ALMOND: Learning an Assembly Language Model for 0-Shot Code Obfuscation Detection

XUEZIXIANG LI, University of California at Riverside, USA
SHENG YU, Deepbits Technology, USA
HENG YIN, University of California at Riverside, USA

Code obfuscation is a technique used to protect software by making it difficult to understand and reverse engineer. However, it can also be exploited for malicious purposes such as code plagiarism or developing malicious programs. Learning-based techniques have achieved great success with the help of supervised learning and labeled training sets. However, when faced with real-life environments involving privately developed and undisclosed obfuscators, these supervised learning methods often raise concerns about generalizability and robustness when facing unseen and unknown classes of obfuscation techniques.

This paper presents ALMOND, a novel zero-shot approach for detecting code obfuscation in binary executables. Unlike previous supervised learning methods, ALMOND does not require labeled obfuscated samples for training. Instead, it leverages a language model pre-trained only on unobfuscated assembly code to identify the linguistic deviations introduced by obfuscation. The key innovation is the use of "error-perplexity" as a detection metric, which focuses on tokens the model fails to predict. Continuous Error Perplexity further enhances this to capture consecutive prediction errors characteristic of obfuscated sequences. Experiments show ALMOND achieves 96.3% accuracy on unseen obfuscation methods, outperforming supervised baselines. On real-world malware samples, it demonstrates an AUC of 0.869 and significantly outperforms the supervise-learning baseline. Our Dataset, pre-trained model, and code of evaluation will be available at https://github.com/palmtreemodel/ALMOND

CCS Concepts: • **Security and privacy → Software reverse engineering**; *Intrusion/anomaly detection and malware mitigation*; • **Theory of computation** → *Program analysis.*

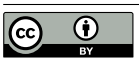
Additional Key Words and Phrases: Deep Learning, Binary Analysis, Language Model, Obfuscation Detection

ACM Reference Format:
Xuezixiang Li, Sheng Yu, and Heng Yin. 2025. ALMOND: Learning an Assembly Language Model for 0-Shot Code Obfuscation Detection. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA017 (July 2025), 22 pages. https://doi.org/10.1145/3728886

1 Introduction

Code obfuscation is a technique used to deliberately make source code, binary code, or program logic difficult to understand, interpret, or reverse-engineer. It alters the code's structure and syntax without changing its functionality. The main objectives are to protect intellectual property, prevent unauthorized access, and hinder reverse engineering by reducing code readability and analysis. However, obfuscation is also commonly exploited for illegal purposes, including malware development [57], code plagiarism [20], and intellectual property theft [54]. Detecting code obfuscation helps security professionals identify hidden or malicious behaviors, making it a crucial element of modern cybersecurity strategies.

Authors' Contact Information: Xuezixiang Li, University of California at Riverside, Riverside, USA, xli287@ucr.edu; Sheng Yu, Deepbits Technology, Riverside, USA, sheng@deepbits.com; Heng Yin, University of California at Riverside, Riverside, USA, heng@cs.ucr.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2994-970X/2025/7-ARTISSTA017
https://doi.org/10.1145/3728886

Obfuscation techniques are mainly divided into data obfuscation, static code rewriting, and dynamic code rewriting [45]. Obfuscation detection for binaries mainly targets the latter two, especially static code rewriting. Security researchers initially used certain statistical features, such as entropy [26] and n-grams [18], to detect code obfuscation. However, these approaches often only work well against specific obfuscation techniques. Some studies have also attempted to use machine learning techniques [5, 11, 42, 49, 51], such as Naïve Bayes (NB), k-Nearest Neighbor (KNN), and Decision Tree (DT), to detect obfuscated code.

In recent years, various deep learning models have been widely applied to the field of binary static analysis at an astonishing speed and have quickly achieved state-of-the-art performance in various tasks [56]. As one of the most important tasks at the forefront of the reverse engineering workflow, obfuscation detection has also greatly benefited from the application of deep learning. Researchers have attempted to use Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN) to encode assembly code [50, 60], along with word2vec [30, 31] word embedding models, achieving better performance compared to traditional machine learning methods.

Many learning-based approaches to obfuscation detection frame the problem as a supervised classification task, relying on known obfuscation methods for training. While this allows these models to perform well on familiar obfuscation techniques, it limits their ability to generalize to novel or proprietary methods. Commercial software vendors and malware authors, seeking to protect their binaries from reverse engineering, often employ non-public, custom obfuscators. This lack of sample diversity in supervised learning can introduce data bias, causing models to overlook features that, while insignificant in the training set, may be critical for detecting previously unseen obfuscation techniques. Consequently, existing learning-based obfuscation detection methods may face similar limitations, raising concerns about their generalizability despite strong performance on standard evaluations. Indeed, our evaluation in §4.6 shows that supervised learning methods have very low detection rates (~ 0.04) for realworld malware samples, which are often protected by custom obfuscators.

In this work, we address the challenge of detecting code obfuscation from the perspective of assembly language modeling. Although both regular and obfuscated binary code "speak" the same assembly language (adhering to its syntax and grammar), how they speak differs significantly. We believe that the assembly language produced by regular binary code is more straightforward, concise, and comprehensible, as it originates from source code written by human developers following sound software engineering practices and is compiled to maximize efficiency. In contrast, obfuscated binary code tends to "speak" assembly in a more convoluted manner, deliberately designed to obscure its logic and hinder analysis by human experts and reverse engineering tools.

Therefore, we propose to train an assembly language model, which can capture how regular binary code "speaks" the assembly language. After training, this assembly language model can detect obfuscated binary code, because its style significantly deviates from that of the regular binary code in training. Specifically, we train a Transformer-based language model using the causal language modeling (CLM) task on a large corpus of regular binary code produced by different compilers and compiler options. Consequently, this model captures the linguistic style of regular binary code. We then detect if a given binary code is obfuscated by measuring how accurate this model predicts the next token in its assembly code sequence. A common metric for this prediction is called "perplexity". So when the perplexity of a given binary code exceeds a predetermined threshold, this input binary is deemed obfuscated. Evidently, the proposed obfuscation detector is a zero-shot detector and is capable of detecting obfuscated code produced by custom and previously-unseen obfuscators, because it is only trained on regular/unobfuscated code.

To further separate obfuscated code apart from regular code, we propose two new metrics: Error Perplexity (EP) and Consecutive Error Perplexity (CEP). The error perplexity metric sets focus

on error predictions only, whereas consecutive error perplexity further stresses on a sequence of mis-predicted tokens. Our evaluation shows that these new metrics indeed improve the detection accuracy.

To evaluate the efficacy of the proposed approach, we implement a prototype called ALMOND [1]. Specifically, we train a GPT-1.0 model on unobfuscated assembly code with a large dataset with 5,200 ELF binaries and 3,000 PE binaries over a wide variety of compilers and configurations, using the causal language modeling (CLM) task. Using our newly proposed CEP metric, ALMOND has an accuracy of 96.3% on binaries with unseen obfuscation methods which is much higher than machine learning and deep learning approaches, and is also superior to the fine-tuned language model. In real-world evaluation, ALMOND significantly outperforms the supervised fine-tuned language model ALMOND-S, demonstrating its effectiveness with a 0.869 AUC score which significantly outperforms fine-tuned language models. Our evaluations demonstrate that, although supervised learning-based models can achieve excellent results on experimental datasets, in complex and unknown real-world environments, an unsupervised, 0-shot model like ALMOND proves to be more reliable.

2 Background and Motivation

2.1 Problem Scope

This paper focuses on the static detection of obfuscated binaries. Code obfuscation, a common protection technique, transforms code to make it more difficult to understand, analyze, or reverse-engineer without altering its functionality [3, 57]. Obfuscation techniques transform the existing code and introduce redundant or junk code to achieve these goals. Broadly, there are three types of obfuscation techniques: static code rewriting and dynamic code rewriting [45].

Static code rewriting is a type of obfuscation that modifies the syntax or control flow structures of code without altering its semantics. Common techniques include control flow obfuscation, string encryption, and code flattening. Strictly speaking, data obfuscation is also a type of static rewriting, but the former primarily focuses on data, while static code rewriting focuses on code. These two categories are also the main focus of this study.

Dynamic Code Rewriting tries to obfuscate the actual execution of the code while still achieving the intended functionality. By doing so, dynamic obfuscation makes it particularly difficult for debuggers or static analysis tools to analyze or trace execution paths. Packers and virtual machine-based obfuscations belong to this category. Dynamic obfuscation techniques are generally difficult for static analysis tools, such as disassemblers, to analyze. This is because the obfuscated code is often only decoded or fully revealed during runtime. Static tools lack the ability to capture the program's runtime behavior, making it hard to reconstruct the original code from a static snapshot. As a result, techniques like dynamic code rewriting are not the primary focus of this paper, as the detection and analysis of these methods require more advanced dynamic analysis approaches rather than static inspection. However, most binaries that use dynamic obfuscation also employ static obfuscation methods to protect their remaining logic. Therefore, static obfuscation detection tools may still be able to detect samples using such techniques.

2.2 Obfuscators

An obfuscator is a tool that applies the aforementioned techniques to obfuscate source code or binaries. Typically, an obfuscator offers various obfuscation techniques, allowing developers to use one or combine multiple techniques as needed. Obfuscators can be categorized into three types based on their target stage [27]: Source Code Obfuscation, Bytecode Obfuscation (Compilation

[1] ALMOND stands for **A**ssembly **L**anguage **M**odel for **O**bfuscatio**N** **D**etection.

Stage), and Binary Obfuscation. Source code obfuscators, such as Tigress, generate obfuscated source code. These tools can obfuscate not only static languages like C/C++ but also dynamic languages like JavaScript. This paper will focus exclusively on obfuscation detection techniques for C/C++ code. Bytecode obfuscation, also known as compilation-time obfuscation, involves obfuscating intermediate code, such as LLVM IR, during the source code compilation process. For example, OLLVM [17] performs obfuscation at this stage. On the other hand, binary obfuscation tools, such as Alcatraz[2] apply obfuscation methods directly by rewriting the binaries. Since this type of tool modifies the binary directly, the available techniques are relatively limited, and it often requires support from debug symbols. However, binary-based obfuscators can fill the binary section with meaningless data or code to increase the cost of reverse engineering.

2.3 Existing Obfuscation Detection Techniques

Obfuscation detection methods can generally be divided into three categories: rule-based approaches, machine learning-based approaches, and deep learning-based approaches. We will discuss each of these methods in detail.

2.3.1 Statistical approaches. As an essential step in obfuscation detection, early methods rely on predefined rules or heuristics to identify patterns or anomalies in the code. Common techniques include the analysis of statistical properties such as entropy, control flow graphs, or n-gram models. To distinguish whether a binary has been packed or encrypted, Lyda et. al. [26] attempted to use entropy as a statistical metric for obfuscation detection. The assumption is that obfuscated binaries exhibit higher entropy than unobfuscated ones due to the randomness introduced by obfuscation techniques. Kanzaki et al. [18] proposed a new metric called Code Artificiality to determine whether the target code has been obfuscated, which is based on an n-gram model. The intuition is that normal code exhibits predictable patterns of n-grams, while obfuscated code disrupts these patterns. Statistical-based methods are straightforward but often limited in their ability to generalize across different obfuscation techniques.

2.3.2 Machine learning based approaches. In subsequent research, researchers attempted to advance the field by using supervised machine learning methods to classify the specific obfuscation methods applied to binaries [5, 42, 49, 51]. The commonality among these methods is that they treat the obfuscation detection task as a pattern recognition problem, employing machine learning techniques such as Naive Bayes (NB), k-Nearest Neighbor (KNN), Decision Tree (DT), and Random Forest (RF) for supervised training, while introducing innovations in the pre-processing step, specifically in how features were extracted. For instance, Salem et. al. [42] treated disassembly code as text and attempted to use Term Frequency-Inverse Document Frequency (TF-IDF) to extract features. This process generated a feature vector for each program, consisting of the TF-IDF values of the top 128 terms encountered across all disassembly files. This 128-dimensional feature vector is then used as input for training and inference with Naive Bayes (NB) and Decision Tree (DT) models. Tofighi-Shirazi et al. [51] chose to apply static symbolic execution to retrieve the semantic representation of the disassembly code. For feature extraction, they used the Bag of Words (BoW) [28] approach to extract features from the semantic-based raw data for machine learning models. Greco et al. [11] employed 19 handcrafted features to train machine learning models. By studying the performance of these different features across various obfuscation methods, they aimed to gain a comprehensive understanding of how obfuscation methods affect the properties of target binaries. Last but not least, on the Android platform, AndrODet [32] uses machine learning models to detect three common

[2]https://github.com/weak1337/Alcatraz

types of obfuscation techniques in Android applications: identifier renaming, string encryption, and control flow obfuscation.

2.3.3 Deep learning based approaches. With the increasing application of neural networks and deep learning in the security domain, several works have emerged that train neural networks to classify obfuscation methods. For instance, Bacci et al. [2] proposed an LSTM-RNN-based approach to detect seven different Android obfuscation techniques. Zhao et al. [60] designed a composite neural network model. They used a CNN to capture local characteristics and an LSTM to identify the instruction sequence, thereby fully capturing the contextual semantic information of the entire target program. Tian et al. [50] took the research a step further by proposing the Reduced Shortest Path Extraction algorithm, which better samples instruction sequences as input for the neural network. They used a network called BiGRU-CNN for classification, where a GRU is employed to extract features from each reduced shortest path, and a CNN is used for aggregation. Compared to traditional machine learning methods, deep learning approaches mostly utilize learned embeddings rather than manually designed feature vectors. These embeddings not only enhance flexibility and learning efficiency but also improve overall performance.

2.4 Challenges

Although the application of deep learning has not only improved the accuracy of obfuscation detection, but learning-based embeddings have also brought flexibility and learning efficiency, the fundamental issue of supervised learning models—generalizability—remains unsolved. Due to the limitations of training data and labels, we can only base our samples and annotations on the data collected. However, in the context of obfuscation detection, the presence of non-public obfuscation tools means that obfuscation detectors must deal with numerous samples obfuscated by unknown methods. Additionally, there is a significant disparity in both the difficulty of obtaining and the number of unobfuscated samples compared to obfuscated samples, leading to dataset imbalance. This poses a major challenge for the training of supervised models.

3 Design

To meet the challenges summarized in §2, we propose ALMOND, a novel zero-shot approach for detecting code obfuscation in binary executables, which does not rely on labeled training data or supervised learning. Figure 1 illustrates the pipeline of ALMOND. We start by training a language model using unobfuscated assembly code. Once training is complete, we can directly use the model for anomaly detection on obfuscated code. We predict the input tokens using the language model's pre-training task, evaluate the prediction results using metrics like error-perplexity, and classify them based on a set threshold. In the following sections, we will provide a detailed explanation of the design for each step. In §3.1 and §3.2, we will first introduce the preprocessing and pre-training processes. In §3.3, we will focus on how we achieve zero-shot detection by reusing the pre-training task and utilizing the newly proposed error-perplexity metric and the consecutive error-prediction penalty operator.

3.1 Pre-processing

In the pre-processing stage, we disassemble the binary and tokenize the assembly code. Natural language models require a tokenizer to convert raw text into numerical vectors, and tokenization methods generally fall into two categories: word-based and subword-based. As the name implies, word-based tokenization treats each word as a separate token, using spaces as delimiters along with some auxiliary rules. While this approach is simple, it results in a large vocabulary size and

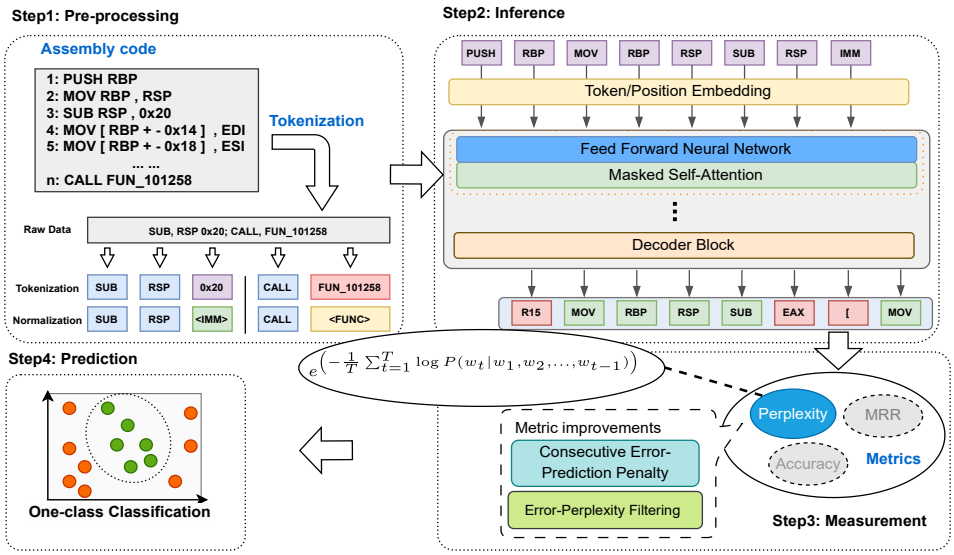


Fig. 1. The Overview of ALMOND

introduces the issue of out-of-vocabulary (OOV) words. For instance, “dog” and “dogs” would be considered entirely different tokens in a word-based tokenizer.

To address the OOV problem, modern NLP models typically use subword-based tokenizers, such as Byte Pair Encoding (BPE) [10] or WordPiece [55]. BPE iteratively merges the most frequent pairs of bytes or characters until the target vocabulary size is reached. Similarly, WordPiece constructs subwords iteratively by selecting token sequences that maximize the likelihood of the text, based on subword frequency data learned during training.

However, assembly language differs significantly from natural languages in terms of structure, syntax, and vocabulary. Assembly code has a more rigid structure and a much smaller vocabulary. For example, in x86-64 assembly, there are only around 1,000 unique mnemonics and 100 registers and symbols. As a result, word-based tokenizers do not encounter the same challenges as they do with natural languages. In this context, subword-based tokenizers offer little advantage. On the contrary, word-based tokenizers are more efficient due to their simplicity and lack of training requirements. Consequently, previous research [7, 23, 52] on assembly language models has commonly adopted the approach of separating opcodes and operands based on spaces.

The application of word-based tokenization to assembly code is not without challenges. Assembly code contains many immediate values and addresses, which can still lead to significant out-of-vocabulary (OOV) issues. Moreover, these tokens vary across different binaries, even when compiled from the same source code, due to variations in compilers and platforms. Immediate values and addresses will differ accordingly. Training a model to predict these specific values reduces its accuracy on non-obfuscated code and increases perplexity, thereby impairing the model’s ability to detect obfuscated code. This issue is present for both subword-based and word-based tokenizers.

Hence, we implemented token normalization [7, 23] to solve our OOV issue. Specifically, as shown in Figure 1, we replace immediate numbers and string tokens within instructions with special tokens. This allows the model to focus on the underlying semantics without being influenced

by specific numerical or address information, which are often subject to configuration changes. These normalized tokens make it easier for the model to learn and make accurate predictions.

3.2 Architecture

For our language model architecture, we choose to utilize state-of-the-art Transformer-based models. There are three main types of popular Transformer architectures: pure Encoder models (e.g., BERT [6]), also known as auto-encoding Transformers; pure Decoder models (e.g., GPT [38]), also known as auto-regressive Transformers; and Encoder-Decoder models [53], which combine elements of both.

Encoder-only models, such as BERT, utilize a bi-directional attention mechanism and are trained through Masked Language Modeling (MLM). However, this approach is not well-suited for solving our problem. The obfuscation detection task is more akin to a stylometric analysis problem, where the goal is to differentiate between the language styles of typical compilers and obfuscators. BERT, being trained on MLM tasks, focuses primarily on predicting masked tokens by leveraging the full context. As a result, it emphasizes semantic and syntactic understanding, with little sensitivity to variations in language style. This makes BERT less effective for obfuscation detection, as the model is likely to predict masked tokens accurately, regardless of whether the code has been obfuscated, as long as it understands the syntax and semantics of the input context.

In contrast, pure Decoder models like GPT use only the Decoder module of the Transformer architecture. At each step, the attention layer can access only the preceding words in the sequence, enabling the model to iteratively predict subsequent words based on the context already generated. This approach is known as Causal Language Modeling (CLM). When predicting the next tokens, the model must consider both fine-grained syntax and semantics, as well as generate sequences that match the style of the preceding text. As a result, if the GPT model has been pre-trained predominantly on unobfuscated code, it will face greater difficulty in predicting sequences for obfuscated code, which exhibits a distinct language style.

We conducted an experiment to confirm this hypothesis. Table 1 presents the accuracy of GPT's CLM task and BERT's MLM task on both obfuscated and unobfuscated code (on validation Dataset during pre-training). The results show that for the MLM task, the top-1 prediction accuracy and perplexity are similar for both types of sequences. This indicates that BERT struggles to distinguish between the two styles. Consequently, GPT and the CLM task are more appropriate design choices.

Table 1. Accuracy(Top-1) and Perplexity on BERT and GPT

Model	Obfuscated	Unobfuscated
BERT(MLM) Accuracy	0.877	0.895
GPT-1.0(CLM) Accuracy	0.725	0.856
BERT(MLM) perplexity	2.728	2.014
GPT-1.0(CLM) perplexity	4.045	2.225

We only use unobfuscated assembly code to train the GPT model. As previously mentioned, this allows our GPT model to learn only the language style of unobfuscated code, which will be used for subsequent obfuscation detection. We train the GPT model using a causal language modeling (CLM) task. More specifically, Transformer architecture is used to model the conditional probabilities $P(w_t \mid w_1, w_2, \ldots, w_{t-1})$ The model is trained to predict the next word in a sequence, given the previous words. The training objective is

$$\text{Loss} = -\sum_{t=1}^{T} \log P(w_t \mid w_1, w_2, \ldots, w_{t-1}) \tag{1}$$

3.3 0-Shot Obfuscation Detection

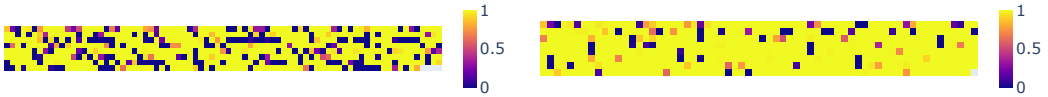
After training, the pre-training task will be reused to perform obfuscation detection. When a query code snippet is fed into the GPT model, it will make predictions from w_2 to w_n if the input length is n. Although obfuscated code functions the same as unobfuscated code, obfuscated instruction sequences create significant logical differences. For a language model trained on unobfuscated code, predicting the logic of obfuscated code becomes challenging. Therefore, in theory, we can evaluate the model's predictions using various metrics designed to assess language model predictions, and then classify the code by setting a threshold. A common example of such a metric is perplexity. For a particular token in a sequence, perplexity is calculated as:

$$\text{Perplexity}(w_t) = \exp\left(-\log P(w_t \mid w_1, w_2, \ldots, w_{t-1})\right) \tag{2}$$

For a sequence, perplexity is calculated as:

$$\text{Perplexity}(w_1, w_2, \ldots, w_T) = \exp\left(-\frac{1}{T}\sum_{t=1}^{T}\log P(w_t \mid w_1, w_2, \ldots, w_{t-1})\right) \tag{3}$$

If the perplexity exceeds the threshold, it indicates poor prediction results, leading us to classify the input sample as obfuscated code.



(a) Probability of an obfuscated binary (b) Probability of a regular binary

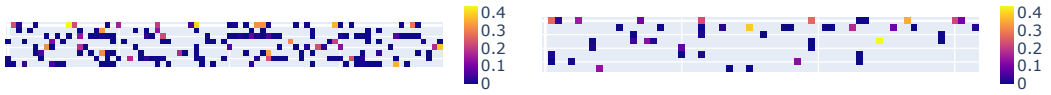
Fig. 2. Comparison of probability between obfuscated and regular binaries

3.4 Further improvement on Obfuscation Detection

Table 1 demonstrates that the perplexity predicted by the GPT model shows a marked difference between obfuscated and unobfuscated code, indicating that perplexity is a suitable metric for zero-shot obfuscation detection. To achieve even higher detection accuracy and robustness, we would like to identify potential areas for improvement. Figure 2 displays the prediction probability for ground truth tokens in an obfuscated code snippet, where each square represents the GPT model's probability of predicting the correct token at a specific position. Lighter colors indicate higher probabilities, while darker colors reflect lower probabilities and may indicate incorrect predictions. Comparing obfuscated code in Figure 2a and unobfuscated code in Figure 2b, some notable differences and patterns emerge.

First, it is evident that for both obfuscated and unobfuscated code, the prediction probability for most tokens is quite high, as supported by the data in Table 1. Both obfuscated and unobfuscated code achieves over 70% accuracy. Our further tests reveal that the perplexity of these correctly predicted tokens is very similar, as shown in Table 2. Therefore, we can conclude that these correctly predicted tokens do not significantly contribute to obfuscation detection. However, the small subset of incorrectly predicted tokens plays a crucial role, as low probabilities result in high perplexity.

Figure 3 presents a heatmap after masking all the correct predictions. We can see that, although the number of dark squares in the obfuscated code is higher than in the unobfuscated code, the unobfuscated code also contains many dark squares. However, the dark squares in the obfuscated



(a) Probability of an obfuscated binary (b) Probability of a regular binary

Fig. 3. Comparison of probability with mispredictions only

code exhibit a clear consecutiveness, while those in the unobfuscated code are more scattered and discrete(Only horizontally connected tokens represent consecutive tokens.).

3.4.1 Error-perplexity. Based on the previous observation, we propose error-perplexity as the metric for classification. Instead of using the perplexity of all tokens, we only consider the perplexity of incorrectly predicted tokens as the evaluation factor. As mentioned earlier, in obfuscated code, many tokens can still be predicted by the GPT model, and for these tokens, the perplexity will be low regardless of whether the code is obfuscated, introducing noise. However, for tokens that the GPT model predicts incorrectly, obfuscated and unobfuscated codes fall into different scenarios. For non-obfuscated code, incorrect predictions for a token are often due to the presence of multiple possibilities within normal logic. For example, after a test instruction, various jump instructions may reasonably follow, leading to potential errors in prediction. As a result, the ground truth token is typically among these possible tokens, leading to a relatively low perplexity value. On the other hand, incorrect predictions are more frequent in obfuscated code than in non-obfuscated code. These errors often arise because the language model cannot predict the obfuscated code's unique logic based on the previous tokens. In such cases, the predictions tend to be more random, and the ground truth token's probability is very low, resulting in a significantly higher perplexity value. Error-perplexity uses Equation 4 as follows.

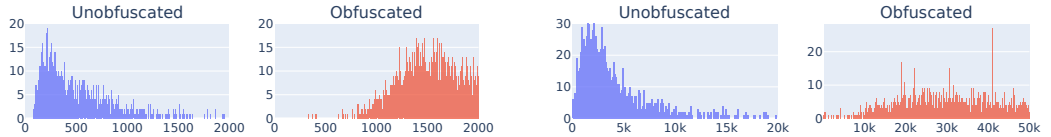
Table 2. Perplexity on correct predictions

Code	Mean	Max	Min
Regular	1.12	4.00	1.00
Obfuscated	1.28	4.14	1.00

$$\text{Error-Perplexity}(w_1, w_2, \ldots, w_T) = \exp\left(-\frac{1}{|M|}\sum_{t \in M} \log P(w_t \mid w_1, w_2, \ldots, w_{t-1})\right) \tag{4}$$

Where:

- M is the set of indices where the model made an incorrect prediction.
- $|M|$ is the size of the set M, i.e., the number of mispredicted tokens.



(a) Distributions of error-perplexity (b) Distributions of error-perplexity with CEP

Fig. 4. Comparison of Distributions of Error-Perplexity with and without CEP

We collected the distribution of error-perplexity for both unobfuscated and obfuscated code, which can be found in Figure 4a. It can be observed that the obfuscated code is primarily distributed in the region above 1000, while the unobfuscated code is concentrated in the range between 1 and 1000.

3.4.2 Consecutive Error Perplexity. On top of error-perplexity, we introduced a new metric called Consecutive Error prediction (CEP). Our investigation into language model predictions highlights two key scenarios where prediction errors arise. First, the semantics are correct, but the model faces multiple valid choices. In this case, the model usually predicts the opcode correctly and can often predict the operands as well. Even if it fails to predict the operands, the perplexity remains relatively low. Second, when an obfuscator rewrites a sequence of instructions rarely seen in regular binaries, the language model tends to make errors in both the opcodes and operands, and sometimes even in subsequent instructions. As a result, the occurrence of consecutive prediction errors is significantly higher in obfuscated code than in regular code. In Table 3, we present the average number of consecutive token prediction errors for both obfuscated and unobfuscated code (This means that a single token prediction error has a length of 1, two consecutive token prediction errors have a length of 2, and so on, with the average being taken.).

The general idea of consecutive error perplexity is to treat a sequence of consecutive mispredicted tokens as one unit. The probability for mispredicting one sequence of tokens can be calculated as a joint probability of mispredicting individual tokens in the sequence. Therefore, CEP can be defined as follows:

Table 3. Avg. length of error predictions

Code	Regular	Obfuscated
Avg. length of error predictions	1.519	2.335

$$\mathrm{CEP}(w_1, w_2, \ldots, w_T) = \exp\left(-\frac{1}{|\mathcal{S}|}\sum_{S_i \in \mathcal{S}} \log \prod_{j=1}^{|S_i|} P(w_{t_j} \mid w_1, w_2, \ldots, w_{t_j-1})\right) \tag{5}$$

Where:

- $\mathcal{S}$ is the set of sequences of consecutive mispredicted tokens.
- $S_i = (w_{t_1}, w_{t_2}, \ldots, w_{|S_i|})$
- $|\mathcal{S}|$ is the number of sequences in $\mathcal{S}$.

Using the logarithmic property $\log \prod = \sum \log$, the equation becomes:

$$\mathrm{CEP}(w_1, w_2, \ldots, w_T) = \exp\left(-\frac{1}{|\mathcal{S}|}\sum_{S_i \in \mathcal{S}} \sum_{j=1}^{|S_i|} \log P(w_{t_j} \mid w_1, w_2, \ldots, w_{t_j-1})\right) \tag{6}$$

So this equation effectively sums up the logarithm of probabilities for all tokens in all mispredicted sequences. Therefore, it can be rewritten as follows:

$$\mathrm{CEP}(w_1, w_2, \ldots, w_T) = \exp\left(-\frac{1}{|\mathcal{S}|}\sum_{t \in \mathcal{M}} \log P(w_t \mid w_1, w_2, \ldots, w_{t-1})\right) \tag{7}$$

Compared to Equation 4 for Error Perplexity, Consecutive Error Perplexity essentially replaces $|\mathcal{M}|$ with $|\mathcal{S}|$. With a fixed number of incorrect predictions, if more mispredicted tokens are next to each other, the number of error sequences $|\mathcal{S}|$ decreases, leading to a higher CEP.

4 Evaluation

In this evaluation, we aim to answer the following research questions:

(1) **RQ1:** How does ALMOND's performance compare to that of a supervised fine-tuned classifier when applied to known obfuscation methods?
(2) **RQ2:** How does ALMOND perform compared to a supervised fine-tuned classifier on previously unseen obfuscation methods?
(3) **RQ3:** How does ALMOND perform under different configurations?
(4) **RQ4:** How does ALMOND perform on real-world cases?

4.1 Implementation

We employed the GPT-1.0 as our model architecture, which is considered small by contemporary standards. It consists of 12 transformer layers, each with 12 heads. It has an output dimension of 768 and an intermediate layer dimension of 3072. We implemented the GPT model using Hugging Face's framework and conducted pre-training on a server with a single A100 40GB GPU.

4.2 Dataset Collection

Obfuscators. We collect four obfuscators for evaluation: OLLVM [17][3], Hikari [4], Tigress [5], and Alcatraz [6]. OLLVM, a modification of LLVM, integrates obfuscation into the compilation process and provides three main obfuscation algorithms: Instructions Substitution, Control Flow Flattening, and Bogus Control Flow. Hikari builds upon OLLVM, offering five additional obfuscation methods: AntiClassDump, FunctionCallObfuscate, FunctionWrapper, IndirectBranching, and StringEncryption. Tigress, in contrast, is a source-to-source transformer designed for the C language. Unlike OLLVM and Hikari, which operate during compilation, Tigress takes a C source program as input and outputs an obfuscated C program. For this evaluation, we selected the AddOpaque, Split, Merge, and Flatten obfuscation techniques from Tigress to obfuscate the source code and then compiled it into binary form.

Table 4. Obfuscators and Transformation Methods

Obfuscators	Transformations
OLLVM	Instruction Substitution, Bogus Control Flow Control Flow Flattening
Hikari	Anti-Class Dump, Function Wrapper Function Call Obfuscate Indirect Branching, String Encryption
Tigress	Add Opaque, Flatten Functions Split Fucntions, Merge Functions
Alcatraz	Obfuscation of Immediate Moves, Control Flow Flattening, ADD Mutation, Lea obfuscation

It is important to note that we used the O0 optimization level during compilation for both OLLVM and Tigress. For source-to-source obfuscators like Tigress, subsequent compiler optimizations could remove or reduce the effectiveness of the obfuscation techniques. Thus, using the O0 optimization level ensures that the original obfuscation algorithms are preserved as much as possible. Table 4 summarizes the obfuscators and the respective transformations used in our training and testing datasets.

[3]https://github.com/obfuscator-llvm/obfuscator
[4]https://github.com/HikariObfuscator/Hikari
[5]https://tigress.wtf/index.html
[6]https://github.com/weak1337/Alcatraz

Alcatraz represents obfuscators that directly modify binaries. This tool works on x64 PE binaries, which is the only platform supported by Alcatraz. It provides powerful obfuscation features including obfuscation of immediate moves, control flow flattening, ADD mutation, and LEA obfuscation. Since it targets PE binaries and is not derived from OLLVM, its implementation differs significantly from OLLVM and Hikari, which were used in pre-training. Alcatraz also includes many dynamic encryption features such as entry point obfuscation and anti-disassembly. However, since these two obfuscation techniques are outside the scope of our research, we modified the Alcatraz source code and recompiled it to disable these methods.

Pre-training Data. In evaluation of StateFormer [35], the authors collected 33 open-source projects in their latest versions, including well-known and large projects such as OpenSSL, ImageMagic, and Coreutils. These projects were compiled for four instruction set architectures including x86, x64, MIPS, and ARM, each with four different optimizations using GCC-7.5. We used the x64 portion of the StateFormer training set. Additionally, the Stateformer dataset includes obfuscated code generated using Hikari and OLLVM, we did not use these obfuscated binaries to pre-train ALMOND.

It is worth noting that our language-based baseline models also used a portion of this dataset set for fine-tuning. To further enhance data diversity, we additionally collected 3,000 PE binaries from Windows systems for pre-training, with the goal of increasing the variety of binaries across different platforms and compilers.

Testing Data. For testing, we selected binaries that were entirely distinct from those used in pre-training. Specifically, we used the POJ-104 dataset, which originates from a pedagogical programming open judge (OJ) system[33] designed to automate the evaluation of submitted source code for specific problems. For our test set, we compiled over 14,000 POJ-104 binaries, encompassing more than 25,000 functions.

Real-world dataset. For evaluations on real-world cases, we collect 5000 Real-world binaries from Linux Distributions and Windows PE files. Similarly, we selected 5000 binaries labeled as malware from VirusTotal feedings. These binaries were compiled for different platforms using various compilers, and the malware likely employed various obfuscators or packers, resulting in significant diversity.

4.3 RQ1: How does ALMOND perform on known obfuscation methods?

In this experiment, we investigate the performance gap between ALMOND and supervised classifiers when dealing with known obfuscation methods. To provide a basis for comparison, we established the following baselines. First, we replicated the method proposed by Salem et al. [43] as the baseline for traditional machine learning. This method encodes assembly code using TF-IDF [48] and utilizes Multinomial Naïve Bayes (MNB) as the classifier. MNB is a variant of the Naïve Bayes algorithm specifically designed for classification tasks involving discrete features and is commonly used in text classification problems [19]. At the same time, it can also be effectively combined with TF-IDF. We refer to this approach as Naïve Bayes.

Second, we implemented Tian et. al.'s OBOB [50] as the deep learning baseline. In this approach, code sequences are sampled from the control flow graph using a shortest path algorithm. A Bi-directional GRU network is then used to encode the sequences, followed by a CNN for further dimensionality reduction. Finally, classification is performed using a softmax classifier. For clarity, we refer to this model as BiGRU-CNN.

Finally, we utilized the pre-trained ALMOND model, attached it to a classifier with a Multi-Layer Perceptron (MLP) network [41], and conducted fine-tuning. This model is referred to as

ALMOND-S, in which S stands for **S**upervised Learning. For ALMOND, we determined the optimal threshold using a set of labeled data from the Stateformer dataset (which contains unobfuscated code and obfuscated code with Hikari and OLLVM), and we applied a fixed threshold value for the evaluation of RQ1 and RQ2. However, different training sets may result in different thresholds. Due to ALMOND's flexibility, the threshold can be adjusted at any time based on the data. It is worth noting that there is no overlap between the obfuscated binaries used during training and fine-tuning and those used in the test set during evaluation.

Table 5 presents the results of different models on a test set containing known obfuscation methods. We observe that, after 18 hours of fine-tuning, ALMOND-S is the best performer, achieving an accuracy of 0.988 and an F1-score of 0.985. Notably, without any fine-tuning or supervision, ALMOND achieves an accuracy of 0.962 and an F1-score of 0.963, demonstrating that in a zero-shot setting, ALMOND still delivers performance comparable to fine-tuned models with the same architecture.

Table 5. Performance on known obfuscation methods. A: Accuracy, P: Precision, R: Recall

Model	A	P	R	F1
Naïve Bayes	0.942	0.911	0.975	0.958
BiGRU-CNN	0.933	0.936	0.932	0.934
ALMOND-S	**0.988**	**0.988**	**0.984**	**0.985**
ALMOND	0.962	0.953	0.975	0.963

Although Naïve Bayes and BiGRU-CNN perform slightly worse than ALMOND, they still achieve accuracy and F1-scores above 0.93, indicating that both supervised learning and zero-shot learning approaches can effectively detect known obfuscation methods.

4.4 RQ2: How does ALMOND perform on previously unseen obfuscation methods?

This experiment examines the performance gap between ALMOND and supervised classifiers when dealing with unseen obfuscation methods. To ensure precise control over the unseen dataset, we trained Naïve Bayes, BiGRU-CNN, and fine-tuned the language model using only the binaries obfuscated with OLLVM. For testing, we used binaries obfuscated with Tigress and Alcatraz, ensuring that the obfuscation methods employed in Tigress were not present in the OLLVM-based training data.

Table 6. Performance on unseen obfuscation methods.

Models	Tigress				Alcatraz			
	A	P	R	F1	A	P	R	F1
Naïve Bayes	0.563	0.943	0.103	0.185	0.522	0.945	0.106	0.190
BiGRU-CNN	0.862	0.866	0.862	0.863	0.566	0.884	0.247	0.247
ALMOND-S	**0.966**	**0.966**	0.957	**0.961**	0.622	0.793	0.452	0.576
ALMOND	0.963	0.952	**0.969**	**0.961**	**0.967**	**0.964**	**0.958**	**0.961**

The experimental results are presented in Table 6. BiGRU-CNN and ALMOND-S demonstrated relatively better generalizability on Tigress binaries. However, there was a significant performance drop on Alcatraz binaries. ALMOND-S achieved the best results among the baselines, showcasing the advantage of language models in semantic modeling.

In contrast, ALMOND remained unaffected when applied to previously unseen obfuscation methods and significantly outperformed all other baselines. For binary-based obfuscation methods with completely different implementations, ALMOND showed higher perplexity and reduced token prediction accuracy. In comparison, supervised methods often classified these obfuscated binaries as benign, resulting in a high false negative rate. This also confirms our earlier assertion that, in an

environment with unknown obfuscation methods, the generalizability of unsupervised training is superior to that of supervised training.

4.5 RQ3: How does ALMOND perform under different configurations?

In this experiment, we examined how different metrics and model sizes affect the performance of ALMOND.

Table 7. Performance of Different Metrics

Method	Acc	MRR	PPL	EP	CEP
regular	0.750	0.836	9.434	570.481	4.53k
sub	0.715	0.801	22.657	2002.313	15.37k
fla	0.811	0.859	11.002	1801.224	24.28k
bcf	0.647	0.724	25.615	1816.941	22.25k
tigress	0.623	0.812	18.494	2295.242	38.80k

4.5.1 Performance of Different Metrics. Table 7 shows results for five metrics on both unobfuscated and obfuscated binaries across different obfuscation methods, highlighting numerical differences. Accuracy (Acc) and Mean Reciprocal Rank (MRR) show no significant difference between the two groups. In some cases, binaries obfuscated with control flow flattening (FLA) even achieve higher accuracy and MRR than their unobfuscated counterparts. This suggests that using accuracy or MRR as a classification metric could result in false negatives, where obfuscated binaries—particularly those obfuscated using control flow flattening—are incorrectly classified as unobfuscated. In contrast, for perplexity-based metrics (i.e., original perplexity (PPL), error perplexity (EP), and consecutive error perplexity (CEP)), we can observe wide numeric gaps between regular and obfuscated binaries.

To further evaluate how these metrics affect ALMOND's performance, we plot their ROC curves in Figure 5 and compute the Area Under the Curve (AUC) for each metric. We can observe that the original perplexity achieves reasonable performance with an AUC of 0.947, whereas error perplexity outperforms it by a big margin with an AUC of 0.978, and consecutive error perplexity further beats error perplexity with an AUC of 0.992.

In summary, this experiment demonstrates that both the proposed error-perplexity and the Consecutive Error Perplexity (CEP) significantly enhance the performance of ALMOND.

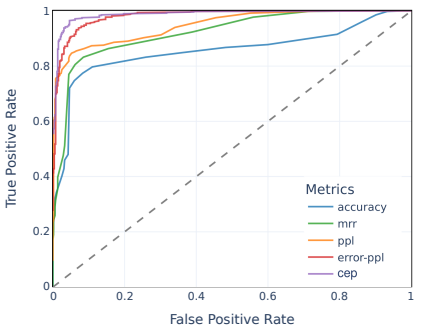


Fig. 5. ROC curves on different metrics

4.5.2 Performance of Different Model Sizes. In this subsection, we will examine the model's performance under different sizes. We adjusted the model size by varying the number of layers and self-attention heads. Including the default size, we trained a total of three different model sizes, labeled as ALMOND-small, ALMOND, and ALMOND-large. Table 8 provides detailed parameters for each of these models.

Table 8. Hyperparameters on different sized models

Models	Layers	Dimensions	Heads	Parameters
Small	6	768	4	14M
Standard	12	768	12	87M
Large	24	768	12	172M

Table 9 shows the precision, recall, F1-score, and AUC scores of ALMOND on different sizes. We observed that in the zero-shot setting, a larger model size does not always lead to better performance; instead, there is a sweet spot. As we scale from ALMOND-small to ALMOND, performance improves with the increase in layers. However, further increasing the model size results in a performance drop. This is because our model relies on its understanding of the semantics of unobfuscated code to distinguish obfuscated code. If the model is too small, it lacks the capacity to fully grasp the semantics of assembly code, leading to excessively high perplexity values on unobfuscated code. Conversely, if the model is too large, its understanding of the assembly code semantics becomes too strong, and its generalization ability too high. As a result, it predicts low perplexity for obfuscated code. Therefore, there is a sweet spot in the model size of ALMOND, where it can adequately understand the semantics of assembly code without losing its ability to distinguish obfuscated code through error-perplexity due to overgeneralization.

Table 9. Performance on different sizes

Models	Recall	Precision	F1-score	AUC
Small	0.944	0.927	0.935	0.982
Standard	0.963	0.958	0.960	0.991
Large	0.939	0.894	0.915	0.977

4.6 RQ4: How does ALMOND perform on real-world cases

This experiment evaluates the performance of ALMOND on large-scale, real-world programs, including both commercial off-the-shelf (COTS) and open-source software. In practice, binaries are typically unobfuscated, so even a low false positive rate can undermine ALMOND's effectiveness due to the high volume of binaries. Additionally, real-world binaries are compiled with various compilers and configurations and may include scientific computation libraries. These libraries often contain extensive mathematical operations that can resemble the logic used by obfuscators. As a result, such operations could be mistakenly identified as obfuscated, leading to false positives. The experiment aims to determine ALMOND's applicability in a real-world malware detection environment. The purpose of the experiment is not to suggest that we should use ALMOND alone for malware detection. Instead, it offers a new perspective on zero-day detection. This is based on a previously discussed assumption that malware will inevitably use some form of obfuscator or packer to obfuscate the source code in order to evade anti-virus detection. Hence, ALMOND can be employed to identify potential zero-day malware, serving as an initial filter. Once an alert is triggered, slower but more comprehensive program analysis tools and reverse engineering techniques can be used to investigate and confirm the nature of the suspected malware.

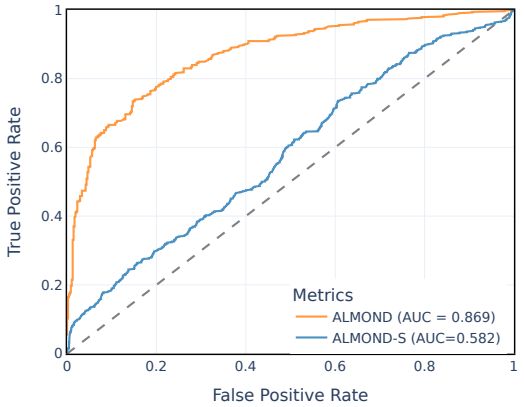


Fig. 6. ROC curves on real-world binaries

Unlike the previous experiments, this experiment requires classification at the binary level, so we adjusted the calculation method for the metrics accordingly. Instead of calculating CEP at the function level, we now compute CEP for the entire binary. Therefore, in this experiment, we plotted ROC curves using different thresholds.

Figure 6 shows ALMOND's performance on real-world binaries. We observed that ALMOND achieved an AUC of 0.869 and an F-1 score of 0.803 on this dataset, indicating that ALMOND can also achieve high accuracy in real-world malware detection tasks. Overall, as an initial filter in a comprehensive malware detection system, ALMOND is fully capable of fulfilling this role.

On the contrary, the classification accuracy of ALMOND-S dropped significantly. As a classifier utilizing a language model, ALMOND-S possesses strong language modeling and feature extraction capabilities, as demonstrated in RQ1. However, as a supervised model, ALMOND-S is unable to handle such tasks in real-world environments.

4.6.1 Real case: Packers.

Case 1 Figure 7 presents a heatmap of the perplexity with CEP on a malware sample. In the heatmap, some distinct highlighted lines can be observed. We extracted one segment for analysis. This function appears to be part of an encryption/decryption or decompression routine, containing complex operations such as multiplications, bit shifts and rotations, and bitwise comparisons along with conditional jumps. It also makes use of bitwise instructions like LZCNT, SWAP, NEG, etc. It is known to all that some packers typically decrypt or unpack compressed executable data on the fly, so we believe this function might be responsible for part of that process—transforming the packed data into executable code just before it is executed. It is worth mentioning that we also studied the implementation of encryption algorithms in OpenSSL. However, we found that the encryption algorithms used in network communications are not the same as the function in this case. Taking the `x86_aes_encrypt` function as an example, this function primarily uses a combination of MOVZX, XOR, SHR, and similar instructions, mainly performing logical and bitshift operations. However, the case function mainly involves various specialized instructions with arithmetic and bit manipulation. Since OpenSSL-related functions are included in ALMOND's pre-training dataset, the language model can also accurately predict these operations.

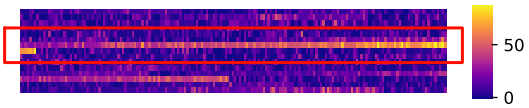


Fig. 7. The heatmap of malware A

Furthermore, we speculate that this segment was either manually crafted by the author or generated using a specialized obfuscation tool, which is completely different from the obfuscated segments observed in other experiments. Hence, ALMOND made consecutive prediction errors in this segment. Besides this, we also observed a few similar segments in the binary. For the rest of the program, ALMOND exhibited lower perplexity.

Case 2 Figure 8 presents the heatmap of another malware sample. Information from VirusTotal indicates that this program likely used some VM-based techniques to protect and encrypt its logic. However, there are still some functions that can be fully disassembled. The heatmap reveals that one part of the binary has higher perplexity compared to other sections. Upon further investigation, this function appears to be a part of a process injection routine, leveraging various Windows API functions to inject code (likely a DLL or shellcode) into another process and execute it remotely. It achieves this by creating a new process or accessing an existing one, allocating memory in that process, writing the payload to the allocated memory, and finally creating a remote thread to execute the injected payload. ALMOND exhibited clear anomalies when dealing with process injection operations, as such operations are rarely found in typical programs. This resulted in very high perplexity and consecutive prediction errors during inference.

4.7 Efficiency

In this experiment, we evaluated the efficiency of ALMOND. We found that on an A100 40G GPU, with a batch size of 32, ALMOND can achieve a throughput of 173.808 inferences or 88989 tokens per second, and with a batch size of 64, it can achieve a throughput of 220.324 inferences or 112640 tokens per second.

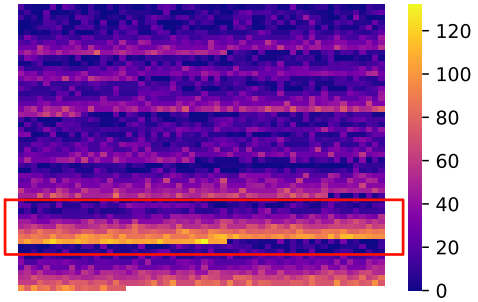


Fig. 8. The heatmap of malware B

5 Related Works

5.1 Obfuscation Detection

As discussed in §2.3, prior research on obfuscation detection primarily aimed to facilitate reverse engineering, employing rule-based, machine learning-based, and deep learning-based methods. The approach proposed in this paper builds on deep learning methods while also leveraging metrics used in rule-based detection. Previous deep learning approaches, such as those by Zhao et al. [60] and Tian et al. [50], utilized CNNs and LSTMs for supervised learning. However, with the application of language models in binary analysis, the use of language models and the pre-training, fine-tuning paradigm has become a superior solution. To our knowledge, this paper is the first to use a language model for obfuscation detection. The connections and distinctions between this work and other studies that apply language models to static binary analysis will be discussed in. After modeling assembly language with a language model, this paper introduces a novel language model-based metric, error perplexity, to detect obfuscated code. This approach is analogous to using rule-based metrics like entropy [26] or n-gram models [18] for obfuscation detection, but with a key difference: the proposed metric is designed to assess the predictions of the language model rather than directly targeting assembly code or raw bytes.

5.2 Language Model for Static Binary Analysis

In recent years, numerous studies have explored the use of language models for static binary analysis. PalmTree [23], for example, proposed using language models to generate instruction embeddings, which can be applied to various downstream tasks. Most of these studies have focused on leveraging language models by introducing specialized pre-training tasks or innovative model architectures to target specific tasks, such as similarity detection [1, 15, 24, 29, 36, 52, 58], type inference [35], function name recovery [4, 16], and value set analysis [12]. These tasks are typically accomplished through fine-tuning or by introducing special pre-training tasks.

The key distinction between this work and those studies is that our approach does not involve any additional pre-training tasks or fine-tuning. Instead, it relies solely on the default pre-training of GPT, enabling obfuscation detection to be performed in a zero-shot manner.

5.3 Zero-shot Classification and Anomaly Detection

Zero-shot classification involves predicting a class the model has never encountered during training, often requiring it to perform tasks not explicitly learned. A notable example is GPT-2, tested on downstream tasks like machine translation without fine-tuning [39]. In this context, the model classifies input text into unseen labels. Two primary approaches exist: Puri et al.[37] utilize generative capabilities of models like GPT by prompting the model with task descriptions and candidate

labels, while Zhang et al. [59] map both labels and documents into a high-dimensional space to predict labels using cosine similarity. Unlike Zhang et al. [59], ALMOND focuses on One-Class Classification, avoiding the need for label mapping.

Similarly, in anomaly detection (AD), the task is to identify data instances that deviate from the norm [40], with applications in security, such as detecting DDoS attacks or monitoring system logs [9, 13, 14, 21]. Zero-shot anomaly detection employs two approaches: one utilizes large-scale data and powerful models through meta-learning or in-context learning, as seen with Liu et al. [25] and RAGLog [34]. The second, more computationally efficient approach studies sample features and applies scoring methods for evaluation, enabling zero-shot detection in tasks like pixel-level anomaly detection [8, 22, 46]. This feature analysis method is the focus of this paper.

6 Discussion

This paper proposes a zero-shot obfuscation detection method based on a pre-trained language model, achieving results comparable to fine-tuned models with significantly less training data. However, there are limitations to this work. Firstly, the paper only explores binary classification using error-perplexity. In reality, the information predicted by the language model could be used for more detailed classification tasks, such as identifying specific obfuscation methods, all without requiring fine-tuning. We believe that building on this work, incorporating few-shot learning algorithms such as Generalized Learning Vector Quantization [44] could enable the prediction of obfuscation methods.

Secondly, this paper presents only a prototype of classification based on error-perplexity and does not thoroughly investigate combining multiple metrics to further enhance performance. However, we have already observed that combining perplexity and error-perplexity outperforms using either metric alone.

Lastly, this paper does not include experiments to thoroughly examine the potential vulnerabilities of ALMOND to evasion or adversarial attacks, which will be discussed here. First, adversarial learning methods targeting binary classifiers must ensure that the modified binary can both mislead the machine learning classifier and maintain functional integrity [47]. Consequently, these methods typically avoid altering the original assembly code and instead insert data or code between the assembly instructions to deceive the classifier. However, the classification method based on error-perplexity proposed in this paper focuses solely on the language model's incorrect predictions. As a result, for an obfuscated binary, it would be challenging to deceive ALMOND by simply inserting unobfuscated code. However, this does not imply that ALMOND is entirely immune to evasion. Attackers would need to design obfuscation methods that more closely resemble regular code to reduce the error-perplexity value, which inherently means a reduction in the effectiveness of the obfuscation itself. Therefore, we have reason to believe that ALMOND offers stronger resistance to adversarial attacks compared to other supervised-learning-based approaches.

7 Conclusion

We present ALMOND, a zero-shot obfuscation detector based on a transformer language model. We employed a metric-based classification technique along with an anomaly detection approach and proposed the error-perplexity and Continuous Error-prediction Penalty to further enhance detection capabilities. Our evaluation demonstrates that ALMOND achieves an accuracy of 96.3% on binaries with previously unseen obfuscation methods, surpassing traditional machine learning and deep learning approaches. Moreover, in a real-world malware detection task, ALMOND achieved a false negative rate of less than 0.001, while maintaining a false positive rate of just 0.269. ALMOND has proven that obfuscation detection can be achieved in a zero-shot setting solely through metric

evaluation of the language model. It has also demonstrated that, when faced with complex and unknown real-world environments, it is more reliable than supervised learning-based models.

8 Acknowledgement

We would like to thank the anonymous reviewers for their helpful and constructive comments. This work was supported by Amazon Research Award. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

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*. 361–374. https://doi.org/10.1145/3564625.3567975

[2] Alessandro Bacci, Alberto Bartoli, Fabio Martinelli, Eric Medvet, and Francesco Mercaldo. 2018. Detection of Obfuscation Techniques in Android Applications. In *Proceedings of the 13th International Conference on Availability, Reliability and Security* (Hamburg, Germany) *(ARES '18)*. Association for Computing Machinery, New York, NY, USA, Article 57, 9 pages. https://doi.org/10.1145/3230833.3232823

[3] Arini Balakrishnan and Chloe Schulze. 2005. Code obfuscation literature survey. *CS701 Construction of compilers* 19 (2005), 31.

[4] Pratyay Banerjee, Kuntal Kumar Pal, Fish Wang, and Chitta Baral. 2021. Variable name recovery in decompiled binary code using constrained masked language modeling. *arXiv preprint arXiv:2103.12801* (2021).

[5] Fabrizio Biondi, Michael A. Enescu, Thomas Given-Wilson, Axel Legay, Lamine Noureddine, and Vivek Verma. 2019. Effective, efficient, and robust packing detection and classification. *Computers & Security* 85 (2019), 436–451. https://doi.org/10.1016/j.cose.2019.05.007

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805* (2018).

[7] 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)*. IEEE, 472–489. https://doi.org/10.1109/SP.2019.00003

[8] Marius Drăgoi, Elena Burceanu, Emanuela Haller, Andrei Manolache, and Florin Brad. 2022. AnoShift: A Distribution Shift Benchmark for Unsupervised Anomaly Detection. *Neural Information Processing Systems NeurIPS, Datasets and Benchmarks Track* (2022).

[9] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 1285–1298. https://doi.org/10.1145/3133956.3134015

[10] Philip Gage. 1994. A new algorithm for data compression. *The C Users Journal* 12, 2 (1994), 23–38.

[11] Claudia Greco, Michele Ianni, Antonella Guzzo, and Giancarlo Fortino. 2024. Enabling Obfuscation Detection in Binary Software through eXplainable AI. *IEEE Transactions on Emerging Topics in Computing* (2024).

[12] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. 2019. DEEPVSA: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1787–1804. https://www.usenix.org/conference/usenixsecurity19/presentation/guo

[13] Ren-Hung Hwang, Min-Chun Peng, Chien-Wei Huang, Po-Ching Lin, and Van-Linh Nguyen. 2020. An unsupervised deep learning model for early network traffic anomaly detection. *IEEE Access* 8 (2020), 30387–30399. https://doi.org/10.1109/ACCESS.2020.2973023

[14] Félix Iglesias and Tanja Zseby. 2015. Analysis of network traffic features for anomaly detection. *Machine Learning* 101 (2015), 59–84. https://doi.org/10.1007/s10994-014-5473-9

[15] Shuai Jiang, Cai Fu, Yekui Qian, Shuai He, Jianqiang Lv, and Lansheng Han. 2022. IFAttn: Binary code similarity analysis based on interpretable features with attention. *Computers & Security* 120 (2022), 102804.

[16] 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* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1631–1645. https://doi.org/10.1145/3548606.3560612

[17] 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, May 19th, 2015*, Brecht Wyseur (Ed.). IEEE, 3–9. https://doi.org/10.1109/SPRO.2015.10

[18] Yuichiro Kanzaki, Akito Monden, and Christian Collberg. 2015. Code Artificiality: A Metric for the Code Stealth Based on an N-Gram Model. In *2015 IEEE/ACM 1st International Workshop on Software Protection*. 31–37. https://doi.org/10.1109/SPRO.2015.14

[19] Ashraf M Kibriya, Eibe Frank, Bernhard Pfahringer, and Geoffrey Holmes. 2005. Multinomial naive bayes for text categorization revisited. In *AI 2004: Advances in Artificial Intelligence: 17th Australian Joint Conference on Artificial Intelligence, Cairns, Australia, December 4-6, 2004. Proceedings 17*. Springer, 488–499.

[20] Sangjun Ko, Jusop Choi, and Hyoungshick Kim. 2017. COAT: Code obfuscation tool to evaluate the performance of code plagiarism detection tools. In *2017 International conference on software security and assurance (ICSSA)*. IEEE, 32–37. https://doi.org/10.1109/ICSSA.2017.29

[21] Van-Hoang Le and Hongyu Zhang. 2022. Log-based anomaly detection with deep learning: How far are we?. In *Proceedings of the 44th international conference on software engineering*. 1356–1367. https://doi.org/10.1145/3510003.3510155

[22] Aodong Li, Chen Qiu, Marius Kloft, Padhraic Smyth, Maja Rudolph, and Stephan Mandt. 2024. Zero-shot anomaly detection via batch normalization. *Neural Information Processing Systems NeurIPS* 36 (2024).

[23] Xuezixiang Li, Yu Qu, and Heng Yin. 2021. Palmtree: Learning an assembly language model for instruction embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 3236–3251. https://doi.org/10.1145/3460120.3484587

[24] Guangming Liu, Xin Zhou, Jianmin Pang, Feng Yue, Wenfu Liu, and Junchao Wang. 2023. Codeformer: A GNN-Nested Transformer Model for Binary Code Similarity Detection. *Electronics* 12, 7 (2023), 1722.

[25] Yilun Liu, Shimin Tao, Weibin Meng, Feiyu Yao, Xiaofeng Zhao, and Hao Yang. 2024. Logprompt: Prompt engineering towards zero-shot and interpretable log analysis. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 364–365.

[26] Robert Lyda and James Hamrock. 2007. Using Entropy Analysis to Find Encrypted and Packed Malware. *IEEE Security & Privacy* 5, 2 (2007), 40–45. https://doi.org/10.1109/MSP.2007.48

[27] Matias Madou, Bertrand Anckaert, Bruno De Bus, Koen De Bosschere, Jan Cappaert, and Bart Preneel. 2006. On the effectiveness of source code transformations for binary obfuscation. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP06)*. CSREA Press, 527–533.

[28] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, UK. http://nlp.stanford.edu/IR-book/information-retrieval-book.html

[29] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. 2019. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 309–329. https://doi.org/10.1007/978-3-030-22038-9_15

[30] Tomas Mikolov. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[31] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* 26 (2013).

[32] O. Mirzaei, J.M. de Fuentes, J. Tapiador, and L. Gonzalez-Manzano. 2019. AndrODet: An adaptive Android obfuscation detector. *Future Generation Computer Systems* 90 (2019), 240–261. https://doi.org/10.1016/j.future.2018.07.066

[33] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 30.

[34] Jonathan Pan, Wong Swee Liang, and Yuan Yidi. 2024. RAGLog: Log Anomaly Detection using Retrieval Augmented Generation. In *2024 IEEE World Forum on Public Safety Technology (WFPST)*. IEEE, 169–174. https://doi.org/10.1109/WFPST58552.2024.00034

[35] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2021. Stateformer: Fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 690–702. https://doi.org/10.1145/3468264.3468607

[36] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray. 2023. Learning Approximate Execution Semantics From Traces for Binary Function Similarity. *IEEE Transactions on Software Engineering* 49, 04 (apr 2023), 2776–2790. https://doi.org/10.1109/TSE.2022.3231621

[37] Raul Puri and Bryan Catanzaro. 2019. Zero-shot text classification with generative language models. *arXiv preprint arXiv:1912.10165* (2019).

[38] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).

[39] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[40] Lukas Ruff, Jacob R Kauffmann, Robert A Vandermeulen, Grégoire Montavon, Wojciech Samek, Marius Kloft, Thomas G Dietterich, and Klaus-Robert Müller. 2021. A unifying review of deep and shallow anomaly detection. *Proc. IEEE* 109, 5 (2021), 756–795.

[41] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *nature* 323, 6088 (1986), 533–536.

[42] Aleieldin Salem and Sebastian Banescu. 2016. Metadata recovery from obfuscated programs using machine learning. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering* (Los Angeles, California, USA) *(SSPREW '16)*. Association for Computing Machinery, New York, NY, USA, Article 1, 11 pages. https://doi.org/10.1145/3015135.3015136

[43] Aleieldin Salem and Sebastian Banescu. 2016. Metadata recovery from obfuscated programs using machine learning. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering* (Los Angeles, California, USA) *(SSPREW '16)*. Association for Computing Machinery, New York, NY, USA, Article 1, 11 pages. https://doi.org/10.1145/3015135.3015136

[44] Atsushi Sato and Keiji Yamada. 1995. Generalized learning vector quantization. *Advances in neural information processing systems* 8 (1995).

[45] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2016. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *Acm computing surveys (csur)* 49, 1 (2016), 1–37. https://doi.org/10.1145/2886012

[46] Eli Schwartz, Assaf Arbelle, Leonid Karlinsky, Sivan Harary, Florian Scheidegger, Sivan Doveh, and Raja Giryes. 2024. MAEDAY: MAE for few-and zero-shot AnomalY-Detection. *Computer Vision and Image Understanding* 241 (2024), 103958. https://doi.org/10.1016/j.cviu.2024.103958

[47] Wei Song, Xuezixiang Li, Sadia Afroz, Deepali Garg, Dmitry Kuznetsov, and Heng Yin. 2022. MAB-Malware: A Reinforcement Learning Framework for Blackbox Generation of Adversarial Malware. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security* (Nagasaki, Japan) *(ASIA CCS '22)*. Association for Computing Machinery, New York, NY, USA, 990–1003. https://doi.org/10.1145/3488932.3497768

[48] Karen Sparck Jones. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation* 28, 1 (1972), 11–21.

[49] Li Sun, Steven Versteeg, Serdar Boztaş, and Trevor Yann. 2010. Pattern Recognition Techniques for the Classification of Malware Packers. In *Information Security and Privacy*, Ron Steinfeld and Philip Hawkes (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 370–390. https://doi.org/10.1007/978-3-642-14081-5_23

[50] Zhenzhou Tian, Hengchao Mao, Yaqian Huang, Jie Tian, and Jinrui Li. 2022. Fine-Grained Obfuscation Scheme Recognition on Binary Code. In *Digital Forensics and Cyber Crime*, Pavel Gladyshev, Sanjay Goel, Joshua James, George Markowsky, and Daryl Johnson (Eds.). Springer International Publishing, Cham, 215–228. https://doi.org/10.1007/978-3-031-06365-7_13

[51] Ramtine Tofighi-Shirazi, Irina Măriuca Asăvoae, and Philippe Elbaz-Vincent. 2019. Fine-grained static detection of obfuscation transforms using ensemble-learning and semantic reasoning. In *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering* (San Juan, Puerto Rico, USA) *(SSPREW9 '19)*. Association for Computing Machinery, New York, NY, USA, Article 4, 12 pages. https://doi.org/10.1145/3371307.3371313

[52] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. jTrans: jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1–13. https://doi.org/10.1145/3533767.3534367

[53] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[54] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. 2018. A large scale investigation of obfuscation use in google play. In *Proceedings of the 34th annual computer security applications conference*. 222–235. https://doi.org/10.1145/3274694.3274726

[55] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv preprint arXiv:1609.08144* (2016). https://doi.org/10.48550/arXiv.1609.08144

[56] Hongfa Xue, Shaowen Sun, Guru Venkataramani, and Tian Lan. 2019. Machine learning-based analysis of program binaries: A comprehensive study. *IEEE Access* 7 (2019), 65889–65912. https://doi.org/10.1109/ACCESS.2019.2917668

[57] Ilsun You and Kangbin Yim. 2010. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*. IEEE, 297–300. https://doi.org/10.1109/BWCCA.2010.85

[58] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 1145–1152. https://doi.org/10.1609/aaai.v34i01.5466

[59] Jingqing Zhang, Piyawat Lertvittayakumjorn, and Yike Guo. 2019. Integrating semantic knowledge to tackle zero-shot text classification. *arXiv preprint arXiv:1903.12626* (2019).

[60] Yujie Zhao, Zhanyong Tang, Guixin Ye, Dongxu Peng, Dingyi Fang, Xiaojiang Chen, and Zheng Wang. 2020. Semantics-aware obfuscation scheme prediction for binary. *Computers & Security* 99 (2020), 102072. https://doi.org/10.1016/j.cose.2020.102072

Received 2024-10-31; accepted 2025-03-31