Improving binary diffing through similarity and matching intricacies

1st Roxane Cohen
Quarkslab
LAMSADE, CNRS, Université Paris Dauphine - PSL
Paris, France
rcohen@quarkslab.com

2nd Robin David
Quarkslab
Paris, France
rdavid@quarkslab.com

3rd Riccardo Mori
Quarkslab
Paris, France
rmori@quarkslab.com

4th Florian Yger
LITIS, INSA Rouen Normandy
Rouen, France
florian.yger@insa-rouen.fr

5th Fabrice Rossi
CEREMADE, CNRS, Université Paris Dauphine - PSL
Paris, France
fabrice.rossi@dauphine.psl.eu

***Abstract*—Reverse-engineering represents a key aspect of cybersecurity as it helps to understand unknown software or systems. From a defender's point of view, it may detect suspicious binaries or existing vulnerabilities inside organization systems. From an attacker's point of view, it offers insights into potential threats or weaknesses of a target. In particular, reverse-engineering relies on binary diffing, whose goal is to find similarities and differences between different binaries or program variants. Such a task is essential as software are constantly evolving over time. However, it is notoriously difficult, despite the large number of research papers on this subject. Many approaches have been explored, ranging from rule-based decision algorithms to advanced Deep Learning (DL) models. Such difficulty can be explained by the inherent nature of binary code, which is unstable and prone to many syntactic differences while semantics are unaltered. Determining if two binary functions are similar is already a complex task as such similarity should describe the semantics and not the syntactic properties. Obtaining a diffing, a correspondence between functions of two binaries, is even more difficult.**

In this work, we present a binary differ, called QBinDiff, and detail how it is adapted for a modular and fine-grained diffing. We conduct an empirical study about its properties, from its algorithm to its implementation, through an ablation study. We present the diffing discipline, the existing approaches, some of them being state-of-the-art, and establish a comparison benchmark between them on standard binaries. We show in particular that QBinDiff performs better than existing differs, thanks to its modularity.

***Index Terms*—Binary Similarity, Binary diffing, Graph Neural Networks, word2vec**

I. Introduction

Reverse-engineering has been widely used for defense purposes such as malware or cryptography analysis and vulnerability search. More specifically, binary diffing, a subdomain of reverse-engineering, consists in identifying similarities and differences between two binaries. It is used for malware diffing [25], patch analysis [31], program similarity [4], backdoor detection, anti-plagiarism, clustering different malware by their families or establishing a malware lineage that could be attributed to a specific Advanced Persistent Threat (APT). Diffing usually operates at the function level and tries to find an assignment between the functions of two binaries. The underlying assumption is that disassembly and function recovery are feasible. However, in practice, this is known as a complex problem [20] and in this paper, we rely on IDA-Pro [1] or Ghidra [2] for disassembly purposes. In addition, the initial binary representation may be considerably altered during the compilation process, due to optimization passes. Among them, inlining or loop unrolling alters the function control-flow logic.

Usually, binary diffing strongly relies on graphs extracted from disassembly, such as Control-Flow Graph (CFG) where nodes are Basic Blocks (BB) and edges represent execution flow within the function scope and, Call Graph (CG) where nodes are program functions and edges denote inter-procedural relationships. Consequently, many graph-centered works have been published with an increasing attention dedicated to Machine Learning (ML) and Deep Learning (DL) based approaches [3], [4], [17], [18], [26], [30]. If these methods show promising results for solving binary similarity, DL methods require large computational resources. Retraining a model or simply using it for inference may be unaffordable. Besides, binary similarity models output, given a candidate function, its closest counterpart inside a pool of functions. They do not compute the direct matching between two binaries. Consequently, obtaining a final diffing between two binaries requires to further apply a matching step using computed similarity scores. Furthermore, DL model source code or datasets are not always available and reproducibility is difficult. For these reasons, mostly BinDiff [5], [7] and Diaphora [14] are used in practice. In fact, in comparison to advanced DL models, they scale relatively well using criteria-based matching. However, for those who want finer and more adapted control over the diffing results, they may not be adapted as they are not modular or difficult to parametrize. In order to provide a more

[1]https://hex-rays.com/ida-pro
[2]https://ghidra-sre.org/

accessible diffing solution, QBinDiff[3] was open-sourced as a modular differ.

The main contributions of this work are:

- We present QBinDiff, an open-source network alignment solver, that can be applied for binary diffing. We analyze its underlying algorithm, in particular its modular functionality given the available features and parameters that can be combined to obtain fined-grained results tailored for each use case.
- We realize an ablation study of QBinDiff components to highlight how they are intertwined and how they contribute to QBinDiff results.
- We perform an empirical comparison between standard binary differs, such as BinDiff and Diaphora. We also analyze more recent DL-based methods such as Graph Matching Networks [17], Asm2vec [3], PalmTree [16] and JTrans [30]. In particular, we show that QBinDiff offers the best performances compared to other approaches.

Section II presents the concepts of binary diffing and similarity and the current approaches. Section III details the QBinDiff algorithm and its ablation study. A fair comparison between state-of-the-art diffing solutions is performed in Section IV whereas Section V establishes some discussions about the limitations of this work and further research that could be conducted. Finally, Section VI concludes this work.

II. Binary analysis

Binary diffing involves identifying similarities and discrepancies between two binary programs. Minor modifications or slight patches resulting from version updates or compilation differences should be detected by diffing tools. Binary diffing is defined as a one-to-one mapping $\phi : (\mathcal{P}, \mathcal{S}) \longmapsto \rho$, where $\mathcal{P}$ represents the primary function set of size n, $\mathcal{S}$ represents the secondary function set of size m, and $\rho : \mathcal{P} \longrightarrow \mathcal{S}$ is a partial and injective assignment function. This ensures that each function in $\mathcal{P}$ is matched to at most one function in $\mathcal{S}$. Other definitions consider various program granularities, such as BB [4]. Additionally, the mapping could be expanded to one-to-many [14] or many-to-many correspondences. Usual diffing is performed without access to source code or symbols. BinDiff [5], [7] and Diaphora [14] are widely used binary diffing tools in the reverse-engineering community. Common metrics for evaluating diffing include recall, precision, and f1-score.

Binary similarity is applied in order to find the most similar function to f inside a pool of candidate functions. A natural application is vulnerability search, because for a given vulnerable function, it is possible to find, from a database of functions, the most similar one, which probably also contains the same vulnerability. It is an active research field relying heavily on Machine Learning (ML) and Deep Learning (DL). Usual ML methods rely on precomputed features derived from assembly code or CFG. For example, TIKNIB [13] computes similarity scores using a specific distance combining various handcrafted features and BinShape [28] starts by extracting features and sorts them to obtain the top-ranked ones that are given to a decision tree.

Most importantly, DL techniques have become prevalent in this research area and are inspired by Natural Language Processing (NLP). Asm2Vec [3] is based on a refined and enhanced version of the word2vec model [24] applied on assembly text. Trex [26] and JTrans [30] are motivated by the recent success of transformers for large language models. The same holds for PalmTree [16], based on BERT but pre-trained on several assembly representation tasks, such as instruction reordering, as it is possible in assembly to switch several instructions without modifying the general semantics of the code. Graph Neural Network (GNN) is a new promising research area, that is gaining more and more popularity. The latest research articles mostly use increasingly complex GNN, with a pretrained language model used to produce initial GNN features and that relies on node assembly instructions [19]. Graph Matching Network (GMN) [17] is the first work that jointly learns graph embeddings on similar graph pairs rather than independent embeddings. Based on this principle, more GNN architectures or language models are explored [8], [18], [29]. Despite the large amount of new academic solutions, few of them provide a maintained implementation. Moreover, training these models can be quite challenging, even with the source code and the dataset. Additionally, such binary similarity models are not exempt from failures, as some adversarial attacks inspired by the ML field have been applied to disrupt specific binary similarity tools [1].

Remark. Binary diffing and binary similarity are two distinct problems. Even though they share many common aspects, their purposes are distinct. Binary diffing aims to find an assignment between the functions of two binaries and may rely on similarity scores to establish matches. Conversely, binary similarity only outputs similarity scores between pairs of candidate functions. To be further used to perform diffing, binary similarity models should be followed by a matching process to perform binary diffing.

III. QBinDiff: a modular differ

This Section presents QBinDiff and the corresponding ablation study of its components. Such ablation experimentation helps understand the intricacies of programs and their representation as graphs.

A. *QBinDiff*

QBinDiff [21]–[23] is a modular one-to-one differ. Given two graphs, respectively called *primary* and *secondary*, it solves an instance of the network alignment problem [6], namely finding a matching between the respective nodes of the two graphs, by expressing the link between objects similarities (as graph nodes) and relationships between these objects (graph edges), using in particular a belief propagation algorithm [21]. This algorithm represents this alignment problem using a graphical model, where nodes indicate variables of the problem and edges denote dependencies between these

[3]https://github.com/quarkslab/qbindiff

variables. It iteratively updates values associated to the variables using "messages" sent through graph edges. Once it has converged, marginal probabilities are used to determine the optimal solution of the problem. This paper focuses on the experimental aspects of QBindiff, while mathematical foundations and details are already provided [22].

Algorithm 1 QBinDiff algorithm

```
Require: Primary binary $p$ , Secondary binary $s$, (features, weights), parameters=($d$, $s_{ratio}$, $\alpha$, $\epsilon$), Optional list of pre-passes and post-passes
Ensure: Matching between $p$ and $s$ functions
 1: $\mathcal{S} \leftarrow Anchoring(p, s)$
 2: for $pass_i \in$ pre-passes do
 3:     $\mathcal{S} \leftarrow pass_i(p, s, S)$    ▷ similarity matrix refinement passes before feature extraction
 4: end for
 5: $features_p \leftarrow FeaturesExtraction(p, features)$
 6: $features_s \leftarrow FeaturesExtraction(s, features)$
 7: $\mathcal{S} \leftarrow Similarity(features_p, features_s, \mathcal{S}, weights, d)$
 8: for $pass_i \in$ post-passes do
 9:     $\mathcal{S} \leftarrow pass_i(p, s, S)$    ▷ refinement passes after feature extraction
10: end for
11: $\mathcal{S} \leftarrow Decimation(\mathcal{S}, s_{ratio})$
12: squares-matrix $\leftarrow Squares(\mathcal{S}, GetCG(p), GetCG(s))$
13: $match \leftarrow$ Belief Propagation($\mathcal{S}$, squares-matrix, $GetCG(p), GetCG(s), \alpha, \epsilon$)
14: return match
```

QBinDiff, whose algorithm is shown in Algorithm 1, consists of three main components and several parameters:

- a similarity matrix (S) encoding the similarity value between each pair of nodes between the two graphs, computed using a preset of heuristics, named *features* that describes the function and the whole program.
- a weight matrix (*squares matrix*) that encodes the induced common edges in both graphs for each possible node assignment,
- a number of user-configurable parameters, among which we distinguish the tradeoff denoted α, the sparsity s_{ratio} and the relaxation parameter ϵ.

First of all, the anchoring phase is used to pre-match imported functions. Optional passes can be defined for a specific initialization of the similarity matrix. Then, features for primary and secondary functions are computed, where *features* denote the data that will be extracted from the binaries: it can be CFG structural features such as the number of BB per function or related to the CG topology like the callee number of a function, or even the assembly instruction mnemonics or the function constants. QBinDiff offers 33 different features in total, that represent the full features set.[4] The similarity matrix S is computed with a weighted linear combination of distances over the primary and secondary feature vectors. Distance candidates are: Canberra, Euclidean, cosine and Haussmann[5]

The similarity matrix S represents the similarity between each node of the primary and each node of the secondary. Intuitively, it should encode domain-specific knowledge coming from the problem instance that the graphs are representing. For binary diffing, an entry $S[i, j]$ close to 1 means that the node i in the primary is really similar to the node j in the secondary, depending on a given metric. On the other hand, a value close to 0 indicates a high dissimilarity between the two. In most cases, the similarity matrix tends to be too large and has to be decimated by using the sparsity ratio $s_{ratio} \in [0, 1]$ that removes the lowest similarity scores that will probably not lead to a match.

The *squares matrix* can be directly derived from the graph structure and the similarity matrix S. A square is defined as a tuple of nodes (A, B, C, D) such that all the following conditions are true:

- Nodes A and D belong to the primary graph.
- Nodes B and C belong to the secondary graph.
- (A, D) is a directed edge in the primary graph.
- (B, C) is a directed edge in the secondary graph.
- Similarity scores for square nodes are positives: S[A,B]>0 and S [D,C]>0.

The tradeoff parameter $\alpha \in [0, 1]$ is similar to a cursor that insists either on the similarity or the graph topology. $\alpha = 0.0$ means that only the graph structure is considered to compute the matches, while the similarity is disregarded. On the contrary, $\alpha = 1.0$ indicates that only the similarity matters. $\epsilon \in [0, 1]$ is a relaxation parameter that helps the Belief Propagation [21] to converge.

B. Ablation : experimental settings

We detail the experimental settings that support the ablation study of QBinDiff. We use the *Dataset-1* [18] that contains various binaries compiled with different options (compilers, compiler versions, optimization levels) from different projects: `zlib`, `unrar`, `curl`, `clamav`, `nmap` and `openssl`. For each project, we randomly split the associated binaries into two sets, named *A* and *B*. As an example, `x64-clang-7-O1-libz.so.1.2.11` will be part of *Dataset-1A* and `x64-clang-5.0-O3-libz.so.1.2.11` of *Dataset-1B*. Similarly, `x64-gcc-7-O0-unrar` will be part of *Dataset-1A* and `x64-clang-3.5-O1-unrar` of *Dataset-1B*.[6] The intersection of binaries from *Dataset-1A* and *Dataset-1B* is empty. In this ablation study, we only use the *Dataset-1A*. It contains 366 binaries from the previous projects, compiled with `clang` or `gcc`, with different versions, from optimization level `-O0` to `-Os`, that represents 425,523 functions. Only `x86-64` binaries are kept. We only diff a binary against another version of itself: this means we can diff a `x64-gcc-7-O2-nmap` against a `x64-clang-3.5-O1-nmap` but not against a `x64-clang-3.5-O1-nping`.[7] Diffing pairs were established at random given the previous conditions. Once they are established, we automatically create ground-truth by matching

[4]A complete list is available on the QBinDiff documentation website [27].

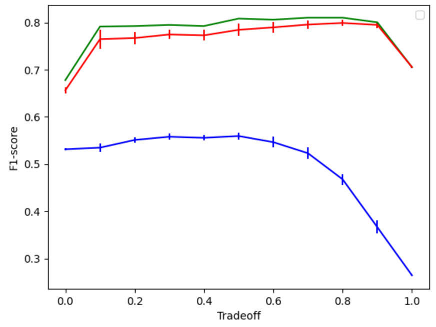
[5]The Haussmann distance is a unique function defined by QBinDiff that combines both the Jaccard index and the Canberra metric, see https://diffing.quarkslab.com/qbindiff/doc/source/params.html#haussmann

[6]Notice that each project has a different number of corresponding binaries. `unrar` has only one binary while `openssl` has 10 binaries.

[7]Such assumption is valid for binary diffing as a reverse-engineer rarely tries to find the exact mapping between two completely different binaries (as `nmap` and `nping` for example), even though they may share some functions. This assumption does not always hold for other binary problems, such as the binary similarity problem [18].

	zlib	curl	clamav	unrar	nmap	openssl
No anchoring	0.72	0.71	0.71	0.72	0.68	0.69
With anchoring	0.81	0.84	0.85	0.79	0.79	0.79
% gain	+12.5	+18.4	+19.8	+9.8	+16.2	+14.5

TABLE I: f1-score anchoring results. Parameters are default ones and the feature set is full.



🟩 standard 🟦 disturbed similarity matrix 🟥 modified adjacency matrices.

Fig. 1: QBinDiff f1-scores on `zlib` for various tradeoffs and settings.

functions with the same name. This ground-truth is globally reliable but may have limits. Compilers can inline or outline functions which add noise to the ground-truth. Next, we strip the binaries so that function names cannot be used for the diffing. At this point, there is only one remaining step before the proper diffing: the export. Most differs rely on binary exporters: BinDiff uses BinExport, Diaphora has its own export format. QBinDiff was created in order to handle different exporters and it now supports BinExport and Quokka [2], respectively developped by Google and Quarkslab. The default QBinDiff parameters p_s were previously defined as the Canberra distance, $\alpha = 0.75$, $s_{ratio} = 0.75$ and $\epsilon = 0.5$ [21].

To evaluate the differ performances, we consider three usual metrics: the recall ($\mathcal{R}$), the precision ($\mathcal{P}$) and the f1-score. They are defined as follows:

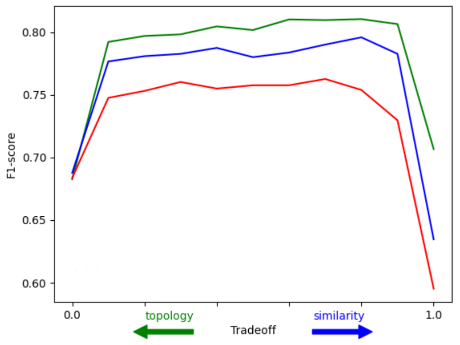
$$\mathcal{P} = \frac{TP}{TP+FP} \quad \mathcal{R} = \frac{TP}{TP+FN} \quad \text{f1-score} = \frac{2\times\mathcal{P}\times\mathcal{R}}{\mathcal{P}+\mathcal{R}}$$

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

Intuitively, the precision denotes how many retrieved items are relevant whereas the recall indicates how many relevant items are retrieved. Precision and recall being complementary metrics, we focus on maximizing the f1-score as it measures a trade-off between precision and recall and requires both metrics to be high.

C. Anchoring

Anchoring (step 1 in Algorithm 1) aims to use imported functions as reliable anchors, especially for dynamically-linked binaries, before any further matching step. We analyze this functionality to determine if it helps QBinDiff to converge, and to evaluate the performance gain compared to the case where we have to match every function candidate. Results are displayed in Table I. f1-score is averaged per project, not per binary. Notice that the f1-score gain with anchoring is significant: instead of having to match every function, we can rely on the imported functions as anchor points to rather match only function clusters that depend on these anchors, which is computationally much easier.



🟥 Data 🟦 Data & CFG 🟩 Data & CFG & CG

Fig. 2: QBinDiff features impact on the `zlib` project.

D. Similarity and CG topology impact

A core QBinDiff parameter is the tradeoff α (step 13 in Algorithm 1) that determines how much QBinDiff should focus on the similarity or the CG topology. Then, a user may decide to focus more on the similarity than the CG structure if it provides better diffing. We choose to observe what happens when we make the tradeoff α vary and respectively, use the default QBinDiff configuration, QBinDiff with a disturbed similarity matrix, and QBinDiff with disturbed adjacency matrices. We disturb the similarity matrix by adding a uniform random noise over its elements. We replace the original adjacency matrices by modified ones using Metropolis-Hasting algorithm [10] (swapping is repeated for 2,000 iterations and self-loops are not allowed). Because these perturbations are mainly built over randomness, we repeat the diffing process with different seeds and average the results. Figure 1 plots results for the `zlib` project. We observe several aspects:

- With standard QBinDiff, the f1-score shows two brutal variations: when we increase the tradeoff from $\alpha = 0.0$ to $\alpha = 0.1$ and from $\alpha = 0.9$ to $\alpha = 1.0$. With $\alpha = 0.1$, the similarity starts to be incorporated to the diffing process. This is highlighted by the significant performance increase. Similarly, when we switch the tradeoff from $\alpha = 0.9$ to $\alpha = 1.0$, the CG topology is not considered anymore, resulting in a lack of information from the CG. Consequently, the f1-score drops suddenly. This demonstrates the necessity to consider both the CG topology and the similarity given by the features extracted from the binaries.
- When we disturb the adjacency matrices and set a lot of weight on the topology, we rely almost only on noisy

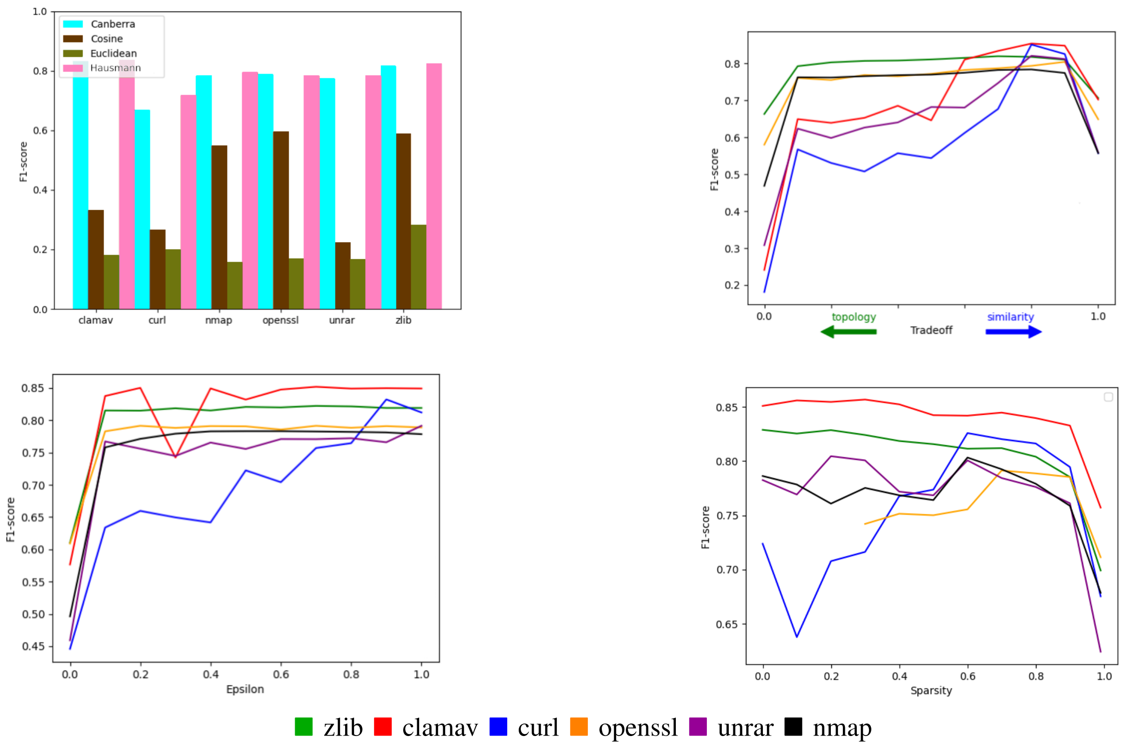


Fig. 3: Distance (d), epsilon (ϵ), tradeoff (α) and sparsity (s_{ratio}) impact on the different projects.

structural information, resulting in a lower f1-score. By increasing α, namely by according more weight on the true similarity, we improve the performances until we reach the standard QBinDiff score. This means that we can deal with a noisy adjacency by using a high tradeoff centered on similarity.

- When the similarity matrix is disturbed, increasing the tradeoff from $\alpha = 0.0$ to around $\alpha = 0.5$ helps to improve the f1-score. Then, the similarity, even noisy, is still necessary for a small α. However, according too much weight on the noisy similarity leads to a significant performance decrease.

E. Features impact

QBinDiff proposes more parameters to adjust than α. Indeed, the similarity matrix is computed by applying a weighted linear combination of distances over features vectors (step 5-6 in Algorithm 1). These features, chosen by the user, describe several aspects of a binary: data-related information, such as the feature *DatName* that indicates data references, CFG-related data like the feature *BBlockNb* which denotes the number of BB inside a function or CG-based features as the *ChildNb* feature, which indicates the number of callees of a given function inside the CG. For this experiment, we define three feature sets, each of them including features from different categories.

The f1-score results for `zlib` binaries, using these features sets and the default parameters p_s, are shown in Figure 2. We observe that with $\alpha = 0.0$, because we do not consider similarity to compute matches, no matter what the features are, the f1-scores are similar for all the features sets. When we start to consider the similarity ($\alpha > 0$), several differences appear. When $\alpha = 1.0$, since we are not considering the CG topology anymore, it becomes particularly useful to choose features extracted from the CG, that make use of that structural information.

The rest of this paper will always consider the default feature set, which includes all the available QBinDiff features at this date[8], especially data, CFG and CG features.

F. Best parameter search

As mentioned before, QBinDiff can be tuned by carefully choosing its parameters $d, \epsilon, \alpha, s_{ratio}$. Finding the best parameters would require testing every combination over the search space, which is not affordable in practice. For this reason, we decided to start with the default parameters p_s (Canberra distance, $\alpha = 0.75$, $s_{ratio} = 0.75$ and $\epsilon = 0.5$) and modify only one parameter at a time (replace the p_s Canberra distance with Haussmann distance for example), to observe the differ's behavior. We make this parameter search on the *Dataset-1A*. Respective plots for d, ϵ, α and s_{ratio} are shown in Figure 3.

From these plots, we deduce that:

- A tradeoff highly focused on the similarity, such as $\alpha = 0.8$ or $\alpha = 0.9$, is better.
- Choosing a high ϵ, such as 0.9 or 1.0, helps QBinDiff to converge faster.

[8]https://diffing.quarkslab.com/qbindiff/doc/source/api/features.html

	zlib	unrar	curl	clamav	nmap	openssl
BinExport	0.73	2.52	5.04	8.89	9.03	2.41
Sqlite	73	223	450	523	968	496
Quokka	0.71	2.19	4.21	8.37	7.93	2.10

TABLE II: Averaged exporting time (s) depending on the exporter for each project of Dataset-1A.

- Canberra or Haussmann are the best distances.
- Increasing parameter s_{ratio} does not imply a significant performance decrease. Indeed, for small projects, such as `zlib` or `unrar`, we notice a slow f1-score loss. For bigger projects, such as `curl`, it may help to converge: indeed, decimating the similarity matrix reduces the number of unlikely candidate matches the differ has to consider, resulting in clear improvement. Choosing a value too high ends up erasing the similarity matrix and matches becomes not possible anymore. We conclude that, for large projects, it may be helpful to choose a middle value (like 0.6) for a better convergence, smaller memory usage and faster computation time.

Each project has its own best parameter set denoted *ppb*: indeed, `zlib` gives the best performance with a Haussmann distance, $\epsilon = 0.7$ (even though the difference with other ϵ values strictly above 0.0 is negligible), $\alpha = 0.8$ and $s_{ratio} = 0$. Moreover, from the best parameters set for all these projects, we can obtain the best averaged parameter set, called *avb*: Haussmann distance, $\epsilon = 0.9$, $\alpha = 0.8$, $s_{ratio} = 0.6$. We finally end up with different parameters sets: *ppb*-zlib, *ppb*-unrar, *ppb*-curl, *ppb*-clamav, *ppb*-nmap, *ppb*-openssl and *avb*, based on *Dataset-1A*.

G. Computational resources

Due to its large complexity, diffing may require a lot of computational resources. We analyze both time and space complexity of diffing solutions using a dedicated server with 64GB of RAM and 16 CPU cores. First, exporting times are compared in Table II for BinExport (used by both QBinDiff and BinDiff), Quokka (used only by QBinDiff) and Diaphora's own exporter, that produces a sqlite database. It is worth mentioning that QBinDiff is not attached to a single exporter but can be used with either Quokka or BinExport. Notice that Quokka is slightly faster than BinExport, whereas the Diaphora sqlite database implies a significant computational overhead. Second, the proper diffing tends to be costly, both in terms of memory usage and computation time. Peak memory usage and computation times can be found in Table III. We especially observe the QBinDiff sparsity effect on diffing. Notice that a low s_{ratio} may be more computationally intensive than higher ratios (leading to an out-of-memory error on large binaries belonging to the `openssl` project) and give lower results due to slower convergence. Contrarily, in some cases, higher ratios help to converge faster and obtain better results, with `openssl` scores that are higher than `nmap` ones for example.

		clamav		nmap		openssl	
		Time	RAM	Time	RAM	Time	RAM
BinDiff		110	298	69	834	56	388
Diaphora3		151	29	651	30	265	30
QBinDiff	$s = 0$	672	823	8675	8339	-	-
	$s = 0.1$	689	801	8702	8329	-	-
	$s = 0.2$	684	712	7424	7167	-	-
	$s = 0.3$	639	617	6712	6171	3844	4884
	$s = 0.4$	656	523	4933	5071	3784	4022
	$s = 0.5$	541	430	4549	3902	2279	3121
	$s = 0.6$	516	347	3146	2900	1509	2301
	$s = 0.7$	489	272	2188	2009	1060	1569
	$s = 0.8$	461	204	1725	1294	1135	999
	$s = 0.9$	447	153	1205	762	528	535
	$s = 0.99$	440	134	880	673	398	408

TABLE III: Required time (s) and memory peaks (MB) needed on the three largest projects for different differs. - means the computation was stopped due to an out-of-memory error.

	ppb-zlib	*ppb*-unrar	*ppb*-curl	*ppb*-clamav	*ppb*-nmap	*ppb*-openssl
zlib	0.83	0.84	0.82	0.84	0.83	0.81
unrar	0.80	0.82	0.81	0.83	0.81	0.78
curl	0.70	0.84	0.83	0.81	0.83	0.80
clamav	0.83	0.77	0.83	0.83	0.82	0.81
nmap	-	-	0.76	-	0.76	0.73
openssl	-	-	-	-	0.75	0.74

TABLE IV: Cross-validation results

IV. Differ's empirical comparison

This Section is dedicated to establishing a comparison between existing binary diffing solutions and QBinDiff. We select BinDiff [5], [7] and Diaphora [14] as they are the most widely used binary differs in reverse-engineering. We also consider GMN, a state-of-the-art binary similarity tool [17], [18], Asm2vec [3], PalmTree [16] and JTrans [30]. Our goal is to show that the QBinDiff capability of arbitrating between CG and similarity with well-chosen features can compete and even outperform standard differs.

A. Experimental setting

The experimental settings are almost identical to the ones used in Section III-B. Because the ablation study was performed on *Dataset-1A*, we consider *Dataset-1B* in this experiment. Notice again there are no binaries in common between the two datasets. *Dataset-1B* contains 770,544 functions distributed in 474 binaries, compiled with either `clang` or `gcc`, for various versions and optimization levels.

The best different parameter sets specific for each project and the best averaged parameter set were previously found in Section III-F. Ideally, we would like to re-use them in this experiment on *Dataset-1B*. However, data leakage is a usual issue when using parameters from one data set to another. However, the performed parameter search is different from hyperparameter search that can be applied on DL models, as there is no learning in the QBinDiff algorithm, which means no overfitting can occur. QBinDiff is before all an optimization algorithm relying on Belief Propagation, which is a statistical ML algorithm.

To demonstrate it, we perform a cross-validation on parameter sets: for each project p, we use its best corresponding

		BinDiff	Diaphora3	QBinDiff-ppb (BinExport)	QBinDiff-ppb (Quokka)	QBinDiff-avb (BinExport)	QBinDiff-avb (Quokka)	GMN	Asm2vec	PalmTree	JTrans
zlib	libz.so.1.2.11	0.85	0.65	0.84	**0.89**	0.82	0.88	0.71	0.19	0.67	0.69
openssl	libssl.so.3	0.81	0.64	0.85	0.85	0.83	**0.86**	0.56	0.17	0.63	0.67
	openssl	0.95	0.68	0.96	**0.98**	0.92	**0.98**	0.59	0.54	0.76	0.72
	libcrypto.so.3	0.76	0.78	0.63	0.80	0.67	**0.82**	0.58	0.01	0.55	0.46
nmap	nping	0.59	0.52	0.74	**0.77**	0.73	**0.77**	0.17	0.17	0.41	0.52
	ncat	0.73	0.58	0.86	**0.92**	0.86	**0.92**	0.24	0.17	0.56	0.67
	nmap	0.8	0.8	0.73	**0.82**	0.73	**0.82**	0.66	0.10	0.61	0.43
clamav	libclamav	0.58	0.46	0.77	**0.81**	0.76	**0.81**	0.43	0.10	0.51	0.53
curl		0.65	0.56	0.83	**0.88**	0.83	**0.88**	0.24	0.22	0.50	0.57
unrar		0.68	0.62	0.82	**0.88**	0.81	0.87	0.22	0.14	0.57	0.69
Averaged		0.74	0.63	0.80	**0.86**	0.80	**0.86**	0.44	0.18	0.58	0.60

TABLE V: f1-scores for different binaries and differs. *ppb* stands for *per project best* and *avb* for *averaged best*.

parameter set *ppb* computed on *Dataset-1A* to perform diffing on *Dataset-1B*. For example, we use *ppb*-zlib to perform diffing on other binaries from *Dataset-1B*. Cross-validation results are available in Table IV, with no significant f1-score differences, meaning there is not artificial inflated f1-score, a phenomenon that could happen with data leakage.

Diaphora results were obtained with its latest version, without any decompiler features. GMN was trained with the default hyperparameters of the available source code and graph attributes (Bag-of-Words over the assembly instruction mnemonics) [18]. Because we cannot train GMN on *Dataset-1B* as it is used for testing, we train GMN using *Dataset-1A*. Once the embeddings are obtained, we compute matches using the Hungarian Algorithm [15]. The same principle is applied to train Asm2vec [3], except the number of random walks is set to 3. PalmTree [16] and JTrans [30] are used with their default parameters. QBinDiff is tested with the two available binary exporters, BinExport [9] and Quokka [2] to see which exporter performs the best. QBinDiff is using all the available features at this date, and we test the two parameter sets found on *Dataset-1A*, as explained in Section III-F: the best parameter set per project, denoted as *ppb* and the best averaged parameter set denoted as *avb*.

B. Results

F1-score results using main standard binaries are displayed in Table V. Several aspects can be highlighted:

- Using QBinDiff with Quokka is much more efficient than using QBinDiff with BinExport, with a f1-score difference of 0.17 for `libcrypto` with *ppb*. This advocates for an increased use of Quokka. Such results can be explained by the fact Quokka exports more information (such as specific cross-references) that BinExport does not.[9]
- Diaphora exhibits lower f1-score results, compared to BinDiff and QBinDiff, because it tends to privilege precision over recall.
- Binary similarity tools show lower scores than standard differs. This can be explained by the fact that HA cannot output a correct mapping if the similarity scores, equivalent to cost, lie in the same range of values with very little standard deviation, which is the case for these trained models. However, PalmTree and JTrans almost compete with Diaphora, with their enriched representations, compared to Asm2vec or GMN.
- There is very little difference in terms of f1-score between results obtained when using *ppb* or *avb*. This means that a QBinDiff user does not have to do the hyperparameter search done in Section III-F, and can simply take the best averaged parameters found earlier to diff its own binaries. Notice that for the `libss.so.3` and `libcrypto.so.3` binaries, *ppb* configuration leads to lower results than the *avb* one. Intuitively, it should not be the case. In fact, if we had performed a complete parameter search over all the parameters $d, \alpha, \epsilon, s_{ratio}$, *ppb* should always produce better results. However, we simply start from a default parameter set, denoted, p_s and make one parameter change at the time. Consequently, *ppb* reflects the best parameter set for each parameter dimension given a default parameter set. Then, it is possible that *avb* outputs slightly better results than *ppb*.

We conclude that QBinDiff using BinExport and even better Quokka, significantly outperforms the other differs and has a f1-score of 12 points better than BinDiff.

V. Discussions

A. Limitations

Initially, we assumed that disassemblies generated by tools like IDA-Pro or Ghidra are accurate. However, these tools may encounter challenges in producing correct disassemblies, especially when faced with techniques like obfuscation or unusual compiler optimizations. Overall, all tools are affected by the limitation of relying solely on disassemblies.

Secondly, these diffing experiments only address one-to-one matching, which may not be adequate for obfuscated or optimized functions where a one-to-many approach would be more appropriate. This issue is quite intricate, and only

[9]See https://blog.quarkslab.com/quokka-a-fast-and-accurate-binary-exporter.htmlformoredetails.

a few solutions have been proposed by researchers, with the effectiveness of these solutions are yet to be definitively established [14].

Third, binaries compiled in -O1 up to -Os apply several optimizations, in particular inlining, which may impact the diffing evaluation as several functions are missing. Studying inlining impact on binary diffing is still an unexplored research area and only few research papers starts to focus it [11], [12].

B. Future work

Extending such a comparison study with cross-architecture binaries seems to be a natural step for binary diffing analysis.

Similarly, adding more binary differs such as SAFE can help to further analyze why binary similarity tools output lower scores compared to binary differs.

VI. Conclusion

In this work, we presented QBinDiff and its core algorithm. We performed an ablation study on its parameters and features and detailed how each of them influences the diffing results. Using the best average parameters, we established a comparison between standard differs and similarity-based diffing and showed that QBinDiff significantly outperforms other differs, especially when the Quokka exporter is used.

References

[1] Gianluca Capozzi, Daniele Cono D'Elia, Giuseppe Antonio Di Luna, and Leonardo Querzoni. Adversarial attacks against binary similarity systems. *arXiv preprint arXiv:2303.11143*, 2023.

[2] Alexis Challande, Robin David, and Guënaël Renault. Quokka: A fast and accurate binary exporter. In *GreHack 2022-10th International Symposium on Research in Grey-Hat Hacking*, 2022.

[3] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 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.

[4] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. Deepbindiff: Learning program-wide code representations for binary diffing. In *Network and distributed system security symposium*, 2020.

[5] Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects (english version). *Sstic*, 5(1):3, 2005.

[6] Frank Emmert-Streib, Matthias Dehmer, and Yongtang Shi. Fifty years of graph matching, network alignment and network comparison. *Information sciences*, 346:180–197, 2016.

[7] Halvar Flake. Structural comparison of executable objects. *DIMVA 2004, July 6-7, Dortmund, Germany*, 2004.

[8] Hao Gao, Tong Zhang, Songqiang Chen, Lina Wang, and Fajiang Yu. Fusion: Measuring binary function similarity with code-specific embedding and order-sensitive gnn. *Symmetry*, 2022.

[9] Google. Binexport. https://github.com/google/binexport, 2016. Accessed: 2023-08-21.

[10] W Keith Hastings. Monte carlo sampling methods using markov chains and their applications. 1970.

[11] Ang Jia, Ming Fan, Wuxia Jin, Xi Xu, Zhaohui Zhou, Qiyi Tang, Sen Nie, Shi Wu, and Ting Liu. 1-to-1 or 1-to-n? investigating the effect of function inlining on binary similarity analysis, 2022.

[12] Ang Jia, Ming Fan, Xi Xu, Wuxia Jin, Haijun Wang, and Ting Liu. Cross-inlining binary function similarity detection, 2024.

[13] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Transactions on Software Engineering*, 49(4):1661–1682, 2022.

[14] Joxean Koret. Diaphora. https://github.com/joxeankoret/diaphora, 2015. Accessed: 2023-08.

[15] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[16] Xuezixiang Li, Yu Qu, and Heng Yin. Palmtree: learning an assembly language model for instruction embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3236–3251, 2021.

[17] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. In *International conference on machine learning*, pages 3835–3845. PMLR, 2019.

[18] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[19] Luca Massarelli, Giuseppe A Di Luna, Fabio Petroni, Leonardo Querzoni, Roberto Baldoni, et al. Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. In *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR)*, pages 1–11, 2019.

[20] Xiaozhu Meng and Barton P Miller. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 24–35, 2016.

[21] Elie Mengin. *Binary Diffing as a Network Alignment Problem*. PhD thesis, Universite Paris 1-Pantheon Sorbonne, 2021.

[22] Elie Mengin and Fabrice Rossi. Binary diffing as a network alignment problem via belief propagation. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 967–978. IEEE, 2021.

[23] Elie Mengin and Fabrice Rossi. Improved algorithm for the network alignment problem with application to binary diffing. *Procedia Computer Science*, 192:961–970, 2021.

[24] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[25] Jiang Ming, Dongpeng Xu, and Dinghao Wu. Memoized semantics-based binary diffing with application to malware lineage inference. In Hannes Federrath and Dieter Gollmann, editors, *ICT Systems Security and Privacy Protection*, pages 416–430, Cham, 2015. Springer International Publishing.

[26] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680*, 2020.

[27] Quarkslab. Qbindiff - github. https://diffing.quarkslab.com/qbindiff/doc/source/features.html#id1, 2023. Accessed: 2023-09-7.

[28] Paria Shirani, Lingyu Wang, and Mourad Debbabi. Binshape: Scalable and robust binary library function identification using function shape. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 301–324. Springer, 2017.

[29] Sami Ullah and Heekuck Oh. Bindiff nn: Learning distributed representation of assembly for robust binary diffing against semantic differences. *IEEE Transactions on Software Engineering*, 48(9):3442–3466, 2021.

[30] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. Jtrans: Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 1–13, New York, NY, USA, 2022. Association for Computing Machinery.

[31] Lei Zhao, Yuncong Zhu, Jiang Ming, Yichen Zhang, Haotian Zhang, and Heng Yin. Patchscope: Memory object centric patch diffing. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 149–165, 2020.