Experimental Study of Binary Diffing Resilience on Obfuscated Programs

Roxane Cohen[1,2], Robin David[1], Riccardo Mori[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. Obfuscation is commonly employed to protect sensitive program assets in legitimate use cases or to conceal malicious behavior in the context of malware. By altering the binary code of a compiled program, obfuscation disrupts binary analysis techniques, such as binary diffing or similarity. However, there is little comprehensive academic research addressing the effects of obfuscation on binary analysis tools and quantifying its impact. In this study, we examine how different types of obfuscation influence binary diffing algorithms. Specifically, we demonstrate a clear relationship between the type of obfuscation and the performance of the diffing algorithms used. Our benchmarks emphasize that, contrary to common assumptions, intra-procedural and data obfuscations have a limited impact on binary diffing when applied alone. In contrast, inter-procedural obfuscations significantly affect the diffing process, degrading performances by up to 40 f1-score points when comparing low and high obfuscation levels. These results highlight the need for modular diffing approaches, where parameters and features can be fine-tuned to handle adversarial scenarios, such as obfuscation. To support this research, we have released a comprehensive dataset comprising pairs of clear and obfuscated compiled programs, along with metadata specifying the type and exact location of each obfuscation. This dataset is intended to facilitate further research in this area.

Keywords: Obfuscation; Binary Diffing; Machine Learning; Binary Similarity; Malware

1 Introduction

Binary diffing is essential for reverse engineering. It is used for malware diffing [33], patch analysis [43], program similarity [10], backdoor detection, anti-plagiarism and to detect statically linked libraries. It consists in identifying similarities between two binaries, typically at the function level. Most binary differs use the disassembly to match the corresponding functions. Likewise, efficient solutions have been proposed to tackle the sub-problem of binary similarity on

standard binaries in a cross-architecture, cross-compiler or cross-optimization setting [25].
However, diffing obfuscated binaries remains an open research question. Static or dynamic obfuscation consists in hiding the true behavior of a program and its syntactical representation, without modifying its semantics. It is widely used to protect application algorithms, data, and more generally, program assets. For example, `MBA (Mixed Boolean Arithmetic)` [44] replaces simple arithmetic operations by complex but strictly equivalent ones. As a consequence, obfuscation may alter binary code and modify features used by binary differs to match functions and more generally to compare programs. The assembly instructions, function data or execution flow, and the relationships between the functions can both be affected. Therefore, current binary differs and similarity tools may not be adapted to this adversarial context, leading to degraded performances. Such a case may occur if only one of the binaries is obfuscated, or if both are.

Contributions. This paper provides the first thorough experimental study of binary diffing in the presence of obfuscated binaries. Our study covers a large panel of state-of-the-art differs and similarity tools. We show how each of them is affected by different obfuscation types, pointing out that intra-procedural and data obfuscations have a limited impact on the diffing results. Contrarily, inter-procedural obfuscation significantly limits differ's efficiency, with a decrease up to 40 f1-score points between obfuscation levels, as most diffing algorithms rely on the CG (Call Graph) to perform matches. Results are valuable from both reverse engineering and software protection perspectives. A reverser may take advantage of existing modular differs and prior knowledge about the applied obfuscation in order to obtain a more resilient binary diffing result. Conversely, inter-procedural obfuscation should be privileged for software protection purposes, as it impedes most differ's work. Our experiments are based on two real-world datasets, the state-of-the-art BinKit dataset [19] and a new dataset dedicated to diffing and obfuscation [5], containing more than 6,700 binaries obfuscated with two obfuscators and more than 10 obfuscation passes. A real-world example of the X-Tunnel malware illustrates the potential of such a robust diffing approach and shows its practical usefulness.

This paper is structured as follows: Section 2 provides an overview of key concepts and related works. The details of our experiments, including the dataset, experimental setup, and the binary diffing and similarity tools used, are outlined in Section 3. Section 4 discusses how binary diffing can be leveraged to enhance resilience against obfuscation. The two binary diffing experiments, focusing on cases where either one or both binaries are obfuscated, are presented in Sections 5 and 6, respectively. BinKit's results are discussed in Section 7. In Section 8, we showcase a real-world example with the XTunnel malware, followed by a discussion in Section 9 and concluding remarks in Section 10.

[5] `https://github.com/quarkslab/diffing_obfuscation_dataset`

2 Background and related works

2.1 Program representation

Programs are often represented by their disassembly, with functions modeled as CFG (Control Flow Graph), where nodes are basic-blocks (sequences of instructions without branching). A CFG encodes the intra-procedural execution flow, while the CG (Call Graph) represents inter-procedural call relationships. Both compilation or obfuscation can significantly alter these graphs. For instance, inlining affects the CG while loop unrolling impacts the CFG.

2.2 Binary diffing and similarity

Binary diffing usually operates at the function level and outputs a correspondence between the functions of two programs, denoted as primary and secondary. Binary diffing can be formalized as a one-to-one assignment $\phi : (\mathcal{P}, \mathcal{S}) \longmapsto \rho$ where $\mathcal{P}$ is the primary function set, $\mathcal{S}$ is the secondary function set. Then, $\rho : \mathcal{P} \longrightarrow \mathcal{S}$ is an assignment function both partial and injective, so that each function of $\mathcal{P}$ should be matched to at most one function of $\mathcal{S}$. This diffing is performed without having access to sources and symbols. Other diffing definitions consider different program granularities, such as basic block like DeepBinDiff[10], or one-to-many assignment [20]. Two programs do not need to be semantically equivalent to be efficiently diffed and small changes or light patches, induced by program versioning or compilation difference, can be used to analyze updates between binaries. BinDiff [12,11] and Diaphora [20] are the most widely used binary differs [6]. BinDiff starts by matching known imported functions and propagates these matches to neighboring functions using the CG, discriminating between neighbors based on function similarity. Diaphora establishes a set of heuristics, from confident to unreliable, used to iteratively match functions. The best heuristics are related to the function address and name, its assembly and pseudo-code and the function hash, with less attention dedicated to CG matching and other similarity measures. Both differs require disassembly as a starting point for diffing.

The **binary similarity** problem, closely related to binary diffing, is applied in order to find the most similar function to f inside a pool of candidate functions. It is an active research field relying heavily on ML (Machine Learning) and is particularly used for vulnerability search and malware analysis. Binary similarity tools either use precomputed assembly-based features or learn them with DL (Deep Learning), such as GNN (Graph Neural Networks). In the former category, TIKNIB [19] computes similarity scores using a specific distance combining various handcrafted features, BinShape [36] starts by extracting features and sorts them to obtain the top-ranked ones that are given to a decision tree. DL techniques tend to dominate the research field and are inspired from NLP (Natural Language Processing): Asm2Vec [9] is based on a refined and improved version of the word2vec model [31]. Trex and JTrans [35,41] are directly

[6] according to popularity metrics (Github downloads, stars, Google Trend).

inspired by the recent success of transformers for large language models. GNN are also gaining more and more popularity as the latest research articles mostly use increasingly complex GNN: current approaches use a pretrained language model on assembly instructions as initialization for further GNN embeddings [26]. GMN (Graph Matching Networks) [23] is the first work that jointly learns graph embeddings on similar graph pairs rather than independent embeddings. The idea is further developed with refined GNN architectures or language models [39,13]. These models are subject to adversarial attacks [3].
Binary similarity and diffing share common aspects but have distinct goals. Similarity algorithms output scores between function pairs, while diffing finds an assignment between functions in two binaries, often using similarity scores to establish these matches. Thus, similarity approaches when combined with a matching algorithm can yield diffing results.

2.3 Obfuscation

Obfuscation enhances code security against reverse engineering. Native code obfuscation (C, C++) can be source-to-source, like Tigress [6], or integrated into compiler toolchains, like OLLVM [18]. Research has explored deobfuscation [7,38], obfuscation detection [14], and diversification [16] which goal is to produce variants that cannot be linked with each other. Static obfuscation protects from static analysis by altering program layout while dynamic obfuscation protects from runtime analysis. Different obfuscation passes have specific effects on programs [34]:

- Data obfuscation alters the data-flow, e.g., hiding a XOR operation.
- Control flow obfuscation blurs the program execution flow logic. It can be divided into :
  - **Intra-procedural** obfuscation mutates the CFG structure. For example, `CFF (CFG Flattening)` puts every basic block at the same level and uses a dispatcher to maintain the function logic [40].
  - **Inter-procedural** obfuscation alters the CG by modifying the relationships between function callers and callees. It is damaging as CG has been demonstrated essential for program analysis [21] (e.g, a `Merge` pass fusions two functions).

While binary diffing, similarity, and obfuscation have been studied separately, no research tackles the problem of diffing obfuscated code. Although some binary similarity techniques include small obfuscated experiments, they are mostly limited to intra-procedural or data obfuscations from OLLVM, lacking generalization across various obfuscations and obfuscators [9,19,36,35,17]. The limitations of intra-procedural obfuscation in a diffing context have been demonstrated [42], and inter-procedural obfuscation has only been studied using a symbolic execution approach [24].
Several reverse engineering use cases can benefit from efficient diffing of obfuscated binaries. Specifically, knowledge transfer refers to the ability to use insights

gained from analyzing one binary to infer information about its subsequent, potentially obfuscated versions. This concept defines a threat model in which an attacker can leverage knowledge from one binary to circumvent the obfuscation applied to a new version. Such an approach has been applied to analyze obfuscated Android bytecode [8]. We denote this framework, where a plain binary (unobfuscated) is diffed with an obfuscated counterpart, as `plain-obfuscated` whereas the `obfuscated-obfuscated` setting indicates that two different obfuscated variants are compared. Conditions to operate such a diffing are commonly encountered, especially for programs that tend to be frequently updated as it is the case for Android applications. Similarly, malware are iteratively modified and improved accross attack campaigns. Finding plain and obfuscated variants is a common use-case [1].

3 Experimental framework

This section outlines the creation of our reference dataset and the experimental settings used for evaluation. The dataset was designed to address the limitations of the existing BinKit dataset [19] (see also Section 7).

3.1 Dataset

Native code obfuscation is challenging to work with as there exist few obfuscators that are free or open-source. Those that are accessible only offer a limited number of obfuscations. For example, the latest official OLLVM [18], based on LLVM-4, exclusively provides intra-procedural obfuscations: `BogusControlflow`, `InstructionSubstitution` and `CFF`. Moreover, most binaries used in the literature are basic code snippets, such as sorting algorithms, and do not represent the complexity of true C projects. To fill the gap, and inspired by the state-of-the-art dataset for binary similarity research [25], we created a large dataset of realistic obfuscated binaries, with both plain binaries, obfuscated versions, and associated ground truths.
Five realistic projects written in C are used: `zlib, lz4, minilua, sqlite` and `freetype`. Obfuscation is applied using Tigress-3.1 [6] and OLLVM [18]. For each obfuscator, various obfuscation passes are selected: if possible, inter-procedural, intra-procedural and data. Because combining passes is essential to enhance security, several obfuscation sequences are created: `Mix` (`CFF` and `EncodeArithmetic` and `OpaquePredicates`)[7], and `Mix + Split` when possible. The latter scheme represents a real-world scenario where all aspects (assembly, CFG, and CG) are altered, making the task of an (adversary) differ challenging. For each project and obfuscation pass, we iteratively obfuscate from 10% to 100% of available functions using a 10% incremental step to observe how differs behave with varying levels of obfuscation. Given the inherent randomness in most obfuscation

[7] Obfuscation pass names are unified as they perform the same modifications but may have been implemented differently, leading to similar passes that can be more or less virulent. See Table 7 for details.

	Passes	Pass type	zlib	lz4	minilua	sqlite	freetype
Tigress	Copy	Inter	✔	✔	✔	✔	✔
	Split	Inter	✔	✔	✔	✔	✔
	Merge	Inter	✔	✔	✗	✗	∼
	CFF	Intra	✔	✔	✔	✔	✔
	Virtualize	Intra	✔	✔	∼	∼	✗
	Opaque	Intra	✔	✔	✔	✗	∼
	EncodeArithmetic (Enc.A)	Data	✔	✔	✔	✔	✔
	EncodeLiterals (Enc.L)	Data	✔	✔	✔	✔	✔
	Mix	Intra & Data	✔	✔	✔	∼	∼
	Mix + Split	All	✔	✔	✔	∼	∼
OLLVM-14	CFF	Intra	✔	✔	✔	✔	✔
	Opaque	Intra	✔	✔	✔	✔	✔
	EncodeArithmetic (Enc.A)	Data	✔	✔	✔	✔	✔
	Mix	Intra & Data	✔	✔	✔	✔	✔

Table 1: Obfuscated dataset sum-up. ✔ compilation success, ✗ no binaries, ∼ some binaries are not available *(depends on obfuscation level or random seed used)*

techniques, this process is repeated 10 times. The resulting dataset, summarized in Table 1, contains 6,718 binaries and 8,910,962 functions, making it the first freely available dataset of realistic obfuscated binaries at this scale.
The binaries are compiled with -O0 and -O2 optimizations for the x86-64 architecture. The -O0 optimization prevents most compiler effects that could attenuate obfuscation, such as inlining obfuscated functions or removing MBA. This allows us to study obfuscation passes in their unaltered form. Conversely, -O2 binaries are more representative of real-world scenarios, despite potential biases. The goal is to demonstrate that diffing obfuscated binaries works well both in a controlled, bias-free environment and on real-world binaries. Using OLLVM and Tigress-3.1 introduces several constraints. OLLVM, originally dependent on LLVM-4, was ported to LLVM-14 for this work and is referred to as OLLVM-14. It offers only three intra-procedural obfuscations. Tigress-3.1 requires amalgamated C files, as its cilly-merge functionality is unreliable to be used in practice. This constraint influenced project choices, limiting them to `zlib`, `lz4`, `minilua`, `sqlite`, and `freetype`, which are available in amalgamated form. Some Tigress binaries could not be produced due to internal errors or compilation issues with GCC 12. Default parameters are used for Tigress, except for the `Split` obfuscation where SplitCount is set to 2, and only top, block, and deep are allowed for SplitKinds. We limit our tests to these parameters as the resulting dataset already contains 6,700 binaries.

3.2 Experimental settings

To support these experiments, we use IDAPro-8.1, assuming it produces a proper disassembly with a correct functions recovery. Most binary differs and binary similarity tools also make this assumption as a basis for further analysis and tackling disassembly issues is out of scope of this research. Various binary differs are selected: BinDiff [12,11] and Diaphora-3.0.0 [20]. Both are tested with default parameters. QBinDiff-1.2.0 [29,30], which is a modular differ that can

be fine-tuned depending on the user goal, is also evaluated with two different configurations, explained in Section 4. Moreover, several binary similarity tools are studied: GMN [23] is considered as state-of-the-art [25] as it outperforms other similarity tools, such as SAFE [27]. Asm2Vec [9], proven to be robust against some obfuscations, and JTrans [41], which has shown promising results, are also considered. GMN is trained to output similarity scores on *Dataset-1A* [25], which contains standard binaries, with the original GMN hyperparameters and attributes for training. The same holds for JTrans and Asm2Vec, except for Asm2vec for which the number of random walks is set to 3 and that assembly instructions are normalized. All these binary similarity approaches are combined with the matching HA (Hungarian Algorithm) [22], with a cost matrix built over embeddings, in order to obtain a proper diffing output, from their similarity scores. We use the same Euclidean distance for all the binary similarity tools, like in the GMN framework [23]. Evaluating a differ efficiency requires to compare its results to the expected matching, called ground truth. Such a ground truth is built as follows:

- Data and intra-procedural obfuscations are applied within the function scope. These techniques will preserve the function name.
- Inter-procedural obfuscation alters the CG structure. Each type of obfuscation has a specific ground truth. For example:
  - A function f in primary split into f_1 and f_2 in the secondary should be matched with both f_1 and f_2.
  - Functions f_1 and f_2 merged into a single function f in the secondary should be matched with both f_1 and f_2 in the primary.
  - A function f cloned into f_1 in the secondary should be matched with both f and f_1 in the secondary.

  This process can be extended to any number of split, merged, or cloned functions.

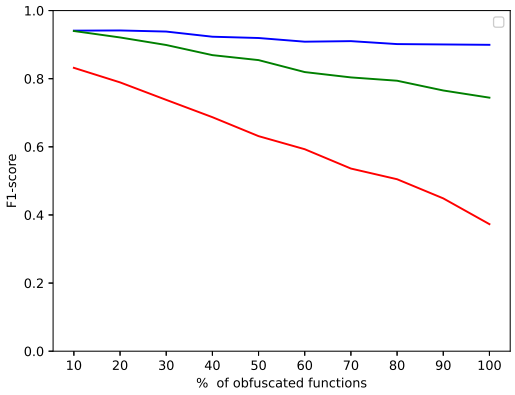
The ground-truth is defined at the function level since both Tigress and OLLVM can target specific functions, preserving the original mappings. Function-level granularity is necessary because basic-block-level ground truth cannot be computed for several obfuscations, such as `Virtualization`, which significantly alters a function's control flow. Maintaining correspondence at the basic-block level becomes impractical. As a result, we focus exclusively on function-level differs, excluding basic-block-level tools like DeepBinDiff [10] or BinSim [32], where accurate ground-truth cannot be established.
This inter-procedural ground-truth penalizes one-to-one diffing methods, as differs recover at most one correct match (either none or one), thus limiting the recall. All diffing and similarity tools evaluated in this paper are subjected to this penalty. After determining matches based on function names, the binaries are stripped to remove symbols, including function names. To compare the ground truth assignments with the diffing results, three standard metrics are considered: recall (R), precision (P), and f1-score, defined as:

$$P = \frac{TP}{TP+FP} \qquad R = \frac{TP}{TP+FN} \qquad \text{f1-score} = \frac{2\times P\times R}{P+R}$$

with *TP* denoting True Positive, *FP* False Positive and *FN* False Negative.

Precision denotes how many retrieved items are relevant whereas the recall indicates how many relevant items are retrieved. Maximizing the f1-score requires both the recall and precision to be high.[8] Notice that even though similarity scores may decrease, due to small patches or differences, differs may still find the correct matches. Two f1-scores can be computed: a general f1-score applied on the whole binary, whereas an obfuscated f1-score is limited to obfuscated functions only. The higher these f1-scores, the better the differ is. If a significant difference exists between the general and obfuscated f1-scores, it implies that binary differs are correctly able to match regular functions but that they face trouble to do the same for obfuscated functions.
Finally, experiments were conducted on a Debian-6.1.27-1 with an Intel Xeon E3-12xx v2, 20 cores and 70GB of RAM. Binary similarity experiments require a GPU and were launched with a Nvidia RTX A6000.

4 Resilient diffing



■ stable features (QBinDiff$_s$) ■ all available features (QBinDiff) ■ unstable features.

Fig. 1: Obfuscated `zlib` (Tigress CFF). QBinDiff f1-scores with respect to the % of obfuscated functions.

This Section illustrates how to obtain an obfuscation-resilient binary diffing on a concrete example. Extended results are available in Section 5 and 6. Existing differs have not been designed to handle obfuscated binaries. Besides changing function, basic-block order strategies, BinDiff cannot be adjusted. Similarly, tuning Diaphora for obfuscation purpose is challenging due to the multitude of heuristics, some considered reliable and others not, for which there is no clear

[8] The exact matching is known and using precision, recall and f1-score is consequently accurate. Binary similarity tools often rely on precision@k or recall@k as they search the most similar functions inside a pool of size k.

analysis of the order in which they should be applied and their influence on the final diffing process. QBinDiff was designed to be modular in order to leverage binary diffing for specific use cases. It enables the precise selection of features used to compute the similarity, which is then used to identify matches [4]. This modularity helps to obtain more accurate diffing results confronted with obfuscation, especially if its type is known. Such a knowledge can be obtained manually with reverse-engineering as some obfuscations, like `CFF` or `MBA`, exhibit very specific patterns, or with machine learning algorithms that directly infer if a function has been obfuscated and how [5]. Even though these machine learning algorithms do not exhibit perfect scores and suffer from several limitations, such as degraded performances for some optimization levels, they are still resilient enough to give an overview on the applied obfuscation within a binary. As a consequence, given an obfuscation found within a binary, it is possible to determine binary features that have likely been impacted by the obfuscation and choose to exclude them from the diffing process. Data obfuscations, such as `MBA`, significantly inflate the basic-block length without altering the CG and CFG structures, meaning that CFG and CG features such as the cyclomatic complexity or the number of children of a function will be unaffected. Similarly, intra-procedural obfuscations, such as `Virtualization`, strongly modify the CFG topology but do not impact the CG. Inter-procedural obfuscations, however, transform the CG topology and tend to also affect the corresponding CFG of functions impacted by the obfuscation. For instance, when two functions are merged, both the overall function relationships within the program and the CFG of the merged functions are restructured resulting in a unique CFG. As a consequence, only a limited set of features remain unaffected by this obfuscation type. We manually establish two obfuscation-dependent feature sets: a set consisting solely of unstable features directly impacted by an obfuscation, and a stable feature set, available in Table 8. These features sets have been determined by reasoning about each obfuscation and comparing unobfuscated functions to their obfuscated counterparts. If no knowledge of the obfuscation can be extracted from a binary, default features can be used, otherwise curated stable features should be selected. In this experiment, we aim to analyze the effect of these feature sets on the final diffing result. For each Tigress and OLLVM-14 obfuscations, QBinDiff is applied with these three configurations on the `zlib` project, compiled with -O0. The Figure 1 highlights the difference between the standard QBinDiff differ (QBinDiff), where all the features are naively enabled, and a refined QBinDiff_s on a Tigress CFF pass applied on `zlib`. Figures concerning other obfuscations are not shown as they depict the same trend. In particular, this experiment reveals how much a differ can be feature-dependent. The standard set of features should be used as a starting point and can be adapted to account for expert knowledge (for instance use the corresponding QBinDiff_s in presence of intra-procedural obfuscation). Moreover, it is fundamental to not use only features that are vulnerable to the same changes, as the unstable features provide degraded performances. For the rest of the paper, QBinDiff denotes the standard QBinDiff algorithm with the default feature set and illustrates the case where a reverse engineer does not have addi-

tional knowledge about the obfuscated binary. QBinDiff$_s$ indicates the adjusted QBinDiff with the stable feature set given a specific obfuscation and represents the best-case scenario, where preliminary knowledge about the obfuscation type was extracted with confidence.

5 Plain against obfuscated

This Section focuses on the `plain-obfuscated` experiment and examines the impact of obfuscation on binary diffing and similarity when only the second binary is obfuscated, while the first remains unmodified. This scenario is relevant for cases where newer binary versions are obfuscated, such as Android applications or malware.

OLLVM-14 `plain-obfuscated`		General f1-score				Obfuscated f1-score			
		Mix	*CFF*	*Opaque*	*Enc.A*	*Mix*	*CFF*	*Opaque*	*Enc.A*
10%	Bindiff	0.81	0.78	0.78	0.79	**0.72**	0.71	0.69	0.75
	Diaphora3	0.79	0.78	0.78	0.80	0.45	0.59	0.61	0.78
	GMN	0.66	0.64	0.65	0.69	0.24	0.27	0.36	0.52
	Asm2vec	0.56	0.53	0.56	0.59	0.32	0.40	0.44	0.61
	JTrans	**0.85**	0.85	0.85	**0.86**	0.67	**0.81**	0.79	**0.82**
	QBinDiff	0.84	0.85	0.85	**0.86**	0.58	0.78	0.76	0.81
	QBinDiff$_s$	-	**0.86**	**0.86**	0.81	-	0.80	**0.81**	0.77
50%	Bindiff	0.72	0.76	0.74	0.79	**0.62**	0.67	0.65	0.73
	Diaphora3	0.64	0.70	0.71	0.79	0.40	0.57	0.59	0.78
	GMN	0.44	0.47	0.50	0.62	0.20	0.25	0.31	0.49
	Asm2vec	0.32	0.37	0.44	0.59	0.23	0.32	0.40	0.61
	JTrans	0.70	0.82	0.80	**0.86**	0.57	0.77	0.73	**0.81**
	QBinDiff	**0.74**	0.84	0.82	**0.86**	0.60	0.76	0.73	0.80
	QBinDiff$_s$	-	**0.85**	**0.85**	0.80	-	**0.79**	**0.79**	0.73
100%	Bindiff	0.53	0.65	0.65	0.78	0.47	0.56	0.57	0.72
	Diaphora3	0.40	0.50	0.61	0.79	0.35	0.49	0.56	0.77
	GMN	0.23	0.26	0.33	0.53	0.18	0.22	0.28	0.48
	Asm2vec	0.15	0.17	0.32	0.59	0.17	0.21	0.35	0.60
	JTrans	0.53	0.71	0.73	**0.86**	0.50	0.70	0.70	**0.81**
	QBinDiff	**0.65**	0.74	0.80	0.85	**0.59**	0.69	0.71	0.80
	QBinDiff$_s$	-	**0.77**	**0.84**	0.79	-	**0.73**	**0.78**	0.73

Table 2: Averaged f1-score comparison on OLLVM-14 obfuscations, `plain-obfuscated` setting. **First**, second and third best differs are displayed for each obfuscation and obfuscation level.

Tables 2 and 3 present the OLLVM-14 and Tigress results, respectively, showing the outcomes for all obfuscation passes across five projects, averaged for 10%, 50%, and 100% obfuscation levels, with both -O0 and -O2 optimizations. The 10% obfuscation level simulates a situation where a defender has a limited obfuscation budget, either due to memory or computational constraints. This is common in embedded systems, where MCUs have limited storage and processing power, and only a small subset of critical functions are obfuscated

for security. The 50% obfuscation level represents a guideline for achieving a reasonable tradeoff between security and efficiency. In practice, applying this amount of obfuscation often requires a deeper understanding of the criticality distribution over the functions, as well as the system constraints. Certain functions, that might precisely be the critical functions that one needs to obfuscate, could be highly sensitive to the obfuscation computational overhead, leading to prohibitive execution time for resource-constrained systems, such as MCUs. The 100% level depicts the worst-case scenario for an attacker.

Tigress `plain-obfuscated`		General f1-score										Obfuscated f1-score									
		Mix	*Mix + Split*	*Copy*	*Merge*	*Split*	*CFF*	*Virtualize*	*Opaque*	*Enc.A*	*Enc.L*	*Mix*	*Mix + Split*	*Copy*	*Merge*	*Split*	*CFF*	*Virtualize*	*Opaque*	*Enc.A*	*Enc.L*
10%	Bindiff	0.80	0.72	0.80	0.76	0.74	0.82	0.81	0.79	0.84	0.84	**0.69**	0.02	0.21	0.56	0.09	0.75	0.72	0.69	0.86	0.81
	Diaphora3	0.75	0.67	0.76	0.73	0.70	0.76	0.74	0.75	0.78	0.79	0.34	0.02	0.46	0.34	0.08	0.52	0.04	0.66	0.77	0.78
	GMN	0.51	0.46	0.57	0.52	0.47	0.53	0.48	0.53	0.54	0.59	0.04	0.01	0.34	0.15	0.02	0.08	0.01	0.25	0.16	0.49
	Asm2vec	0.49	0.42	0.51	0.55	0.46	0.49	0.45	0.51	0.56	0.57	0.15	0.01	0.28	0.35	0.03	0.20	0.03	0.45	0.52	0.58
	JTrans	0.80	0.74	0.82	0.78	0.76	0.84	0.80	0.81	0.85	0.85	0.55	0.02	**0.54**	0.35	0.04	0.72	0.36	**0.74**	0.87	0.88
	QBinDiff	**0.84**	**0.77**	**0.86**	0.82	0.81	0.87	0.83	**0.84**	**0.89**	**0.89**	0.55	**0.05**	0.25	0.59	0.19	0.73	0.35	0.69	**0.92**	**0.91**
	QBinDiff$_s$	-	-	0.85	**0.84**	**0.82**	**0.89**	**0.89**	0.79	0.83	0.84	-	-	0.25	**0.60**	**0.22**	**0.85**	**0.79**	0.65	0.90	0.86
50%	Bindiff	0.63	0.38	0.65	0.63	0.45	0.73	0.66	0.65	0.81	0.83	**0.52**	0.02	0.19	0.43	0.07	0.67	0.60	0.57	0.83	0.79
	Diaphora3	0.57	0.33	0.69	0.59	0.43	0.63	0.46	0.69	0.73	0.78	0.28	0.01	0.48	0.29	0.08	0.46	0.01	0.64	0.74	0.82
	GMN	0.30	0.19	0.49	0.35	0.26	0.32	0.27	0.38	0.38	0.58	0.02	0.01	0.36	0.13	0.02	0.06	0.00	0.20	0.13	0.50
	Asm2vec	0.27	0.16	0.41	0.40	0.26	0.30	0.18	0.40	0.49	0.59	0.10	0.00	0.29	0.28	0.04	0.14	0.01	0.39	0.50	0.64
	JTrans	0.64	0.41	0.71	0.56	0.47	0.76	0.52	**0.74**	0.63	0.85	0.46	0.01	**0.54**	0.16	0.03	0.69	0.23	**0.71**	0.67	**0.90**
	QBinDiff	**0.68**	**0.45**	**0.75**	0.70	0.56	0.79	0.68	**0.74**	**0.87**	**0.89**	0.49	**0.03**	0.26	0.51	0.17	0.72	0.53	0.66	**0.90**	**0.90**
	QBinDiff$_s$	-	-	0.74	**0.73**	**0.58**	**0.87**	**0.81**	0.68	0.82	0.83	-	-	0.26	**0.58**	**0.20**	**0.84**	**0.77**	0.59	0.88	0.87
100%	Bindiff	0.33	0.10	0.48	0.44	0.22	0.60	0.51	0.47	0.77	0.80	0.23	0.01	0.20	0.28	0.06	0.56	0.48	0.41	0.80	0.68
	Diaphora3	0.27	0.09	0.64	0.38	0.25	0.46	0.10	0.61	0.66	0.76	0.19	0.01	0.50	0.28	0.07	0.43	0.01	0.61	0.71	0.78
	GMN	0.10	0.05	0.42	0.23	0.13	0.12	0.11	0.24	0.25	0.57	0.01	0.01	0.35	0.12	0.02	0.05	0.00	0.19	0.12	0.49
	Asm2vec	0.08	0.04	0.29	0.29	0.13	0.11	0.02	0.32	0.43	0.56	0.08	0.00	0.24	0.32	0.03	0.11	0.00	0.37	0.48	0.65
	JTrans	**0.46**	**0.21**	0.62	0.32	0.28	0.68	0.20	**0.66**	0.60	0.83	**0.43**	0.01	**0.54**	0.14	0.03	0.68	0.16	**0.68**	0.66	**0.89**
	QBinDiff	0.40	0.19	**0.65**	0.57	0.36	0.71	0.49	0.63	**0.85**	**0.87**	0.33	**0.02**	0.26	0.48	0.15	0.70	0.46	0.61	**0.89**	0.87
	QBinDiff$_s$	-	-	0.64	**0.61**	**0.39**	**0.84**	**0.72**	0.56	0.81	0.82	-	-	0.27	**0.55**	**0.18**	**0.83**	**0.72**	0.53	0.86	0.84

Table 3: Averaged f1-score comparison on Tigress obfuscations, `plain-obfuscated` setting. **First**, second and third best differs are displayed for each obfuscation and obfuscation level.

To begin with, the most promising diffing tools are QBinDiff, QBinDiff$_s$ and JTrans, all of which achieve an f1-score of at least 0.80 for many obfuscation passes, particularly for data and intra-procedural obfuscations. Overall, real diffing tools like BinDiff, Diaphora, and QBinDiff tend to outperform some binary similarity methods combined with a matcher, such as GMN or Asm2vec. JTrans, however, does not face the same performance limitations as GMN or Asm2vec, thanks to its powerful embeddings. QBinDiff generally surpasses other diffing

tools by balancing both CG structure and function similarity, whereas JTrans relies solely on function embedding similarity, and BinDiff uses CG structure as the foundation for match propagation.

Additionally, the differences between the general and obfuscated f1-scores depend on both the type of obfuscation and the diffing tool used. Similarity tools like GMN and Asm2vec struggle to maintain consistent f1-score range, often favoring matching unobfuscated functions over obfuscated ones. More effective diffing tools, such as BinDiff and, to a lesser extent, Diaphora, reduce the gap between these scores to around 10 f1-score points for intra-procedural and data obfuscations. For JTrans and both versions of QBinDiff, the difference is even smaller, typically only about 5 f1-score points for lower levels of obfuscation. This is particularly true for QBinDiff$_s$ which incorporates features that make it more resilient to specific obfuscations. This result is surprising, as one might expect obfuscation to completely hinder binary diffing, but this is not the case. However, for inter-procedural obfuscations, this trend no longer holds, and all the diffing tools exhibit significant discrepancies between the f1-scores, sometimes exceeding 60 f1-score points. This is because either the tools fail to account for inter-procedural data, like similarity tools do, resulting in a major loss of information, or they rely on the CG for matching, which is altered during obfuscation and becomes unreliable, leading to poor performance, further exacerbated by the one-to-one mapping constraint.

Moreover, QBinDiff and QBinDiff$_s$ show different behaviors. Using QBinDiff$_s$ significantly improves performance for most inter and intra-procedural obfuscations compared to QBinDiff, both for general and obfuscated f1-scores. In some cases, it can boost the obfuscated f1-score by as much as 44 points, particularly for the `Virtualization` pass. However, this improvement does not hold for data obfuscations and the `Opaque` pass, where QBinDiff outperforms QBinDiff$_s$. This difference is due to the stable feature set used in QBinDiff$_s$, which was specifically designed for those obfuscations. For data obfuscations, features like assembly mnemonics were removed, reducing the obfuscation noise, but some deleted features may still provide useful information for matching.

In general, Tigress f1-scores are lower than those of OLLVM-14. While OLLVM-14 lacks inter-procedural obfuscations, which would penalize diffing, its intra-procedural and data obfuscations are less severe than Tigress's. Tigress scores tend to be 10 to 20 points lower in terms of f1-score. This is because Tigress includes more advanced obfuscations, such as `Virtualization`, and the obfuscations it shares with OLLVM-14, like `Opaque`, are more aggressive.

Note that the results above include both -O0 and -O2 binaries.[9] In general, -O0 results tend to be higher than -O2, with the f1-score degradation in -O2 being due to inlining caused by the optimization, which causes some functions, both obfuscated and unobfuscated, to disappear. As a result, diffing tools can no longer match those functions, leading to a drop of f1-score.

[9] Due to space limitations, we combined the -O0 and -O2 experiments, even though they exhibit slightly different behavior in terms of diffing results. Specific -O0 and -O2 results will be released as artifacts along with the dataset.

Overall, these results show that most diffing tools perform well on intra-procedural and data-obfuscated binaries, even when large portions of the binary are obfuscated. This is primarily because these tools rely on the CG structure, which remains unaffected by these types of obfuscations, to start matching functions. However, inter-procedural obfuscations lead to a significant drop in f1-scores, especially for obfuscated f1-scores. As a result, reverse engineers will face more challenges dealing with such obfuscated variants using diffing and should prioritize adaptive diffing tools like QBinDiff, which maintain the highest performances. From a software protection standpoint, these obfuscations should be prioritized if the goal is to counter reverse engineering attacks through diffing.

6 Obfuscated against obfuscated

In this Section, we compare two obfuscated versions of a binary to assess the effectiveness of binary diffing and similarity tools in this complex scenario. This use case is valuable for tracking the evolution of software obfuscation techniques over time or for identifying variants that are more vulnerable than others.

Tigress `obfuscated-obfuscated`		General f1-score										Obfuscated f1-score									
		Mix - Mix	*Split - Merge*	*Merge - Copy*	*Split - Copy*	*CFF - Split*	*CFF - Merge*	*CFF - Copy*	*CFF - Opaque*	*Opaque - Enc.A*	*CFF - Enc.A*	*Mix - Mix*	*Split - Merge*	*Merge - Copy*	*Split - Copy*	*CFF - Split*	*CFF - Merge*	*CFF - Copy*	*CFF - Opaque*	*Opaque - Enc.A*	*CFF - Enc.A*
10%	BinDiff	0.81	0.68	0.79	0.72	0.77	0.80	0.83	0.82	0.83	0.87	**0.70**	0.08	0.24	0.16	**0.35**	**0.68**	0.50	**0.75**	**0.83**	**0.77**
	Diaphora3	0.74	0.65	0.74	0.68	0.71	0.73	0.77	0.76	0.77	0.79	0.40	0.09	**0.42**	0.17	0.18	0.51	**0.53**	0.71	0.79	0.64
	GMN	0.62	0.50	0.67	0.56	0.59	0.62	0.68	0.64	0.67	0.70	0.09	0.03	0.30	0.06	0.05	0.16	0.10	0.22	0.33	0.11
	Asm2vec	0.55	0.47	0.55	0.46	0.50	0.37	0.56	0.56	0.58	0.61	0.19	0.05	0.31	0.08	0.11	0.27	0.21	0.40	0.57	0.25
	JTrans	0.79	0.70	0.79	0.71	0.77	0.78	0.82	0.82	0.83	0.86	0.43	0.05	0.37	0.15	0.26	0.49	0.50	**0.75**	**0.83**	0.73
	QBinDiff	**0.83**	**0.73**	**0.82**	**0.75**	**0.81**	**0.83**	**0.85**	**0.85**	**0.86**	**0.88**	0.53	**0.15**	0.25	**0.20**	0.33	0.66	0.48	0.69	0.80	0.69
50%	BinDiff	0.55	0.40	0.57	0.38	0.46	0.59	0.60	0.63	0.71	0.80	0.40	0.06	0.18	0.11	0.24	0.43	0.38	0.51	0.66	**0.69**
	Diaphora3	0.53	0.40	0.60	0.37	0.39	0.52	0.58	0.65	0.69	0.67	0.41	0.07	**0.35**	0.15	0.13	0.36	**0.43**	0.64	**0.72**	0.55
	GMN	0.32	0.25	0.43	0.24	0.27	0.32	0.36	0.34	0.40	0.42	0.09	0.02	0.18	0.04	0.03	0.07	0.06	0.11	0.18	0.05
	Asm2vec	0.32	0.21	0.29	0.19	0.20	0.27	0.29	0.32	0.50	0.39	0.20	0.04	0.16	0.06	0.07	0.20	0.13	0.28	0.53	0.16
	JTrans	0.57	0.43	0.59	0.36	0.45	0.53	0.62	**0.72**	0.55	0.61	0.38	0.02	0.24	0.11	0.19	0.24	**0.43**	**0.66**	0.50	0.47
	QBinDiff	**0.61**	**0.48**	**0.63**	**0.43**	**0.52**	**0.67**	**0.66**	**0.72**	**0.78**	**0.81**	**0.43**	**0.11**	0.19	**0.17**	**0.28**	**0.55**	**0.43**	0.61	0.70	0.64
100%	BinDiff	0.49	0.25	0.36	0.14	0.13	0.38	0.24	0.40	0.39	0.53	0.32	0.04	0.08	0.05	0.06	0.15	0.13	0.21	0.40	0.48
	Diaphora3	0.67	0.23	0.32	0.22	0.14	0.32	0.37	0.55	0.39	0.33	**0.61**	0.03	**0.15**	**0.14**	0.09	0.24	0.36	**0.53**	**0.54**	0.42
	GMN	0.30	0.18	0.25	0.10	0.08	0.23	0.13	0.16	0.16	0.13	0.18	0.01	0.04	0.03	0.02	0.04	0.04	0.02	0.06	0.04
	Asm2vec	0.41	0.06	0.08	0.08	0.05	0.16	0.08	0.18	0.33	0.14	0.35	0.03	0.06	0.05	0.05	0.17	0.09	0.18	0.44	0.13
	JTrans	**0.71**	0.27	0.35	0.19	0.21	0.34	0.43	0.64	0.36	0.41	0.52	0.01	0.06	0.09	0.15	0.12	**0.38**	0.51	0.29	0.34
	QBinDiff	0.70	**0.32**	**0.41**	**0.25**	**0.24**	**0.61**	**0.46**	**0.66**	**0.48**	**0.56**	0.51	**0.07**	0.12	0.13	**0.18**	**0.49**	**0.38**	0.49	0.39	**0.60**

Table 4: Averaged f1-score comparison on Tigress obfuscations, `obfuscated-obfuscated` setting. **First**, second and third best differs are displayed for each obfuscation and obfuscation level.

For brevity, we only present Tigress results in Table 4, as the OLLVM results follow the same pattern but show higher f1-scores, similar to what we observed in the previous `plain-obfuscated` experiment. We have chosen to focus on

a subset of possible obfuscated-obfuscated pairs, prioritizing intra-intra, inter-inter, and inter-intra pairs when available. This decision is based on the results of the first `plain-obfuscated` experiment, where data obfuscations yielded less insightful results compared to other obfuscation types. The `Mix-Mix` pair refers to two binaries obfuscated with the same `Mix` schema but using different seeds. Results for QBinDiff$_s$ are not available because when comparing two obfuscated variants, unless both variants are obfuscated using the same type of obfuscation, it is difficult to identify a set of features that are robust across different obfuscation types. Neither the CG nor the CFG are reliable when comparing a variant obfuscated with `Split` to one obfuscated with `Virtualization`. Overall, the results reflect a similar trend to previous findings, with QBinDiff, JTrans, and BinDiff generally being the most effective differ tools. In this scenario, JTrans performs worse because the variants are both obfuscated, whereas JTrans was trained on unobfuscated code, making it more reliant on plain binary code.
Although this framework may appear more challenging than the `plain-obfuscated` one, f1-scores, both obfuscated and unobfuscated, remain acceptable for obfuscated variants without inter-procedural obfuscation, with scores reaching at least 0.70 for many pairs at lower obfuscation levels. While this trend diminishes as the number of obfuscated functions reaches 100%, it still demonstrates that diffing obfuscated variants is feasible, yielding satisfactory results, particularly with intra-procedural or data obfuscations that preserve the CG structure for binary diffing. These findings show that data and intra-procedural obfuscations, when used alone, do not fully prevent binary similarity or diffing tools from working, even though they are effective at confusing reverse engineers conducting manual analysis. However, these obfuscations are the most commonly implemented in open-source or free obfuscators [18,37,15]. In contrast, diffing two samples obfuscated with inter-procedural passes is much more challenging, with a significant gap between the f1-scores for unobfuscated and obfuscated pairs, which remain low, typically below 0.50 of f1-score when obfuscation exceeds 50%. Therefore, from a reverse engineering perspective, more focus should be placed on developing new matching algorithms that can better handle transformations that disrupt the relationships between functions. From a software protection standpoint, this suggests a stronger emphasis on inter-procedural obfuscation to limit knowledge transfer between binaries.

7 BinKit results

As previously mentioned, we expand on our earlier experiment using the BinKit dataset, which includes plain binaries compiled with various compilers, versions, optimizations, and architectures. The obfuscated binaries are exclusively compiled with Clang using OLLVM, with different optimizations and architectures. We focus only on x64 binaries and limit our analysis to five projects. The obfuscations applied are solely intra-procedural and data obfuscations at the binary level. The results, presented in Table 5, follow the same pattern as our previous dataset experiments, but the effects are even more pronounced. Specifically,

binary similarity tools perform notably worse than binary diffing tools, with QBinDiff outperforming both BinDiff and Diaphora.

Binkit	Plain-obfuscated					Obfuscated-obfuscated				
	bool	*cpio*	*cflow*	*ccd2cue*	*a2ps*	*bool*	*cpio*	*cflow*	*ccd2cue*	*a2ps*
BinDiff	0.9	0.63	0.78	0.94	0.7	0.8	0.42	0.61	0.84	0.44
Diaphora3	0.66	0.6	0.71	0.71	0.63	0.57	0.45	0.4	0.57	0.39
GMN	0.41	0.39	0.30	0.53	0.22	0.40	0.39	0.31	0.53	0.23
Asm2vec	0.37	0.29	0.22	0.55	0.15	0.34	0.25	0.19	0.38	0.13
JTrans	0.86	0.80	0.84	0.90	0.69	0.70	0.55	0.55	0.66	0.42
QBinDiff	0.96	0.92	0.91	0.98	0.82	0.9	0.82	0.82	**0.91**	0.7
QBinDiff$_s$	**0.97**	**0.94**	**0.93**	**0.99**	**0.87**	**0.92**	**0.86**	**0.86**	**0.91**	**0.80**

Table 5: Averaged f1-score comparison for the BinKit obfuscated dataset. **First**, second and third best differs are displayed for each obfuscation and obfuscation level.

These results, tested on a different dataset, further confirm that intra-procedural and data obfuscations do not pose a significant barrier to binary diffing.

8 Real-world example: XTunnel

This Section aims to extend the previous experiments, conducted on a realistic but yet simulated dataset, to real-world malware samples. Finding any two malware samples is simple, whereas identifying two obfuscated versions of the same malware is more challenging, though still achievable. However, locating two obfuscated malware samples for which establishing a reliable ground truth is practically feasible proves to be significantly more difficult, due to a limited number of functions and limited obfuscation. XTunnel is among the few malware that satisfies these requirements [1]. Establishing a match between different versions eases the reverse engineering of new variant, especially if new functionalities have been added to the obfuscated version. We replicate only the previous `plain-obfuscated` experiment, for brevity, by diffing two XTunnel samples: the plain *42DEE* and the obfuscated *99B45*. The two samples contain 3,196 and 3,693 functions, respectively, with about 400 of them heavily obfuscated using `Opaque Predicates` [1]. The remaining functions appear to be third-party libraries statically linked within the executables.

To create the ground truth for this example, we manually match functions between the plain binary *42DEE* and the obfuscated variant *99B45*, starting with functions that have the same hash and then using manual reverse engineering to identify the rest. Creating such ground truth requires significant effort and may introduce bias, especially due to discrepancies in the number of primary and secondary functions caused by inlining, which can affect its accuracy. As a result, 417 primary functions and 913 secondary functions remain unmatched due to uncertainty in their assignment.

Only QBinDiff$_s$ and BinDiff are used in this experiment. An effective differ should produce high f1-scores for both binaries. The results, shown in Table 6,

reveal that while QBinDiff$_s$ and BinDiff are almost identical on the full set of functions, BinDiff performs poorly on the obfuscated functions. Despite potential bias in the ground truth, this example demonstrates that our earlier findings remain true in real-world use cases.

	General f1-score	Obfuscated f1-score
BinDiff	0.966	0.303
QBinDiff$_s$	0.97	0.915

Table 6: f1-scores for the `plain-obfuscated` experiment variant between samples *42DEE* and *99B45*.

9 Discussions

Limitations and threats to validity First, this research relies on disassembly and functions recovered by IDA-Pro, which may lead to incomplete results due to the complexity of these problems [28]. Additionally, some obfuscation techniques are specifically designed to hinder disassemblers from working correctly.
Second, most differ tools are limited to one-to-one matching, which may not be enough for obfuscated or optimized functions where a one-to-many approach would be more appropriate. This is a complex issue, and only a few differ tools offer solutions, with no clear way to assess their effectiveness [20].
Third, obfuscation may not achieve the desired effect in the final binary, as the compiler can interfere with or even reverse obfuscation, given their conflicting purposes. The higher the optimization level, the greater the risk that the obfuscation will be altered. This is especially true for optimization level -O2, which is more realistic but can remove obfuscations or apply inlining. This issue is noticeable with newer compiler versions, such as clang-14, when using older obfuscation techniques like OLLVM. Even at optimization level -O0, this behavior can still occur, though to a lesser extent (e.g., OLLVM `EncodeArith` may be simplified through constant propagation). Preventing these optimizations is compiler-dependent and often requires tweaking internals, if even possible. For example, constant propagation occurs both in the clang front-end and optimization phase, making it difficult to disable. Source-to-source obfuscators, like Tigress, cannot directly interact with the compiler, and most OLLVM passes are applied before any optimization, meaning both are susceptible to simplification. On the other hand, other compilation-pass-based obfuscators apply their transformations within an optimization chain, not at the beginning, to avoid slowing down the final binary, and not at the end, to slightly optimize the obfuscated code for performance [2]. Determining the best way to combine optimization and obfuscation is beyond the scope of this paper.
Fourth, this study primarily focuses on single obfuscation types, except the `Mix` and `Mix + Split` combinations. In the `Mix` configuration, `CFF` is followed by another intra-procedural pass and a data obfuscation. The pass order here was fixed, and it may not be the most efficient for resisting an attacker. Similarly, `Mix` combines `CFF`, shared by both Tigress and OLLVM-14, with slightly different

passes that modify control flow (`BogusCF` and `Opaque`) and data (`InsSub` and `Enc.A`). This comparison evaluates an obfuscator's effectiveness based on its different obfuscation types, rather than the overall robustness of each pass.
Fifth, to level the playing field for all the differs, Diaphora3's decompiler-based features were disabled. Including them will be considered in future work.

Future Work. Studying obfuscation in all its facets remains an ongoing research challenge, as functions, optimizations, obfuscation passes, and obfuscators are all intertwined. There is still much work to be done. Specifically, the dataset that we created is an initial effort to address a gap in obfuscation research. Expanding it by incorporating additional obfuscators or more projects would be a valuable step toward improving generalization. We plan to gradually enhance it over time. Additionally, the relationship between an obfuscator and a differ is similar to the dynamic between a defender and an attacker. This opens the door for adversarial refinement on both sides. A defender could use diffing tools to identify and analyze weaknesses in their obfuscations, improving their resilience against attackers. This line of research could ultimately lead to more robust and resilient obfuscation schemes.

10 Conclusion

This paper presents a comprehensive study of binary diffing in both `plain-obfuscated` and `obfuscated-obfuscated` settings. We evaluate various binary differ and similarity tools, revealing that, surprisingly, standard binary diffing methods perform well against intra-procedural and data obfuscations. On the other hand, the most effective differ tools, which rely heavily on the CG, are susceptible to disruption from inter-procedural obfuscations. The modularity offered by QBinDiff proves useful in obtaining resilient diffing results based on the type of obfuscation applied, showing promising outcomes. These findings demonstrate that diffing obfuscated binaries is feasible and can yield satisfactory results in certain cases. From a software protection perspective, they also highlight the advantages of using more extensive inter-procedural obfuscation.

Availability

The dataset of obfuscated binaries and the artifacts are publicly released. [10]

Acknowledgments. We would like to thank Bruno Mateu for porting OLLVM-4 to OLLVM-14 and the Agence Innovation Defense (AID) that supports this research.

References

1. 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). `https://doi.org/10.1109/SP.2017.36`

[10] `https://github.com/quarkslab/diffing_obfuscation_dataset`

2. Brunet, P., Creusillet, B., Guinet, A., Martinez, J.M.: Epona and the obfuscation paradox: Transparent for users and developers, a pain for reversers. In: Proceedings of the 3rd ACM Workshop on Software Protection. p. 41–52. SPRO'19, Association for Computing Machinery, New York, NY, USA (2019)
3. Capozzi, G., D'Elia, D.C., Di Luna, G.A., Querzoni, L.: Adversarial attacks against binary similarity systems. arXiv preprint arXiv:2303.11143 (2023)
4. Cohen, R., David, R., Mori, R., Yger, F., Rossi, F.: Improving binary diffing through similarity and matching intricacies. In: Conference on Artificial Intelligence for Defense (CAID) (11 2024)
5. Cohen, R., David, R., Yger, F., Rossi, F.: Identifying obfuscated code through graph-based semantic analysis of binary code. In: The 13th International Conference on Complex Networks and their Applications (2024)
6. Collberg, C.: The tigress C obfuscator. `https://tigress.wtf/index.html` (2016)
7. David, R., Coniglio, L., Ceccato, M.: Qsynth-a program synthesis based approach for binary code deobfuscation. In: BAR 2020 Workshop (2020)
8. De Ghein, R., Abrath, B., De Sutter, B., Coppens, B.: Apkdiff: Matching android app versions based on class structure. In: Proceedings of the 2022 ACM Workshop on Research on Offensive and Defensive Techniques in the Context of Man At The End (MATE) Attacks. p. 1–12. Checkmate '22, Association for Computing Machinery, New York, NY, USA (2022)
9. Ding, S.H., Fung, B.C., Charland, P.: Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In: 2019 IEEE Symposium on Security and Privacy (SP). IEEE (2019)
10. Duan, Y., Li, X., Wang, J., Yin, H.: Deepbindiff: Learning program-wide code representations for binary diffing. In: Network and distributed system security symposium (2020)
11. Dullien, T., Rolles, R.: Graph-based comparison of executable objects (english version). Sstic **5**(1), 3 (2005)
12. Flake, H.: Structural comparison of executable objects. DIMVA 2004, July 6-7, Dortmund, Germany (2004)
13. Gao, H., Zhang, T., Chen, S., Wang, L., Yu, F.: Fusion: Measuring binary function similarity with code-specific embedding and order-sensitive gnn. Symmetry (2022)
14. Greco, C., Ianni, M., Guzzo, A., Fortino, G.: Explaining binary obfuscation. pp. 22–27 (07 2023). `https://doi.org/10.1109/CSR57506.2023.10224825`
15. Hikari: Hikari-llvm15. `https://github.com/61bcdefg/Hikari-LLVM15` (2019)
16. Hosseinzadeh, S., Rauti, S., Lauren, S., Makela, J.M., Holvitie, J., Hyrynsalmi, S., Leppanen, V.: Diversification and obfuscation techniques for software security: A systematic literature review. Information and Software Technology (2018)
17. Hu, Y., Zhang, Y., Li, J., Wang, H., Li, B., Gu, D.: Binmatch: A semantics-based hybrid approach on binary code clone analysis. In: 2018 IEEE international conference on software maintenance and evolution (ICSME). IEEE (2018)
18. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: 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 (2015)
19. Kim, D., Kim, E., Cha, S.K., Son, S., Kim, Y.: Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. IEEE Transactions on Software Engineering pp. 1–23 (2022)
20. Koret, J.: Diaphora. `https://github.com/joxeankoret/diaphora` (2015)

21. Kostakis, O., Kinable, J., Mahmoudi, H., Mustonen, K.: Improved call graph comparison using simulated annealing. In: Proceedings of the 2011 ACM Symposium on Applied Computing. pp. 1516–1523 (2011)
22. Kuhn, H.W.: The hungarian method for the assignment problem. Naval research logistics quarterly **2**(1-2), 83–97 (1955)
23. Li, Y., Gu, C., Dullien, T., Vinyals, O., Kohli, P.: Graph matching networks for learning the similarity of graph structured objects. In: International conference on machine learning. pp. 3835–3845. PMLR (2019)
24. Luo, L., Ming, J., Wu, D., Liu, P., Zhu, S.: Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering. pp. 389–400 (2014)
25. 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) (2022)
26. Massarelli, L., Di Luna, G.A., Petroni, F., Querzoni, L., Baldoni, R., et al.: Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. In: 2nd Workshop on Binary Analysis Research (BAR) (2019)
27. Massarelli, L., Di Luna, G.A., Petroni, F., Querzoni, L., Baldoni, R.: Safe: Self-attentive function embeddings for binary similarity. In: Proceedings of 16th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA) (2019)
28. Meng, X., Miller, B.P.: Binary code is not easy. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. pp. 24–35 (2016)
29. Mengin, E., Rossi, F.: Binary diffing as a network alignment problem via belief propagation. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 967–978. IEEE (2021)
30. Mengin, E., Rossi, F.: Improved algorithm for the network alignment problem with application to binary diffing. Procedia Computer Science **192**, 961–970 (2021)
31. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781 (2013)
32. Ming, J., Xu, D., Jiang, Y., Wu, D.: BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 253–270 (2017)
33. Ming, J., Xu, D., Wu, D.: Memoized semantics-based binary diffing with application to malware lineage inference. In: Federrath, H., Gollmann, D. (eds.) ICT Systems Security and Privacy Protection. Cham (2015)
34. Nagra, J., Collberg, C.: Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Pearson Education (2009)
35. Pei, K., Xuan, Z., Yang, J., Jana, S., Ray, B.: Trex: Learning execution semantics from micro-traces for binary similarity. arXiv preprint arXiv:2012.08680 (2020)
36. Shirani, P., Wang, L., Debbabi, M.: Binshape: Scalable and robust binary library function identification using function shape. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 301–324. Springer (2017)
37. Thomas, R.: O-mvll. `https://github.com/open-obfuscator/o-mvll` (2022)
38. 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)

39. Ullah, S., Oh, H.: Bindiff nn: Learning distributed representation of assembly for robust binary diffing against semantic differences. IEEE Transactions on Software Engineering **48**(9), 3442–3466 (2021)
40. Wang, C.: A security architecture for survivability mechanisms. University of Virginia (2001)
41. Wang, H., Qu, W., Katz, G., Zhu, W., Gao, Z., Qiu, H., Zhuge, J., Zhang, C.: Jtrans: Jump-aware transformer for binary code similarity detection. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 1–13. ISSTA 2022, New York, NY, USA (2022)
42. Zhang, P., Wu, C., Peng, M., Zeng, K., Yu, D., Lai, Y., Kang, Y., Wang, W., Wang, Z.: Khaos: The impact of inter-procedural code obfuscation on binary diffing techniques. In: Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization. p. 55–67. CGO '23, Association for Computing Machinery, New York, NY, USA (2023)
43. Zhao, L., Zhu, Y., Ming, J., Zhang, Y., Zhang, H., Yin, H.: Patchscope: Memory object centric patch diffing. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. pp. 149–165 (2020)
44. 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)

	Passes	Pass type	Description	Unify name
Tigress	Copy	Inter	Clone a function	Copy
	Split	Inter	Split a function into chunks	Split
	Merge	Inter	Merge multiple function into one	Merge
	CFF	Intra	The function basic blocks are put at the same level	CFF
	Virtualize	Intra	Transforms a function into an interpreter	Virtualize
	Opaque	Intra	Insert OpaquePredicates	Opaque
	EncodeArithmetic	Data	Replace arithmetic with complex expressions	Enc.A
	EncodeLiterals	Data	Hide literals with less obvious expressions	Enc.L
	Mix	Intra & Data	Combination of CFF, EncodeArithmetic and Opaque	Mix
	Mix + Split	All	Mix + Split	Mix + Split
OLLVM-14	CFF	Intra	The function basic blocks are put at the same level	CFF
	BogusCF	Intra	Insert basic blocks and OpaquePredicates	Opaque
	InsSub	Data	Substitute operators by more complicated instructions	Enc.A
	Mix	Intra & Data	Combination of CFF, InsSub and BogusCF	Mix

Table 7: Obfuscation description

	BBlockNb	SCComponents	BytesHash	Cyclomatic Complexity	MDIndex	JumpNb	SmallPrimeNumbers	MaxParentNb	MaxChildNb	MaxInsNb	MeanInsNb	InsNb	GraphMeanDegree	GraphDensity	GraphNbComponents	Graph Diameter	GraphTransitivity	GraphCommunities	Address	DatName	FuncName	ChildNb	ParentNb	RelativeNb	LibName	ImpName	Constant	StrRefs	MnemonicSimple	MnemonicTyped	GroupsCategory	ReadWriteAccess
Merge Split	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✔	✔	✔	✗	✗	✗	✗	✗	✗	✔	✔	✔	✔	✔	✔
Copy	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✗	✗	✗	✗	✗	✗	✔	✔	✔	✔	✔	✔
Intra	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Data	✔	✔	✗	✔	✔	✔	✔	✔	✔	✗	✗	✗	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✗	✗	✗	✗	✗	✗

Table 8: QBinDiff stable (✔) and unstable (✗) features list depending on the applied obfuscation.