Identifying Obfuscated Code through Graph-Based Semantic Analysis of Binary Code

Roxane Cohen[1,2], Robin David[1], Florian Yger[3], and Fabrice Rossi[4]

[1] Quarkslab,
[2] LAMSADE, CNRS, Université Paris-Dauphine - PSL, Paris, France
[3] LITIS, INSA Rouen Normandy, Rouen, France
[4] CEREMADE, CNRS, Université Paris-Dauphine - PSL, Paris, France

Abstract. Protecting sensitive program content is a critical issue in various situations, ranging from legitimate use cases to unethical contexts. Obfuscation is one of the most used techniques to ensure such protection. Consequently, attackers must first detect and characterize obfuscation before launching any attack against it. This paper investigates the problem of function-level obfuscation detection using graph-based approaches, comparing algorithms, from elementary baselines to promising techniques like GNN (Graph Neural Networks), on different feature choices. We consider various obfuscation types and obfuscators, resulting in two complex datasets. Our findings demonstrate that GNNs need meaningful features that capture aspects of function semantics to outperform baselines. Our approach shows satisfactory results, especially in a challenging 11-class classification task and in a practical malware analysis example.

Keywords: Graphs, Graph representation learning, GNN, Obfuscation, Security

1 Introduction

Binary programs and their sensitive contents are often protected from reverse engineering through obfuscation techniques [19], which aim to obscure a program's underlying logic without altering its functionality. While reverse engineers seek to comprehensively understand program semantics [22], developers try to conceal them, at least partially. Their motivations range from legitimate concerns like intellectual property protection to less ethical practices such as hiding malicious payloads [26]. Consequently, detecting obfuscated programs is useful for program protection assessment and malware detection complementing, for instance, traditional signature-based methods. ML (Machine Learning) approaches have emerged as effective tools for this detection task [10].

In this paper, we focus on obfuscation detection at the function level, with the aim of identifying both obfuscated functions and the specific obfuscation techniques employed. Our objective is to pinpoint obfuscated functions within a binary. While this can help determine if a program is obfuscated overall [10], it has broader implications. First, as obfuscation negatively impacts program efficiency, developers tend to only obfuscate important functions, which ultimately are the ones of interest for an analyst. Second, automated attacks have been developed against specific obfuscation schemes, assuming the obfuscation is already detected and located [4, 32, 2, 24, 27]. By detecting functions obfuscated with these schemes, analysts can effectively employ the corresponding attacks. Finally, it provides an evaluation of how stealthy an obfuscation is, which is crucial for developers, as automated detection tools can provide a measure of the undetectability of obfuscation methods at a fine-grained level [15].

This problem has been studied from a ML perspective in several previous works. Many of these studies extract various features from binary programs, such as the distribution of instructions in the assembly code [23], the function complexity metrics [25] or semantic reasoning [28]. Some of the more advanced features are derived from a graph-based representation of the binary functions. Specifically, each function can be represented by its CFG (Control Flow Graph), a graph where each node is a BB (Basic Block), an atomic sequence of instructions without any branching and where a directed edge between two nodes corresponds to a candidate execution path. The cyclomatic complexity is one example of a feature computed from the CFG.

While extracting domain-specific features from the CFG has been proved to be effective in some cases [10], leveraging the full structural information of the graph can enhance classification performance in certain applications. This can be achieved using kernel methods [17] or GNN (Graph Neural Networks). GNNs are under very active development since their introduction [9] and have shown promise in outperforming feature-based approaches in specific scenarios [5].

Function-level obfuscation detection and classification were investigated using a specific type of GNN, a GCN (Graph Convolutional Network) combined with a LSTM (Long Short-Term Memory) neural network [13]. This approach outperformed baseline methods based on manually extracted features. However, these features were limited to the BB level and combined using a simple sum, lacking structural features that could be derived from the CFG.

In this paper, we compare function-level C code obfuscation detection and classification methods based on advanced features, including structural features extracted from the CFGs, processed by classical ML algorithms (Random Forest and Gradient Boosting), to methods based on GNNs that directly process attributed CFGs. We extend the previous GNN approach [13] by comparing different collections of features, including graph-level ones, and exploring various GNNs architectures. We investigate more advanced obfuscation techniques than those provided by OLLVM [14] by incorporating Tigress [3] in our experiments. We use a larger dataset and investigate two data splits to control the classification difficulty. Our experiments demonstrate that obfuscation detection is best achieved through a GNN processing of fine-grained semantic level features.

The remainder of this paper is organized as follows. Section 2 introduces the obfuscation techniques considered in this study and briefly discusses their impact on CFG. Section 3 reviews fundamental concepts related to GNNs. Section 4 describes the proposed dataset and Section 5 outlines our experimental setup.

Section 6 presents the initial task of binary classification, followed by the extended multi-class experiment in Section 7. Section 8 shows a real-world malware example dedicated to obfuscation detection. A discussion in Section 9 concludes this study.

2 Binary representation and obfuscation

Binary code is often described by its corresponding disassembly, a symbolic representation of the machine code. At function level, this disassembly is naturally represented with an attributed CFG that details the function execution flow between atomic blocks of code, denoted as BB. Such a representation is particularly useful as semantic information can be extracted from it, describing the function behavior.

However, binary code is extremely sensitive to compilation parameters, such as the optimization level. A given function can have multiple CFGs representations that all convey the same underlying semantics. Conversely, two different functions may share the same CFG structure but differ on the instructions in their BBs.

A different source of variability is induced by program obfuscation. It aims at altering a program syntax but not its behavior. It consists in specific transformation passes that try to increase code security against reverse engineering. Obfuscation is widely used to protect binary assets, such as data, keys or algorithms. Each obfuscation pass has specific effects on binaries [19]. In particular, a **data-related obfuscation** consists in modifying the function data-flow. For example, a MBA (Mixed Boolean Arithmetic) [33] replaces integer values with a sequence of complex arithmetic computations that is strictly equivalent. A **control-flow obfuscation** modifies the true program execution flow, either at the function level or at the program level. One elementary obfuscation among this type is the CFF (CFG Flattening), that, inside the function, puts every BB at the same level and uses a dispatcher to preserve the execution flow logic [30].

Figure 1 and Table 1 illustrate the high variability of binary code, depending on the compilation effect or obfuscation. Intuitively, detecting a pass that has a subtle effect on binary code, such as MBA, is tedious. The resulting function may be confused with a legitimate complex one with dozens of arithmetic operations.

```
int ZEXPORT inflateReset(strm)
z_streamp strm;
{
    struct inflate_state FAR *state;

    if (strm == Z_NULL || strm->state == Z_NULL) return Z_STREAM_ERROR;
    state = (struct inflate_state FAR *)strm->state;
    state->wsize = 0;
    state->whave = 0;
    state->wnext = 0;
    return inflateResetKeep(strm);
}
```

Fig. 1: `gzerror` function source code (`zlib` project)

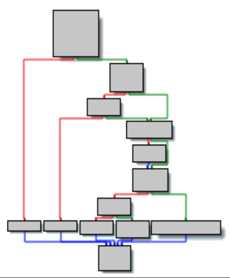
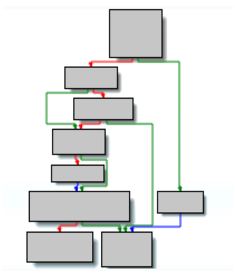
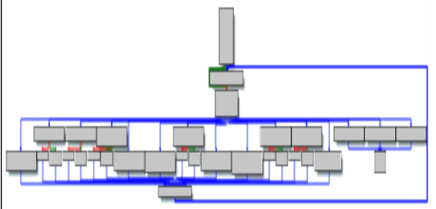
-O0 optimization	-O2 optimization	Obfuscated with CFF
		

Table 1: Optimization and obfuscation effects on the `gzerror` function CFG.

3 Machine learning for graphs

Graph representation learning aims at encoding graph data into a low-dimensional vector. There exist two main classes of algorithms for this task: feature-based approaches and GNNs. Feature-based approaches consist in using various expertly designed features to describe a graph, that are then often processed by traditional ML algorithms, such as Random Forest, for classification purpose [5]. Standard features are the number of nodes and edges, the mean node degree, the density, etc.

GNN (Graph Neural Networks) have experienced recent popularity, despite having been theorized quite early [9]. GNNs use graph structure and initial features assigned to the graph nodes to iteratively learn either node or graph representation, with a message passing mechanism. Formally, the k-th layer of a message passing GNN is described as follows [31]:

$$a_v^{(k)} = AGG^{(k)}\left(\{h_u^{(k-1)} : u \in \mathcal{N}(v)\}\right), \quad h_v^{(k)} = COMB^{(k)}\left(h_v^{(k-1)}, a_v^{(k)}\right),$$

where $h_v^{(k)}$ is the feature associated to the node v at the k-th iteration, $h_v^{(0)} = X_v$ is the initial feature of node v and $\mathcal{N}(v)$ denotes the neighborhood of node v. Such node representation can be directly used for node-level tasks. For graph-level tasks, a graph embedding can be derived using a readout function that will combine all the node representations into a final graph vector:

$$h_G = RO(\{h_v^{(K)} | v \in G\})$$

The AGG, COMB and the optional RO functions vary depending on the message passing GNN model that is used. GCN [16] was the first applicable GNN using convolutional layers. SAGE [11] is a refined GCN version, with a more advanced COMB method. GIN is a model architecture that offers the best theoretical foundations as it has been shown to be as powerful as the 1-Weisfeiler-Lehman test [31]. GAT includes an attention mechanism in the message passing framework, that should give more weight to important nodes [29]. UNet is inspired from the usual UNet architecture for computer vision, where the dimension of the input data is first downsampled and then expended again [8].

As mentioned before, GNNs take as input a graph with n nodes and a feature matrix of dimension (n, d) with d, the feature dimension. This node feature matrix should ideally describe semantically the content or the nature of graph nodes. Contrarily to feature-based approaches, GNNs use node-level features. They are known to have a huge impact during the GNN training [5].

Beyond the GNN popularity, one must remember that simple baselines should always be used as a comparison. Previous works highlight the fact GNNs do not always provide the best results compared to baselines that are less costly [5, 6].

4 Dataset

In order to study obfuscation, we have built an open-source dataset [21] that consists of C program sources obfuscated by two different obfuscators and with different obfuscation types (control-flow and data obfuscations). The dataset is based on five open source projects: `zlib`, `lz4`, `minilua`, `sqlite` and `freetype`, compiled for x86-64, that are obfuscated with Tigress [3], a source-to-source obfuscator, and OLLVM [14], that directly interfaces with the compiler. Several Tigress obfuscations are selected:

- Data obfuscation: EncodeArithmetic, EncodeLiterals;
- Control-flow obfuscation: Virtualize, OpaquePredicates, CFF, Split, Merge, Copy;
- Combined: a mix of CFF, EncodeArithmetic and OpaquePredicates, and the same mix with an additional Split.

OLLVM unfortunately offers less obfuscations: only OpaquePredicates, CFF and a pass similar to EncodeArithmetic.

Data leakage can be a serious issue when building such a dataset. A particular risk is that an obfuscation could be negated by compiler optimizations. Another potential issue is functions shared by different projects. To avoid any leakage, we use two data split strategies to produce a training set, a validation set and a test set.

In the **per function split strategy** (*Dataset-1*), a function and its obfuscated versions are within the same set. To implement this strategy, all the functions of the five projects are collected, ensuring the common functions between projects, such as the `libc` functions, are completely removed. The resulting function list is randomly split into three sets: train, validation and test with a ratio of (64%, 16%, 20%) of the functions, using stratified sampling to ensure a similar distribution of BB number across sets. For each function in a subset, we include both the obfuscated and unobfuscated versions. Notice that this leads to an unbalanced class ratio, as the original dataset contains 11 obfuscation classes for only one unobfuscated class. This class unbalance ratio is unusual in a context of anomaly detection, as in general there exist much more normal data than

Table 2: Characteristics of the two datasets

	-O0 optimization	**-O2 optimization**
Dataset-1	Train* : 3,225 / 48,813 Validation* : 803 / 12,135 Test* : 1012 / 15,403 Ratio binary= (0.11, 0.11, 0.11)	Train* : 1,846 / 23,151 Validation* : 459 / 5,753 Test* : 583 / 7,162 Ratio binary: (0.17, 0.17, 0.17)
Dataset-2	Train* : 1,137 / 18,759 Validation* : 279 / 4,652 Test* : 3,948 / 57,627 Ratio binary : (0.13, 0.13, 0.11)	Train* : 610 / 9,019 Validation* : 150 / 2,238 Test* : 3012 / 31,760 Ratio binary : (0.14, 0.14, 0.17)

* values expressed in functions/samples

abnormal ones. However, obtaining abnormal data is accessible in this context, even if in practice, obfuscated functions (abnormal data) are often limited inside a binary for computational reasons.

In the **per binary split strategy** (*Dataset-2*), we use all the functions belonging to `zlib,` `lz4` and `minilua`, and all their obfuscated versions to create the train and validation sets. These two sets are split according to the same procedure used to generate the *Dataset-1*, with a ratio of (80%, 20%). The test set is made of all the functions of `sqlite` and `freetype`, with all their obfuscated versions. This setting represents a real-world scenario for detecting obfuscation: we want to detect and characterize obfuscated functions in a completely new executable using a model trained on controlled binaries that are potentially unrelated to the new one.

Dataset-2 should be more challenging than *Dataset-1* as two projects may have a different coding style or may use a vastly different number of functions that are valid candidates for certain types of obfuscation. Having different projects during training/validation and testing prevents the model from leveraging per project regularity.

For these two datasets, -O0 and -O2 binaries are separated. Indeed, compiler optimizations tend to remove, sometimes completely, the applied obfuscation. This is particularly true for the OLLVM obfuscator, leading to many obfuscated variants that are identical to non-obfuscated versions. The dataset descriptions, especially the number of functions and samples, and the class ratio are available in Table 2.

5 Methods and feature vectors

Quantifying precisely to what extent each aspect of the obfuscation detection problem contributes to the final classification score is essential. In fact, various features are potential candidates to be used for further classification: textual data from assembly text, statistical data, etc. In this work, graph representation and features are gradually enriched, starting from existing ML works, before diving into more advanced GNN algorithms.

All our experiments were conducted on a Linux-based server equipped of Nvidia RTX A6000, with 20 cores, 40 threads and 32GiB of RAM.

This study analyzes various graph-related algorithms. Indeed, CFG are an intuitive graph representation for a binary function, where nodes represent BBs and edges denote the execution flow between these nodes inside the function. This graph is attributed and each node contains its associated assembly code, a sequence of instructions. Each instruction, e.g. *mov eax, 0* in x86-64, combines a mnemonic (the action to operate, here *mov*) and operands (the arguments of this action, here *eax, 0*) As mentioned before, two main algorithm classes are used for graph classification.

We use as baseline models traditional ML models, namely **random forests** and **gradient boosting**. They operate on **graph-level** features extracted from the functions. More precisely, we consider two types of features. We extract first CFG-related features such as the number of nodes, edges, the cyclomatic complexity, etc. They are complemented by **assembly TF-IDF features**. We use the approach proposed by Salem and al. [23], that is the counts of the 128 most used assembly mnemonics inversely weighted by the global frequencies.

We compare these baselines to five **message passing GNN models**: GCN, SAGE, GIN, GAT and UNet. As mentioned in Section 3, GNNs need to have access to both graph structure and node features that are given by the user. These **node-level** features are distinct to the ones of previous baselines that directly handle graph-level features. These node feature vectors are iteratively refined as follows.

As a reference, we use an **identity feature vector**, a one-dimensional vector filled with 1's. This forces the GNN to rely only on the graph structure.

The first non-trivial feature vector is based on **counting assembly mnemonic classes**. We adopt here an existing strategy [13] which provides a coarse representation of the assembly mnemonic distribution per BB.

To provide a more robust and refined representation, we use **semantic and counting Pcode mnemonic**. Pcode is an IR (Intermediary Representation) for which each assembly instruction is semantically represented by one or more Pcode instructions in an architecture agnostic way. Consequently, all the CPU architectures (ARM, Aarch64, MIPS) share the same underlying Pcode. This feature vector combines several CFG BB features such as the number of instructions per node with the Pcode mnemonic counts.

Finally, we consider **semantic and counting assembly mnemonic** which is similar to the previous feature vector but uses counts of all possible assembly instructions (1,828 mnemonic counts for binaries compiled in x86-64). Such a feature is specific to an architecture, contrary to Pcode.

The above models are evaluated with their corresponding candidate feature vectors, on both *Dataset-1* and *Dataset-2*. To select the best hyperparameters on the validation set, GridSearchCV and Optuna [1] are respectively used for the baselines and the GNNs. The Optuna search is applied with three seeds, with the best run leading to the chosen hyperparameters, such as the number of layers or the hidden dimensions. Each Optuna run is restricted to 20 trials in

Table 3: Binary classification scores, depending on features, algorithms and datasets.

Graph	**Features**	**Algorithm**	**Balanced accuracy**	
			Dataset-1	***Dataset-2***
CFG	Graph features & assembly (Dim: #**23**)	RandomForest	0.702	0.60
		GradientBoosting	0.725	0.649
	TF-IDF on assembly mnemonics (Dim: #**128**)	RandomForest	0.76	0.607
		GradientBoosting	0.80	0.683
	Identity (Dim: #**1**)	GCN	0.634	0.608
		Sage	0.615	0.574
		GIN	0.603	0.531
		GAT	0.589	0.539
		UNet	0.616	0.555
	Counting mnemonic classes (Dim: #**27**)	GCN	0.659	0.658
		Sage	0.694	0.66
		GIN	0.701	0.673
		GAT	0.655	0.667
		UNet	0.66	0.654
	Semantic & counting PCode mnemonics (Dim: #**78**)	GCN	0.789	0.736
		Sage	0.801	0.755
		GIN	0.80	0.766
		GAT	0.805	0.731
		UNet	0.779	0.672
	Semantic & counting assembly mnemonics (Dim: #**1839**)	GCN	0.792	0.758
		Sage	0.802	0.727
		GIN	0.793	0.727
		GAT	0.797	0.729
		UNet	0.785	0.701

order to limit the computational burden. Baselines and GNNs are respectively implemented using scikit-learn [20] and Pytorch-Geometric [7].

Because of the unbalanced classes, both in binary and multi-class settings, our benchmarks are evaluated using the balanced accuracy. This metric heavily penalizes cases where a class is not properly detected compared to the others.

In this work, results for -O2 are omitted for brevity since they are based on the same principles as -O0 and show the same trends.

6 Binary classification

In this Section, we address a simplified binary classification problem where the goal is simply to determine if a function is obfuscated or not. Results for -O0 are available in Table 3.

We note first that baselines with graph-level features demonstrate satisfactory results, with a balanced accuracy that is better for the *Dataset-1*. Such behavior is expected as the *Dataset-2* framework is more challenging. Gradient Boosting outperforms slightly Random Forest. Besides, the TF-IDF baseline respectively

Table 4: Multi-class classification scores, depending on features, algorithms and datasets.

Graph	**Features**	**Algorithm**	**Balanced accuracy**	
			Dataset-1	***Dataset-2***
CFG	Graph features & assembly (Dim: #**23**)	RandomForest	0.65	0.57
		GradientBoosting	0.66	0.594
	TF-IDF on assembly mnemonics (Dim: #**128**)	RandomForest	0.697	0.593
		GradientBoosting	0.724	0.579
	Identity (Dim: #**1**)	GCN	0.323	0.326
		Sage	0.341	0.347
		GIN	0.414	0.407
		GAT	0.192	0.195
		UNet	0.362	0.299
	Counting mnemonic classes (Dim: #**27**)	GCN	0.431	0.462
		Sage	0.498	0.499
		GIN	0.488	0.474
		GAT	0.45	0.342
		UNet	0.439	0.448
	Semantic & counting PCode mnemonics (Dim: #**78**)	GCN	0.721	0.675
		Sage	0.737	0.549
		GIN	0.732	0.657
		GAT	0.729	0.637
		UNet	0.704	0.655
	Semantic & counting assembly mnemonics (Dim: #**1839**)	GCN	0.723	0.633
		Sage	0.718	0.535
		GIN	0.713	0.427
		GAT	0.723	0.646
		UNet	0.709	0.611

reaches 0.80 and 0.68 of balanced accuracy for *Dataset-1* and *Dataset-2*. This highlights the fact that a fine-grained representation of the assembly mnemonic distribution characterizes better the abnormality induced by obfuscation than graph-level features with a coarse-grained assembly representation.

We observe that enriching GNN initial features is fundamental: elementary features, such as the identity, offer poor performances compared to more accurate features. This is consistent with the inferior performances of structural graph features with baseline methods. Moreover, the best performances are obtained with the richest representation.

Using a feature based on assembly mnemonic counts provides approximately the same results than relying on the Pcode instructions, even though it is much larger and restricted to a specific architecture, which is not the case for Pcode, applicable to any CPU architecture, not only x86-64. This demonstrates the Pcode potential to extract semantic features. Besides, theoretical results [31] about GIN expressivity power are confirmed by these experiments as GIN is generally slightly more efficient than other GNN.

Table 5: Obfuscation detection results on two XTunnel samples.

	Binary balanced accuracy	Multi-class balanced accuracy
Sample C637E	0.726	0.533
Sample 99B45	0.711	0.55

7 Multi-class classification

In multi-class classification, the goal is to determine the type of obfuscation that has been applied to a function. Results for -O0 are available in Table 4. Many observations from the binary case are confirmed in this experiment. Indeed, elementary baselines perform remarkably well, given the fact there exists 11 classes. As a comparison, previous works consider at most 4 classes [13]. Similarly, GNNs outperform baselines, especially on *Dataset-2*, when the features are enriched with semantic information originated from Pcode mnemonic.

8 Real-world example: XTunnel

XTunnel is a malware, developed by APT28 hacking group, that can relay traffic between a victim and a server used by cybercriminals to control compromised devices and exfiltrate data. Multiple variants have been found on governmental and institutional networks, for which some of them were obfuscated. This obfuscation has been used to evade security products. Malware deobfuscation helps to highlight and determine malicious functionalities hidden inside these obfuscated executables [2]. Then, locating and determining the obfuscation type is necessary before any deobfuscation attempt. These tasks are performed on two XTunnel obfuscated samples[5]. Results are compared with a previously built ground truth, that was computed using a costly approach of symbolic execution [2]. Such ground truth asserts with a satisfactory confidence that these samples were heavily obfuscated using OpaquePredicates. The binary and multi-class classification are handled by the model that, on averaged, gives the best results, which is GIN with Pcode-based feature. The binary scores are computed over all the executable functions, whereas the multi-class scores are limited to obfuscated functions only. Results are available in Table 5 and show the validity of our approach. Interestingly, many functions are considered obfuscated with EncodeArithmetic instead of OpaquePredicates. Indeed, distinguishing these two obfuscations is tedious as an OpaquePredicate is simply an EncodeArithmetic followed by a branching instruction, with one fake branching. Consequently, our models can still detect a suspicious pattern related to OpaquePredicates and we consider these predictions as valid.

[5] Their corresponding hashes are C637E01F50F5FBD2160B191F6371C5DE2AC56DE4 and 99B454262DC26B081600E844371982A49D334E5E.

9 Conclusion

To conclude, this work provides a general study about obfuscation detection, at both binary and multi-class levels. It demonstrates the efficiency of standard baselines and most importantly the potential of GNNs that, combined with meaningful features conveying part of function semantics, achieve the best results. These results are confirmed with a real-world malware example.

If this work seeks to be as complete as possible, it is subject to specific limitations. First, building a real-word obfuscated dataset implies a lot of implementation constraints. We try our best to represent the large variety of obfuscators and obfuscations, given accessible resources. Because obfuscation and optimizations are intertwined, it is difficult to ensure that the obfuscation was correctly applied and that the compiler optimizations do not remove or attenuate initial obfuscation, especially in -O2. As a result, our dataset may contain specific functions that differ from what they should be. Second, GNN hyperparameters were obtained with a budget constraint. As a consequence, specific GNNs may have been advantaged compared to others. As an example, GAT takes a long time to train compared to simpler models such as GCN.

Finally, this works constitutes only a first step of a more general study on obfuscation detection. More attention should be dedicated to innovating graph types and features that should capture as much as possible the function semantics. The binary similarity problem [18] faces the same challenge, leading to the development of new graphs, such as SOG (Semantic Oriented Graph) [12] that is, to the best of our knowledge, the first attempt that tries to represent binary code by combining multiple edge types (data, control-flow, effects) inside a graph using solely disassembly. This representation seems promising as it brings together all the key aspects of a function, in particular part of its semantics.

Acknowledgments

The authors thank the Agence Innovation Defense (AID) for its financial support.

References

1. Akiba, T., Sano, S., Yanase, T., Ohta, T., Koyama, M.: Optuna: A next-generation hyperparameter optimization framework. In: Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining, pp. 2623–2631 (2019)
2. Bardin, S., David, R., Marion, J.Y.: Backward-bounded dse: Targeting infeasibility questions on obfuscated codes. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 633–651 (2017). DOI 10.1109/SP.2017.36
3. Collberg, C.: The tigress c obfuscator. https://tigress.wtf/index.html. Accessed: 2023-08-17
4. David, R., Coniglio, L., Ceccato, M.: Qsynth-a program synthesis based approach for binary code deobfuscation. In: BAR 2020 Workshop (2020)

5. Errica, F., Podda, M., Bacciu, D., Micheli, A.: A fair comparison of graph neural networks for graph classification. In: International Conference on Learning Representations (2020). URL https://openreview.net/forum?id=HygDF6NFPB
6. Ferrari Dacrema, M., Cremonesi, P., Jannach, D.: Are we really making much progress? a worrying analysis of recent neural recommendation approaches. In: Proceedings of the 13th ACM Conference on Recommender Systems, RecSys '19. ACM (2019). DOI 10.1145/3298689.3347058. URL http://dx.doi.org/10.1145/3298689.3347058
7. Fey, M., Lenssen, J.E.: Fast graph representation learning with PyTorch Geometric. In: ICLR Workshop on Representation Learning on Graphs and Manifolds (2019)
8. Gao, H., Ji, S.: Graph u-nets. In: K. Chaudhuri, R. Salakhutdinov (eds.) Proceedings of the 36th International Conference on Machine Learning, *Proceedings of Machine Learning Research*, vol. 97, pp. 2083–2092. PMLR (2019). URL https://proceedings.mlr.press/v97/gao19a.html
9. Gori, M., Monfardini, G., Scarselli, F.: A new model for learning in graph domains. In: Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005., vol. 2, pp. 729–734 vol. 2 (2005). DOI 10.1109/IJCNN.2005.1555942
10. Greco, C., Ianni, M., Guzzo, A., Fortino, G.: Explaining binary obfuscation. pp. 22–27 (2023). DOI 10.1109/CSR57506.2023.10224825
11. Hamilton, W.L., Ying, R., Leskovec, J.: Inductive representation learning on large graphs. In: Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS'17, p. 1025–1035. Curran Associates Inc., Red Hook, NY, USA (2017)
12. He, H., Lin, X., Weng, Z., Zhao, R., Gan, S., Chen, L., Ji, Y., Wang, J., Xue, Z.: 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 (2024)
13. Jiang, S., Hong, Y., Fu, C., Qian, Y., Han, L.: Function-level obfuscation detection method based on graph convolutional networks. Journal of Information Security and Applications **61**, 102,953 (2021). DOI https://doi.org/10.1016/j.jisa.2021.102953. URL https://www.sciencedirect.com/science/article/pii/S2214212621001654
14. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-llvm–software protection for the masses. In: B. Wyseur (ed.) Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015, pp. 3–9. IEEE (2015). DOI 10.1109/SPRO.2015.10
15. Kanzaki, Y., Monden, A., Collberg, C.: Code artificiality: A metric for the code stealth based on an n-gram model. In: 2015 IEEE/ACM 1st International Workshop on Software Protection, pp. 31–37. IEEE (2015)
16. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: International Conference on Learning Representations (2017). URL https://openreview.net/forum?id=SJU4ayYgl
17. Kriege, N.M., Johansson, F.D., Morris, C.: A survey on graph kernels. Applied Network Science **5**(1), 6 (2020). DOI 10.1007/s41109-019-0195-3. URL https://doi.org/10.1007/s41109-019-0195-3
18. Marcelli, A., Graziano, M., Ugarte-Pedrero, X., Fratantonio, Y., Mansouri, M., Balzarotti, D.: How machine learning is solving the binary function similarity problem. In: 31st USENIX Security Symposium (USENIX Security 22), pp. 2099–2116 (2022)
19. Nagra, J., Collberg, C.: Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Pearson Education (2009)

20. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al.: Scikit-learn: Machine learning in python. the Journal of machine Learning research **12**, 2825–2830 (2011)
21. Quarkslab: Obfuscation dataset. https://github.com/quarkslab/diffing_obfuscation_dataset. Accessed: 2024-09-01
22. Raja, V., Fernandes, K.J.: Reverse engineering: an industrial perspective. Springer Science & Business Media (2007)
23. Salem, A., Banescu, S.: Metadata recovery from obfuscated programs using machine learning. In: Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering, pp. 1–11 (2016)
24. Salwan, J., Bardin, S., Potet, M.L.: Symbolic deobfuscation: From virtualized code back to the original. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 372–392. Springer (2018)
25. Schrittwieser, S., Wimmer, E., Mallinger, K., Kochberger, P., Lawitschka, C., Raubitzek, S., Weippl, E.R.: Modeling obfuscation stealth through code complexity. In: European Symposium on Research in Computer Security, pp. 392–408. Springer (2023)
26. Sharif, M.I., Lanzi, A., Giffin, J.T., Lee, W.: Impeding malware analysis using conditional code obfuscation. In: NDSS (2008)
27. Tofighi-Shirazi, R., Asavoae, I.M., Elbaz-Vincent, P., Le, T.H.: Defeating opaque predicates statically through machine learning and binary analysis. In: Proceedings of the 3rd ACM Workshop on Software Protection, pp. 3–14 (2019)
28. Tofighi-Shirazi, R., Asăvoae, I.M., Elbaz-Vincent, P.: 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, SSPREW9 '19. Association for Computing Machinery, New York, NY, USA (2019). DOI 10.1145/3371307.3371313. URL https://doi.org/10.1145/3371307.3371313
29. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., Bengio, Y.: Graph attention networks. In: International Conference on Learning Representations (2018). URL https://openreview.net/forum?id=rJXMpikCZ
30. Wang, C.: A security architecture for survivability mechanisms. University of Virginia (2001)
31. Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How powerful are graph neural networks? In: International Conference on Learning Representations (2019). URL https://openreview.net/forum?id=ryGs6iA5Km
32. Yadegari, B., Johannesmeyer, B., Whitely, B., Debray, S.: A generic approach to automatic deobfuscation of executable code. In: 2015 IEEE Symposium on Security and Privacy, pp. 674–691 (2015). DOI 10.1109/SP.2015.47
33. Zhou, Y., Main, A., Gu, Y.X., Johnson, H.: Information hiding in software with mixed boolean-arithmetic transforms. In: International Workshop on Information Security Applications, pp. 61–75. Springer (2007)