Journal of Computer Virology and Hacking Techniques (2019) 15:307–323
https://doi.org/10.1007/s11416-019-00339-6

ORIGINAL PAPER

Function matching between binary executables: efficient algorithms and features

Chariton Karamitas[1,2] · Athanasios Kehagias[2]

Received: 23 April 2019 / Accepted: 5 September 2019 / Published online: 19 September 2019

Abstract
Binary diffing consists in comparing *syntactic* and *semantic* differences of two programs in binary form, when source code is unavailable. It can be reduced to a *graph isomorphism* problem between the *Control Flow Graphs*, *Call Graphs* or other forms of graphs of the compared programs. Here we present REveal, a prototype tool which implements a binary diffing algorithm and an associated set of features, extracted from a binary's CG and CFGs. Additionally, we explore the potential of applying *Markov lumping* techniques on function CFGs. The proposed algorithm and features are evaluated in a series of experiments on executables compiled for *i386*, *amd64*, *arm* and *aarch64*. Furthermore, the effectiveness of our prototype tool, code-named REveal, is assessed in a second series of experiments involving clustering of a corpus of 18 malware samples into 5 malware families. REveal's results are compared against those produced by Diaphora, the most widely used binary diffing software of the public domain. We conclude that REveal improves the state-of-the-art in binary diffing by achieving higher matching scores, obtained at the cost of a slight running time increase, in most of the experiments conducted. Furthermore, REveal successfully partitions the malware corpus into clusters consisting of samples of the same malware family.

1 Introduction

In the current paper we present REveal, a prototype tool which implements a *binary diffing* algorithm and an associated set of function features extracted from the CFGs (*Control Flow Graphs*) and CG (*Call Graph*) of a binary program.

Binary diffing is the process of reverse engineering two programs (when source code is not available) in order to study their *syntactic* and *semantic* differences [12,25,26]. It can be performed by *function matching*, which assists in spotting differences between the two programs and minimizes the manual labor required to understand and process code and data modifications. Function matching can be reduced to the *graph isomorphism* problem [8] between the compared subjects' CFGs, CGs or other graphs modeling the subjects' flow of information. Since graph isomorphism belongs to the *NP* class, in the current paper we present computationally practical methods to obtain an approximate solution.

REveal is introduced to attain these goals; it is evaluated in a series of experiments involving open-source software, as well as unpacked versions of malware samples found in the wild; in both cases it achieves excellent results, as will be seen in the sequel. REveal is currently a work-in-progress prototype tool which is being actively developed. The project will be open-sourced in the near future, once a production milestone has been reached.

We propose a set of function features extracted from a binary's CG and CFGs; these can be used by variants of the BinDiff algorithm [8,9,11] to (i) build a set of initial exact

This paper is a significantly expanded version of [22] which was presented at 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). Since our last publication, we have received useful and constructive feedback from various individuals we would like to thank; Joxean "matalaz" Koret for developing and open-sourcing Diaphora. Balint "buherator" Varga-Perke for the feedback and discussions on binary diffing. J-Michael Roberts for running VirusShare, a priceless resource for malware related research, and for giving us access to his incredible malware database. Last but not least, we would like to thank Shaul Holtzman and Intezer for giving us access to their platform, code-named Analyze, and for providing us with unpacked executables of the malware samples analyzed in Sect. 6.

✉ Chariton Karamitas
huku@census-labs.com

Athanasios Kehagias
kehagiat@auth.gr

[1] CENSUS S.A., Thessalonıki, Greece

[2] Department of Electrical and Computer Engineering, Aristotle University of Thessaloniki, Thessalonıki, Greece

matches with minimal false positives, by scanning for unique perfect matches, and (ii) to propagate approximate matching information, for example, by using a nearest-neighbor scheme. One of the proposed features is obtained by applying *Markov lumping* to function CFGs (to our knowledge, this technique has not been previously studied). The feature extraction process is repeated for each program under comparison and the resulting information is utilized by a number of algorithms implementing graph matching strategies. The aforementioned algorithms produce a mapping between functions of the compared subjects, where each element in the map associates a function from the primary program to a similar function in the secondary program.

We evaluate our approach by a series of experiments on binary executables compiled for the four, most widely used, computer architectures: *i386*, *amd64*, *arm* and *aarch64*. Additionally, REveal is assessed in a second series of experiments involving clustering of a corpus of 18 malware samples into 5 malware families. Malware classification has always been an actively researched field with many applications in the computer industry and this latter set of experiments highlights our tool's ability to solve real life problems. Our results are compared to those obtained by Diaphora [7]. We conclude that REveal improves the state-of-the-art in binary diffing by achieving higher matching scores, at the cost of higher running times. Furthermore, REveal successfully partitions the malware corpus into clusters consisting of samples of the same malware family.

The remaining of this paper is organized as follows. In Sect. 2 we review previous work on binary diffing. In Sect. 3 we present mathematical notation, definitions and assumptions on which our prototype system is based. In Sect. 4 we propose the set of function features, utilized by the binary diffing algorithms of Sect. 5. In Sect. 6 we evaluate REveal in the two aforementioned batches of experiments. Furthermore, we compare our results to those obtained by Diaphora [7]. As already mentioned, we conclude that REveal improves the binary diffing state-of-the-art. Finally, in Sect. 7 we summarize and propose several future research directions.

2 Previous work

Binary diffing has been used in the software engineering industry in several application domains.

1. **Malware classification** [3,4,17]: Given a binary executable, classify the subject as either innocent or suspicious. Malware classification works by extracting features from the binary executable and locating matching candidates within a database of known, pre-analyzed malware samples. Since the database size may be hundreds of terabytes, both performance and effectiveness are very important. With the number of sophisticated cyber-attacks, ransomware, malware and other forms of malicious software constantly increasing [27,28], both antivirus companies and independent researchers have entered this research field.
2. **Patch analysis** [20]: Quickly study, analyze and evaluate product updates for the presence of software vulnerabilities. By diffing shared libraries and binary executables of the investigated system, pre- and post-update, a security researcher reduces the set of changes that need to be studied.
3. **Plagiarism detection** [30]: In an online course management system, for example, diffing can be used to assign similarity scores to solutions submitted by students and detect cheating. In the software industry, diffing can be employed to detect copyright infringement and other forms of intellectual property theft.
4. **Propagation of profiling information** [33,34]: Binary diffing techniques may be used to port existing profile information, which was initially assembled from a previous version of the same program, to a newer one, thus allowing the software testing team to only focus on evaluating new features of a software suite.

As already mentioned, many binary diffing algorithms reduce binary diffing to the study of graph isomorphism of CFGs and/or CGs and separating the diffing process in two phases.

1. During *pre-filtering*, two binary executables, the matching candidates, are disassembled and their CGs are recovered through a combination of standard disassembly techniques. For each vertex in the two aforementioned CGs (that is, for each function in each executable) the corresponding CFG is formed and a feature vector is extracted. Then an initial 1–1 mapping between the CFGs of the two matching candidates is formed by looking for unique entries in the aforementioned feature vector sets. This initial mapping constitutes a set of *fixed points* and effectively reduces the problem space by decreasing the number of possible vertex permutations theoretically required to find the isomorphism between the two CGs.
2. In the *propagation phase*, the initial 1–1 mapping is expanded. This is usually done by examining neighbors of already matched vertices in the compared CGs. Neighbors which, according to their features, match exactly or differ only slightly, are added in the mapping thus expanding the isomorphism. Other forms of propagation may also be used, as discussed in Sect. 5.

The two-phase process described above can be repeatedly executed until the mapping cannot be expanded any more.

Exact differences between the compared executables can be spotted by examining each basic block of each matched CFG.

For example, Dullien et al. [8,9,11] form *feature vectors* (composed of the number of vertices and edges as well as the out-degree of each CFG) which they use to locate perfect matches in the compared executables. Later publications [2,10] extend Dullien's work by adding or modifying features in the aforementioned feature vector. Variants of these algorithms have been extensively used in commercial tools like Zynamics BinDiff [35]. Also, Diaphora [7] (an open source application and nowadays the industry standard in binary diffing) uses a set of several graph metrics, computed over function CFGs, to match unique functions in two executables.[1]

Other binary diffing algorithms are based on dynamic analysis techniques [25,26], back-tracking [33,34], MCS [10] or *simulated annealing* [23]. Even though our work is mostly applicable to BinDiff algorithm variants, it may be possible to apply it to the abovementioned algorithms.

3 Preliminaries

We define a program p to be a set of N functions; $p = \{f_i \mid i = 0, 1, \ldots, N-1\}$. Binary diffing involves comparing two programs, namely p_1, the primary subject, and p_2, the secondary subject, and forming a 1–1 mapping M, that corresponds functions of p_1 to functions of p_2, $M = \{f_i^{p_1} \rightarrow f_j^{p_2} \mid i < |p_1|,\ j < |p_2|\}$.

We represent *digraphs* (i.e., directed graphs) with the notation $G = \langle V, E\rangle$, where V is the digraph's vertex set and $E \subseteq V \times V$ is the digraph's edge set (or arch set). Given a vertex $v \in V$, we define the set of *successors* of v as $succ(v) = \{s \mid (v,\ s) \in E\}$ and the set of *predecessors* of v as $pred(v) = \{p \mid (p,\ v) \in E\}$.

For each function f, in a program p, we define a digraph $CFG_f = \langle V_{CFG_f},\ E_{CFG_f}\rangle$, where V_{CFG_f} is the set of f's basic blocks (straight-line machine code sequences with no branches in, except to the entry, and no branches out, except at the exit). The set of edges E_{CFG_f} denotes the possible execution flow paths between the function's basic blocks. If $e = (b_{src},\ b_{dst}) \in E_{CFG_f}$, then control flow can reach basic block b_{dst} immediately after b_{src} (i.e., $b_{dst} \in succ(b_{src})$). Digraph CFG_f is usually referred to as f's *Control Flow Graph*.[2]

A program p can also be treated as a *digraph of digraphs*, referred to as the program's *Call Graph* (CG). CG_p of program p is a digraph whose vertices correspond to individual function CFGs, that is $V_{CG} = \{CFG_f \mid f \in p\}$. For each control transfer instruction in basic block $b_{src} \in V_{CFG_{f_{src}}}$ that transfers execution to basic block $b_{dst} \in V_{CFG_{f_{dst}}}$, where $f_{src} \neq f_{dst}$, an edge $(CFG_{f_{src}},\ CFG_{f_{dst}})$ exists in E_{CG}.

We assume that, $succ(v)$ and $pred(v)$ return the corresponding vertex sets ordered by their addresses. I.e., if $succ(v) = \{s_0,\ s_1, \ldots, s_{n-1}\}$ for some $v \in V$, then $address(s_0) < address(s_1) < \cdots < address(s_{n-1})$, where $address : V \rightarrow \mathbb{N}$ returns the address of a vertex in program memory.

A *Markov chain* [15,16] is a discrete time, discrete state space stochastic process which satisfies the *Markov property*: the probability of the process entering the next state, depends only on the present state. The *transition matrix* p has elements (*transition probabilities*) p_{ij} such that:

$$\begin{aligned}\forall i, j : p_{ij} = &\Pr(\text{Process at time}\\ &t+1 \text{ is in state } j \mid \text{Process at time } t \text{ is in state } i)\end{aligned}$$

Markov chain lumping [6,32] is a process introduced to reduce computations on Markov chains. It merges equivalent states from the original Markov chain into super-states, called *components*, of a new system, also obeying the Markov property. For any two components P_i and P_j the following property holds true:

$$\forall n_1,\ n_2 \in P_i : \quad \sum_{s\in P_j} p_{n_1 s} = \sum_{s\in P_j} p_{n_2 s}$$

Standard Markov analysis algorithms are then applied to the new, reduced system.

To evaluate our algorithm and test the effectiveness of the proposed features, we need the ground truth on the function matching results. To this end we use an oracle function $oracle : p_1 \times p_2 \rightarrow \{True,\ False\}$. Given two compared subjects, p_1 and p_2 and corresponding functions f_{p_1} and f_{p_2}:

$$oracle(f_{p_1},\ f_{p_2}) = \begin{cases} True & \text{if } \ f_{p_1} = f_{p_2} \\ False & \text{otherwise} \end{cases}$$

Implementing such an oracle is critical for experimental evaluation. The simplest approach makes use of debug information embedded in the binary executables. For each reported match, the two function names are extracted from each executable's debug section, demangled and checked for equality. If the two names are identical, or if they are sufficiently similar (e.g. their Levenshtein distance [24] is small), the match is considered successful.

In the rest of the paper, we make the following assumptions.

[1] It should also be mentioned that Diaphora does not implement any kind of propagation phase.

[2] Even though the CFG (and the CG mentioned in the sequel) are *digraphs* we will follow standard usage and call them *graphs*.

1. Symbol names and debugging information of compared binaries is not taken into account during binary diffing, except for ground truth computations.
2. Source code of compared subjects is not available.
3. We only consider exact, perfect matches.
4. Speed is not our primary interest. Even if binary diffing takes several hours to complete, it still saves hundreds of hours of manual processing.
5. Compared binaries are neither obfuscated nor packed.[3]

4 Features

For each function, we extract a feature vector composed of 10 features. These describe the CFG's (i) structural characteristics (vertex and edge classification, graph signatures), (ii) spatial characteristics (inlinks, outlinks, lumped transition matrix) and (iii) semantic content (function type, instruction and data histogram).

4.1 Function type

To make use of advanced runtime facilities, offered by modern operating systems, executable code usually depends on the presence of standard or third-party libraries, from which it references code and data symbols (e.g. use of *printf*() residing in **libc.so**). This dependency manifests itself either during link or run time. When a dynamically-linked executable is loaded, for example, the dynamic loader is responsible for resolving these external symbols at run time. On the other hand, when a program is statically linked against a library, the linker is responsible for performing the same task at link time. Such external symbols are referred to as *imported symbols* or just *imports*. In contrast, executable code can expose certain functionality to the operating system, so that other programs can freely make use of, via a series of *exported symbols* or just *exports*. Last but not least, executable code usually consists of code and data not belonging to any of the aforementioned categories. This is usually code, which performs the actual computations of the program, and data it acts on. We refer to these as *hidden* symbols.

Naturally, with only a few exceptions not discussed here, when matching functions of an executable against functions of another, imports from the first should be matched against imports from the second. The same principle should be followed for exports and hidden symbols. Towards this, our very first feature is a simple integer signifying the category each function belongs to. During the matching process, functions in the same group are matched with each other.

[3] Initial pre-processing steps of de-obfuscating and unpacking the the executables may be necessary. In fact, such preparatory steps have been used in the past by various authors (e.g. [3,4,17]).

4.2 Vertex and edge taxonomy

In previous publications [2,8–11] feature vectors extracted from function CFGs include two features which reflect a digraph's overall structure; the number of vertices (basic blocks) and the number of edges in the corresponding function. These two important structural characteristics can significantly speed up the pre-filtering phase, by discarding candidates with incompatible vertex and edge counts. But their pruning power is limited and they can easily introduce non-negligible inaccuracies and/or latencies in the overall process.
For example, take the digraphs depicted in Fig. 1a and b. These might be the CFGs of two functions under comparison. They are both composed of 6 basic blocks and 7 edges. Consequently, a traditional BinDiff algorithm variant would pick those two as potential matching candidates. However they have obvious differences. Figure 1a represents a function that executes linearly, while Fig. 1b represents a function whose basic block **F** *traps* the execution flow; it might be a dispatch loop (e.g. malware virtualization obfuscation dispatch loop as described in [26]), or a *no-return* vertex (as recognized by IDA Pro [13]).

We propose the following vertex taxonomy (in which a basic block may belong to more than one category) as a better way to distinguish CFGs by their topological vertex characteristics.

1. **Normal**. All vertices, including those that do not belong to any of the remaining categories.
2. **Entry points**. Program execution enters the function in question via one of these vertices. A single function may have more than one entry points.
3. **Exit points**. Control flow leaves the function via one of these basic blocks (e.g. basic blocks ending with a RET instruction, or a CALL instruction when *tail-call optimization* is used). Each function may have several exit points.
4. **Traps**. Every vertex with a single edge looping into itself.
5. **Self-loops**. A vertex with an edge to itself and, possibly, edges to other vertices. Traps are also self-loops.
6. **Loop heads**. Loops' *header* [29] vertices.
7. **Loop tails**. Vertices within a loop's *body* [29] with edges to the loop's *header*.

So we propose a feature sub-vector with 7 elements; the i-th element holds the number of vertices of category i defined above. E.g., the feature vector extracted from the digraph at Fig. 1a is $\begin{bmatrix}6 & 1 & 1 & 0 & 0 & 0 & 0\end{bmatrix}$ and that from Fig. 1b is $\begin{bmatrix}6 & 1 & 1 & 1 & 0 & 0\end{bmatrix}$. These clearly expose the dissimilarity of the two CFGs.

Compare Fig. 1a and c. Both digraphs have the same number of vertices and edges and are only composed of *normal*

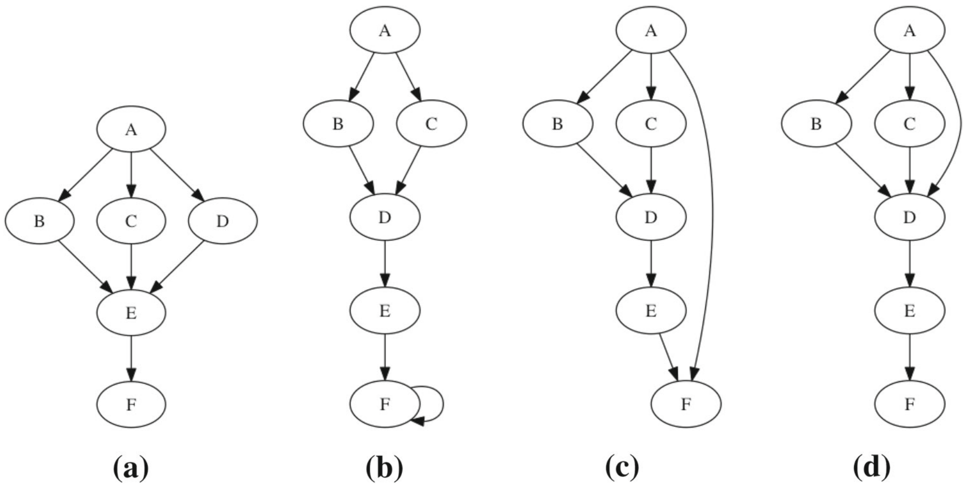

Fig. 1 Several example CFGs. They all share the same number of vertices and edges, yet, we need to come up with ways of distinguishing one from another

vertices, a single *entry point* and a single *exit point* (according to the previously defined taxonomy). Yet they are very different; 1c, unlike 1a, appears to perform a sanity checking on its input arguments and, if checks are not passed, execution flows directly to basic block **F**, the function's exit point. The number of edges alone, is not enough to distinguish 1a and c; we need more robust features.

Hence we introduce a taxonomy of edges, similar to the one described by Tarjan [31]. Using DFS traversal, we classify edges into the following categories.

1. **Basis edges**. Given a topological sort of the vertices of graph $G = \langle V, E \rangle$ and a DFS which respects this topological sort such that: if $(u,\ v) \in E$ then $depth(u) < depth(v)$ unless v is an ancestor of u[4] (this is satisfied by the graphs of Fig. 1a, d), then $(u,\ v)$ is a basis edge if $depth(v) = depth(u) + 1$.
2. **Forward edges**. Like basis edges above but $depth(v) > depth(u)+1$. Forward edges connect ancestors with non-direct descendants.
3. **Back edges**. Back edges connect descendants with their ancestors (i.e. $depth(v) < depth(u)$). Self-loops are also, sometimes, considered back edges as well.
4. **Cross-links**. Edges between vertices belonging to different DFS sub-trees.

Hence we propose a second feature sub-vector of 4 elements. The i-th element holds the number of edges falling in category i. Accordingly 1a is described by $\begin{bmatrix}7 & 0 & 0 & 0\end{bmatrix}$ and 1c by $\begin{bmatrix}6 & 1 & 0 & 0\end{bmatrix}$. The two CFGs can now be separated.

[4] Where the *depth*() function is the one resulting from the aforementioned DFS.

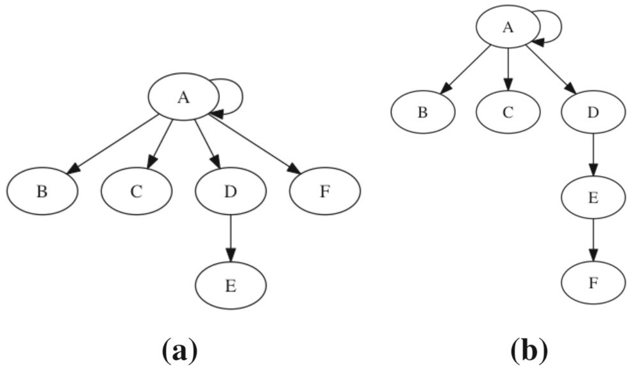

Fig. 2 Immediate dominator trees for 1c and d

4.3 Digraph signatures

The exact vertex and edge classification features of two digraphs do not give any insight on how the latter are actually laid out. Consider Fig. 1c and d. Both digraphs have a vertex feature vector of $\begin{bmatrix}6 & 1 & 1 & 0 & 0 & 0 & 0\end{bmatrix}$ and an edge feature vector of $\begin{bmatrix}6 & 1 & 0 & 0\end{bmatrix}$ and hence cannot be distinguished based solely on these two feature vectors (i.e., there exist many different digraphs with 6 vertices and 7 edges).

Note that the above digraphs have different *dominance relations*. E.g., in Fig. 1c the immediate dominator of vertex **F** is **A**, while in Fig. 1d is **E**. The *dominator trees* of Fig. 1c and d can be seen in Figs. 2a and b respectively.

To encode this information in a feature vector we first build the immediate dominator tree of the subject digraph and then visit its vertices in a depth-first fashion. During the DFS traversal, we incrementally construct a bit-vector that reflects the digraph's layout in the following way; whenever a vertex is first visited, we append a **1** to the bit-vector. When DFS leaves the vertex in question, a **0** is appended. For example, given 2a, a DFS traversal of **A B C D E F**, produces the bit signature **110101100100**, or **D64** in hexadecimal. Simi-

```
tk_debug_sleep_time_open proc near
  55                    push rbp
  48 89 F7              mov rdi, rsi
  31 D2                 xor edx, edx
  48 C7 C2 20 67 0B 81 mov rsi, offset tk_debug_show_sleep_time
  48 89 E5              mov rbp, rsp
  E8 7B 66 0C 00        call single_open
  5D                    pop rbp
  C3                    ret
tk_debug_sleep_time_open endp
```

(a)

```
tk_debug_sleep_time_open proc near
  55                   XED_IFORM_PUSH_GPRv_50
  48 89 F7             XED_IFORM_MOV_GPRv_GPRv_89
  31 D2                XED_IFORM_XOR_GPRv_GPRv_31
  48 C7 C2 20 67 0B 81 XED_IFORM_MOV_GPRv_IMMz
  48 89 E5             XED_IFORM_MOV_GPRv_GPRv_89
  E8 7B 66 0C 00       XED_IFORM_CALL_NEAR_RELBRd
  5D                   XED_IFORM_POP_GPRv_51
  C3                   XED_IFORM_RET_NEAR
tk_debug_sleep_time_open endp
```

(b)

Fig. 3 Example of **a** assembly instructions and **b** their instruction forms

larly, in 2b, traversing the vertices in the order **A B C D E F** produces the bit signature **110101110000**, or **D70**.

Hence our next two features are the bit signatures of both the original function CFG and its immediate dominator tree. The corresponding features extracted from 1c and d are the hexadecimal numbers **F84**, **D64** and **F84**, **D70** accordingly. Notice how, in this specific example, the CFG signatures happen to be the same and the two functions only differ in their dominator trees. More elaborate signature schemes have been used for indexing graph structures [4]; such schemes are also applicable in our case.

4.4 Inlinks and outlinks

For each vertex we add as features the number of inlinks (number of CALL sites of this function) and the number of outlinks (number of CALL instructions in the function body); most previous work uses only the latter. Following [4,10] we include the former in our feature vector as an extra step towards a more complete solution.

4.5 Instruction histogram

The features discussed so far encode *structural* characteristics. We also want features that encode an approximation over the *semantics* of a function. For instance, in [10] the authors present the *discovRE* system, which classifies instructions in 4 categories based on their functionality (arithmetic, logic, data transfer, redirection). This classification is then used in a 4-element feature vector identifying each function in the program's CFG. Even though we believe this to be the right choice, separating instructions in a so abstract taxonomy results in extended information loss and consequently in reduced unique matches.

Based on this idea, we make use of Intel's XED library [18] and *pyxed* [21], for *i386* and *amd64*, and IDA Pro's built-in disassembler, for *arm* and *aarch64*, to classify instructions based on their instruction form. As opposed to the classification used in [10], the instruction form also gives an overview of the type of operands used in each instruction, effectively abstracting away constant values and register names. Last but not least, the histogram of the distribution of instruction forms is computed and appended in each function's feature vector.

Figure 3a shows a random function, consisting of a single basic block, taken from a compiled Linux kernel binary. Figure 3b shows the same function, however, instructions have now been replaced with their instruction forms.[5]

Clearly, different compilers and optimization levels can have a significant impact on the instruction and operand types used in the program. This, in turn, affects the binary diffing process, as these variations can produce dissimilar machine code for the exact same source, resulting in reduced exact matches. A similar problem arises when trying to compare two variants of the same program compiled for different computer architectures. Indeed, assembly code produced for **arm64** cannot be directly compared to this produced for **amd64**. A widely accepted solution for cross-architecture operations on machine code is lifting instructions to an architecture-independent intermediate representation and comparing the result. Future research towards this direction is discussed in the Conclusion.

4.6 String histogram

Functions usually perform simple operations with ASCII strings. For example, when a function calls *printf*(), the first argument passed to *printf*() (the format string) is usually a constant. Such distinctive constants are extremely useful for reverse engineers as they can disclose a wealth of information for the inner workings of a function. In binary diffing, examining the string constants referenced from a CFG can reveal potential matching candidates; thus, we would like to encode this information in our feature vector.

An obvious, yet naive, choice is to extend each feature vector with all strings accessed by the corresponding function. However this has two drawbacks: (i) the overall size of the feature vector is increased linearly with respect to the

[5] For the exact meaning of the instruction form constants see [19].

number of strings accessed by a function and (ii) computing a similarity metric between the feature vectors of two functions becomes more complicated, as one has to resort to computing a similarity score between individual strings. Instead, we compute the histogram of all characters in all strings accessed by the examined function and use the result as a feature. This allows for detecting exact matches more efficiently, by just comparing two histograms for equality, and partial matches more naturally, by, for example, using a cross-entropy metric to compute a similarity score between two seemingly unrelated histograms.

This feature can be thought of as the program's data elements manipulated by the function in question. It can also be extended to account not only for strings, but for any type of constant data accessed by a function. However this approach introduces complications which are outside of the scope of this paper.

4.7 Markov lumping of CFGs

Treating a CFG as a Markov chain is not a new idea [5]. Each basic block, in a function CFG, is assumed to be a separate state and each edge a transition with a certain assigned probability. What is most important in such a treatment, however, is the model/assumptions used in order to assign the aforementioned transition probabilities. Care must be taken as this will eventually affect the effectiveness of the overall analysis and the soundness of its results. For example, at [5] (Figs. 11–15), the edge weights of the example program seem to be known in advance and no insight on how they were deduced is given. When specific program inputs are not known, determining the probability of a program reaching a specific state is generally undecidable.

Abstractly, transition probabilities are just a set of numbers obeying the following properties:

$$\forall i : \sum_{j=0}^{n-1} p_{ij} = 1, \quad \forall i, j : 0 \leq p_{ij} \leq 1.$$

It is not necessary to treat the p_{ij}'s as probabilities; they can be understood as weights whose interpretation depends on the application domain and reflects the relationship between a source and destination. In our case, the entities are basic blocks and the associated weights can be assigned based on their properties. The resulting *transition* matrix can then be viewed as a summary of the system's overall entity relationships to which standard Markov analysis tools can be applied. Note also that such a transition matrix reflects the corresponding CFG's spatial characteristics, as each p_{ij} quantifies the relationship between *i* and one of its neighbors. By extracting and comparing the matrices of two CFGs, we gain insight on how similar or different the corresponding functions are. In our experiments we evaluated the following weight assignment schemes.

1. Each transition receives a uniform probability. We did not find this scheme to be of any practical use, apart from an analysis similar to the one described at chapter 15–3 of [5].
2. Edges towards basic blocks with a higher number of instructions are assigned larger weights. The rationale here is that computer programs generally receive and process user input and thus execution is more likely to reach a basic block with more instructions, as this is where (most likely) actual processing will take place. Even though this scheme gives a more detailed overview of the CFG's layout, it is inappropriate for detecting exact matches, as, small rearrangements of instructions result in significant changes in the transition matrix. This, however, makes it more suitable for detecting inexact matches.
3. The last scheme takes into account various *locality optimizations* [1] performed by the compilers which built the executables. During certain optimization passes, a function's CFG may be split into several *chunks* which might be placed in distant physical locations within the executable (and consequently in distant virtual memory addresses as well). Basic blocks are distributed among the aforementioned chunks, according to the compiler's view of how often each one might be reached when the function in question is executed. In this manner a function is split into *hot paths* (highly likely to be executed) and *cold* ones (less likely to be reached). Hence, we assign to an edge weight inversely proportional to the distance between its source and destination (where distance is measured by the addresses the code is located). This choice seems natural and conforms to the operation of an optimizing compiler.

We found the third of the above schemes to yield the best trade-off between accuracy and matching power.

Algorithm 1 Weight assignment

```
1: procedure $assign\_edge\_weights(CFG_f)$
2:     $chunks \leftarrow cfg\_chunks(CFG_f)$
3:     $chunk\_weights \leftarrow \{|chunks|, |chunks| - 1, \ldots, 1\}$
4:     for all $v \in V_{CFG_f}$ do
5:         $weight \leftarrow 0$
6:         for all $s \in succ(v)$ do
7:             $i \leftarrow chunk\_index(chunks, s)$
8:             $weight \leftarrow weight + chunk\_weights[i]$
9:         $weight \leftarrow 1/weight$
10:        for all $s \in succ(v)$ do
11:            $i \leftarrow chunk\_index(chunks, s)$
12:            $(v, s)_{weight} = weight * chunk\_weights[i]$
```

Algorithm 1 shows the process of assigning weight values to the edges of an arbitrary CFG. It begins by retrieving the list of chunks of function with CFG CFG_f and stores them in variable *chunks* (line 2). For chunk i, $chunk\_weights[i]$ contains an initial reference weight (line 3). For example, for a function with 4 chunks, the first chunk (index 0) is given a weight of 4, the highest weight, while remaining chunks are given successively lower weight values ranging from 3 to 1. The core of the algorithm is a single loop that traverses all CFG vertices (line 4). For each vertex, the sum of its edge weights is assembled in *weight* by examining its successors' chunk indexes one by one (line 8). Finally *weight* is converted to a weight factor (line 9), which is then multiplied with the corresponding weight of each successor's chunk (line 12) to give the final value of an edge's weight.

Before a lumping algorithm can be used, the states of the original system have to be split in a set of, so called, *initial components* $P_0 = \{P_{0,0}, P_{0,1}, \ldots, P_{0,N-1}\}$. Our reference implementation begins by distributing basic blocks into components so that no basic block is in the same component with any of its successors (except if the basic block in question has a self-loop):

$$\forall v \in V_{CFG_f} : v \in P_{0,i} \Rightarrow (succ(v) - \{v\}) \cap P_{0,i} = \varnothing$$

Algorithm 2, which is composed of several procedures, is responsible for returning the set of initial components given an arbitrary CFG. We begin our description from procedure $find\_component()$. Given the current set of components P_0 and an arbitrary CFG vertex v, $find\_component()$ iterates through all components defined so far (line 3) and locates one which contains vertices (i) that are not v's successors (line 4) (ii) that do not have v as successor (line 6). If no such component exists, the empty set is returned.

Procedure $process\_component()$, as its name suggests, processes one component $P \in P_0$. It iterates through all vertices in P (line 15) and looks for incompatibilities (line 16). If such an incompatibility is found, the vertex under examination is removed from its component (line 18) and a another, compatible, component is looked up (line 19). Recall that, if no compatible component is found, a new empty set is returned. Continuing, v is added to the newly found component Q (line 20) and the latter is added in the component set if not already there (line 22). Practically, $process\_component()$ splits P into one compatible and several incompatible parts. The incompatible vertices are moved to either a new component, or an existing compatible one.

The main procedure of algorithm 2, which we have named $create\_initial\_components()$, is just a fixed-point loop that processes components one-by-one, until no more components can be split.

Algorithm 2 Initial partitioning algorithm

```
1:  procedure $find\_component(v, P_0)$
2:      $Q \leftarrow \varnothing$
3:      for all $P \in P_0$ do
4:          if $(succ(v) - \{v\}) \cap P = \varnothing$ then
5:              $r \leftarrow True$
6:              for all $q \in P$ do
7:                  if $v \in succ(q)$ then
8:                      $r \leftarrow False$
9:                      break
10:             if $r = True$ then
11:                 $Q \leftarrow P$
12:                 break
        return $Q$

13: procedure $process\_component(P, P_0)$
14:     $change \leftarrow False$
15:     for all $v \in P$ do
16:         if $(succ(v) - \{v\}) \cap P \neq \varnothing$ then
17:             $change \leftarrow True$
18:             $P \leftarrow P - \{v\}$
19:             $Q \leftarrow find\_component(v, P_0)$
20:             $Q \leftarrow Q \cup \{v\}$
21:             if $Q \notin P_0$ then
22:                 $P_0 \leftarrow P_0 \cup \{Q\}$
        return $change$

23: procedure $create\_initial\_components(CFG_f)$
24:     $P_0 = \{V_{CFG_f}\}$
25:     $change \leftarrow True$
26:     while $change = True$ do
27:         $change \leftarrow False$
28:         for all $P \in P_0$ do
29:             $change \leftarrow change \vee split(P, P_0)$
        return $P_0$
```

Finally we apply a lumping algorithm [6] to refactor the sets in P_0, and use the lumped transition matrix as our feature vector's last element (other matrix characteristics, like its eigenvalues, might be used instead). An overview of the overall lumping process appears in Algorithm 3; $markov\_lumping()$ is described in [6].

Algorithm 3 Overall lumping

```
1: procedure $lump(CFG_f)$
2:     $P_0 \leftarrow create\_initial\_components(CFG_f)$
3:     $P \leftarrow markov\_lumping(P_0)$ return $P$
```

5 A binary diffing algorithm

We now present REveal, our proof-of-concept implementation. REveal consists of two main components and several helper libraries, all coded in Python. The first component is an IDA Python plug-in for IDA Pro. It computes the feature vectors of all functions found in an IDA Pro database and saves them in a standard Pickle format. The second component implements our binary diffing algorithms as described in this section. Given two Pickle files, like those mentioned previously, it attempts to produce a mapping between the functions of the corresponding programs, by locating feature vectors with equal features in the exported data. Last but not least, the helper libraries implement various graph theory abstractions and algorithms.

Given two multisets of function feature vectors, FV_{p_1} and FV_{p_2}, extracted from two programs under comparison, p_1 and p_2, various strategies, referred to as *matchers* from now on, might be used for mapping elements from the first set to the second, or vice versa. REveal implements the following three:

1. **Singleton matcher**: This matching strategy is shown in algorithm 5. Given FV_{p_1} and FV_{p_2}, *singletons* (feature vectors which exist only once in each set) are compared and matched by iteratively scanning over the Cartesian product of the aforementioned multisets.
2. **Structural matcher**: Shown in algorithm 6, the structural matcher iteratively matches predecessors and successors of already matched functions. This is done by creating multisets, FV'_{p_1} and FV'_{p_2}, consisting solely of feature vectors of callers and callees, respectively, of functions matched in previous steps. This process carries on until no more new matches can be found.
3. **Monotonic matcher**: When linking programs consisting of multiple object files, linkers don't move code belonging to a single object file around, as this would introduce several complications in the overall compilation process. The monotonic matcher exploits this fact by matching functions which are physically close to other functions that have already been matched. More formally, if f_1, f_2 are functions in program p_1 and g_1, g_2 functions in program p_2 and $f_1 \rightarrow g_1$, $f_2 \rightarrow g_2$, $address(f_1) < address(f_2)$ and $address(g_1) < address(g_2)$, the monotonic matcher attempts to find matching singletons between functions of p_1 whose addresses fall in the interval $(address(f_1),\ address(f_2))$ and functions of p_2 in $(address(g_1),\ address(g_2))$. In other words, assuming that we have two pairs of matched functions, whose matched elements are monotonically sorted in terms of their addresses, we try to find matching singletons in the sets consisting of feature vectors of functions that *lie between* f_1 and f_2 in p_1 with functions that *lie between* g_1 and g_2 in p_2. The monotonic matcher is shown in algorithm 7.

Other strategies that might be used in real life applications include the **import name matcher** and the **export name matcher**, which match imported and exported symbols respectively, based on their names and ordinals. These matchers can be very effective at creating an initial set of reliable matches which can be enriched, with the help of the abovementioned matching strategies, to produce an even larger set of results. It is essential to note that, in the experiments presented in the following sections, no symbol names were taken into account and consequently, the import and export name matchers were not enabled.

The core of REveal is based on algorithm 4, which is a simple loop that executes all matching strategies in sequence until no more matches can be found. The input to algorithm 4 are the multisets of function feature vectors FV_{p_1} and FV_{p_2} of programs p_1 and p_2 respectively. The singleton matching logic is invoked at line 6, the structural matching logic at line 8 and the monotonic matching logic at line 10. Variable M is the set that holds function matches, elements of the form $f \rightarrow g$, where f is a function in program p_1 and g a function in program p_2. The procedures, implementing the aforementioned matching strategies, populate M with elements of this form and return the total number of matches discovered.

Algorithm 4 REveal's main algorithm

```
1:  $M \leftarrow \emptyset$
2:  procedure $match(FV_{p_1},\ FV_{p_2})$
3:      $change \leftarrow True$
4:      while $change = True$ do
5:          $change \leftarrow False$
6:          if $singleton\_matcher(FV_{p_1},\ FV_{p_2},\ M) > 0$ then
7:              $change \leftarrow True$
8:          if $structural\_matcher(FV_{p_1},\ FV_{p_2},\ M) > 0$ then
9:              $change \leftarrow True$
10:         if $monotonic\_matcher(FV_{p_1},\ FV_{p_2},\ M) > 0$ then
11:             $change \leftarrow True$
```

The singleton matching logic, implemented in $singleton\_matcher()$, is shown in algorithm 5. Lines 4–5 iterate through the Cartesian product of the input feature vector multisets and try to match equal elements that appear only once in both. In such an event, the corresponding functions are retrieved (lines 6 and 7), the set of matches is updated (line 8), and the feature vectors are removed from their multisets (lines 9 and 10). Last but not least, if our oracle signifies a match (line 11), the number of correct matches is updated (line 12). On the opposite case, the number of mismatches is increased instead (line 12). The number of correct matches is eventually returned.

Algorithm 5 Singleton matcher

```
1: procedure $singleton\_matcher(FV_{p_1}, FV_{p_2}, M)$
2:     $num\_matches \leftarrow 0$
3:     $num\_mismatches \leftarrow 0$
4:     for all $(fv_{p_1}, fv_{p_2}) \in FV_{p_1} \times FV_{p_2}$ do
5:         if $fv_{p_1} = fv_{p_2}$ and $|\{fv_{p_1}\} \cap FV_{p_1}| = 1$ and $|\{fv_{p_2}\} \cap FV_{p_2}| = 1$ then
6:             $f_{p_1} \leftarrow FV_{p_1}^{-1}[fv_{p_1}]$
7:             $f_{p_2} \leftarrow FV_{p_2}^{-1}[fv_{p_2}]$
8:             $M \leftarrow M \cup \{f_{p_1} \to f_{p_2}\}$
9:             $FV_{p_1} \leftarrow FV_{p_1} - \{f_{p_1} \to fv_{p_1}\}$
10:            $FV_{p_2} \leftarrow FV_{p_2} - \{f_{p_2} \to fv_{p_2}\}$
11:            if $oracle(f_{p_1}, f_{p_2})$ = True then
12:                $num\_matches \leftarrow num\_matches + 1$
13:            else
14:                $num\_mismatches \leftarrow num\_mismatches + 1$
    return $num\_matches$
```

The structural matcher is shown in algorithm 6. By the time it is invoked, in the loop body of algorithm 4, $singleton\_matcher()$ has already been called once and thus M has already been populated with a set of initial matches. At lines 8–10, temporary sets, consisting solely of feature vectors of matched functions' successors, are created and $singleton\_matcher()$ is used once again to further expand M. The exact same procedure is repeated at lines 11–13, but, this time, predecessors are taken into account. The aforementioned steps are repeated for as long as M grows (guaranteed by the outer *while* loop). From a higher level perspective, algorithm 6 is the equivalent of traversing the compared programs' call-graphs and matching neighbors of already matched nodes.

Algorithm 6 Structural matcher

```
1: procedure $structural\_matcher(FV_{p_1}, FV_{p_2}, M)$
2:     $num\_matches \leftarrow 0$
3:     $change \leftarrow True$
4:     while $change = True$ do
5:         $change \leftarrow False$
6:         for all $f_{p_1} \to f_{p_2} \in M$ do
7:             $tmp\_num\_matches \leftarrow 0$
8:             $FV'_{p_1} \leftarrow \{FV_{p_1}[f] \mid f \in succ(f_{p_1})\}$
9:             $FV'_{p_2} \leftarrow \{FV_{p_2}[f] \mid f \in succ(f_{p_2})\}$
10:            $tmp\_num\_matches \leftarrow tmp\_num\_matches + singleton\_matcher(FV'_{p_1}, FV'_{p_2}, M)$
11:            $FV'_{p_1} \leftarrow \{FV_{p_1}[f] \mid f \in pred(f_{p_1})\}$
12:            $FV'_{p_2} \leftarrow \{FV_{p_2}[f] \mid f \in pred(f_{p_2})\}$
13:            $tmp\_num\_matches \leftarrow tmp\_num\_matches + singleton\_matcher(FV'_{p_1}, FV'_{p_2}, M)$
14:            if $tmp\_num\_matches > 0$ then
15:                $change \leftarrow True$
16:            $num\_matches \leftarrow num\_matches + tmp\_num\_matches$
    return $num\_matches$
```

The monotonic matching logic, briefly discussed above, is shown in algorithm 7. The algorithm starts by sorting M based on the addresses of the matched functions at line 6 (that is, for each $f_1 \to g_1, f_2 \to g_2 \in M$, $f_1 \to g_1 > f_2 \to g_2$ iff $address(f_1) > address(f_2)$ or $address(g_1) > address(g_2)$). The main body of the monotonic matching logic consists of a loop (lines 8–9 and 14), which iterates through M's elements in pairs (that is, the first iteration will examine $M[0]$ and $M[1]$, the second iteration $M[1]$ and $M[2]$ and so on). If a pair whose elements are monotonically sorted, in terms of their addresses, is found, multisets consisting of feature vectors of functions, whose addresses lie between the range of the matched functions, are constructed for both programs and $singleton\_matcher()$ is again used for discovering new matches (lines 11–13). The while loop at line 4 guarantees that the overall monotonic matching logic is iteratively executed until no more new matches can be found.

Algorithm 7 Monotonic matcher

```
1: procedure $monotonic\_matcher(FV_{p_1}, FV_{p_2}, M)$
2:     $num\_matches \leftarrow 0$
3:     $change \leftarrow True$
4:     while $change = True$ do
5:         $change \leftarrow False$
6:         $M \leftarrow sort(M)$
7:         $tmp\_num\_matches \leftarrow 0$
8:         $f_1 \to g_1 \leftarrow M[0]$
9:         for all $f_2 \to g_2 \in M[1\ldots]$ do
10:            if $address(g_1) < address(g_2)$ then
11:                $FV'_{p_1} \leftarrow \{FV_{p_1}[f] \mid address(f_1) < address(f) < address(f_2)\}$
12:                $FV'_{p_2} \leftarrow \{FV_{p_2}[g] \mid address(g_1) < address(g) < address(g_2)\}$
13:                $tmp\_num\_matches \leftarrow tmp\_num\_matches + match\_singletons(FV'_{p_1}, FV'_{p_2}, M)$
14:            $f_1, g_1 \leftarrow f_2, g_2$
15:        if $tmp\_num\_matches > 0$ then
16:            $change \leftarrow True$
    return $num\_matches$
```

6 Experimental results

Our experimental setup consists of an Intel Core-i7 2.4 Ghz system equipped with 8 GB of RAM. We evaluate our prototype binary diffing tool, code-named REveal, and compare it with Diaphora, which we believe to be the current industry standard in binary diffing. To further clarify, the version of Diaphora used in the experiments is this of commit 566bfce (Dec. 14, 2018), on IDA Pro 6.95, without the decompiler suite (purchase and use of the decompiler package is currently scheduled for a future publication).

For the purpose of evaluating our tool, we conducted two types of experiments. In the first type, we try to find similarities between two different versions of the same program P_i, say P_i^0 and P_i^1. For each such version pair, we use our binary diffing tool and Diaphora to find similarities between binary executables compiled for **ia32**, **amd64**, **arm** and **aarch64** ($P_{i,ia32}^0$ vs. $P_{i,ia32}^1$, $P_{i,amd64}^0$ vs. $P_{i,amd64}^1$ and so on). The second batch of experiments concerns malware variants. We collect 18 unpacked executable binaries belonging to 5 malware families and diff them with one another. We use the diffing results to classify malware samples into families using the following, oversimplified approach; each sample is added to the same family as the sample with which it shares the most common functions. Despite the aforementioned naive way of classifying malware, the second batch of experiments serves as a good example of detecting similarities in real world software.

6.1 Binary diffing open source software

Each experiment conducted in this batch, consists of the following steps. First, the IDA Pro databases of two versions of a program are opened. An IDAPython script is, then, executed that obfuscates the names of all functions. This preprocessing step is required for the experiments to be more fair as the SQLite queries internally used by Diaphora, during the matching process, do take symbol names into account. By obfuscating them we simulate a real life scenario where no symbols are available. Next, Diaphora is used once in order to export information from both databases and a second time in order to actually perform the diffing process. The same procedure is followed for our tool, REveal. Last but not least, the results produced by the two tools are extracted, parsed and formatted for easier evaluation. To further stress the effectiveness of REveal, we compare its results against those produced by Diaphora when used on a non-obfuscated database, thus clearly giving the latter an early advantage.

Both tools are evaluated based on the number of exact matches found. That is, functions which are not exactly similar are not considered. In the case of Diaphora, given a SQLite database holding binary diffing results, the exact number of perfect matches can be computed using the following SQL query. Notice how we avoid comparing the names of matched anonymous functions and always count them as true positives.

```
SELECT COUNT(*) FROM results WHERE type =
"best" AND (name = name2 OR name LIKE "sub_%%"
OR name2 LIKE "sub_%%");
```

The full list of experiments is shown in Table 1. Numbers, corresponding to experiment identifiers, are assigned to pairs of lines. Each pair shows information on the two programs that were compared; their versions, their architectures and the compilers used for building them (according to metadata found in each executable).

It should be noted that the Linux kernel binaries were compiled by us, while the remaining binary executables were directly downloaded from the respective public domain GitHub repositories. For the sake of having ground truth, we had to take into account only open source projects that publish precompiled executables with populated symbol tables and/or debug symbols for all the aforementioned architectures.

Table 2 summarizes the results of this series of experiments. Entries in the table consist of triplets of rows with the first line in each triplet showing the results produced by

Table 1 List of experiments conducted

	Program	Version	Architecture	Compiler
1	ncmc	v0.1.7	aarch64	GCC 4.2.0
		v0.1.8	aarch64	GCC 4.2.0
2	ncmc	v0.1.7	amd64	GCC 4.2.0
		v0.1.8	amd64	GCC 4.2.0
3	ncmc	v0.1.7	arm	GCC 4.3.0
		v0.1.8	arm	GCC 4.3.0
4	ncmc	v0.1.7	ia32	GCC 4.2.0
		v0.1.8	ia32	GCC 4.2.0
5	ffmpeg	20180408	aarch64	GCC 4.9.x
		20180731	aarch64	GCC 4.9.x
6	ffmpeg	20180408	amd64	GCC 4.9.x
		20180731	amd64	GCC 4.9.x
7	ffmpeg	20180408	arm	GCC 4.9.x
		20180731	arm	GCC 4.9.x
8	ffmpeg	20180408	ia32	GCC 4.9.x
		20180731	ia32	GCC 4.9.x
9	nmap	7.12	aarch64	GCC 4.9.x
		7.31	aarch64	GCC 4.9
10	nmap	7.12	amd64	GCC 4.9.x
		7.31	amd64	GCC 4.9
11	nmap	7.12	arm	GCC 4.9.x
		7.31	arm	GCC 4.9
12	nmap	7.12	ia32	GCC 4.9.x
		7.31	ia32	GCC 4.9
13	vmlinux	4.4.1	aarch64	GCC 4.9.x
		4.4.40	aarch64	GCC 4.9.x
14	vmlinux	4.4.1	amd64	GCC 8.2.0
		4.4.40	amd64	GCC 8.2.0
15	vmlinux	4.4.1	arm	GCC 4.9.x
		4.4.40	arm	GCC 4.9.x
16	vmlinux	4.4.1	ia32	GCC 8.2.0
		4.4.40	ia32	GCC 8.2.0

Table 2 Summary of experimental results

	Tool	Time	Matches	Mismatches
1	Diaphora	12 s	427 (52.073%)	46 (5.610%)
	Diaphora (unobfuscated)	10 s	633 (77.195%)	77 (9.390%)
	REveal	2 s	577 (70.366%)	81 (9.878%)
2	Diaphora	10 s	388 (51.255%)	54 (7.133%)
	Diaphora (unobfuscated)	9 s	600 (79.260%)	75 (9.908%)
	REveal	2 s	542 (69.487%)	89 (11.410%)
3	Diaphora	9 s	514 (60.329%)	57 (6.690%)
	Diaphora (unobfuscated)	10 s	618 (72.535%)	73 (8.568%)
	REveal	2 s	675 (72.503%)	91 (9.774%)
4	Diaphora	7 s	264 (34.921%)	23 (3.042%)
	Diaphora (unobfuscated)	9 s	277 (36.640%)	25 (3.307%)
	REveal	1 s	496 (59.401%)	77 (9.222%)
5	Diaphora	373 s	6658 (52.620%)	0 (0.000%)
	Diaphora (unobfuscated)	410 s	10951 (86.549%)	0 (0.000%)
	REveal	340 s	11,599 (91.381%)	61 (0.481%)
6	Diaphora	356 s	8877 (55.447%)	0 (0.000%)
	Diaphora (unobfuscated)	369 s	13,732 (85.707%)	0 (0.000%)
	REveal	475 s	13,074 (80.361%)	65 (0.400%)
7	Diaphora	384 s	9970 (59.608%)	0 (0.000%)
	Diaphora (unobfuscated)	474 s	12,499 (74.728%)	0 (0.000%)
	REveal	586 s	13,888 (81.752%)	73 (0.430%)
8	Diaphora	380 s	6417 (42.396%)	0 (0.000%)
	Diaphora (unobfuscated)	473 s	6519 (43.058%)	0 (0.000%)
	REveal	522 s	11,402 (74.266%)	52 (0.339%)
9	Diaphora	101 s	2212 (20.577%)	50 (0.465%)
	Diaphora (unobfuscated)	93 s	5907 (54.949%)	51 (0.474%)
	REveal	186 s	6205 (57.598%)	46 (0.427%)
10	Diaphora	89 s	2390 (22.838%)	55 (0.526%)
	Diaphora (unobfuscated)	98 s	6228 (59.513%)	57 (0.545%)
	REveal	184 s	5993 (55.475%)	36 (0.333%)
11	Diaphora	90 s	2275 (20.902%)	55 (0.505%)
	Diaphora (unobfuscated)	91 s	3866 (35.54%)	51 (0.469%)
	REveal	193 s	5897 (51.163%)	34 (0.295%)
12	Diaphora	89 s	1780 (16.804%)	83 (0.784%)
	Diaphora (unobfuscated)	91 s	1824 (17.219%)	74 (0.699%)
	REveal	174 s	5974 (55.095%)	29 (0.267%)
13	Diaphora	216 s	7310 (27.708%)	56 (0.212%)
	Diaphora (unobfuscated)	215 s	20,557 (77.921%)	34 (0.129%)
	REveal	960 s	22,832 (86.541%)	5 (0.019%)
14	Diaphora	328 s	9856 (23.735%)	119 (0.287%)
	Diaphora (unobfuscated)	312 s	31,663 (76.249%)	234 (0.564%)
	REveal	1991 s	33,094 (79.693%)	262 (0.631%)
15	Diaphora	142 s	5212 (23.907%)	5 (0.023%)
	Diaphora (unobfuscated)	148	11,758 (53.933%)	4 (0.018%)
	REveal	433 s	19,513 (89.456%)	4 (0.018%)
16	Diaphora	610 s	9479 (23.686%)	73 (0.182%)
	Diaphora (unobfuscated)	611 s	10,101 (25.241%)	64 (0.160%)
	REveal	1536 s	33,722 (84.261%)	167 (0.417%)

Diaphora when used on obfuscated IDA Pro databases. On the contrary, the second row shows the outcome of Diaphora on normal IDA Pro databases, without the aforementioned obfuscation step. Last but not least, the third row in each triplet displays the results produced by REveal. Since REveal does not take into account symbol names, an obfuscated IDA Pro database makes no difference on the outcome. Notice that the number of each triplet corresponds to the experiment ordinal as shown in Table 1.

One of the most notable conclusions that can be drawn from Table 2 is that REveal outperforms Diaphora, in terms of correctly, according to ground truth, matched functions. In fact, on obfuscated experiments REveal always outperforms Diaphora. Even in unobfuscated experiments, Diaphora outperforms REveal in only 4 out of 16 experiments (highlighted in grey), despite receiving a significant boost from using unobfuscated symbol names. The **ffmpeg** and **vmlinux** experiments, for all four architectures, are notable examples of this behavior. In the **ffmpeg** case, REveal reports 11599/13074/13888/11402 matches, corresponding to a coverage of about 74–91% of the compared binaries, while in the **vmlinux** case it finds 960/1991/433/1536 correct matches, successfully matching almost 80% of the given Linux kernels.

The increased match rate, however, comes at a cost, which brings us to our second conclusion. When comparing large, real life programs, REveal is much slower than Diaphora. Take experiments 14 and 16 as an example. REveal takes 1991/1536 s to complete respectively, while Diaphora only needs 312/611 when symbol information is available and 328/610 when not. Evidently, REveal can be quite slower than Diaphora.

6.2 Malware classification

As it has already been discussed, in this batch of experiments we try to classify 18 malware samples in 5 families. The actual families, the aforementioned samples originally belong to, are known in advance and used as the ground truth for the evaluation of our oversimplified classification process. Once again, we compare the results produced by REveal with those reported by Diaphora. This time, no obfuscation step is required, as the unpacked malware samples come with no symbol names.

The malware families in question, along with the corresponding number of samples are listed in Table 3. All unpacked executables are compared with one another and are classified into families based on the outcome of the diffing process as follows; each sample is added to the same family with the sample it shares the most common functions with. As it has already been mentioned, this batch of experiments focuses on measuring the effectiveness of our binary diffing technique rather than the classification process itself.

Table 3 List of malware samples used grouped by family

Malware family	Number of samples
WannaCry	4
Loki	3
Formbook	4
AZOrult	2
Duqu	5

That said, the classification procedure, followed in this section, requires the execution of 18! experiments, and thus the full table of results would be very long. For this purpose, Table 4 shows only those experiment instances which eventually draw the classification verdict. That is, we only show the results of these experiments where the compared malware executables had the highest number of matches and consequently classified into the same malware family. The relevant results are shown in Table 5. The first column in the latter corresponds to the experiment ordinal of Table 4.

First thing to note is that Table 5 has no column showing the number of mismatches. This is reasonable, as malware samples come with no symbol names and thus ground truth is not available. We emphasize that, since we are only interested in the clustering outcome, the exact number of true and false positives are of no interest to us, as the classification verdict is drawn by comparing the relative similarity scores between the compared samples.

As it can be seen, both REveal and Diaphora perform very well with success rates ranging from 79 to 100%, which are quite impressive. Evidently, malware samples in our dataset, belonging to the same family, share a lot of common code and both tools detect this adequately. The lines highlighted in grey, pinpoint those experiment instances where Diaphora won by matching more functions than REveal. In 2 of the aforementioned 8 cases (9 and 10), Diaphora detects a single extra function which seems to be common in both samples (415 matches reported by REveal vs. 416 by Diaphora). In the remaining 6 cases (17, 19, 20, 22, 23 and 26) Diaphora finds an additional 6 for a final score of 111 versus 105 achieved by REveal.

Careful readers may have noticed a certain discrepancy in experiment number 1 in Table 5, which might be less obvious in other experiment instances of both this and the previous section. One can see that Diaphora matches 234 functions and the percentage reported is 100%, while REveal matches 475 with a reported percentage of 93.874%. Clearly this shouldn't be possible. This anomaly, however, can be explained by taking a closer look at the list of functions Diaphora considers for diffing purposes. As it can be seen by studying the source code, *thunk* and *library* functions are, by default, ignored and

Table 4 List of experiments conducted

	Malware family	Sample SHA256
1	AZORult	301b69021ba9cfa4dc77b6727f54f15ee5c2cbe8b900c33284b673ef95a0f875
	AZORult	83f80a5fd364f30b185a3127d73990681977887e7c74d419ce47e50d35cf9b63
2	Formbook	233fd6693019a87459d2f244b288548309b823b1a8880231160ecfc40c9fdd67
	Formbook	3ea7fefb5b0e4eab0df80a20e5ae83c5baaaf84dff229d4c3f0ce1032b92e774
3	Formbook	233fd6693019a87459d2f244b288548309b823b1a8880231160ecfc40c9fdd67
	Formbook	ba510ea2775f9148b188eab984e7e35ca5208a5b375497017e7720b9e9fc3640
4	Formbook	233fd6693019a87459d2f244b288548309b823b1a8880231160ecfc40c9fdd67
	Formbook	f7dbb04865c110568e2af9dd6b5397b8080d19cb7c6b34007c722913b6af7de1
5	Formbook	3ea7fefb5b0e4eab0df80a20e5ae83c5baaaf84dff229d4c3f0ce1032b92e774
	Formbook	ba510ea2775f9148b188eab984e7e35ca5208a5b375497017e7720b9e9fc3640
6	Formbook	3ea7fefb5b0e4eab0df80a20e5ae83c5baaaf84dff229d4c3f0ce1032b92e774
	Formbook	f7dbb04865c110568e2af9dd6b5397b8080d19cb7c6b34007c722913b6af7de1
7	Formbook	ba510ea2775f9148b188eab984e7e35ca5208a5b375497017e7720b9e9fc3640
	Formbook	f7dbb04865c110568e2af9dd6b5397b8080d19cb7c6b34007c722913b6af7de1
8	Loki	243b9d53bca8e3eb0c0e67a196a5f9021afea1b82e75e89a45f90d560f8a2270
	Loki	bf88c1dbe7222c5360e08bc0261049a731ca967b2fbfe7a7ac285dcd8bd0fc60
9	Loki	243b9d53bca8e3eb0c0e67a196a5f9021afea1b82e75e89a45f90d560f8a2270
	Loki	ca6a1824220944d262451d0854e3a2d15aa111c3ac5244662f32bf3b8e377386
10	Loki	bf88c1dbe7222c5360e08bc0261049a731ca967b2fbfe7a7ac285dcd8bd0fc60
	Loki	ca6a1824220944d262451d0854e3a2d15aa111c3ac5244662f32bf3b8e377386
11	WannaCry	066e9388d68db89aaca79c7ad0ced6fab9921c2c8e849ae863725df651129bf0
	WannaCry	6973990a7e9eab21d19b40ef64c167343b2679a73ca2be639607d8a0baf51b33
12	WannaCry	066e9388d68db89aaca79c7ad0ced6fab9921c2c8e849ae863725df651129bf0
	WannaCry	a942718565fd880b3c12299786c82ad65986fdc6dfc7ed9ee131a8665abb16f7
13	WannaCry	066e9388d68db89aaca79c7ad0ced6fab9921c2c8e849ae863725df651129bf0
	WannaCry	fe4ab64b2aaa8a22adbcc6948e81a9eb600dea9be6f1bcafd8289b6f562db78d
14	WannaCry	6973990a7e9eab21d19b40ef64c167343b2679a73ca2be639607d8a0baf51b33
	WannaCry	a942718565fd880b3c12299786c82ad65986fdc6dfc7ed9ee131a8665abb16f7
15	WannaCry	6973990a7e9eab21d19b40ef64c167343b2679a73ca2be639607d8a0baf51b33
	WannaCry	fe4ab64b2aaa8a22adbcc6948e81a9eb600dea9be6f1bcafd8289b6f562db78d
16	WannaCry	a942718565fd880b3c12299786c82ad65986fdc6dfc7ed9ee131a8665abb16f7
	WannaCry	fe4ab64b2aaa8a22adbcc6948e81a9eb600dea9be6f1bcafd8289b6f562db78d
17	Duqu	2c9c3ddd4d93e687eb095444cef7668b21636b364bff55de953bdd1df40071da
	Duqu	2ecb26021d21fcef3d8bba63de0c888499110a2b78e4caa6fa07a2b27d87f71b
18	Duqu	2c9c3ddd4d93e687eb095444cef7668b21636b364bff55de953bdd1df40071da
	Duqu	6c803aac51038ce308ee085f2cd82a055aaa9ba24d08a19efb2c0fcfde936c34
19	Duqu	2c9c3ddd4d93e687eb095444cef7668b21636b364bff55de953bdd1df40071da
	Duqu	6e09e1a4f56ea736ff21ad5e188845615b57e1a5168f4bdaebe7ddc634912de9
20	Duqu	2c9c3ddd4d93e687eb095444cef7668b21636b364bff55de953bdd1df40071da
	Duqu	d12cd9490fd75e192ea053a05e869ed2f3f9748bf1563e6e496e7153fb4e6c98
21	Duqu	2ecb26021d21fcef3d8bba63de0c888499110a2b78e4caa6fa07a2b27d87f71b
	Duqu	6c803aac51038ce308ee085f2cd82a055aaa9ba24d08a19efb2c0fcfde936c34
22	Duqu	2ecb26021d21fcef3d8bba63de0c888499110a2b78e4caa6fa07a2b27d87f71b
	Duqu	6e09e1a4f56ea736ff21ad5e188845615b57e1a5168f4bdaebe7ddc634912de9
23	Duqu	2ecb26021d21fcef3d8bba63de0c888499110a2b78e4caa6fa07a2b27d87f71b
	Duqu	d12cd9490fd75e192ea053a05e869ed2f3f9748bf1563e6e496e7153fb4e6c98

Table 4 continued

	Malware family	Sample SHA256
24	Duqu	6c803aac51038ce308ee085f2cd82a055aaa9ba24d08a19efb2c0fcfde936c34
	Duqu	6e09e1a4f56ea736ff21ad5e188845615b57e1a5168f4bdaebe7ddc634912de9
25	Duqu	6c803aac51038ce308ee085f2cd82a055aaa9ba24d08a19efb2c0fcfde936c34
	Duqu	d12cd9490fd75e192ea053a05e869ed2f3f9748bf1563e6e496e7153fb4e6c98
26	Duqu	6e09e1a4f56ea736ff21ad5e188845615b57e1a5168f4bdaebe7ddc634912de9
	Duqu	d12cd9490fd75e192ea053a05e869ed2f3f9748bf1563e6e496e7153fb4e6c98

Table 5 Summary of experimental results

	Tool	Matches
1	Diaphora	234 (100.000%)
	REveal	475 (93.874%)
2	Diaphora	333 (80.825%)
	REveal	378 (91.525%)
3	Diaphora	376 (91.262%)
	REveal	379 (91.768%)
4	Diaphora	342 (82.609%)
	REveal	379 (91.325%)
5	Diaphora	332 (80.583%)
	REveal	373 (90.315%)
6	Diaphora	329 (79.854%)
	REveal	369 (89.346%)
7	Diaphora	343 (83.252%)
	REveal	375 (90.799%)
8	Diaphora	416 (100.000%)
	REveal	416 (99.284%)
9	Diaphora	416 (100.000%)
	REveal	415 (99.045%)
10	Diaphora	416 (100.000%)
	REveal	415 (99.045%)
11	Diaphora	91 (98.913%)
	REveal	107 (85.600%)
12	Diaphora	91 (98.913%)
	REveal	107 (85.600%)
13	Diaphora	91 (98.913%)
	REveal	107 (85.600%)
14	Diaphora	91 (98.913%)
	REveal	107 (85.600%)
15	Diaphora	91 (98.913%)
	REveal	107 (85.600%)
16	Diaphora	92 (100.000%)
	REveal	108 (86.400%)
17	Diaphora	111 (100.000%)
	REveal	105 (84.677%)
18	Diaphora	67 (60.909%)
	REveal	84 (68.293%)
19	Diaphora	111 (100.000%)
	REveal	105 (84.677%)
20	Diaphora	111 (100.000%)
	REveal	105 (84.677%)
21	Diaphora	67 (60.909%)
	REveal	84 (68.293%)
22	Diaphora	111 (100.000%)
	REveal	105 (84.677%)
23	Diaphora	111 (100.000%)
	REveal	105 (84.677%)
24	Diaphora	67 (60.909%)
	REveal	84 (68.293%)
25	Diaphora	67 (60.909%)
	REveal	84 (68.293%)
26	Diaphora	111 (100.000%)
	REveal	105 (84.677%)

this results in the overall match percentage computed using a denominator smaller than the one used by REveal.

Last but not least, it is evident that both tools perform really well in terms of execution time, requiring from 1 to 2 s to finish the overall matching process. This suggests that both REveal and Diaphora might be used in real life applications involving malware clustering.

7 Conclusion

We have presented a combination of (i) algorithms and (ii) features which, as our experiments indicate, perform efficient function matching of two binary programs. Our proof-of-concept implementation, planned to be released as an open-source project, compares favorably with Diaphora, the leading industry standard in binary diffing. We conclude by listing several issues which require further research.

1. Coming up with equally efficient *distance* metrics for the proposed features is a challenging task. Using distance metrics, our algorithms will be capable of considering inexact, non-perfect matches and will allow for a fair comparison of our approach with other binary diffing software. REveal currently implements limited inexact matching capabilities with more elaborate research scheduled for the future.
2. We believe the Markovian properties of computer programs merit further research. For example, the weight assignment schemes presented in Sect. 4 can be considerably improved and extended, so as to mirror more distinctive characteristics of the subject CFGs. Development of more elaborate and efficient Markov-based features is definitely possible.
3. We have successfully performed binary diffing experiments on four widespread computer architectures. Lifting machine code to an IR and encoding its semantics is of paramount importance in real life applications of cross-architecture binary diffing. An added benefit is that, a carefully chosen IR will also reduce noise caused by variance in code generated by different versions of a compiler, different levels of code optimizations, instruction set idiosyncrasies and so on.
4. A new version of Zynamics BinDiff was released at the time the current paper was completed. Comparing REveal against it, as well as against other tools like YaDiff or DarunGrim, is essential at evaluating the effectiveness of our approach. Additionally, Diaphora is said to produce better results with the help of the IDA decompiler plug-in; this is also something that should be considered in future research.
5. Last but not least, a complete binary diffing approach should not only depend on the CG and CFG, the control-flow dependencies, but should also take into account data dependence relations present in the examined programs. Towards this, PDG (*Program Dependence Graph*) based techniques have already been proposed and studied [14]. In such an approach, feature vectors can be extracted from the compared programs' data elements to aid in the overall diffing process.

References

1. Aho, A., Lam, M., Sethi, R., Ullmanr, J.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison-Wesley Longman Publishing Co., Boston (2006)
2. Bourquin, M., King, A., Robbins, E.: BinSlayer: accurate comparison of binary executables. In: 2nd ACM SIGPLAN Program Protection and Reverse Engineering (2013)
3. Cesare, S., Xiang, Y.: Classification of malware using structured control flow. In: Proceedings of the 8th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2010) (2010)
4. Cesare, S., Xiang, Y., Zhou, W.: Control flow-based malware variant detection. IEEE Trans. Dependable Secur Comput **11**, 307–317 (2013)
5. Deo, N.: Graph Theory with Applications to Engineering and Computer Science. Prentice-Hall Inc, Upper Saddle River (1974)
6. Derisavi, S., Hermanns, H., Sanders, W.: Optimal state-space lumping in Markov chains. Inf. Process. Lett. **87**, 309–315 (2003)
7. Koret, J.: Diaphora: A Free and Open Source Program Diffing Tool [Online]. http://diaphora.re/. Accessed 15 Apr 2019
8. Dullien, T., Rolles, R.: Graph-based comparison of executable objects. In: Proceedings of the Symposium sur la Securite des Technologies de l'Information et des Communications (2005)
9. Dullien, T., Carrera, E., Eppler, S. M., Porst, S.: Automated attacker correlation for malicious code. In: NATO Information Systems Technology (IST) 091 (2010)
10. Eschweiler, S., Yakdan, K., Gerhards-Padilla, E.: discovRE: efficient cross-architecture identification of bugs in binary code. In: SP '15 Proceedings of the 2015 IEEE Symposium on Security and Privacy (2016)
11. Flake, H.: Structural comparison of executable objects. In: Proceedings of the IEEE Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA) (2004)
12. Gao, D., Reiter, M., Song, D.: BinHunt: automatically finding semantic differences in binary programs. In: Information and Communications Security, pp. 238–255 (2008)
13. Hex-Rays: IDA Pro [Online]. https://www.hex-rays.com/products/ida/. Accessed 15 Apr 2019
14. Henderson, T.A.D., Podgurski, A.: Sampling code clones from program dependence graphs with GRAPLE. In: SWAN 2016 Proceedings of the 2nd International Workshop on Software Analytics (2016)
15. Howard, R.: Dynamic Probabilistic Systems: volume I: Markov Models. Wiley, Hoboken (1971)
16. Howard, R.: Dynamic Probabilistic Systems. Volume II: Semi-Markov and Decision Processes. Wiley, Hoboken (1971)
17. Hu, X., Chiueh, T., Shin, K.G.: Large-scale malware indexing using function-call graphs. In: Computer and Communications Security, pp. 611–620 (2009)
18. Intel: Intel X86 Encoder Decoder Software Library [Online]. https://software.intel.com/en-us/articles/xed-x86-encoder-decoder-software-library. Accessed 15 Apr 2019
19. Intel: Intel X86 Encoder Decoder [Online]. https://intelxed.github.io/ref-manual/xed-iform-enum_8h.html. Accessed 15 Apr 2019
20. Jurczyk, M.: Using Binary Diffing to Discover Windows Kernel Memory Disclosure Bugs [Online]. https://googleprojectzero.blogspot.gr/2017/10/using-binary-diffing-to-discover.html. Accessed 15 Apr 2019
21. Karamitas, C.: Python Bindings for Intel's XED [Online]. https://github.com/huku-/pyxed. Accessed 15 Apr 2019
22. Karamitas, C., Kehagias, A.: Efficient Features for function matching between binary executables. In: 2018 IEEE 25th Int Conf Softw Anal Evol Reengineering (SANER), vol. 1, pp. 335–345 (2018)
23. 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 (2011)
24. Levenshtein, V.: Binary codes capable of correcting deletions, insertions and reversals. In: Soviet Physics Doklady, pp. 707–710 (1966)
25. Ming, J., Pan, M., Gao, D.: iBinHunt: binary hunting with interprocedural control flow. In: Lecture Notes in Computer Science, pp. 92–109 (2013)
26. 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) (2017)

27. McAfee: McAfee Labs Threats Report April (2017) [Online]. https://www.mcafee.com/us/resources/reports/rp-quarterly-threats-mar-2017.pdf. Accessed 15 Apr 2019
28. Panda Security: Pandalabs Quarterly Report Q1 (2017) [Online]. http://www.pandasecurity.com/mediacenter/src/uploads/2017/05/Pandalabs-2017-T1-EN.pdf. Accessed 15 Apr 2019
29. Ramalingam, G.: On loops, dominators, and dominance frontiers. In: PLDI'00 Proceedings of the ACM SIGPLAN 2000 conference on Programming Language Design and Implementation, pp. 233–241 (2000)
30. SafeCorp: Detecting Software IP Theft Using CodeMatch [Online]. https://www.safe-corp.com/documents/CodeMatch_Whitepaper.pdf. Accessed 15 Apr 2019
31. Tarjan, R.: Testing flow graph reducibility. In: STOC'73 Proceedings of the Fifth Annual ACM Symposium on Theory of Computing, pp. 96–107 (1973)
32. Valmari, A., Franceschinis, G.: Simple O(mlogn) time Markov chain lumping. In: TACAS'10 Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 38–52 (2010)
33. Wang, Z., Pierce, K., McFarling, S.: BMAT: a binary matching tool. In: Second ACM Workshop on Feedback-Directed and Dynamic Optimization (1999)
34. Wang, Z., Pierce, K., McFarling, S.: BMAT: a binary matching tool for stale profile propagation. J Instr Level Parallel **2**, 1–20 (2000)
35. Zynamics: BinDiff [Online]. https://www.zynamics.com/bindiff.html. Accessed 15 Apr 2019