Cohen *et al. Applied Network Science* (2025) 10:49
https://doi.org/10.1007/s41109-025-00733-8

Applied Network Science

RESEARCH Open Access

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*]

*Correspondence:
Roxane Cohen
roxane.cohen@dauphine.eu
Robin David
rdavid@quarkslab.com
Florian Yger
florian.yger@insa-rouen.fr
Fabrice Rossi
rossi@ceremade.dauphine.fr
[1]Quarkslab, Paris, France
[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 concern in various situations, ranging from legitimate use cases to unethical contexts. Obfuscation is one of the most used techniques to ensure such a 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 classical baselines to advanced techniques like Graph Neural Networks (GNN), 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 two practical binary analysis examples. It highlights how much obfuscation and optimization are intertwined in binary code and that a better comprehension of these two principles are fundamental in order to obtain better detection results.

Keywords Obfuscation, Classification, GNN, Graphs, Graph representation learning, Security, Optimization

Introduction

Binary programs and their sensitive contents are often protected from reverse engineering through obfuscation techniques (Nagra and Collberg 2009), which aim to obscure a program's underlying logic without altering its functionality. While reverse engineers seek to comprehensively understand program semantics (Raja and Fernandes 2007), 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 (Sharif et al. 2008). 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 (Greco et al. 2023).

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. 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 (David et al. 2020; Yadegari et al. 2015; Bardin et al. 2017; Salwan et al. 2018; Tofighi-Shirazi et al. 2019). 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 (Kanzaki et al. 2015).

In this paper, we compare function-level C code obfuscation detection and classification methods based on advanced features, including structural features extracted from the CFG (Control Flow Graph), 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 (Jiang et al. 2021) 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 (Junod et al. 2015) by incorporating Tigress (Collberg 2023) in our experiments. We use a larger dataset and investigate two data splits to control the classification difficulty.

Our experiments demonstrate that obfuscation detection and characterization are efficient with classical baselines and features describing the distribution of mnemonic inside a function. More importantly, the best scores are achieved through a GNN processing of fine-grained semantic level features. These results are considerably modified in some cases, especially when the initial functions are optimized, highlighting the importance of a better comprehension between obfuscation and optimization.

The remainder of this paper is organized as follows. Section "Binary representation and obfuscation" introduces the concepts used throughout this paper, the obfuscation techniques considered in this study and briefly discusses their impact on CFG. Section "Machine learning for grap" reviews fundamental concepts related to graph learning. Section "Related papers" provides a description of the works related to our study. Section "Experimental settings" describes our experiments technical details. Section "Binary classification" presents the initial task of binary classification, followed by the extended multi-class experiment in Section "Multi-class classification". Section "Optimization against obfuscation" details how optimization may alter an initial obfuscation and its impact on the general detection and characterization tasks. Section "Real-world examples" shows two real-world examples dedicated to obfuscation detection. A discussion in Section "Conclusion" concludes this study.

Binary representation and obfuscation

Binary code is often described by its corresponding disassembly, a symbolic representation of the machine code. At the function level, this disassembly is naturally represented with an attributed CFG that details the function execution flow between blocks of code, denoted as BB (Basic Block). A BB contains a sequence of assembly instructions without any branching, meaning that the block will be executed from its beginning until its end. 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*). Such a

representation is particularly useful as semantic information can be extracted from it, describing the function behavior.

This disassembly is, however, architecture-dependent. Indeed, one source code, for which an example is displayed in Fig. 1a, can be compiled to a specific architecture, which defines how a CPU executes instructions. These architectures have different instruction sets and registers. For example, the x86 and x86-64, denoted x64, architectures[1] use a specific set of branching mnemonics, like *jz, jnz, je, jne, ja, jne*, etc. whereas the Arm architecture[2] uses instead *b, bl, bx*. As a consequence, using an architecture-dependent disassembly to build a graph representation lacks of generalization, as the same source code, compiled with different architectures, as shown in Fig. 1b, c, will end up with completely dissimilar CFG. Consequently, one may want to use a unified graph representation, valid for any compiled code source, regardless of its architecture. Pcode (Borgerson 2024) is the IR (Intermediary Representation) behind the Ghidra disassembler,[3] used to obtain a higher-level abstraction compared to machine code. Each assembly instruction is semantically represented by one or more Pcode instructions in an architecture agnostic way, like in Fig. 1d. Consequently, all the CPU architectures (ARM, Aarch64, MIPS) share the same underlying language. Figure 1 illustrates the link between source code, architecture-dependent code and IR.

Apart from the assembly language or the underlying Pcode IR, binary code is extremely sensitive to compilation parameters, such as the optimization level. In fact, the more optimized the code is, the faster it will execute, and the more its assembly will

```
int main() {
   int x = 0;
   return x;
}
```

(a) A source code function assigning a value to a variable.

```
push    rbp
mov     rbp, rsp
mov     [rbp+var_4], 0
mov     eax, [rbp+var_4]
pop     rbp
retn
```

(b) x86-64 instructions

```
push    {r11}
add     r11, sp, #0
sub     sp, sp, #0xc
mov     r3, #0
str     r3, [r11,#var_8]
ldr     r3, [r11,#var_8]
mov     r0, r3
add     sp, r11, #0
pop     {r11}
bx      lr
```

(c) Arm instructions

```
IMARK ram[8244:4]
sp = sp + 0xfffffffc
*[ram]sp = r11
IMARK ram[8248:4]
unique[2e80:4] = 0x0 >> 0x1f
unique[2f00:1] = 0x0 == 0x0
unique[2f80:1] = unique[2f00:1] && CY
unique[3000:1] = 0x0 != 0x0
unique[3080:1] = SUBPIECE unique[2e80:4], 0x0
unique[3100:1] = unique[3000:1] && unique[3080:1]
shift_carry = unique[2f80:1] || unique[3100:1]
tmpCY = carry(sp, 0x0)
tmpOV = scarry(sp, 0x0)
r11 = sp + 0x0
tmpNG = r11 s< 0x0
tmpZR = r11 == 0x0
IMARK ram[824c:4]
```

(d) First two instructions (delimited by IMARK opcode) translated into Pcode

Fig. 1 An original source code, the resulting binary code compiled for x64 and Arm and their common underlying Pcode instructions

[1] https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[2] https://developer.arm.com/documentation.

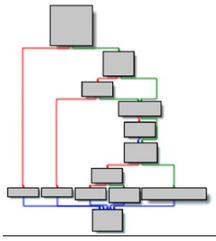
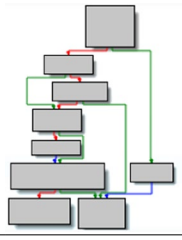
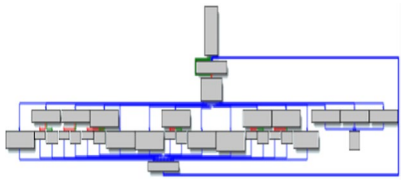
[3] https://ghidra-sre.org/

```
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. 2 gzerror function source code (zlib project)

Table 1 Optimization and obfuscation effects on the gzerror function CFG

O0 optimization	O2 optimization	Obfuscated with CFF
		

differ from non-optimized assembly code. Thus, entire BBs may disappear, sometimes even entire functions. However, the optimization preserves the program semantics and that a given function, optimized differently, will exhibit 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 of 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 (Nagra and Collberg 2009):

- A **data-related obfuscation** consists in modifying the function data-flow. For example, a MBA (Mixed Boolean Arithmetic) (Zhou et al. 2007) 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 (Wang 2001).

Figure 2 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.

Machine learning for graphs

Since graphs naturally represent functions execution flow, graph learning is adapted to our task and particularly graph vector representations. 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 GNN (Graph Neural Networks). 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 (Errica et al. 2020). Standard features are the number of nodes and edges, the mean node degree, the density, etc.

GNNs have experienced recent popularity, despite having been theorized quite early (Gori et al. 2005). GNNs use graph structure and initial features assigned to the graph nodes to iteratively learn either node or graph representations, with a message passing mechanism. Each node v is associated to an initial feature X_v, that will be further refined depending on its node neighbors, denoted $\mathcal{N}(v)$, in the graph structure in order to account for neighbor representation. A schema of a general GNN model (Wu et al. 2021) is displayed in Fig. 3. Given a graph and a feature matrix, where each entry describes a node by a feature, a GNN model stacks multiple message-passing or convolution layers. At each layer, node representations are iteratively refined by incorporating information from the node itself and its neighbors, with each node potentially having an arbitrary degree. This process propagates node data further inside the graph, the model depth deciding how far an initial node feature can be spread into the graph. Between each message-passing layer, it is possible to use a pooling layer that reduces the graph order by extracting features for sub-graphs. At the end, each node is associated to an embedding, which is a learned vector representation of the node that both captures the structural role of it and its semantic properties.

Formally, the k-th message-passing layer of a GNN is described as follows Xu et al. (2019):

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

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. In this work, edge features are not included in the GNN design as CFG do not provide any.

After K steps of node embedding refinement, final node representations can be directly used for node-level tasks. For graph-level tasks, a graph embedding of the graph G can be derived using a readout function that will combine all the node representations into a final graph vector:

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

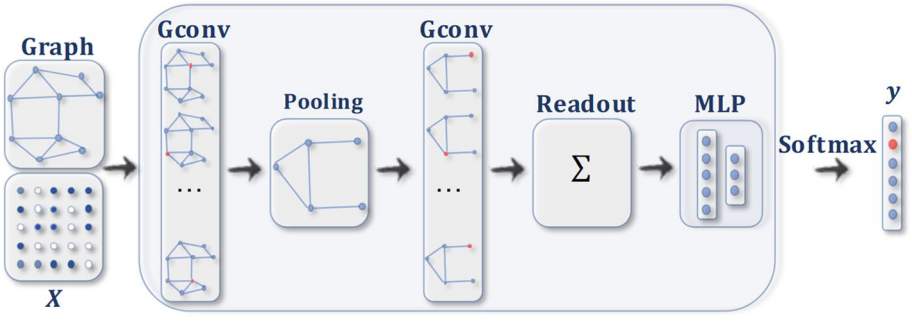


Fig. 3 GNN framework (Wu et al. 2021)

The AGGREGATE, COMBINE and the optional READOUT functions vary depending on the message passing GNN model that is used. GCN (Graph Convolutional Network) (Kipf and Welling 2017) was the first applicable GNN using convolutional layers. SAGE (Hamilton et al. 2017) is a refined GCN version, with a more advanced COMBINE 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 (Xu et al. 2019). GAT includes an attention mechanism in the message passing framework, that should give more weight to important nodes (Velčković et al. 2018). UNet is inspired from the usual UNet architecture for computer vision, where the dimension of the input data is first downsampled and then expanded again (Gao and Ji 2019).

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 (Errica et al. 2020).

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 (Errica et al. 2020; Ferrari Dacrema et al. 2019).

Related papers

The problem of detecting and characterizing an obfuscation is related to the stealth property of the obfuscation (Nagra and Collberg 2009). This problem has been studied from a heuristic and ML perspectives in several previous works.

Heuristic-based approaches either aim to detect if a code portion is obfuscated (Blazytko 2021a, 2023) or how (Blazytko 2021b). They often rely on a prior knowledge about the obfuscation and use assembly statistical analysis or graph properties, such as dominance relationships. These techniques do not use any learning process.

Most ML approaches extract various features from binary programs, such as the distribution of instructions in the assembly code (Salem and Banescu 2016), the function complexity metrics (Schrittwieser et al. 2023) or semantic reasoning (Tofighi-Shirazi et al. 2019). Some of the more advanced features are derived from a graph-based representation of the binary functions, such as the CFG-based cyclomatic complexity. These features are then used with standard machine-learning algorithms such as Decision Trees, Random Forests or Gradient Boosting. In particular, a focus is made on explainability (Greco et al. 2023) for obfuscation applied at the program level.

While extracting domain-specific features from the CFG has been proved to be effective in some cases (Greco et al. 2023), leveraging the full structural information of the graph can enhance classification performance in certain applications. This can be achieved using kernel methods (Kriege et al. 2020) or GNN (Graph Neural Networks). GNNs are under very active development since their introduction (Gori et al. 2005) and have shown promise in outperforming feature-based approaches in specific scenarios (Errica et al. 2020).

More recent works try to adapt GNN to this problem. In particular, function-level obfuscation detection and classification were investigated using a GCN and a LSTM (Long Short-Term Memory) neural network applied on the resulting graph embedding

(Jiang et al. 2021). 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. Besides, applying a LSTM on a graph embedding seems to lack of a stable theoretical background. We extend this work by studying multiple GNN models and exploring different types of features than only classes of mnemonics.

Compared to all of these papers, we evaluate novel feature types, such as Pcode, and new graph types. The impact of the optimization on the obfuscation is particularly considered as it is often not taken in consideration in existing research.

Experimental settings

Detecting and characterizing obfuscations using graph-based ML requires a large dataset containing various types of obfuscations. Section "Dataset" details how two datasets were created for these purposes. From these datasets, different types of graphs are extracted, explained in Section "Graph types", on which different models are trained, described in Section "Models and features".

Dataset

In order to study obfuscation, we have built an open-source dataset (Quarkslab 2024) 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 x64, that are obfuscated with Tigress [13], a source-to-source obfuscator, and OLLVM (Junod et al. 2015), that directly interfaces with the compiler. Several Tigress obfuscations are selected:

- Data obfuscations: EncodeArithmetic, EncodeLiterals;
- Control-flow obfuscations: 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.

These five projects are selected due to Tigress−3.1 limitations. Specifically, it can only be applied to C projects contained within a single source file, as the provided merging tool does not function reliably on realistic projects. In practice, most prior research relies on toy examples, and finding realistic projects fullfilling the aformentioned constraint remains challenging. Projects from diverse application domains were carefully sought to satisfy this constraint. As a result, zlib, lz4, minilua, sqlite and freetype were identified as the only publicly realistic available projects meeting these criteria. The dataset creation has been documented in previous work (Cohen et al. 2025).

Data leakage can be a serious issue when building such a dataset. A particular risk 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%), 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 abnormal ones. Unlike standard anomaly detection settings, where anomalous data is sparse compared to normal behavior, our experimental setup allows us to obtain abnormal data without difficulty. However, in real-world scenarios, 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, the class ratio and various graph characteristics, are available in Table 2. It illustrates the variability between the *Dataset-1* and *Dataset-2*, the -O0 and -O2 optimizations and the two types of graphs considered in this work and detailed in Section Graph Types. Some features, in particular the graph order and size, and the average node degree are features that impact most graph learning algorithms, especially GNN (Xu et al. 2019).

Quantifying precisely to what extent each aspect of the obfuscation detection problem contributes to the final classification score is essential. We explore different graph types and various features that 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 64 GiB of CPU RAM and 48 GiB of GPU RAM.

Graph types

In this study, two types of graphs are considered:

- CFG, 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

Table 2 Characteristics of the two datasets depending on the graph type and the optimization level

		-O0 optimization		-O2 optimization	
	Train*	3,225 / 48,813		1,846 / 23,151	
	Validation*	803 / 12,135		459 / 5,753	
	Test*	1012 / 15,403		583 / 7,162	
	Ratio binary	(0.11, 0.11, 0.11)		(0.17, 0.17, 0.17)	
		CFG	CFG-IR	CFG	CFG-IR
Dataset-1	Min graph order	1	–	1	2
	Max graph order	5,344	–	1,432	2,761
	Mean graph order	15.6	–	17.54	36,54
	Median graph order	7	–	10	21
	Mean graph density	0.116	–	0.132	0.118
	Min graph size	0	–	0	1
	Max graph size	7,122	–	2,396	3,725
	Mean graph size	21.64	–	26.57	45.45
	Median graph size	8	–	13	24
	Min average node degree	0	–	0	1
	Max average node degree	3.93	–	4.44	3.31
	Mean average node degree	1.84	–	2.28	2.16
	Median average node degree	2.4	–	2.67	2.31
	Train*	1,137 / 18,759		610 / 9,019	
	Validation*	279 / 4,652		150 / 2,238	
	Test*	3,948 / 57,627		3012 / 31,760	
	Ratio binary	(0.13, 0.13, 0.11)		(0.14, 0.14, 0.17)	
		CFG	CFG-IR	CFG	CFG-IR
Dataset-2	Min graph order	1	–	1	2
	Max graph order	9454	–	9,829	23,305
	Mean graph order	22.48	–	25.81	54
	Median graph order	7	–	10	22
	Mean graph density	0.112	-	0.127	0.117
	Min graph size	0	–	0	1
	Max graph size	12,764	–	15,620	28,944
	Mean graph size	31.73	–	40,35	68.35
	Median graph size	9	–	14	25
	Min average node degree	0	-	0	1
	Max average node degree	3.95	-	4.85	3.48
	Mean average node degree	1.89	-	2.31	2.17
	Median average node degree	2	-	2.67	2.33

The symbol "-" indicates that computing the corresponding graph is unaffordable given given our current means

*Values expressed in functions/samples

associated assembly code, namely a sequence of architecture-specific instructions, where an instruction is composed from a mnemonic and several operands. Such a graph representation is the one that is mostly used by current disassemblers, such as IDA-Pro[4] or Ghidra.[5]

- CFG-IR. This graph is based on the elementary CFG, except it uses the Pcode representation instead of the raw assembly code. Each assembly instruction is equivalent to a list of more abstract Pcode instructions. Then, it is possible to create a CFG-IR, where each node corresponds to a Pcode block of code, linked by Pcode execution flow. This CFG-IR differs from the standard CFG as several assembly

[4] https://hex-rays.com/ida-pro.

[5] https://ghidra-sre.org/

instructions inside a BB can be translated as a list of Pcode instructions that contains a branching one (which means the block is no more basic and may include a flow redirection). When it happens, the structures of the CFG and CFG-IR differ, leading to the creation of new Pcode blocks and edges in the CFG-IR graph. As a consequence, the CFG-IR contains more nodes with more instructions and more edges compared to the CFG. If the assembly instructions that are transformed into a sequence of instructions containing a branch are limited, the general CFG-IR topology should be preserved to some extent.

Models and features

In this study, two types of model are considered:

- **Classical ML** as baseline models, in particular **random forests** and **gradient boosting**.
- **GNN**, with five **message-passing** algorithms: GCN, SAGE, GIN, GAT and UNet.

These two types of algorithms do not operate at the same granularity. Classical ML methods extract **graph-level** features from the functions whereas GNN methods need **node-level** features as input, apart from the graph structure, meaning that each BB, wheter inside the CFG or the CFG-IR needs to be associated with a feature.

Classical baselines are evaluated with two types of features:

- Graph-related features, both common to CFG and CFG-IR, are computed: the number of nodes, edges, the cyclomatic complexity, the number of strongly connected components, the mean degree, density, diameter, transitivity, its number of components, the maximum number of instructions per node, the averaged number of instructions per node, the total number of instructions per function, the maximum degree and the minimum degree. Specifically, these features are complemented with assembly data when available with the CFG, such as the number *imul*, *shr*, *shl*, *sar*, *div*, *xor*, *add*, *sub* instructions in the function, the number of Immediate in the function and the number of read and write. The corresponding feature dimensions are respectively **23** and **13** for the CFG and CFG-IR.
- An assembly TF-IDF feature, proposed by Salem and Banescu (2016), is analyzed. It consists in the counts of the **128** most used assembly mnemonics inversely weighted by the global frequencies for the assembly. Concerning the Pcode, because Pcode has a limited set of operands, this feature has a dimension of **72**.

Conversely, GNN features are gradually enriched as follows:

- As a reference, we use an **identity** feature vector, a **one-dimensional** vector filled with 1's. Such a feature is valid for both the CFG and the CFG-IR. It does not convey any meaning about the nodes. Consequently, GNN are relying only on the graph structure to iteratively refine the node embeddings.
- The first non-trivial feature vector is based on **counting assembly mnemonic classes**. We adopt here an existing strategy (Jiang et al. 2021) which provides a coarse representation of the assembly mnemonic classes distribution per BB. Each assembly mnemonic is attributed to a general class of operations, among **27** classes, such as conditional transfer mnemonics (e.g *jne*) or logical mnemonics (e.g. *and*). Such an approach is based on assembly code exclusively, hence used only with the CFG

representation. Transferring this feature to Pcode is not relevant as defining Pcode operation classes is less intuitive and would only create a small number of classes, insufficient to correctly describe a Pcode block in a case of CFG-IR.

- Besides counting user-defined mnemonics classes, we directly compute a counting mnemonic feature. Specifically to the CFG and its corresponding assembly, it counts all possible assembly instructions, with 1828 different mnemonics for binaries compiled in x64. Such a feature is specific to the x64 architecture, thus the CFG. This feature is complemented by several BB features.[6]
- Counting assembly mnemonics results in a high-dimensional features specific to a given architecture. To provide a lighter and more robust feature, we define the Pcode counterpart of the previous counting feature. Its dimension is at most 72 and is available for any architecture. This feature is available for both the CFG-IR and the CFG, as Pcode can be obtained from assembly. It is further augmented by specific graph node features, like the number of instructions per node,[7] resulting in a **77**-dimensional feature.
- Motivated by the recent success of NLP (Natural Language Processing)-inspired state-of-the art approaches addressing the binary similarity problem (Massarelli et al. 2019), each BB code is given to PalmTree (Li et al. 2021), a transformer trained on x86 assembly code. The resulting text embedding feature, of size **128**, is used as the initial feature for the GNN. Because there exists no transformer model trained directly on Pcode, this feature is limited to the CFG graph.

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 (Akiba et al. 2019) 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 order to limit the computational burden. Baselines and GNNs are respectively implemented using scikit-learn (Pedregosa et al. 2011) and Pytorch-Geometric (Fey and Lenssen 2019).

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.

Results for -O0 and -O2 are shown separately. Indeed, even if they show the same trend, -O2 results tend to be significantly lower, due to the subsequent optimization that is applied and may modify the original obfuscation. More details are available in Section "8".

Binary classification

This Section is dedicated to the binary classification problem that tries to determine if a function has been obfuscated or not.

[6] The complete additional features are a boolean value indicating if the BB is the first one of the function, label encoding depending of the last mnemonic of a node (conditional jump, unconditional jump, call, ret, other), the number of instructions per node, the BB number of successors and predecessors, the number of read-write accesses, of immediates, of syscall and the different number of calls (external, internal or register).

[7] The complete additional features are a boolean value indicating if the basic block is the first one of the function, label encoding depending of the last mnemonic of a node (conditional jump, unconditional jump, call, ret, other), the number of instructions of a node, the number of its successors and the number of its predecessors.

Results dedicated to the CFG, both in -O0 and -O2, are available in Tables 3 and 5, whereas Tables 4 and 6 contain classification scores using CFG-IR.

First, baselines demonstrate satisfactory results, especially for -O0 CFG, achieving at least 0.60 of balanced accuracy. The best balanced accuracy is achieved with a Gradient Boosting model and the mnemonic TF-IDF feature. These model and features are almost all the time slightly better than their respective alternatives, Random Forest and graph-based features. This result is explained by the fact the graph-based features remain coarse-grained and high-level and cannot describe precisely enough the semantics of the function, contrarily to a mnemonic distribution represent that characterizes better the abnormality induced by obfuscation.

Second, for all the available benchmarks, results are significantly better for the *Dataset-1*, compared to the *Dataset-2*. Such a behavior is expected as this latter dataset is more challenging. This difference is of at most 0.10, showing that even in a disadvantageous context, it is possible to effectively classify obfuscated functions from unobfuscated ones.

GNN results are contrasted. In fact, using the identity feature reflects disappointing performances, actually outperformed by simpler baselines. The graph structure only is not sufficient to discriminate between obfuscated functions and unobfuscated ones

Table 3 Binary classification scores on the CFG, depending on features, algorithms and datasets for -O0 optimization. "-" indicates GPU Out-Of-Memory error

Features	Algorithm	Balanced accuracy	
		Dataset-1	Dataset-2
Graph features &	RandomForest	0.702	0.60
assembly (Dim: **#23**)	GradientBoosting	0.725	0.649
TF-IDF on assembly	RandomForest	0.76	0.607
mnemonics (Dim: **#128**)	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
	GCN	0.659	0.658
Counting mnemonic	Sage	0.694	0.66
classes	GIN	0.701	0.673
(Dim: **#27**)	GAT	0.655	0.667
	UNet	0.66	0.654
	GCN	0.761	0.733
Semantic & counting	Sage	0.782	0.70
PCode mnemonics	GIN	0.775	0.69
(Dim: **#77**)	GAT	0.77	0.73
	UNet	0.753	0.724
	GCN	0.795	0.756
Semantic & counting	Sage	0.746	**0.761**
assembly mnemonics	GIN	**0.806**	0.716
(Dim: **#1840**)	GAT	0.801	0.733
PalmTree embeddings	UNet	0.788	0.756
	GCN	0.763	–
	Sage	0.718	–
	GIN	0.715	–
(Dim: **#128**)	GAT	0.773	–
	UNet	0.768	–

Table 4 Binary classification scores on the CFG-IR, depending on features, algorithms and datasets for -O0 optimization. - indicates that the experiments could not be completed due to the -O0 CFG-IR processing

Features	Algorithm	Balanced accuracy	
		Dataset-1	Dataset-2
Graph features &	RandomForest	0.66	–
Pcode (Dim: **# 13**)	GradientBoosting	0.678	–
TF-IDF on Pcode	RandomForest	0.708	0.561
mnemonics (Dim: **#72**)	GradientBoosting	0.742	0.619
Identity (Dim: **#1**)	GCN	0.614	–
	Sage	0.624	–
	GIN	0.618	–
	GAT	0.588	–
	UNet	0.573	–
	GCN	0.783	–
Semantic & counting	Sage	0.809	–
Pcode mnemonics	GIN	0.77	–
(Dim: **#77**)	GAT	0.768	–
	UNet	0.763	–

Table 5 Binary classification scores on the CFG, depending on features, algorithms and datasets for -O2 optimization

Features	Algorithm	Balanced accuracy	
		Dataset-1	Dataset-2
Graph features &	RandomForest	0.661	0.583
assembly (Dim: **#23**)	GradientBoosting	0.674	0.606
TF-IDF on assembly	RandomForest	0.70	0.565
mnemonics (Dim: **#128**)	GradientBoosting	0.749	0.595
Identity (Dim: **#1**)	GCN	0.581	0.547
	Sage	0.506	0.541
	GIN	0.594	0.56
	GAT	0.508	0.476
	UNet	0.589	0.575
Counting mnemonic	GCN	0.62	0.576
classes	Sage	0.615	0.57
(Dim: **#27**)	GIN	0.63	0.565
	GAT	0.613	0.585
	UNet	0.601	0.549
	GCN	0.728	0.568
Semantic & counting	Sage	0.712	0.559
PCode mnemonics	GIN	0.672	0.554
(Dim: **#77**)	GAT	0.59	0.575
	UNet	0.697	0.572
	GCN	**0.787**	0.601
Semantic & counting	Sage	0.786	**0.616**
assembly mnemonics	GIN	0.782	0.603
(Dim: **#1840**)	GAT	0.778	0.612
	UNet	0.783	0.593
	GCN	0.763	0.56
PalmTree on assembly	Sage	0.718	0.566
code (Dim: **#128**)	GIN	0.792	0.589
	GAT	0.779	0.554
	UNet	0.776	0.564

Table 6 Binary classification scores on the CFG-IR, depending on features, algorithms and datasets for -O2 optimization

Features	Algorithm	Balanced accuracy	
		Dataset-1	Dataset-2
Graph features &	RandomForest	0.631	0.568
assembly (Dim: **#13**)	GradientBoosting	0.634	**0.594**
TF-IDF on Pcode	RandomForest	0.643	0.556
mnemonics (Dim: **#72**)	GradientBoosting	0.675	0.584
Identity (Dim: **#1**)	GCN	0.579	0.534
	Sage	0.518	0.481
	GIN	0.575	0.502
	GAT	0.5	0.472
	UNet	0.5	0.5
	GCN	0.70	0.574
Semantic & counting	Sage	**0.736**	0.578
Pcode mnemonics	GIN	0.703	0.525
(Dim: **#77**)	GAT	0.689	0.549
	UNet	0.694	0.586

as there exist obfuscations that are only applied within a BB, namely a node inside the graph, without modifying the graph topology. As a consequence, the identity feature cannot capture these obfuscated patterns.

The counting mnemonic classes feature exhibits better scores but remains lower than baselines. Apart from being coarse-grained, establishing these classes depends on an expert knowledge and does not necessarily reflect the presence of obfuscated schema, as obfuscation can be performed with standard mnemonics.

Enriching GNN initial features is fundamental, as using a more precise feature directly based on the mnemonic counting, either Pcode or assembly, considerably increases balanced accuracy, around 10%. They are able to compete with classical baselines and outperform them on the *Dataset-2*. In particular, the Pcode counting shows slightly lower scores than the assembly counting feature. This behavior is explained by the fact that Pcode, by using a more concise set of mnemonics, is less capable to express specific behaviors that are diluted. Thus, its discriminative power is hindered, as its ability to distinguish obfuscated functions. Nevertheless, it is less costly to train, due to reduced features dimensions, and can be applied on any compiled binary.

Concerning PalmTree embeddings, contrary to many state-of-the-art papers on binary similarity (Massarelli et al. 2019; Gao et al. 2022; Ullah and Oh 2021), using first a model language trained on assembly binaries and initializing GNN features with the corresponding embeddings shows lower performances compared to simpler counting features, while being challenging to use, for both memory and time perspectives. As an example, several PalmTree results cannot be obtained due to an out-of-memory error for -O0 functions, that are considerably larger than -O2 functions.

Some of these results apply when using the CFG-IR, instead of the regular CFG. In particular, the underlying Pcode of the CFG-IR present lower scores, compared to their available CFG counterpart, even for the classical baselines. This difference is however not meaningful when using the counting Pcode features. This indicates that the resulting graph topology, induced by the Pcode branching, may include some noise or redundancies that cannot be compensated by basic features.

Moreover, building these CFG-IR is considerably more costly than usual CFG. In fact, one assembly instruction is translated into multiple Pcode ones. The assembly instruction that leads to the largest translation in terms of Pcode instruction in our dataset is *packuswb xmm0, xmm2* that gives 164 Pcode instructions. Among these multiple Pcode instructions, some of them are potentially branching instructions. Building the corresponding CFG-IR implies new nodes and edges, leading to a significant graph size increase. This effect is particularly pronounced for -O0 graphs that are larger that -O2. These reasons explain the sudden computational overhead, both in terms of memory and running time, for building -O2 CFG-IR graphs. Consequently, it is untracktable to process large functions compiled in -O0 for the Dataset-2, leading to missing results in Tables 4 and 8. As such, building CFG-IR graphs can only be performed at a smaller scale and would imply a significant computing power to extend it to larger graphs.

Concerning the optimization level, -O0 results present up to 10% better results than -O2. This behavior is explained by the fact that optimization can remove totally or partially some obfuscated patterns from the function. Distinguishing obfuscated functions from unobfuscated ones is consequently more difficult, especially for basic features, such as the GNN identity feature, that exhibit up to 10% of loss in terms of balanced accuracy.

Multi-class classification

In multi-class classification, the goal is to determine the type of obfuscation that has been applied to a function. In this experiment, given the dataset presented in Section "5.1", there exists 11 classes.

CFG results for -O0 and -O2 are respectively available in Tables 7 and 9, whereas Tables 8 and 10 contain classification scores for CFG-IR experiment.

These results confirm many observations that were drawn in the binary case. In particular, in general, for -O0 optimization, multi-class scores are 10% lower than in the binary case. Classical baselines perform remarkably well, given the fact there are 11 classes. As a comparison, previous works consider at most 4 classes (Jiang et al. 2021). GNNs outperform baselines, especially on *Dataset-2*, when the features are enriched with semantic information originated from a counting mnemonic vector. On the contrary, GNNs feeded with naive features, like the identity or counting mnemonic classes features, suffer from disappointing results, far from the ones obtained with more meaningful features. This difference between features is reinforced in the multi-class experiment.

Conversely, multi-class -O2 results show a high discrepancy compared to the binary case. Previously, the gap between -O0 and -O2 was limited, whereas -O2 exacerbates this variation. GNNs, even when using relevant features conveying part of the function semantics, are not able to compete with simpler baselines. This is due to the fact distinguishing between precise and sometimes subtle obfuscations when they are optimized, consequently partially removed or altered, is much more difficult.

Optimization against obfuscation

Optimization and obfuscation are deeply intertwined. If obfuscation tries to hide the program behavior by deliberately inserting complex code sequences, optimization conversely attempts to get rid of them. As a consequence, applying an obfuscation does not guarantee that the final code will be obfuscated. Many parameters influence such a

Table 7 Multi-class classification scores on the CFG, depending on features, algorithms and datasets for -O0 optimization. "-" indicates GPU Out-Of-Memory error

Features	Algorithm	Balanced accuracy	
		Dataset-1	Dataset-2
Graph features &	RandomForest	0.65	0.57
assembly (Dim: **#23**)	GradientBoosting	0.66	0.594
TF-IDF on assembly	RandomForest	0.697	0.593
mnemonics (Dim: **#128**)	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	GCN	0.431	0.462
classes	Sage	0.498	0.499
(Dim: **#27**)	GIN	0.488	0.474
	GAT	0.45	0.342
	UNet	0.439	0.448
	GCN	0.699	0.693
Semantic & counting	Sage	0.611	**0.729**
PCode mnemonics	GIN	0.706	0.71
(Dim: **#77**)	GAT	0.684	0.65
	UNet	0.704	0.627
	GCN	0.74	0.659
Semantic & counting	Sage	0.738	0.714
assembly mnemonics	GIN	**0.744**	0.69
(Dim: **#1840**)	GAT	0.733	0.723
	UNet	0.733	0.68
	GCN	0.696	–
PalmTree on assembly	Sage	0.698	–
code (Dim: **#128**)	GIN	0.693	–
	GAT	0.685	–
	UNet	0.67	–

Table 8 Multi-class classification scores on the CFG-IR, depending on features, algorithms and datasets for -O0 optimization. - indicates that the experiments could not be completed due to the -O0 CFG-IR processing

Features	Algorithm	Balanced accuracy	
		Dataset-1	Dataset-2
Graph features &	RandomForest	0.535	–
assembly (Dim: **#13**)	GradientBoosting	0.538	–
TF-IDF on Pcode	RandomForest	0.645	0.566
mnemonics (Dim: **#72**)	GradientBoosting	0.675	0.589
Identity (Dim: **#1**)	GCN	0.301	–
	Sage	0.306	–
	GIN	0.381	–
	GAT	0.16	–
	UNet	0.287	–
	GCN	0.716	–
Semantic & counting	Sage	**0.741**	–
Pcode mnemonics	GIN	0.70	–
(Dim: **#77**)	GAT	0.679	–
	UNet	0.692	–

Table 9 Multi-class classification scores for CFG, depending on features, algorithms and datasets for -O2 optimization. "-" indicates GPU Out-Of-Memory error

Features	Algorithm	Balanced accuracy	
		Dataset-1	Dataset-2
Graph features &	RandomForest	0.418	0.362
assembly (Dim: **#23**)	GradientBoosting	0.422	0.368
TF-IDF on assembly	RandomForest	0.469	**0.387**
mnemonics (Dim: **#128**)	GradientBoosting	**0.481**	0.33
Identity (Dim: **#1**)	GCN	0.246	0.177
	Sage	0.135	0.156
	GIN	0.253	0.174
	GAT	0.128	0.1
	UNet	0.245	0.198
Counting mnemonic	GCN	0.268	0.196
classes	Sage	0.28	0.249
(Dim: **#27**)	GIN	0.265	0.144
	GAT	0.261	0.221
	UNet	0.257	0.212
	GCN	0.36	0.275
Semantic & counting	Sage	0.317	0.247
PCode mnemonics	GIN	0.364	0.271
(Dim: **#77**)	GAT	0.353	0.256
	UNet	0.374	0.261
	GCN	0.38	0.327
Semantic & counting	Sage	0.386	0.269
assembly mnemonics	GIN	0.381	0.309
(Dim: **#1840**)	GAT	0.242	0.252
	UNet	0.373	0.307
	GCN	0.36	0.288
PalmTree on assembly	Sage	0.355	0.278
code (Dim: **#128**)	GIN	0.371	0.248
	GAT	0.347	0.282
	UNet	0.28	–

Table 10 Multi-class classification scores on the CFG-IR, depending on features, algorithms and datasets for -O2 optimization

Features	Algorithm	Balanced accuracy	
		Dataset-1	Dataset-2
Graph features &	RandomForest	0.374	0.319
assembly (Dim: **#13**)	GradientBoosting	0.393	0.322
TF-IDF on Pcode	RandomForest	0.402	**0.363**
mnemonics (Dim: **#72**)	GradientBoosting	**0.41**	0.307
Identity (Dim: **#1**)	GCN	0.218	0.18
	Sage	0.111	0.197
	GIN	0.244	0.158
	GAT	0.09	0.104
	UNet	0.222	0.1
	GCN	0.372	0.244
Semantic & counting	Sage	0.364	0.265
Pcode mnemonics	GIN	0.367	0.245
(Dim: **#77**)	GAT	0.148	0.236
	UNet	0.345	0.233

result: the compiler version compared to the obfuscation passes, the optimization level that is enabled, the order of the obfuscation passes among the compiler,[8] etc.

Studying the intricacies of obfuscation and optimization is challenging. Studying why the -O2 results were so different from -O0 is a first step over a better understanding of these subjects.

The -O0 optimization level corresponds to the optimization level that offers the minimal transformation of the source code. Only small optimizations, such as constant propagation, intervene, both in the compiler front-end and optimization phase. It consists in replacing values that are known constant in further variables or computation in order to limit the binary code size. Removing it is difficult and makes the final binary running considerably slower than standard -O0, which is already slow compared to the -O2 optimization level. -O2, contrary to -O0, activates various optimization passes, including inlining, and then may remove light obfuscations. For example, among the three obfuscations provided by OLLVM, CFF and BogusControlFlow are applied before any optimization, meaning that they might be further removed by the compiler.

As a consequence, the -O2 dataset we built is potentially subject to label noise, as some functions that were labeled obfuscated finally are not, due to the compiler optimization. We locate these functions, considered obfuscated, but that do not differ from their plain optimized counterparts and correct their label to unobfuscated in the dataset. 12,057, 3,013 and 3,787 functions are concerned respectively for the train, validation and test sets for the *Dataset-1* and 4,283, 1,094 and 19,395 for the *Dataset-2*.

Once *Dataset-1* and *Dataset-2* are fixed for -O2 functions, we repeat the previous experiments and quantify the performance divergence between mislabeled data and fixed dataset. Results for binary classification using the CFG and CFG-IR are available in Tables 11 and 12, whereas multi-class results are displayed in Tables 13 and 14. Fixing the mislabeled functions within the -O2 dataset has a contrasted effect. In the binary case, baselines are boosted with a 10% balanced accuracy increase, sometimes outperforming GNNs on the *Dataset-2*. GNNs results are more contrasted and this refinement seems to have little effect for models trained on the *Dataset-1*, contrary to the *Dataset-2* where the performances significantly increase. For the multi-class setting, this performance boost is even more pronounced, for any dataset and model. This illustrates the necessity to consider carefully obfuscation when optimization is involved.

Real-world examples

These two experiments aim to extend the previous classification tasks to real-world binaries. A malware and a banking application, both obfuscated, are studied. There is no information available about how they were compiled and deciding between a -O0 or -O2 model is difficult. Consequently, we analyze both the best -O0 and -O2 model, in binary and multi-class classification, privileging models trained on *Dataset-1*, as they have seen functions more diverse compared to the *Dataset-2*. Binary classification is then evaluated with a 1840-sized GIN and the refined SAGE model with 1840 as input dimension for -O2. For multi-class classification, the evaluated models are again a 1840-sized GIN and a refined Gradient Boosting feeded with a assembly mnemonic TF-IDF.

[8] if the obfuscator directly interfaces with the compiler, such as OLLVM.

Table 11 Binary classification scores on the CFG, depending on features, algorithms for -O2 optimization, on a refined dataset

Features	Algorithm	Balanced accuracy	
		Dataset-1	Dataset-2
Graph features &	RandomForest	0.723	0.693
assembly (Dim: **#23**)	GradientBoosting	0.741	0.679
TF-IDF on assembly	RandomForest	0.788	**0.722**
mnemonics (Dim: **#128**)	GradientBoosting	0.791	0.688
Identity (Dim: **#1**)	GCN	0.604	0.524
	Sage	0.545	0.511
	GIN	0.635	0.552
	GAT	0.524	0.528
	UNet	0.579	0.5
Counting mnemonic	GCN	0.646	0.5
classes	Sage	0.646	0.582
(Dim: **#27**)	GIN	0.657	0.5
	GAT	0.65	0.558
	UNet	0.537	0.602
	GCN	0.786	0.533
Semantic & counting	Sage	0.778	0.59
PCode mnemonics	GIN	0.762	0.567
(Dim: **#77**)	GAT	0.756	0.612
	UNet	0.692	0.546
	GCN	0.734	0.608
Semantic & counting	Sage	**0.793**	0.689
assembly mnemonics	GIN	0.781	0.655
(Dim: **#1840**)	GAT	0.776	0.65
	UNet	0.773	0.623
	GCN	0.759	0.588
PalmTree on assembly	Sage	0.752	0.56
code (Dim: **#128**)	GIN	0.776	0.586
	GAT	0.772	0.528
	UNet	0.764	0.63

Table 12 Binary classification scores on the CFG-IR, depending on features, algorithms for -O2 optimization, on a refined dataset

Features	Algorithm	Balanced accuracy	
		Dataset-1	Dataset-2
Graph features &	RandomForest	0.697	0.638
assembly (Dim: **#13**)	GradientBoosting	0.676	0.627
TF-IDF on Pcode	RandomForest	0.737	**0.702**
mnemonics (Dim: **#72**)	GradientBoosting	0.748	0.677
Identity (Dim: **#1**)	GCN	0.588	0.524
	Sage	0.496	0.493
	GIN	0.645	0.565
	GAT	0.529	0.523
	UNet	0.583	0.506
	GCN	0.771	0.566
Semantic & counting	Sage	0.685	0.547
Pcode mnemonics	GIN	**0.772**	0.575
(Dim: **#77**)	GAT	0.769	0.51
	UNet	0.747	0.525

Table 13 Multi-class classification scores for CFG, depending on features, algorithms and datasets for -O2 optimization, on a refined dataset. - indicates OOM error

Features	Algorithm	Balanced accuracy	
		Dataset-1	Dataset-2
Graph features &	RandomForest	0.553	0.378
assembly (Dim: **#23**)	GradientBoosting	0.597	0.389
TF-IDF on assembly	RandomForest	0.663	0.441
mnemonics (Dim: **#128**)	GradientBoosting	**0.685**	**0.452**
Identity (Dim: **#1**)	GCN	0.263	0.226
	Sage	0.245	0.267
	GIN	0.227	0.185
	GAT	0.2	0.194
	UNet	0.262	0.234
Counting mnemonic	GCN	0.283	0.287
classes	Sage	0.319	0.244
(Dim: **#27**)	GIN	0.285	0.123
	GAT	0.293	0.193
	UNet	0.293	0.25
	GCN	0.451	0.331
Semantic & counting	Sage	0.455	0.319
PCode mnemonics	GIN	0.443	0.334
(Dim: **#77**)	GAT	0.391	0.326
	UNet	0.434	0.354
	GCN	0.529	0.435
Semantic & counting	Sage	0.46	0.406
assembly mnemonics	GIN	0.484	0.225
(Dim: **#1840**)	GAT	0.509	0.404
	UNet	0.521	0.411
	GCN	0.507	0.384
PalmTree on assembly	Sage	0.499	0.395
code (Dim: **#128**)	GIN	0.495	0.239
	GAT	0.487	0.336
	UNet	0.444	0.34

Table 14 Multi-class classification scores on the CFG-IR, depending on features, algorithms and datasets for -O2 optimization

Features	Algorithm	Balanced accuracy	
		Dataset-1	Dataset-2
Graph features &	RandomForest	0.478	0.315
assembly (Dim: **#13**)	GradientBoosting	0.484	0.312
TF-IDF on Pcode	RandomForest	0.561	**0.393**
mnemonics (Dim: **#72**)	GradientBoosting	**0.566**	**0.393**
Identity (Dim: **#1**)	GCN	0.25	0.225
	Sage	0.27	0.24
	GIN	0.286	0.235
	GAT	0.206	0.183
	UNet	0.195	0.13
	GCN	0.465	0.349
Semantic & counting	Sage	0.359	0.331
Pcode mnemonics	GIN	0.40	0.302
(Dim: **#77**)	GAT	0.399	0.242
	UNet	0.433	0.262

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 (Bardin et al. 2017). Then, locating and determining the obfuscation type is necessary before any deobfuscation attempt. These tasks are performed on two XTunnel obfuscated samples,[9] having respectively 3,693 and 3,982 functions. Results are compared with a previously built ground truth, that was computed using an approach based on symbolic execution (Bardin et al. 2017). Such a ground truth asserts with a satisfactory confidence that these samples were heavily obfuscated using OpaquePredicates. 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 15. By considering a -O2 model, the binary balanced accuracy is disappointing, whereas the multi-class one is more satisfactory. The -O0 model achieves decent scores, close to the ones obtained in the previous experiments in Sections "6" and "7". This does not imply that the XTunnel samples were compiled with -O0, just that the -O0 model recognizes obfuscation better than the -O2 one on these two samples.

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 that turns out to always evaluate to the same boolean value, making the function to always branch to the same next BB. Consequently, our models can still detect a suspicious pattern related to OpaquePredicates and we consider these predictions as valid.

This shows that these detection and characterization methods scale well on real-world malware.

Rabobank

Rabobank is a financial institution that is based in Netherlands. An Android application is available for customers, that is regularly updated. The version 35.0 was released in March 2024. Apart from the Android bytecode, it contains several native binaries, that are heavily obfuscated. The *libnative-lib.35-0.so* binaries is particularly interesting: it contains only 172 functions and a preliminary manual analysis quickly reveals that most of them are obfuscated. For each function, we try our best to build a reliable ground truth. This task is especially challenging in this example, as multiple obfuscations are combined within the same function. The previous multi-class framework, detailed in Section "7", is not adapted for multi-label task. Multi-label in a multi-class setting, with

Table 15 Obfuscation detection results on two XTunnel samples for both -O0/ -O2

	Binary balanced accuracy	Multi-class balanced accuracy
Sample C637E	0.737/ 0.357	0.657 / 0.559
Sample 99B45	0.734 / 0.369	0.576 / 0.55

[9] Their corresponding hashes are C637E01F50F5FBD2160B191F6371C5DE2AC56DE4 and 99B454262DC-26B081600E844371982A49D334E5E.

Table 16 Obfuscation detection results on the Rabobank—35.0 *libnative*

	Binary balanced accuracy	Multi-class balanced accuracy
Rabonbank—35.0	0.921 / 0.768	0.017 / 0.319

a high number of classes, is little addressed in literature (Tarekegn et al. 2024) and particularly challenging.

In this example, for a given function obfuscated with multiple passes, any single obfuscation that is part of the applied obfuscation passes is considered as a correct label for the function. Such a labeling is limited as ideally, a reverse engineer would need to predict all the obfuscation passes that are applied on a function, not a single one.

Functions obfuscated with CFF are easily recognized. However, it is difficult to distinguish between MBA and OpaquePredicates, even from dead code insertion (that is no part of our dataset). As a consequence, we only establish a fixed ground-truth for functions for which we are certain about the obfuscation that was applied. The rest is discarded, resulting in 118 functions used for final classification.

Results are available in Table 16. Binary scores, both in -O0 and -O2, are satisfactory and aligned with the previous reported scores in Section "6". For the multi-class setting, results are disappointing, especially in -O0. This shows how difficult it is for a model to determine a class of obfuscation when multiple passes are applied on the same time, making the model choosing either one of the passes and most of the time, completely choose another pass.

Conclusion

To conclude, this work provides a general study about obfuscation detection for both binary and multi-class settings. It demonstrates the efficiency and the robustness of standard baselines, that achieve satisfactory performances with simple and fast models. They however cannot compete with GNNs that, combined with meaningful features conveying part of function semantics, present higher scores and a better generalization power, especially in a context where test data can be far from training data. The superiority of GNN is assessed only if the initial node features correctly describe the BBs as a GNN with poor quality features is considerably less efficient than baselines. In particular, Pcode and assembly mnemonics are well adapted to be used in a feature. The first one is CPU-agnostic, memory and time saving while being less efficient that the second one, as it is less discriminative than assembly mnemonics. In particular, using Pcode is useful to build a CFG representation completely independent from the architecture, even though the corresponding obfuscation detection scores are slightly lower than the regular CFG. These results are valid for the binary and multi-class settings. The only setting where all the tested algorithms in general exhibit disappointing performances are the -O2 scenario, where the obfuscation are harder to distinguished because of the subsequent optimization applied on the code. In this context, baselines are the ones that present more robustness than GNN. Finally, these results are confirmed with a two real-world examples.

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 optimization 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 (Marcelli et al. 2022) faces the same challenge, leading to the development of new graphs, such as SOG (Semantic Oriented Graph) (He et al. 2024) 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.

Author Contributions
R.C drafted the main manuscript. R.D, F.Y, and F.R contributed to the overall conceptual design of the paper and provided substantial revisions and suggestions. All authors reviewed and approved the final version of the manuscript.

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

Data availability
The dataset used in this work is available at https://github.com/quarkslab/diffing_obfuscation_dataset

Declarations

Conflict of interest
The authors declare no Conflict of interest.

Code availability
The artifacts related to this work are available at https://github.com/quarkslab/obfuscation_benchmark_code_artifacts

Received: 31 March 2025 / Accepted: 20 August 2025
Published online: 30 September 2025

References

Akiba T, Sano S, Yanase T, Ohta T, Koyama M (2019) Optuna: a next-generation hyperparameter optimization framework. In: Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining, pp 2623–2631
Bardin S, David R, Marion J-Y (2017) Backward-bounded dse: targeting infeasibility questions on obfuscated codes. In: 2017 IEEE symposium on security and privacy (SP), pp 633–651 . https://doi.org/10.1109/SP.2017.36
Blazytko T (2021) Automated detection of obfuscated code. https://synthesis.to/2021/08/10/obfuscation_detection.html. Accessed 20 Nov 2023
Blazytko T (2021) Statistical analysis to detect uncommon code. https://synthesis.to/2021/03/03/flattening_detection.html. Accessed 20 Nov 2023
Blazytko T (2023) Statistical analysis to detect uncommon code. https://synthesis.to/2023/01/26/uncommon_instruction_sequences.html. Accessed 20 Nov 2023
Borgerson M Pypcode. https://docs.angr.io/projects/pypcode/en/latest/. Accessed 05 Aug 2024
Cohen R, David R, Mori R, Yger F, Rossi F (2025) Experimental study of binary diffing resilience on obfuscated programs. In: International conference on detection of intrusions and malware, and vulnerability assessment. Springer
Collberg C The Tigress C obfuscator. https://tigress.wtf/index.html. Accessed 17 Aug 2023
David R, Coniglio L, Ceccato M (2020) Qsynth-a program synthesis based approach for binary code deobfuscation. In: BAR 2020 workshop
Errica F, Podda M, Bacciu D, Micheli A (2020) A fair comparison of graph neural networks for graph classification. In: International conference on learning representations. https://openreview.net/forum?id=HygDF6NFPB
Ferrari Dacrema M, Cremonesi P, Jannach D (2019) 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. https://doi.org/10.1145/3298689.3347058
Fey M, Lenssen JE (2019) Fast graph representation learning with PyTorch Geometric. In: ICLR workshop on representation learning on graphs and manifolds
Gao H, Zhang T, Chen S, Wang L, Yu F (2022) Fusion: measuring binary function similarity with code-specific embedding and order-sensitive gnn. Symmetry 14(12):2549

Gao H, Ji S (2019) Graph u-nets. In: Chaudhuri K, Salakhutdinov R (eds.) Proceedings of the 36th international conference on machine learning. Proceedings of machine learning research, vol 97, pp 2083–2092. PMLR. https://proceedings.mlr.press/v97/gao19a.html
Gori M, Monfardini G, Scarselli F (2005) A new model for learning in graph domains. In: Proceedings. 2005 IEEE international joint conference on neural networks, vol 2, pp 729–7342 . https://doi.org/10.1109/IJCNN.2005.1555942
Greco C, Ianni M, Guzzo A, Fortino G (2023) Explaining binary obfuscation, pp 22–27 . https://doi.org/10.1109/CSR57506.2023.10224825
Hamilton WL, Ying R, Leskovec J (2017) Inductive representation learning on large graphs. In: Proceedings of the 31st international conference on neural information processing systems. NIPS'17, pp. 1025–1035. Curran Associates Inc., Red Hook, NY, USA
He H, Lin X, Weng Z, Zhao R, Gan S, Chen L, Ji Y, Wang J, Xue Z (2024) Code is not natural language: unlock the power of semantics-oriented graph representation for binary code similarity detection. In: 33rd USENIX security symposium (USENIX Security 24), PHILADELPHIA, PA
Jiang S, Hong Y, Fu C, Qian Y, Han L (2021) Function-level obfuscation detection method based on graph convolutional networks. J Inf Secur Appl 61:102953. https://doi.org/10.1016/j.jisa.2021.102953
Junod P, Rinaldini J, Wehrli J, Michielin J (2015) Obfuscator-llvm–software protection for the masses. In: Wyseur, B. (ed.) Proceedings of the IEEE/ACM 1st international workshop on software protection, SPRO'15, Firenze, Italy, May 19th, 2015, pp 3–9. IEEE. https://doi.org/10.1109/SPRO.2015.10
Kanzaki Y, Monden A, Collberg C (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, pp 31–37 . IEEE
Kipf TN, Welling M (2017) Semi-supervised classification with graph convolutional networks. In: International conference on learning representations. https://openreview.net/forum?id=SJU4ayYgl
Kriege NM, Johansson FD, Morris C (2020) A survey on graph kernels. Appl Netw Sci 5(1):6. https://doi.org/10.1007/s41109-019-0195-3
Li X, Qu Y, Yin H (2021) Palmtree: Learning an assembly language model for instruction embedding. In: Proceedings of the 2021 ACM SIGSAC conference on computer and communications security. CCS '21, pp 3236–3251. Association for Computing Machinery, New York, NY, USA . https://doi.org/10.1145/3460120.3484587
Marcelli A, Graziano M, Ugarte-Pedrero X, Fratantonio Y, Mansouri M, Balzarotti D (2022) How machine learning is solving the binary function similarity problem. In: 31st USENIX security symposium (USENIX Security 22), pp 2099–2116
Massarelli L, Di Luna GA, Petroni F, Querzoni L, Baldoni R et al (2019) Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. In: Proceedings of the 2nd workshop on binary analysis research (BAR), pp 1–11
Nagra J, Collberg C (2009) Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection: obfuscation, watermarking, and tamperproofing for software protection. Pearson Education, London
Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V et al (2011) Scikit-learn: machine learning in python. J Mach Learn Res 12:2825–2830
Quarkslab: obfuscation dataset. https://github.com/quarkslab/diffing_obfuscation_dataset. Accessed 09 Jan 2024
Raja V, Fernandes KJ (2007) Reverse engineering: an industrial perspective. Springer, Berlin
Salem A, Banescu S (2016) Metadata recovery from obfuscated programs using machine learning. In: Proceedings of the 6th workshop on software security, protection, and reverse engineering, pp 1–11
Salwan J, Bardin S, Potet M-L (2018) Symbolic deobfuscation: from virtualized code back to the original. In: International conference on detection of intrusions and malware, and vulnerability assessment. Springer, pp 372–392
Schrittwieser S, Wimmer E, Mallinger K, Kochberger P, Lawitschka C, Raubitzek S, Weippl ER (2023) Modeling obfuscation stealth through code complexity. In: European symposium on research in computer security. Springer, pp 392–408
Sharif MI, Lanzi A, Giffin JT, Lee W (2008) Impeding malware analysis using conditional code obfuscation. In: NDSS
Tarekegn AN, Ullah M, Cheikh FA (2024) Deep learning for multi-label learning: a comprehensive survey . https://arxiv.org/abs/2401.16549
Tofighi-Shirazi R, Săvoae IM, Elbaz-Vincent P (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. SSPREW9 '19. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3371307.3371313
Tofighi-Shirazi R, Asavoae I-M, Elbaz-Vincent P, Le T-H (2019) Defeating opaque predicates statically through machine learning and binary analysis. In: Proceedings of the 3rd ACM workshop on software protection, pp 3–14
Ullah S, Oh H (2021) Bindiff nn: learning distributed representation of assembly for robust binary diffing against semantic differences. IEEE Trans Softw Eng 48(9):3442–3466
Velčković P, Cucurull G, Casanova A, Romero A, Liò P, Bengio Y (2018) Graph attention networks. In: International conference on learning representations . https://openreview.net/forum?id=rJXMpikCZ
Wang C (2001) A security architecture for survivability mechanisms. University of Virginia, Virginia
Wu Z, Pan S, Chen F, Long G, Zhang C, Yu P (2021) A comprehensive survey on graph neural networks. IEEE Trans Neural Netw Learn Syst 32(1):4–24. https://doi.org/10.1109/TNNLS.2020.2978386
Xu K, Hu W, Leskovec J, Jegelka S (2019) How powerful are graph neural networks? In: International conference on learning representations. https://openreview.net/forum?id=ryGs6iA5Km
Yadegari B, Johannesmeyer B, Whitely B, Debray S (2015) A generic approach to automatic deobfuscation of executable code. In: 2015 IEEE symposium on security and privacy, pp 674–691 . https://doi.org/10.1109/SP.2015.47
Zhou Y, Main A, Gu YX, Johnson H (2007) Information hiding in software with mixed boolean-arithmetic transforms. In: International workshop on information security applications. Springer, pp 61–75

Publisher's Note