploit existing malware with subtle modifications, our methodology offers a proactive defense mechanism.

As we navigate the intricacies of closed-source software and cybersecurity threats, our research contributes not only to the academic understanding of binary-level analysis but also holds practical implications for enhancing software security and malware detection. By pushing the boundaries of traditional detection methods, our work aims to empower cybersecurity professionals and software developers with a potent tool to safeguard against code reuse, unauthorized content, and evolving malware threats in the dynamic digital landscape. Through this innovative approach, we anticipate a significant stride toward bolstering the trustworthiness and authenticity of closed-source applications, ultimately contributing to a more secure and resilient software ecosystem.

References

[1] 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*, ASIA CCS '23, (New York, NY, USA), p. 443–456, Association for Computing Machinery, 2023.

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

[3] 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*, CCS '23, (New York, NY, USA), p. 2351–2365, Association for Computing Machinery, 2023.