On the Learnability, Robustness, and Adaptability of Deep Learning Models for Obfuscation-applied Code

Jiyong Uhm
Sungkyunkwan University
Suwon, Republic of Korea
jiyong423@g.skku.edu

Yujeong Kwon
Sungkyunkwan University
Suwon, Republic of Korea
shr2008@g.skku.edu

Hyungjoon Koo*
Sungkyunkwan University
Suwon, Republic of Korea
kevin.koo@skku.edu

Abstract

Code obfuscation is a common technique to impede software analysis on purpose by concealing a program's structure, logic, or behavior for both benign and malicious purposes. Its widespread adoption poses a significant challenge to security analysts. While deep learning has shown promise across various binary analysis tasks, the learnability and the applicability from obfuscation-applied code have received less attention to date. Prior work often overlooks code obfuscation or defers it to future research, as the complexity of obfuscation may differ significantly depending on the design and implementation differences across obfuscation tools. In this paper, we investigate how well obfuscation-applied code can be learned on a state-of-the-art model for the binary code similarity detection task. Training the model with obfuscated codes from two source-based and IR-based obfuscation tools (*e.g.,* Tigress, Obfuscator-LLVM), we evaluate: i) learnability on obfuscated code, ii) generalizability for both obfuscated and non-obfuscated code, iii) robustness to known obfuscation techniques, and iv) adaptability to unknown obfuscation techniques. Our findings show that learning a task directly from obfuscated code is feasible, outperforming models trained on large volumes of non-obfuscated code even with a comparatively small dataset. However, achieving generalizability across obfuscated and non-obfuscated code remains challenging. Furthermore, we find that the model's robustness and adaptability to previously (un)known obfuscations is closely tied to the inherent complexity of an obfuscation technique.

CCS Concepts

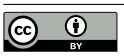
• **Security and privacy** → *Software reverse engineering*.

Keywords

Binary Analysis, Similarity Detection, Obfuscation, Deep Learning

ACM Reference Format:
Jiyong Uhm, Yujeong Kwon, and Hyungjoon Koo. 2025. On the Learnability, Robustness, and Adaptability of Deep Learning Models for Obfuscation-applied Code. In *Proceedings of the 2025 Workshop on Software Understanding and Reverse Engineering (SURE '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3733822.3764676

*Corresponding author.



SURE '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1910-3/25/10
https://doi.org/10.1145/3733822.3764676

1 Introduction

Code obfuscation encompasses a wide spectrum of techniques intentionally designed to hinder software analysis or reverse engineering by obscuring its internal structures, logic, or behaviors. These techniques involve the transformation of boolean and arithmetic expressions, modification of control flow graphs, or the encryption of code into randomized bytecode intended for execution by a specialized interpreter. While developers may utilize obfuscation to enhance software security by thwarting unauthorized inspection, malware authors can exploit it to evade detection and impede reverse engineering efforts. The pervasive use of obfuscation in malware significantly burdens security analysts, as it not only increases the reliance on manual inspection but also challenges the effectiveness of automated analysis tools. As a result, obfuscated code presents an ongoing and critical challenge in the field of cybersecurity.

A number of obfuscation tools are readily available, either as commercial products (*e.g.,* Themida [53], ASPack [54], VMProtect [51]) or as publicly accessible solutions (*e.g.,* UPX [52], Tigress [10], Obfuscator-LLVM [32]). Those accessible tools automate the process of code obfuscation, enabling users to efficiently produce obfuscated executables in a handy way. Furthermore, such code obfuscation tools can be applied at multiple stages of the compilation workflow, including the source code, intermediate representation (IR), or binary level, thereby expanding their applicability across a wide range of contexts.

While the complexity of obfuscation may differ substantially depending on the design and implementation of a given tool, obfuscation techniques can be systematically categorized according to their underlying conceptual similarities. For instance, instruction substitution in O-LLVM [10] and encode arithmetic in Tigress [32] can be classified under the category of mixed Boolean-arithmetic operations. However, the introduction of randomness, along with differences in design and implementation, can result in the generation of infinitely many variations.

Recent advancements in deep learning-assisted binary code analysis have demonstrated promising results by effectively capturing high-level code semantics, such as function name recovery [27, 28, 44], binary code similarity detection (BCSD) [1, 22, 26, 58], function boundary detection [45, 61], decompilation [18, 23, 57], and type prediction [5, 60, 63]. While a few studies have incorporated obfuscated code in their evaluations [12, 46, 47, 56, 61], the majority of prior work has focused primarily on non-obfuscated code alone. Given the ambiguous code semantics and complex representations that are rare in non-obfuscated code, the feasibility and applicability of learning from obfuscated code remain far from straightforward.

In this work, we explore the extent to which semantics can be learned from obfuscation-applied code. We choose one of the state-of-the-art models [1] for the BCSD task. With two publicly available obfuscation tools, Tigress [10] that operates at the source level and Obfuscator-LLVM (O-LLVM) [32] that works at at the IR level, we generate obfuscated executables. We define the following research questions: ① how well the model can be trained on obfuscated code (learnability); ② how consistent the model maintains performance for both obfuscated and non-obfuscated code (generalizability); ③ how robust the model is on known obfuscation techniques during training (robustness); and ④ how well the model can be adapted to unknown (*i.e.,* unseen) obfuscation techniques (adaptability).

Our investigation reveals that while large-scale training on non-obfuscated code offers some robustness against obfuscation, training directly on obfuscated code yields comparable or better performance, even with substantially less amount of data. However, generalizability across both obfuscated and non-obfuscated code is challenging, due to inherent variations in the design and implementation of obfuscation tools. Meanwhile, we observe that the model exhibits reasonable performance on complex obfuscation techniques such as virtualization. Finally, our findings indicate that a model's robustness and adaptability to both known and unknown obfuscation techniques are closely connected to the inherent differences in implementing those techniques.

This paper makes the following contributions:

- We provide insights by systematically investigating the learnability, robustness, and adaptability of deep learning models for obfuscation-applied code.
- We use O-LLVM [32] and Tigress [10] to generate 4,362 executables that apply varying obfuscation techniques, followed by training BCSD models to support comprehensive evaluation.
- We publicly release our binaries, trained models, and compilation scripts to facilitate reproducibility for further research[1].

2 Background and Related Work

Binary Code Similarity Detection. BCSD models aim to gauge the similarity between two code snippets. BCSD has a wide range of real-world applications [21] (*e.g.,* vulnerability detection [19, 39], malware detection [2, 4], malware family classification [24, 36]). For example, a malware variant that applies code obfuscation can be identified via similarity detection models. A similar binary code snippet refers to binary code originating from the same source code. The same source code can result in numerous different executable binaries depending on a specific compilation pipeline. A compilation pipeline can include a compiler (*e.g.,* GCC [17], Clang [48], MSVC [41]), a target architecture (*e.g.,* ARM, x86-64), an optimization level (*e.g.,* `O0`-`3`), a target operating system (*e.g.,* Windows, Linux), and various obfuscation techniques. Since the advent of the transformer [55] numerous highly performant state-of-the-art BCSD models emerged. While these models demonstrate robust performance against non-obfuscated code, their performance on obfuscated code remains underexplored.

Code Obfuscation. Obfuscation encompasses a set of techniques designed to hinder software analysis and reverse engineering. An adversary may apply obfuscation to various components of a program, including executable code, program headers, and string literals. Such transformations can be applied at different stages of the compilation pipeline. We specifically examine obfuscation techniques that operate at two stages: one that transforms source code before compilation(*e.g.,* Tigress [10]), and another that introduces obfuscation during compilation (O-LLVM [32]). Note that we do not explore binary-level obfuscation [51–54] as most of these obfuscations are a form of packing, which we do not consider (Section 6).

Obfuscator-LLVM. O-LLVM is built on top of the LLVM compiler infrastructure, which provides a flexible framework for code transformation during compilation. LLVM supports function passes – modular transformations that can be applied to the intermediate representation during compilation. O-LLVM implements three core obfuscation techniques as function passes: control flow flattening, bogus control flow, and substitution. We discuss these techniques in more detail in Section 3. Because O-LLVM integrates directly into the compiler (*i.e.,* Clang), it can be applied with minimal effort during the build process.

Tigress. Tigress is a source-to-source obfuscation tool that transforms C source code into an obfuscated version before compilation. While Tigress offers a wide range of obfuscation techniques, in this work we select five transformations: add opaque, encode branches, encode arithmetic, flatten, and virtualization. These techniques are described in detail in Section 3. Unlike O-LLVM, integrating Tigress into the build process requires additional effort, particularly for programs with multiple source files. In such cases, source files must first be merged before obfuscation. We describe the compilation workflow we use in Section 4.

Previous Approaches. Obfuscated executables are occasionally employed to evaluate the robustness and generalization capability of a binary analysis model. FID [56] detects function boundaries by extracting semantic information from binary code. It is trained on non-obfuscated code and evaluated on code obfuscated by O-LLVM to assess robustness against obfuscation. Similarly, XDA [46] and DeepDi [61] perform function boundary detection and use a modified O-LLVM [42] for evaluation, with DeepDi additionally including binaries from Linn and Debray [37]. For binary code similarity detection, both Asm2Vec [12] and Trex [47] evaluate model performance using O-LLVM, with Trex employing a modified O-LLVM. BinFinder [50], a binary function clone detection model, is trained jointly on binaries obfuscated with both O-LLVM and Tigress. Except for BinFinder, all other baseline models are trained exclusively on non-obfuscated code. In contrast to BinFinder's joint training strategy, our approach trains a seperate model for each obfuscation tool. This allows us to evaluate a model's performance on (un)known obfuscation techniques during training.

3 Obfuscation Techniques

While O-LLVM and Tigress operate at different stages of the compilation process, their obfuscation techniques can be mainly categorized into three types of code obfuscation.

Mixed Boolean-Arithmetic. Mixed Boolean-Arithmetic (MBA) obfuscation [62] transforms arithmetic and logical expressions into more complex but semantically equivalent forms. MBA disrupts program analysis techniques such as symbolic execution [34]. Both

[1] https://github.com/SecAI-Lab/bcsd_obf_sure2025

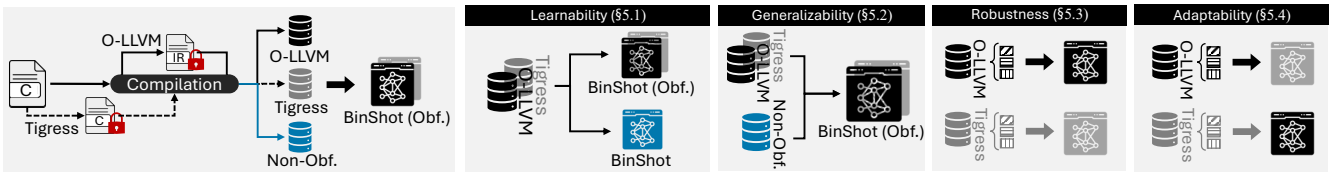


Figure 1: Overview of our experimental pipeline. We construct three BinShot-based models: the original BinShot [1] model as a baseline, and two models trained on code obfuscated with O-LLVM and Tigress. Then, we evaluate these models based on four research questions (Section 5): learnability from obfuscated code; generalizability for both non-obfuscated and obfuscated code; robustness to known obfuscations; and adaptability to unknown obfuscations.

Tigress and O-LLVM implement simplified variants of MBA through encode arithmetic [7] and instruction substitution [31] transformations, respectively. While neither tool generates highly intricate MBA expressions, they systematically replace basic arithmetic and logical operations with sufficiently complex equivalents. Specifically, O-LLVM provides four randomly selected alternative transformations for addition, three for subtraction, and two for each logical AND, OR, and XOR operations. Although Tigress's source code is not publicly available, its documentation for encode arithmetic illustrates four alternate translations for addition, similar to O-LLVM. A key distinction between the two tools lies in their transformation strategies: Tigress frequently adopts logical operators, while O-LLVM prefers inserting randomly generated constants during transformations.

Control Flow Graph Alteration. The control flow graph (CFG) represents a function's execution logic, and obfuscating it reduces interpretability. Various techniques can achieve this, with O-LLVM implementing bogus control flow [29] and control flow flattening [30], while Tigress offers similar capabilities via the add opaque [6] and flatten [9] transformations, respectively. For clarity, we refer to O-LLVM's control flow flattening and Tigress's flatten as "flattening" and O-LLVM's bogus control flow and Tigress's add opaque as "opaque predicate". Flattening restructures a function into a loop with a large switch statement. Both O-LLVM and Tigress support switch statements, but Tigress also includes variants using goto, indirect jumps, function calls, and concurrent calls (where each function runs in a separate thread). Opaque predicate relies on expressions that always evaluate to true or false and inserts unreachable code. O-LLVM duplicates a basic block, inserts junk instructions, and ensures the copy is unreachable. Tigress, in contrast, provides various types of unreachable blocks, including calls to random or non-existent functions, buggy or obfuscated versions of existing code, and random byte sequences. Tigress also implements encode branches [8], an obfuscation where no equivalent technique is implemented in O-LLVM. Encode branches obscure the target of jumps and calls, complicating static analysis. Its default variant, inspired by Linn and Debray [37], replaces direct branches with calls to special functions that manipulate the return address.

Virtualization. Virtualization transforms a function into an interpreter and encodes the original function body as bytecode. Only Tigress supports virtualization, using a technique similar to flattening, where the interpreter is implemented as a switch statement. Tigress also offers eight additional approaches, including direct and indirect threading, as well as nested if statements. Recently, xVMP [59] introduced a virtualization obfuscation extension for O-LLVM. However, xVMP suffers from several limitations – most notably, instability when certain compiler options (*e.g.,* optimization or debug flags) are enabled. The ability to compile with these flags is crucial for generating our dataset. Due to these limitations, we exclude xVMP from our evaluation.

4 Experimental Setup

Overview. Figure 1 depicts the overview of our experimental workflow. We train BCSD models on obfuscated code and investigate four research questions, detailed in Section 5. Our study focuses on two publicly available obfuscation tools: O-LLVM [32] and Tigress [10], selected for their distinct integration points in the compilation pipeline—O-LLVM operates at the IR level, while Tigress targets source code. Interested readers refer to Appendix A for obfuscation examples. To examine the impact of these tools, we train two BinShot [1]-based models, BinShot-O-LLVM and BinShot-Tigress, on code obfuscated with O-LLVM and Tigress, respectively. While our goal is to evaluate the model's ability to learn and generalize across different obfuscation techniques, we are also interested in its capacity to transfer knowledge across tools that implement similar transformations. This motivates our use of BinShot, which is explicitly designed for transferability in BCSD tasks.

Applying Tigress Obfuscations. Tigress is designed to process a single C source file at a time, rendering direct application to multi-source packages difficult, such as coreutils. That is, each binary is built from multiple source files typically with many dependencies and complex paths, being unable to utilize the description of compilation (*e.g.,* Makefile). To compile coreutils with Tigress, we manually merge the necessary source files into a single file for each binary and provide the appropriate linker flags (*e.g.,* `lselinux` for cp, mkdir, and mv). To address this, we wrote a script that compiles coreutils with Tigress across five obfuscation techniques and four optimization levels. During this process, we encountered an undeclared identifier error in the `dcpgettext_expr` function. We found that Tigress misinterpreted a macro expansion, causing the declaration and usage of the `msg_ctxt_id` variable to be separated into different scopes. We fixed it by including a patching script that automatically detects and resolves the error. We use GCC for compilation, as Clang produces machine-specific type errors due to incompatibilities with Tigress's custom C Intermediate Language (CIL) parser. Nonetheless, some binaries (*e.g.,* od, ptx, stat) have

been failed for compilation when applying obfuscation techniques, such as virtualization.

Preprocessing and Preparation. We prepare binaries by disassembling them with IDA Pro 8.2 and the Capstone Python library [3]. To eliminate a bias, we remove redundant functions, filtering out 168,868 functions in total from executables. We further filter out functions with five or fewer instructions, removing 93,731 functions in total. Finally, we split the remaining functions into training and testing sets with an 8:2 ratio, resulting in 886,615 samples for training and 70,056 samples for testing.

Model Generation. While we largely follow BinShot's architecture for our custom models (*i.e.,* BinShot-O-LLVM, BinShot-Tigress), we introduce two key modifications. First, we replace the pretrained BERT encoder with RoBERTa [38], increasing the maximum input length from 256 to 512 tokens. The extended sequence enables the model to capture a broader context. Second, we adopt byte pair encoding (BPE) instead of the well-balanced normalization method [33]. BPE preserves more semantic detail in tokens, helping the model handle the semantic complexity caused by obfuscation. Both models are trained in two phases: a pretraining phase, where the model learns instruction-level semantics, followed by a fine-tuning phase to adapt the model to the downstream BCSD task. Each model is pretrained for 15 epochs and fine-tuned for an additional 10 epochs. For training BinShot-O-LLVM, we use binaries obfuscated with instruction substitution, opaque predicates, flattening, and the combination of these three techniques. For BinShot-Tigress, we use binaries obfuscated with encode arithmetic, opaque predicates, flattening, encode branches, and virtualization. BinShot-O-LLVM requires approximately 20 minutes for pretraining and 9.3 hours for fine-tuning, while BinShot-Tigress was pretrained in 5 minutes and fine-tuned for 2.41 hours. We use the Adam optimizer with a learning rate of 0.0005, $\beta_1 = 0.9$, $\beta_2 = 0.999$, weight decay of 0.01, $\epsilon = 1e\text{-}6$, and a warm-up proportion of 0.1. For evaluation, we also use the original BinShot model released by the authors without any modifications.

5 Evaluation

We evaluate BinShot [1], BinShot-O-LLVM, and BinShot-Tigress on our four datasets. All experiments are conducted on a 64-bit Ubuntu 20.04 system equipped with an AMD EPYC 7763 64-Core Processor, 512GB of RAM, and two NVIDIA A100 GPU.

Datasets. We generate both obfuscated and non-obfuscated binaries from GNU libraries (*e.g.,* coreutils, binutils), a variety of open-source projects (*e.g.,* PuTTY, Nginx), and a benchmark suite (*e.g.,* SPEC2006). Table 1 shows the number of obfuscated binaries generated. Due to the complexity of generating obfuscated source for multi-source packages with Tigress, we limit its dataset to coreutils. Binaries are compiled in the ELF format for the x86-64 architecture, with optimizations ranging from `O0` to `O3`. O-LLVM binaries are compiled using Clang (19.1.4), and Tigress binaries with GCC (7.2.0). Note that some binaries failed to generate due to compiler compatibility issues with O-LLVM and unresolved statements in the CIL representation with Tigress, resulting in unknown exceptions. For evaluation, we construct four test datasets: ① **Dataset-Clang**: 2,500 positive and 2,500 negative pairs of non-obfuscated functions compiled with Clang; ② **Dataset-GCC**: 2,074 positive and 2,074 negative pairs of non-obfuscated functions compiled with GCC; ③ **Dataset-O-LLVM**: 4,000 positive and 4,000 negative pairs of O-LLVM-obfuscated and non-obfuscated functions; and ④ **Dataset-Tigress**: 5,000 positive and 5,000 negative pairs of Tigress-obfuscated and non-obfuscated functions.

Table 1: We generate 4,362 obfuscated-applied executables using O-LLVM and Tigress, compiled with Clang and GCC, respectively. For training and evaluation, we also compile 604 non-obfuscated binaries with Clang and 408 with GCC.

		O-LLVM		Tigress	
Package	**Version**	**#Binaries**	**#Functions**	**#Binaries**	**#Functions**
binutils [15]	2.27	150	218,135	–	–
coreutils [14]	8.2	1,648	131,142	2,020	59,088
diffutils [16]	3.2	64	6,353	–	–
gzip [20]	1.8	16	1,685	–	–
lighttpd [35]	1.4.43	16	5,897	–	–
lvm2 [40]	2.02.168	32	37,295	–	–
Nginx [43]	1.8.1	16	17,908	–	–
PuTTy [49]	0.66	112	90,682	–	–
vsftpd [13]	3.0.3	16	6,953	–	–
miniweb [25]	–	16	1,081	–	–
SPEC2006 [11]	–	256	167,559	–	–
Total		2,342	684,690	2,020	59,088

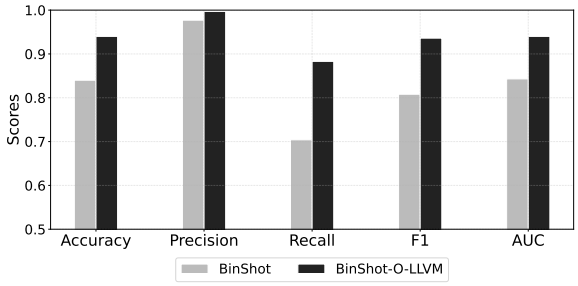


Figure 2: Performance comparison between BinShot [1] (gray) and BinShot-O-LLVM (black) for O-LLVM-obfuscated code. After training on O-LLVM-obfuscated code, BinShot-O-LLVM achieves higher performance on obfuscated code compared to BinShot, despite trained on a smaller dataset (RQ1).

Research Questions. We define the following four research questions to investigate the learnability, robustness, and adaptability of a state-of-the-art BCSD model for obfuscation-applied code.

- **RQ1**: How well can the deep learning model be trained from obfuscated code (Section 5.1)?
- **RQ2**: How well can the model be generalizable for both obfuscated and non-obfuscated code (Section 5.2)?
- **RQ3**: How robust is the model on known obfuscation techniques (Section 5.3)?
- **RQ4**: How well the model be adapted to unknown obfuscation techniques (Section 5.4)?

5.1 Learnability from Obfuscated Code

This section evaluates the learnability of obfuscated code by comparing the performance of BinShot that has not seen obfuscated code and our own models trained with obfuscated code. Specifically,

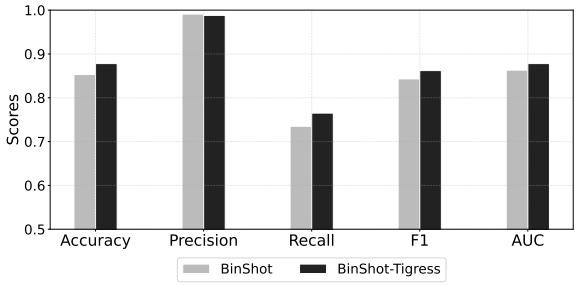


Figure 3: Performance comparison between BinShot [1] (gray) and BinShot-Tigress (black) for Tigress-obfuscated code. After training on Tigress-obfuscated code, BinShot-Tigress achieves comparable performance on obfuscated code compared to BinShot, despite trained on a smaller dataset (RQ1).

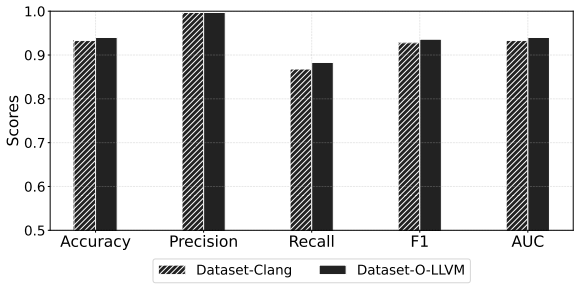


Figure 4: Performance of BinShot-O-LLVM on non-obfuscated (left) and obfuscated code (right). After training on obfuscated code, BinShot-O-LLVM retains similar performance on regardless of obfuscation (RQ2).

we evaluate BinShot and BinShot-O-LLVM on Dataset-O-LLVM, and BinShot and BinShot-Tigress on Dataset-Tigress.

Results. Figure 2 and Figure 3 present how well BinShot-O-LLVM and BinShot-Tigress can learn from obfuscation-applied code. We compare the performance of each model against the original BinShot [1] model. BinShot-O-LLVM exhibits better performance compared to BinShot (Figure 2) while BinShot-Tigress demonstrates comparable results to the original (Figure 3). We hypothesize that the performance gap may arise from a relatively small dataset when training models with obfuscated code: recall that BinShot was trained on 15 distinct software packages, whereas BinShot-O-LLVM and BinShot-Tigress on only 11 and 1 package(s), respectively.

> **Takeaway #1** Deep learning-based models are learnable to some extent by training directly on obfuscated code, achieving performance improvement on obfuscated samples, even with a relatively small training dataset.

5.2 Model Generalizability

We evaluate the generalization capability of BinShot-O-LLVM and BinShot-Tigress on both non-obfuscated and obfuscated code.

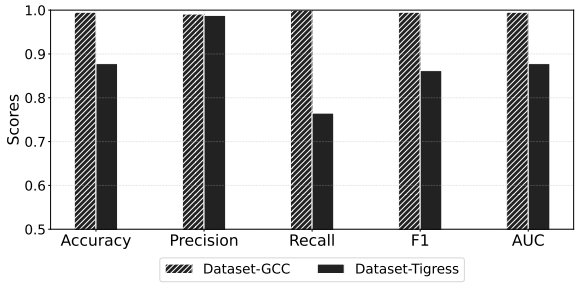


Figure 5: Performance of BinShot-Tigress on non-obfuscated (left) and obfuscated code (right). Interestingly, unlike BinShot-O-LLVM, we observe discrepancies in performance between non-obfuscated and obfuscated code: *i.e.,* the BCSD performance on non-obfuscated code is significantly better. We hypothesize that relatively small training data for the BinShot-Tigress model constraints its performance (RQ2).

BinShot-O-LLVM is evaluated on Dataset-Clang and Dataset-O-LLVM, while BinShot-Tigress is evaluated on Dataset-GCC and Dataset-Tigress.

Results. Figure 4 and Figure 5 illustrate the generalizability of BinShot-O-LLVM and BinShot-Tigress across (non-)obfuscated code. BinShot-O-LLVM achieves comparable performance on both Dataset-Clang and Dataset-O-LLVM, indicating that it performs well on both obfuscated and non-obfuscated code. However, BinShot-Tigress performs better on Dataset-GCC compared to Dataset-Tigress, demonstrating high performance on non-obfuscated code alone. The results suggest that persistent model behaviors may vary between obfuscated and non-obfuscated code, depending on the design and implementation of a given obfuscation tool. We hypothesize that this discrepancy may partially stem from the limited training corpus of BinShot-Tigress.

> **Takeaway #2** Achieving generalizability in deep learning models remains challenging, even when trained on obfuscated code. This limitation arises from inherent complexities caused by variations in the design and implementation of obfuscation tools.

5.3 Robustness to Known Obfuscations

We evaluate the performance of BinShot-O-LLVM and BinShot-Tigress on individual obfuscation techniques they encountered during training. To ensure a balanced comparison, Dataset-O-LLVM and Dataset-Tigress are organized such that each obfuscation technique contributes an equal number of function pairs. BinShot-O-LLVM is evaluated on Dataset-O-LLVM and BinShot-Tigress on Dataset-Tigress.

Results. Table 2 (white background) presents the performance of BinShot-O-LLVM and BinShot-Tigress on known obfuscation techniques encountered during training. While both models perform well on these techniques, we observe a notable discrepancy in instruction substitution and encode arithmetic. Although both techniques are forms of MBA, BinShot-Tigress performs relatively

Table 2: Performance of BinShot-O-LLVM and BinShot-Tigress on obfuscation techniques encountered during training. Obfuscation techniques are categorized as mixed boolean-arithmetic (MBA), control flow graph alteration, and others. A, P, R, F1, and AUC denote accuracy, precision, recall, harmonic mean of precision and recall, and area under the curve, respectively. Both models demonstrate decent performance on known obfuscation techniques (white background; RQ3). For unknown obfuscation techniques (gray background; RQ4), BinShot-Tigress shows better adaptability compared to BinShot-O-LLVM.

	Mixed Boolean-Arithmetic						Control Flow Graph Alteration						Other				
	Type	A	P	R	F1	AUC	Type	A	P	R	F1	AUC	Type	A	P	R	F1	AUC
BinShot-O-LLVM	Substitution	0.960	0.997	0.922	0.958	0.960	Flattening	0.932	0.998	0.866	0.927	0.932	Combined	0.920	0.995	0.845	0.914	0.920
							OpaquePredicate	0.948	0.997	0.899	0.945	0.948						
BinShot-Tigress	EncodeArithmetic	0.871	0.988	0.752	0.854	0.871	Flattening	0.884	0.986	0.780	0.871	0.884	Virtualization	0.876	0.989	0.760	0.859	0.876
							OpaquePredicate	0.878	0.990	0.764	0.862	0.878						
							EncodeBranch	0.880	0.987	0.771	0.865	0.880						
BinShot-O-LLVM	EncodeArithmetic	0.589	0.994	0.180	0.304	0.589	Flattening	0.606	1.000	0.212	0.349	0.606	Virtualization	0.599	0.980	0.203	0.336	0.599
							OpaquePredicate	0.604	0.995	0.210	0.346	0.604						
							EncodeBranch	0.595	1.000	0.190	0.319	0.595						
BinShot-Tigress	Substitution	0.695	0.891	0.445	0.593	0.695	Flattening	0.636	0.816	0.352	0.492	0.636	Combined	0.582	0.747	0.248	0.372	0.582
							OpaquePredicate	0.595	0.787	0.260	0.391	0.595						

worse on encode arithmetic compared to BinShot-O-LLVM on instruction substitution. This may be attributed to differences in implementation: instruction substitution frequently employs random constants, whereas encode arithmetic relies more heavily on logical operators. These results suggest that the specific implementation details of an obfuscation technique can influence its learnability.

Takeaway #3 Deep learning-based models trained with obfuscated code demonstrate a decent performance for known obfuscation techniques. However, the specific implementation of an obfuscation technique can substantially affect this performance.

5.4 Adaptability to Unknown Obfuscations

We evaluate the adaptability of BinShot-O-LLVM and BinShot-Tigress on unknown obfuscation techniques during training. We achieve this by testing BinShot-O-LLVM on Dataset-Tigress and Tigress on Dataset-O-LLVM. In essence, each model is evaluated on code generated by a different obfuscation tool.

Results. Table 2 (gray background) presents the performance of BinShot-O-LLVM and BinShot-Tigress on unknown obfuscation techniques. While both models maintain high precision, we observe a notable drop in other metrics across the board. For BinShot-O-LLVM, the lowest F1 score of 0.304 is recorded for encode arithmetic, despite this technique being conceptually similar to instruction substitution. However, BinShot-Tigress exhibits the lowest performance (F1: 0.372) in combined obfuscation, which is an unknown obfuscation technique for this model. Despite BinShot-O-LLVM being trained on a larger dataset, BinShot-Tigress achieves a better overall performance on unknown obfuscations. This observation suggests that the implementation of an obfuscation tool impacts the robustness of a model on unknown obfuscation techniques. Overall, both models struggle to handle unknown obfuscation techniques, even when they are similar to those used during training.

Takeaway #4 The model's ability to adapt to previously unknown obfuscation during training is closely tied to the complexity of the obfuscation techniques it has encountered.

6 Discussion and Limitations

Effect of Obfuscation Techniques on Model Performance. In general, we observe that incorporating obfuscated code during training improves model performance on obfuscated inputs. We also find that training on a large non-obfuscated dataset can provide a degree of robustness against obfuscation. Despite this, models trained specifically on obfuscated code achieve better or comparable results while requiring a significantly smaller dataset. However, generalization to unknown obfuscation techniques yield mixed results. BinShot-Tigress outperforms BinShot-O-LLVM despite being trained on a smaller dataset, which may be attributed to differences in how Tigress and O-LLVM implements obfuscation techniques. We further observe implementation-specific effects in the case of MBA. O-LLVM incorporates random constants, while Tigress relies more heavily on logical operators. We observe that both models struggled more with Tigress's implementation, suggesting that the specific implementation of an obfuscation technique can substantially influence model performance.

Limited Obfuscation Tools. We do not consider binary-level obfuscation tools such as Themida [53], ASPack [54], VMProtect [51], and UPX [52]. These tools perform packing, an obfuscation technique that conceals executable code by encrypting it and only revealing it at runtime. Packing and other advanced obfuscation techniques that apply encryption to executable code are beyond the scope of our work.

Scope and Variability Constraints in Obfuscation Experiments. We do not explore the extended configuration options provided by O-LLVM and Tigress. Both tools offer parameters to control the frequency and complexity of their obfuscation techniques. However, for brevity, we restrict our experiments to the default setting of each obfuscation technique. The obfuscation techniques used in our paper inherently introduce variability—repeated

runs with the same configuration can yield different binaries. However, in our experiments, we generate each obfuscated binary only once and do not investigate the effect of this variance. We leave a deeper analysis of the variability of obfuscation to future work.

Imbalanced Dataset. Our experiments with Tigress limits scalability (compared to O-LLVM) due to dataset generation constraints. We leave expanding the dataset as part of future work, which includes a broader range of programs can provide deeper insight into the impact of obfuscation on deep learning models.

Future Work. Commercial tools such as VMProtect and Themida employ sophisticated virtualization-based obfuscation techniques, which we investigate as part of our future work. Besides, interpreting undesirable model behavior (*e.g.,* false positives and false negatives) through explainable AI techniques is essential for developing robust models.

7 Conclusion

Despite the wide adoption of code obfuscation, its applicability and learnability in deep learning models have received less attention. In this paper, we examine how well a state-of-the-art BCSD model can learn and perform on obfuscated code. Our findings show that training directly on obfuscated code is not only feasible but also outperforms models trained on much larger datasets of non-obfuscated code. However, generalizability across obfuscated and non-obfuscated code is challenging to achieve. Finally, we observe the model's robustness and adaptability to some extent depending on the complexity of an obfuscation technique.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. This work was supported by the grants from Institute of Information & communications Technology Planning & Evaluation (IITP), funded by the Korean government (MSIT; Ministry of Science and ICT): No. RS-2022-II221199, No. RS-2024-00437306, and No. RS-2024-00337414. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

References

[1] Sunwoo Ahn, Seonggwan Ahn, Hyungjoon Koo, and Yunheung Paek. 2022. Practical Binary Code Similarity Detection with BERT-based Transferable Similarity Learning. In *Proceedings of the 38th Annual Computer Security Applications Conference.*

[2] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *2008 IEEE Symposium on Security and Privacy (sp 2008).*

[3] Capstone. 2025. Capstone The Ultimate Disassembler. https://www.capstone-engine.org/

[4] Silvio Cesare, Yang Xiang, and Wanlei Zhou. 2014. Control Flow-Based Malware VariantDetection. *IEEE Transactions on Dependable and Secure Computing* (2014).

[5] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2022. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security 22).*

[6] Christian Collberg. 2025. Add Opaque. https://tigress.wtf/addOpaque.html

[7] Christian Collberg. 2025. Encode Arithmetic. https://tigress.wtf/encodeArithmetic.html

[8] Christian Collberg. 2025. Encode Branches. https://tigress.wtf/encodeBranches.html

[9] Christian Collberg. 2025. Flatten. https://tigress.wtf/flatten.html

[10] Christian Collberg. 2025. the tigress c obfuscator. https://tigress.wtf/

[11] Standard Performance Evaluation Corporation. 2025. SPEC CPU 2006. https://www.spec.org/cpu2006/

[12] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 ieee symposium on security and privacy (sp).*

[13] Chris Evans. 2025. vsftpd. https://security.appspot.com/vsftpd.html

[14] Free Software Foundation. 2025. Coreutils - GNU core utilities. https://www.gnu.org/software/coreutils/

[15] Free Software Foundation. 2025. GNU Binutils. https://www.gnu.org/software/binutils/

[16] Free Software Foundation. 2025. GNU Diffutils. https://www.gnu.org/software/diffutils/

[17] Inc. Free Software Foundation. 2025. GCC, the GNU Compiler Collection. http://gcc.gnu.org/

[18] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. 2019. Coda: An end-to-end neural program decompiler. *Advances in Neural Information Processing Systems* (2019).

[19] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.*

[20] gzip. 2025. The gzip home page. https://www.gzip.org/

[21] Irfan Ul Haq and Juan Caballero. 2021. A Survey of Binary Code Similarity. *ACM Comput. Surv.* (2021).

[22] Haojie He, Xingwei Lin, Ziang Weng, Ruijie Zhao, Shuitao Gan, Libo Chen, Yuede Ji, Jiashui Wang, and Zhi Xue. 2024. Code is not natural language: Unlock the power of semantics-oriented graph representation for binary code similarity detection. In *33rd USENIX Security Symposium (USENIX Security 24), PHILADELPHIA, PA.*

[23] Iman Hosseini and Brendan Dolan-Gavitt. 2022. Beyond the C: Retargetable decompilation using neural machine translation. *arXiv preprint arXiv:2212.08950* (2022).

[24] Xin Hu, Kang G. Shin, Sandeep Bhatkar, and Kent Griffin. 2013. MutantX-S: Scalable Malware Clustering Based on Static Features. In *2013 USENIX Annual Technical Conference (USENIX ATC 13).*

[25] Stanley Huang. 2025. MiniWeb. https://miniweb.sourceforge.net/

[26] Ang Jia, Ming Fan, Xi Xu, Wuxia Jin, Haijun Wang, and Ting Liu. 2024. Cross-inlining binary function similarity detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering.*

[27] Linxi Jiang, Xin Jin, and Zhiqiang Lin. 2025. Beyond Classification: Inferring Function Names in Stripped Binaries via Domain Adapted LLMs. In *Proceedings of the 2025 on ACM SIGSAC Conference on Computer and Communications Security.*

[28] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 2022. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security.*

[29] Pascal Junod. 2025. Bogus Control Flow. https://github.com/obfuscator-llvm/obfuscator/wiki/Bogus-Control-Flow

[30] Pascal Junod. 2025. Control Flow Flattening. https://github.com/obfuscator-llvm/obfuscator/wiki/Control-Flow-Flattening

[31] Pascal Junod. 2025. Instruction Substitution. https://github.com/obfuscator-llvm/obfuscator/wiki/Instructions-Substitution

[32] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, June 19th, 2015.*

[33] Hyungjoon Koo, Soyeon Park, Daejin Choi, and Taesoo Kim. 2023. Binary code representation with well-balanced instruction normalization. *IEEE Access* (2023).

[34] Jaehyung Lee and Woosuk Lee. 2023. Simplifying mixed boolean-arithmetic obfuscation by program synthesis and term rewriting. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security.*

[35] lighttpd. 2025. Lighttpd - fly light. https://www.lighttpd.net/

[36] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. 2012. Lines of malicious code: insights into the malicious software industry. In *Proceedings of the 28th Annual Computer Security Applications Conference.*

[37] Cullen Linn and Saumya Debray. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security.*

[38] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2020. Roberta: A robustly optimized bert pretraining approach. In *Proceedings of the International Conference on Learning Representations (ICLR).*

[39] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. 2023. VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search.. In *NDSS.*

[40] lvm2. 2025. LVM2. https://github.com/lvmteam/lvm2

[41] Microsoft. 2025. Microsoft C++, C, and Assembler documentation. https://docs.microsoft.com/en-us/cpp/
[42] Naville. 2025. Hikari. https://github.com/HikariObfuscator/Hikari
[43] nginx. 2025. nginx. https://nginx.org/
[44] James Patrick-Evans, Moritz Dannehl, and Johannes Kinder. 2023. Xfl: Naming functions in binaries with extreme multi-label learning. In *2023 IEEE Symposium on Security and Privacy (SP)*.
[45] Kexin Pei, Jonas Guan, David Williams King, Junfeng Yang, and Suman Jana. 2021. XDA: Accurate, Robust Disassembly with Transfer Learning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS)*.
[46] Kexin Pei, Jonas Guan, David Williams King, Junfeng Yang, and Suman Jana. 2021. XDA: Accurate, Robust Disassembly with Transfer Learning. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS)*.
[47] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2022. Trex: Learning execution semantics from micro-traces for binary similarity. *IEEE Transactions on Software Engineering* (2022).
[48] The LLVM Project. 2025. Clang: a C language family frontend for LLVM. https://clang.llvm.org/
[49] putty. 2025. PuTTY: a free SSH and Telnet client. https://www.chiark.greenend.org.uk/~sgtatham/putty/
[50] Abdullah Qasem, Mourad Debbabi, Bernard Lebel, and Marthe Kassouf. 2023. 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*.
[51] VMProtect Software. 2025. Advanced code security made simple and reliable. https://vmpsoft.com/
[52] The UPX Team. 2025. the Ultimate Packer for eXecutables. https://upx.github.io/
[53] Oreans Technologies. 2025. Themida Overview. https://www.oreans.com/Themida.php
[54] Ltd Union Liyuan Consulting Co. 2025. ASPack Software. http://www.aspack.com/
[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*.
[56] Shuai Wang, Pei Wang, and Dinghao Wu. 2017. Semantics-aware machine learning for function recognition in binary code. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*.
[57] Josh Wiedemeier, Elliot Tarbet, Max Zheng, Sangsoo Ko, Jessica Ouyang, Sang Kil Cha, and Kangkook Jee. 2024. PYLINGUAL: Toward Perfect Decompilation of Evolving High-Level Languages. In *2025 IEEE Symposium on Security and Privacy (SP)*.
[58] Wai Kin Wong, Huaijin Wang, Zongjie Li, and Shuai Wang. 2024. Binaug: Enhancing binary similarity analysis with low-cost input repairing. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*.
[59] Xuangan Xiao, Yizhuo Wang, Yikun Hu, and Dawu Gu. 2023. xVMP: An LLVM-based Code Virtualization Obfuscator. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*.
[60] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. 2024. Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*.
[61] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. 2022. {DeepDi}: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly. In *31st USENIX Security Symposium (USENIX Security 22)*.
[62] Yongxin Zhou, Alec Main, Yuan X Gu, and Harold Johnson. 2007. Information hiding in software with mixed boolean-arithmetic transforms. In *International Workshop on Information Security Applications*.
[63] Chang Zhu, Ziyang Li, Anton Xue, Ati Priya Bajaj, Wil Gibbs, Yibo Liu, Rajeev Alur, Tiffany Bao, Hanjun Dai, Adam Doupé, et al. 2024. {TYGR}: Type Inference on Stripped Binaries using Graph Neural Networks. In *33rd USENIX Security Symposium (USENIX Security 24)*.

Appendices

A Obfuscated Code Examples

The following code examples illustrate the Mixed Boolean-Arithmetic (MBA) obfuscation technique in practice. The `dot_or_dotdot`, with Tigress, function's original conditional operation at line 2 in Listing 1 has been transformed into the obfuscated (source) code at lines 4–8 in Listing 2. With O-LLVM, the `add` function's original addition operation at line 9 in Listing 3 has been replaced in the obfuscated IR at lines 9–12 in Listing 4. Note that both transformations preserve the original semantics while increasing its complexity.

Listing 1: Original `dot_or_dotdot` function.

```
1  static inline bool dot_or_dotdot(char const *file_name) {
2    if (file_name[0] == '.') {
3      char sep = file_name[(file_name[1] == '.') + 1];
4      return (!sep || ISSLASH(sep));
5    } else
6      return false;
7  }
```

Listing 2: Obfuscated source code for `dot_or_dotdot` with Tigress' MBA.

```
1  static inline bool dot_or_dotdot(char const *file_name) {
2    char sep;
3    int tmp;
4    if ((int)(((unsigned int const)(
5    ((((int const) * (file_name+0) - 46) + (1<<31)) ^
6    ((((int const) * (file_name+0) - 46) + (1<<31)) >>
7    (int const)31))-((((int const)*(file_name+0)-46)+(1<<31)) >>
8    (int const)31)) >> (unsigned int const)31) & 1)) {
9      sep=(char)*(file_name+(((int const)*(file_name+1)==46)+1));
10     if (!sep) {
11       tmp = 1;
12       } else if
13         (((unsigned int)(((((int)sep - 47) + (1 << 31)) +
14         ((((int)sep - 47) + (1 << 31)) >> 31)) ^
15         ((((int)sep - 47) + (1 << 31)) >> 31)) >> 31U) & 1){
16       tmp = 1;
17       } else {
18       tmp = 0;
19       }
20       return ((_Bool)tmp);
21   } else {
22     return ((_Bool)0);
23   }
24   }
```

Listing 3: Original `add` function.

```
1  define dso_local i32 @add(i32 noundef %a, i32 noundef %b) #0 {
2    entry:
3    %a.addr = alloca i32, align 4
4    %b.addr = alloca i32, align 4
5    store i32 %a, ptr %a.addr, align 4
6    store i32 %b, ptr %b.addr, align 4
7    %0 = load i32, ptr %a.addr, align 4
8    %1 = load i32, ptr %b.addr, align 4
9    %add = add nsw i32 %0, %1
10   ret i32 %add
11 }
```

Listing 4: Obfuscated IR for `add` with O-LLVM's MBA.

```
1  define dso_local i32 @add(i32 noundef %a, i32 noundef %b) #0 {
2    entry:
3    %a.addr = alloca i32, align 4
4    %b.addr = alloca i32, align 4
5    store i32 %a, ptr %a.addr, align 4
6    store i32 %b, ptr %b.addr, align 4
7    %0 = load i32, ptr %a.addr, align 4
8    %1 = load i32, ptr %b.addr, align 4
9    %2 = add i32 %0, 1949326951
10   %3 = add i32 %2, %1
11   %4 = sub i32 %3, 1949326951
12   %add = add nsw i32 %0, %1
13   ret i32 %4
14 }
```