Efficient Features for Function Matching between Binary Executables

Chariton Karamitas
CENSUS S.A.,
Greece
huku@census-labs.com

Athanasios Kehagias
Department of Electrical and Computer Engineering,
Aristotle University of Thessaloniki,
Greece
kehagiat@auth.gr

***Abstract*—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. For large programs, binary diffing can be performed by *function matching* which, in turn, is reduced to a *graph isomorphism* problem between the compared programs' CFGs (*Control Flow Graphs*) and/or CGs (*Call Graphs*). In this paper we provide a set of carefully chosen features, extracted from a binary's CG and CFG, which can be used by BinDiff algorithm variants to, first, build a set of initial exact matches with minimal false positives (by scanning for unique perfect matches) and, second, propagate approximate matching information using, for example, a nearest-neighbor scheme. Furthermore, we investigate the benefits of applying *Markov lumping* techniques to function CFGs (to our knowledge, this technique has not been previously studied). The proposed function features are evaluated in a series of experiments on various versions of the Linux kernel (Intel64), the OpenSSH server (Intel64) and Firefox's *xul.dll* (IA-32). Our prototype system is also compared to Diaphora, the current state-of-the-art binary diffing software.**

I. Introduction

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 [15], [26], [27]. As the size of the programs increases, so does the complexity of binary diffing, necessitating the use of automated difference analysis methods. In these cases, binary diffing can be performed by *function matching* of the two programs, which assists in spotting critical differences and minimizes any manual labor required to understand and process code and data modifications between the examined executables.

Specifically, function matching is reduced to a *graph isomorphism* problem [11] between the compared subjects' CFGs (*Control Flow Graphs*), ICFGs (*Inter-procedural CFGs*) and/or CGs (*Call Graphs*). Since graph isomorphism is known to belong to the *NP* class, practical methods must be considered for creating a maximal set of matches and approximately solving the graph isomorphism problem. In this paper, we propose a set of carefully chosen function features extracted from a binary's CG and CFG, which can be used by BinDiff algorithm variants [11], [12], [14] to (i) build a set of initial exact matches with minimal false positives by scanning for unique perfect matches, and (ii) propagate approximate matching information using, for example, a nearest-neighbor scheme. The proposed features are evaluated in a series of experiments on various versions of the Linux kernel (Intel64), the OpenSSH server (Intel64) and Firefox's *xul.dll* (IA-32). Our prototype system is also compared to Diaphora [10]. The contributions of the current paper are the following: (i) we provide a set of carefully chosen features; (ii) we investigate the benefits of applying *Markov lumping* techniques to function CFGs (to our knowledge, this technique has not been previously studied).

The rest of the paper is organized as follows. Section II gives a brief overview of previous work on the subject of binary diffing. In Section III preliminary material is presented: the mathematical notation and definitions we will use and the assumptions we made while developing our prototype system. A detailed description of each of the proposed features is given in Section IV. Experimental results are presented in Section V, followed by a brief summary and proposals for future work in Section VI.

II. Previous Work

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

1) **Malware classification** [3], [4], [20]. Given a possibly malicious binary executable, the problem is to classify the subject as either innocent or suspicious. At a higher level, malware classification works by extracting various features from the analyzed binary executable and locating matching candidates within a large database of known, pre-analyzed malware samples. As such databases may be hundreds of terabytes in size, performance and effectiveness are both of great importance in this application. With the number of sophisticated cyber-attacks, ransomware, malware and other forms of malicious software constantly increasing [29], [30], both antivirus companies and independent researchers have entered this research field.
2) **Patch analysis** [23]. Software complexity increases as more features are implemented, especially in various commercial software suites like Microsoft Office. At the same time, security researchers need to quickly study, analyze and evaluate product updates for the presence of software vulnerabilities. In this case, binary diffing comes in handy; by diffing shared libraries and binary

executables of the investigated system, pre- and post-update, a security researcher practically minimizes the set of changes that need to be studied.

3) **Plagiarism detection**. Not all applications of binary diffing are related to computer security. Diffing methods, applied both on source and executable code, have been used in the past to detect plagiarism. Consider an online system in a university where students submit solutions to their assignments; a diffing technique may be used to assign similarity scores to submitted solutions and eventually detect cheating students. In the software industry, binary diffing has been employed in order to detect copyright infringement and other forms of intellectual property theft. [32]
4) **Propagation of profiling information**. As explained in [35], [36], porting profiling information from an older version of a program to a new one is an error prone and time consuming process. 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.

Many binary diffing algorithms in the literature approach the underlying graph isomorphism problem by 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 well known 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. The higher the number of detected fixed points and matching ratio, the better the binary diffing results.
2) In the *propagation phase*, the initial 1-1 mapping is expanded. This is 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. Exact differences between the compared executables are then spotted by examining each basic block of each matched CFG. Such algorithms are usually referred to as *BinDiff variants*.

For example in [35], [36] a series of heuristics of various fuzziness levels is used to form a 1-1 mapping of functions. As a second step, the aforementioned mapping is used to narrow down the problem space by matching basic blocks of matched procedures. Another notable example is the work of Dullien et al. in [11], [12], [14]. Feature vectors (composed of the number of vertices and edges as well as the out-degree of each CFG) are formed and used to locate perfect matches in the compared executables. Later publications [2], [13] 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 [37]. Last but not least, consider Diaphora [10] an open source application and nowadays the industry standard in binary diffing. It uses a set of several graph metrics, computed over function CFGs, to match unique functions in two executables; it should also be mentioned that Diaphora does not implement any kind of propagation phase.

There are other binary diffing algorithms as well; some of them based on dynamic analysis techniques [26], [27], others use back-tracking [35], [36], MCS-based algorithms [13] and others follow a more simplistic *simulated annealing* approach [25]. Even though our work is mostly applicable to BinDiff algorithm variants, it may be possible to apply it to the abovementioned algorithms.

III. Preliminaries

A. Graph Theory Preliminaries

We define a program p to be a set of N functions; $p = \{f_i \mid i = 0, 1, ..., N-1\}$. 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 correlates functions of p_1 with 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 $pred(v) = \{p \mid (p, v) \in E\}$. In the following, we give definitions for various digraph types encountered in programs' analysis.

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*. [1]

A program p can also be treated as a *digraph of digraphs*, referred to as the program's *Call Graph*, or CG for short. CG_p, of program p, is a digraph whose vertices correspond to individual function CFGs, that is $V_{CG} = \{CFG_f \mid$

[1] Even though the CFG (and the CG mentioned in the sequel) are *digraphs* we will follow standard usage and call them *graphs*. In general, any "graph" mentioned in this paper will be a directed graph, unless explicitly mentioned otherwise.

$f \in p\}$. 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 $f_{src} \neq f_{dst}$ [2], an edge $(CFG_{f_{src}}, CFG_{f_{dst}})$ exists in E_{CG}.

When traversing a digraph, be it a CFG, a CG or any other digraph type, it is often useful to enforce an ordering in the way digraph edges or vertices are visited. During a DFS traversal, for example, it may be imperative to sort $succ(v) \forall v \in V$ under a certain ordering, so that, neighbor vertices are visited in a certain fashion. In the following, we make the assumption that, whenever $succ(v)$ or $pred(v)$ is used, the corresponding vertex sets are returned ordered by their addresses in the program's virtual address space. That is, if $succ(v) = \{s_0, s_1, ..., s_{n-1}\}$ for some $v \in V$, then $address(s_0) < address(s_1) < ... < address(s_{n-1})$, where $address : V \rightarrow \mathbb{N}$, a function defined over V, that returns the address of a vertex in program memory.

B. Markov Preliminaries

A Markov process is a stochastic process that satisfies the Markov property; the probability of the process entering the next state, depends only on the present state. Markov processes are usually represented using a two-dimensional *transition matrix* (or *stochastic matrix*) with elements

$$\forall i, j : p_{ij} = \Pr(\text{Process enters state } j \,|\, \text{Process is in state } i),$$

called the process' *transition probabilities*. Extensive treatments of Markov processes appear in [18] and [19]. In the rest of the paper, the terms *Markov system* and *Markov process* are used interchangeably.

Markov systems with a high number of states result in extremely large transition matrices, which introduce computational complexities. For this purpose, a technique referred to as *lumping* has been proposed. Lumping is used to merge equivalent states from the original Markov system into superstates, called *components*, of a new system, also obeying the Markov property. The new system (which generally is an abstraction of the old one and thus introduces information loss) is then used for applying standard Markov analysis algorithms.

For any two components P_i and P_j of a lumped Markov system, the following property holds true:

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

Several algorithms for testing and computing the lumpability of a Markov system have been proposed, with [9] and [34] being two notable examples.

C. Oracle Preliminaries

In order to test the effectiveness of the proposed features and evaluate our algorithm, we need to establish some ground truth on the function matching results. Towards this, we make use of an oracle $oracle : p_1 \times p_2 \rightarrow \{True, False\}$, defined over the two compared subjects, p_1 and p_2. Given two functions f_{p_1} and f_{p_2}:

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

Practically, the oracle defined above, can be used to tell whether a match, reported by our algorithm, is valid or not. Implementing such an oracle, for experimental use, is of paramount importance to our overall evaluation. The most simple approach involves making use of debug information embedded in the binary executables under comparison. 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 [28] distance is small), the match is considered successful.

D. Assumptions

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

1) Compared binaries come with no debug information whatsoever. In our experiments, debugging information was only used as an oracle for evaluating the soundness of matching results.
2) Source code of compared subjects is not available.
3) We only consider exact, perfect matches, discovered using a simple matching algorithm that resembles [5], [6], [7].
4) Even though our algorithm has good execution times, speed is not our primary interest. Even if binary diffing takes more than, say, 24 hours to complete, it still saves hundreds of man hours of manual labor.
5) Compared binaries should be neither obfuscated nor packed. 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], [20]).

IV. Features

For each function in each binary executable, we extract a feature vector composed of 9 features. Some of the aforementioned features give an abstract view of the CFG's structural characteristics (vertex and edge classification, graph signatures), others reflect its spatial characteristics (inlinks, outlinks, lumped transition matrix) and others its semantic content (instruction and data histogram). In the following sections, we elaborate on each one in more detail.

A. Vertex and Edge Classification

In previous publications [2], [11], [12], [13], [14], feature vectors extracted from function CFGs, among others, include two features that reflect a digraph's overall structure; the number of vertices (basic blocks) and the number of edges in the corresponding function. The aforementioned features, even though naive at first sight, are good examples of two important structural characteristics that can significantly speed up the pre-filtering phase, by discarding candidates with incompatible

[2]Examples of such control transfers where $f_{src} = f_{dst}$, are sometimes used by compilers emitting PIC code and should be ignored.

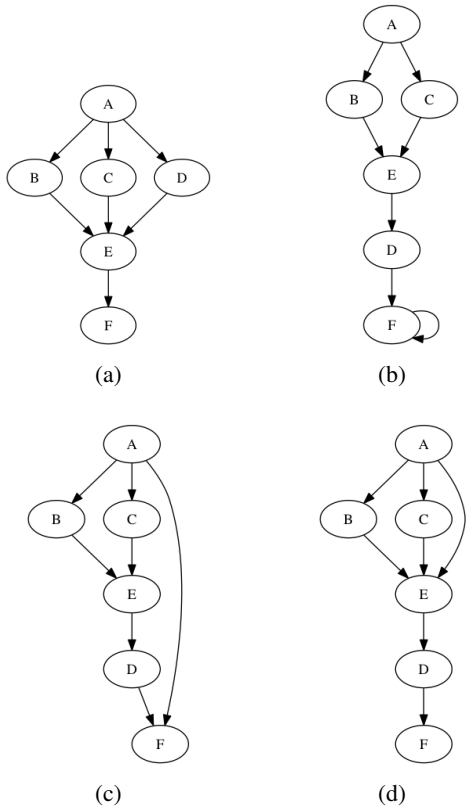


Fig. 1: Several example CFGs. They all share the same number of vertices and edges, yet, we need to come up with ways of distinguishing one from another.

vertex and edge counts during the early stages of diffing. At the same time, however, their pruning power is limited and they can easily introduce non-negligible inaccuracies and/or latencies in the overall process.

Take the digraphs depicted in figures 1(a) and 1(b), for example. These figures show two arbitrary digraphs which might, as well, 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. Notice their obvious differences, however. Figure 1(a) represents a function that executes linearly. On the contrary, 1(b) shows the CFG of 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 [27]), or a *no-return* vertex (as recognized by IDA Pro [16]). We, thus, need a better way of distinguishing CFGs by certain topological vertex characteristics.

Towards this, instead of using the number of vertices as a feature, we came up with the following taxonomy of vertices:

1) **Normal**. All vertices, including those that do not belong to any of the remaining categories.
2) **Entry points**. Vertices which are CFG entry points, that is, program execution enters the function in question via one of these vertices. Notice that a single function may have more than one entry points.
3) **Exit points**. Control flow leaves the corresponding 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**. Vertices with an edge to themselves that also have edges to other vertices. Notice that traps, defined above, are also self-loops.
6) **Loop heads**. Loops' *header* [31] vertices.
7) **Loop tails**. Vertices within a loop's *body* [31] with edges to the loop's *header*.

Categories in the above taxonomy are not mutually exclusive; a basic block may belong to more than one at the same time.

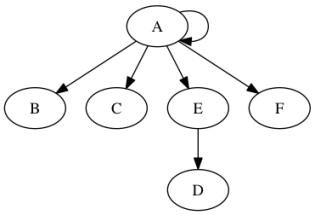
The first proposed feature is a vector with 7 elements. The element at index i holds the number of vertices of category i defined above. So, for example, the first feature extracted from the digraph at 1(a) is $\begin{bmatrix}6 & 1 & 1 & 0 & 0 & 0 & 0\end{bmatrix}$. The corresponding feature extracted from figure 1(b) is $\begin{bmatrix}6 & 1 & 1 & 1 & 0 & 0 & 0\end{bmatrix}$. Clearly, by taking into account this kind of classification, the two CFGs are not similar and, thus, cannot be considered matching candidates.

Now consider figure 1(a) and 1(c). Both digraphs have the same number of vertices and edges. Additionally, both digraphs are only composed of *normal* vertices, a single *entry point* and a single *exit point* (according to the previously defined taxonomy), yet they are very different. As opposed to 1(a), 1(c) looks like a function that, probably, performs some kind of sanity checking on its input arguments and, if checks are not passed, execution flows directly to basic block **F**, which happens to be the function's exit point. Once again we are faced with the same problem; we need more robust features for telling the difference between 1(a) and 1(c). The number of edges alone, is not enough for performing this task.

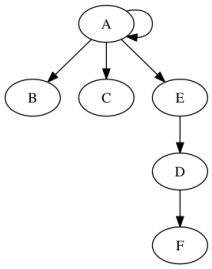
To solve this problem we adopted a taxonomy of edges, similar to this originally described by Tarjan [33]. Briefly, using DFS traversal, we classify edges into the following categories:

1) **Basis edges**. Suppose there exists 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 [1] (the graph layouts of figures 1(a)-1(d) satisfy the above requirement). 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

[1] Where the $depth()$ function is the one resulting from the aforementioned DFS.



(a)



(b)

Fig. 2: Immediate dominator trees for 1(c) and 1(d).

also, sometimes, considered back edges as well.

4) **Cross-links**. Edges between vertices belonging to different DFS sub-trees.

Accordingly, we propose a second feature, a vector of 4 elements. Element at index i holds the number of edges falling in category i. As such, 1(a) is described by $\begin{bmatrix}7 & 0 & 0 & 0\end{bmatrix}$ while 1(c) by $\begin{bmatrix}6 & 1 & 0 & 0\end{bmatrix}$. As it can be seen, the two CFGs can now be separated.

B. Digraph Signatures

The exact vertex and edge classification features of two digraphs do not give any insight on how the latter are actually layed out. Consider 1(c) and 1(d). Both digraphs have a vertex classification feature vector of $\begin{bmatrix}6 & 1 & 1 & 0 & 0 & 0 & 0\end{bmatrix}$ and an edge classification feature vector of $\begin{bmatrix}6 & 1 & 0 & 0\end{bmatrix}$. Distinguishing these two digraphs based solely on the two aforementioned features is not feasible. After all, 6 vertices and 7 edges may be combined in several ways in order to form a digraph.

Careful readers, however, would have probably noticed that the two aforementioned digraphs differ in their dominance relations. For example, in figure 1(c), the immediate dominator of vertex **F** is **A**, while in figure 1(d) is **D**. The *dominator trees* of figure 1(c) and 1(d) can be seen in 2(a) and 2(b) respectively. Encoding this information in a feature will bring out the obvious difference between the two compared digraphs.

In our implementation, we follow a very simple, yet effective, approach. 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 2(a), a DFS traversal of **A B C E D F**, produces the bit signature **110101100100**, or **D64** in hexadecimal. Similarly, in 2(b), traversing the vertices in the order **A B C E D F** produces the bit signature **110101110000**, or **D70** in hexadecimal.

```
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.

The next two features we propose are the bit signatures of both the original function CFG and its immediate dominator tree. The corresponing features extracted from 1(c) and 1(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. It should be noted that, in the public literature, more elaborate signature schemes have been used [4] for the purpose of indexing graph structures. Such schemes are also applicable in our case.

C. Inlinks & Outlinks

For each vertex in the program's CG we add two more features to our feature vector, the number of inlinks (number of callers of this function) and the number of outlinks (number of CALL instructions in the function body). Most previous publications only use the latter. Following the example of few previous publications ([4], [13]), we also include the first in our feature vector as a logical step towards a more complete solution.

D. Instruction Histogram

Features discussed so far can be used to match two digraphs based solely on structural characteristics. Ideally, we would also like a feature that encodes an abstraction over the semantic content of a function. For example, in [13], the authors present a promising system called *discovRE*, 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, and as we are currently only targeting the IA-32 and Intel64 architectures, we make use of Intel's XED library [21] and pyxed [24] to classify instructions based on their instruction form. As opposed to the classification used in [13], 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 3(a) shows a random function, consisting of a single basic block, taken from a compiled Linux kernel binary. Figure 3(b) shows the same function, however, instructions have now been replaced with their instruction forms. The exact meaning of the instruction form constants are outside of the scope of this paper (interested readers can see [22]).

E. String Histogram

Functions usually perform simple operations with ASCII strings. Consider, for example, a function that calls $printf()$. The first argument passed to $printf()$, also referred to as the format string, is usually a constant value. 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 in the two compared subjects and, thus, it would be beneficial to encode this information in our feature vector as well.

An obvious, yet naive, choice is to extend each feature vector with all strings accessed by the corresponding function. This, however, comes with two drawbacks: (i) the overall size of the feature vector is increased linearly with respect to the 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 an abstraction over a subset of the program's data elements manipulated by the function in question. Additionally, it can be extended to take into account, not only strings, but any type of constant data accessed by a function. Such an approach, however, comes with many complications which are outside of the scope of this paper.

F. Markov Lumping of CFGs

The idea of treating a CFG as a Markov system is not new [8]. 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 [8] (figure 15-11), 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.

At a higher level, outgoing transition probabilities, at state i of a Markov system with $n > 0$ transitions, are just a set of numbers obeying the following properties:

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

Having said that, probabilities need not be treated necessarily as the likelihood of a system actually transitioning from one state to another; they can instead be understood as general weights, whose actual interpretation depends on the application domain and reflects the relationship between the source entity and the destination entity. For our purposes, the entities in question are, in fact, basic blocks and the various weights might be computed based on their properties. The resulting transition matrix can, then, be viewed as a summary of the system's overall entity relationships and standard Markov analysis tools may be used to evaluate it. It should also be noted 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 stochastic matrices of two CFGs, we can gain an insight on how similar or different the corresponding functions are.

During our experiments, we evaluated the following weight assignment schemes:

1) A simple scheme where 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 [8].
2) A scheme where edges towards basic blocks with a higher number of instructions are assigned larger weights. The rationale behind this choice 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 probably, actual processing might 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 re-arrangements of instructions result in significant changes in the transition matrix. This, however, makes it more suitable for detecting inexact matches.

3) The last approach involves taking into account various *locality optimizations* [1], that the compilers (which built the executables under comparison) might have performed. 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. Practically, this technique splits a function into *hot paths*, that are highly likely to be executed, and *cold paths*, which are less likely to be reached.
In this scheme we assign a uniform probability to each outgoing transition (like weight assigning scheme 1 above), but we give lower weights to CFG edges leading to basic blocks that belong to distant function chunks (where distance is measured by the addresses the code is located). This choice seems natural, as this is actually in accordance with the rationale used by an optimizing compiler when building the executable.

Of the three schemes described above, we found the third to be the best trade-off between accuracy and matching power. Algorithm 1 shows the actual process of assigning weight values to the edges of an arbitrary CFG.

Algorithm 1 Weight assignment

```
1:  procedure assign_edge_weights(CFG_f)
2:      chunks ← cfg_chunks(CFG_f)
3:      chunk_weights ← {|chunks|, |chunks| − 1, ..., 1}
4:      for all v ∈ V_CFG_f do
5:          weight ← 0
6:          for all s ∈ succ(v) do
7:              i ← chunk_index(chunks, s)
8:              weight ← weight + chunk_weights[i]
9:          weight ← 1/weight
10:         for all s ∈ succ(v) do
11:             i ← chunk_index(chunks, s)
12:             (v, s)_weight = weight * chunk_weights[i]
```

Algorithm 1 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 consists of 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). Last but not least, $weight$ 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* $P_0 = \{P_{0,0}, P_{0,1}, ..., P_{0,N-1}\}$. 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):

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

Algorithm 2 Initial partitioning algorithm

```
1:  procedure find_component(v, P_0)
2:      Q ← ∅
3:      for all P ∈ P_0 do
4:          if (succ(v) − {v}) ∩ P = ∅ then
5:              r ← True
6:              for all q ∈ P do
7:                  if v ∈ succ(q) then
8:                      r ← False
9:                      break
10:             if r = True then
11:                 Q ← P
12:                 break
        return Q

13: procedure process_component(P, P_0)
14:     change ← False
15:     for all v ∈ P do
16:         if (succ(v) − {v}) ∩ P ≠ ∅ then
17:             change ← True
18:             P ← P − {v}
19:             Q ← find_component(v, P_0)
20:             Q ← Q ∪ {v}
21:             if Q ∉ P_0 then
22:                 P_0 ← P_0 ∪ {Q}
        return change

23: procedure create_initial_components(CFG_f)
24:     P_0 = {V_CFG_f}
25:     change ← True
26:     while change = True do
27:         change ← False
28:         for all P ∈ P_0 do
29:             change ← change ∨ split(P, P_0)
        return P_0
```

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.

Last but not least, we apply a standard lumping algorithm [9], in order to refactor the sets in P_0, and use the lumped system's transition matrix as our feature vector's last element (other matrix characteristics, like its eigenvalues, might be used instead). For a simplified overview of the overall lumping process, see algorithm 3. The procedure $markov\_lumping()$ is the one described in [9].

Algorithm 3 Overall lumping

```
1: procedure lump(CFG_f)
2:     P_0 ← create_initial_components(CFG_f)
3:     P ← markov_lumping(P_0) return P
```

V. Experimental Results

Our proof of concept implementation, which was developed with the sole purpose of demonstrating the effectiveness of the proposed features, consists of two utilities, along with accompanying helper libraries, all coded in Python. The first utility is an IDA Python plug-in for IDA Pro, which also makes use of pyxed [24], the Python bindings library for Intel's XED, in order to bypass the various limitations that the built-in IDA Pro disassembler API suffers from. It extracts the feature vectors of all functions found in an IDA Pro database and saves them in a standard Pickle format. The second utility is the main program that, given the two pickle files of two IDA Pro databases, performs the matching process using the greedy algorithm 4. The helper libraries are simple Python modules that implement various graph theory abstractions and algorithms. The overall implementation consists of about 1800 lines of Python code.

Algorithm 4 consists of two procedures, namely $match()$, the main entry point, and $match\_fvs()$ which performs a single round of our overall matching process. We start by describing the first. Lines 19 and 20 build two key-value datastructures that map program functions to their feature vectors, as extracted by our tools, for programs p_1 and p_2, named FV_{p_1} and FV_{p_2} respectively. In each matching round, $match\_fvs()$ is called three times. One for detecting unique

Algorithm 4 Greedy matching algorithm

```
1: matches ← 0
2: mismatches ← 0

3: procedure match_fvs(FV_{p1} FV_{p2})
4:     M ← ∅
5:     for all (fv_{p1}, fv_{p2}) ∈ FV_{p1} × FV_{p2} do
6:         if fv_{p1} = fv_{p2} then
7:             if |{fv_{p1}} ∩ FV_{p1}| = 1 then
8:                 if |{fv_{p2}} ∩ FV_{p2}| = 1 then
9:                     f_{p1} ← FV_{p1}^{-1}[fv_{p1}]
10:                    f_{p2} ← FV_{p2}^{-1}[fv_{p2}]
11:                    M ← M ∪ {f_{p1} → f_{p2}}
12:                    FV_{p1} ← FV_{p1} − {f_{p1} → fv_{p1}}
13:                    FV_{p2} ← FV_{p2} − {f_{p2} → fv_{p2}}
14:                    if oracle(f_{p1}, f_{p2}) = True then
15:                        matches ← matches + 1
16:                    else
17:                        mismatches ← mismatches + 1
       return M

18: procedure match(p_1, p_2)
19:     FV_{p1} ← {f → fv_f | f ∈ V_{CG_{p1}}}
20:     FV_{p2} ← {f → fv_f | f ∈ V_{CG_{p2}}}
21:     do
22:         M ← match_fvs(FV_{p1}, FV_{p2})
23:         while |M| increases do
24:             for all f_{p1} → f_{p2} ∈ M do
25:                 FV'_{p1} ← {FV_{p1}[f] | f ∈ succ(f_{p1})}
26:                 FV'_{p2} ← {FV_{p2}[f] | f ∈ succ(f_{p2})}
27:                 M ← M ∪ match_fvs(FV'_{p1}, FV'_{p2})
28:                 FV'_{p1} ← {FV_{p1}[f] | f ∈ pred(f_{p1})}
29:                 FV'_{p2} ← {FV_{p2}[f] | f ∈ pred(f_{p2})}
30:                 M ← M ∪ match_fvs(FV'_{p1}, FV'_{p2})
31:     while M ≠ ∅
```

matches in the, still unmatched, feature vector sets (line 22) and twice for matching the successors and predecessors of already matched functions (lines 27 and 30). This process is repeated as long as new entries are added in M (line 23), a third datastructure that temporarily maps functions from p_1 to functions of p_2.

Lines 5-8 in $match\_fvs()$ iterate through the input feature vectors and try to match elements that appear once in both multisets. In such an event, the corresponding functions are retrieved (lines 9 and 10), the set of matches M is updated (line 11), and the feature vectors are removed from their multisets (lines 12 and 13). Last but not least, if our oracle signifies a match (line 14), the number of correct matches is updated (line 15). On the opposite case, the number of mismatches is increased instead (line 17).

To further stress the effectiveness of our features, we decided to compare our implementation with Diaphora, the current industry standard in binary diffing, developed by Jox-

TABLE I: Test corpus information.

Software	Architecture	Compiler
Linux 4.4.1	Intel 64	GCC 6.3.1
Linux 4.4.40	Intel 64	GCC 5.4.0
Linux 4.4.40	Intel 64	GCC 6.3.1
Firefox xul.dll 41.0	IA-32	MSVC
Firefox xul.dll 44.0.2	IA-32	MSVC
OpenSSH 7.0	Intel 64	GCC 4.8.4
OpenSSH 7.6	Intel 64	Clang 700.1

Software	Functions	Chunked functions
Linux 4.4.1	39661	71 (0.179%)
Linux 4.4.40	39723	70 (0.176%)
Linux 4.4.40	39727	72 (0.181%)
Firefox xul.dll 41.0	168113	12641 (7.519%)
Firefox xul.dll 44.0.2	178032	13421 (7.539%)
OpenSSH 7.0	2515	0 (0.000%)
OpenSSH 7.6	1755	0 (0.000%)

ean Koret. Additionally, since we are only interested in finding exact matches, we only compare our tool with Diaphora's capability of detecting such perfect matches and set the latter to ignore function names in executables that come with debug symbols. Given the Diaphora results in an SQLite format, we count the number of perfect matches by using the following SQL query:

```
SELECT COUNT(*) FROM results WHERE type = "best";
```

Names of matched functions, however, have to go through the same oracle that was used for evaluating our own implementation, which in this case, is just a matter of augmenting the above SQL query with the following additional predicates:

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

Briefly, the above query assumes that a match is correct only as long as the two function names either match exactly, or any of the two has the *sub_* prefix, automatically assigned by IDA Pro to functions which lack debugging information. Such pairs are, for the sake of this experiment, considered valid matches, as there is no easy way of manually verifying the subjects' similarity.

Our test corpus is presented in table I where one can see the architecture, the compiler used and the number of functions and chunked functions in each executable. All of the following open source projects where built without explicitly disabling compiler optimizations (the optimization level used is either the one hardcoded in each project's Makefile, or the default used by the corresponding compiler). The experiments we performed are listed below and corresponding results are shown in table II (first column on the left holds the experiment number as listed below).

TABLE II: Summary of experimental results.

	Tool	Time	Matches	Mismatches
1	Our Approach	1720s	32958 (83.099%)	0 (0.000%)
	Diaphora	5999s	32726 (82.514%)	71 (0.179%)
2	Our Approach	1965s	20505 (51.700%)	10 (0.025%)
	Diaphora	5845s	18943 (47.762%)	175 (0.441%)
3	Our Approach	1953s	21466 (54.039%)	8 (0.020%)
	Diaphora	5814s	20587 (51.826%)	150 (0.378%)
4	Our Approach	48834s	79395 (47.227%)	1966 (1.169%)
	Diaphora	99940s	73638 (42.802%)	2774 (1.650%)
5	Our Approach	1s	2 (0.114%)	2 (0.114%)
	Diaphora	298s	0 (0.000%)	1 (0.057%)

1) Linux 4.4.1 against Linux 4.4.40 both compiled with GCC 6.3.1.
2) Linux 4.4.1 compiled with GCC 6.3.1 against Linux 4.4.40 compiled with GCC 5.4.0.
3) Linux 4.4.40 compiled with GCC 5.4.0 against Linux 4.4.40 compiled with GCC 6.3.1.
4) Firefox *xul.dll* 41.0 against Firefox *xul.dll* 44.0.2.
5) OpenSSH 7.0 compiled with GCC 4.8.4 against OpenSSH 7.6 compiled with Clang 700.1.

In the first experiment we can see that Diaphora detected 32726 perfect matches in 5999 seconds, while our tool 32958 in 2259 seconds. Evidently, our tool performs about 2.5 times faster and detects 232 more perfect matches. However, as it can be seen, both tools perform very well, as they manage to match about 82-83% of the functions found in the kernel binary. The Linux kernel source code is very strictly organized and all commits go through scrutiny before being accepted. Kernel developers are not allowed to make groundbreaking changes and they have to stick to specific rules when writing code. This makes the Linux kernel source code ideal for evaluating greedy algorithms like algorithm 4. Another interesting thing to note is that, according to our oracle, our tool detected no false positives, while Diaphora detected 71.

The next experiment involves comparing the same Linux kernel versions as before, but one of the two is now compiled with an older GCC version. More specifically, we compare Linux 4.4.1 compiled with GCC 6.3.1 against 4.4.40 compiled with GCC 5.4.0. In this case, our tool detects 1562 more matches in 3880 seconds less with only 10 false positives.

The third Linux kernel experiment is also a very interesting example, showcasing the effects of different compiler versions used on the same kernel source tree with the exact same kernel configuration. Obviously, since we only look for exact matches, we cannot expect the two binaries to match perfectly, as even minor variances in versions of GCC might introduce different compile-time optimizations. Once again, our proof-of-concept tool outperforms Diaphora, both in speed and efficiency. Diaphora finds 20587 best matches in 5814 seconds

(51.826% of all functions), while our tool 21466 in 1953 seconds (54.039%). The number of false positives detected by our tool is, again, classes of magnitude less than Diaphora's (8 vs. 150).

In our fourth experiment we used our approach and Diaphora on an even larger sample, namely Firefox's *xul.dll*, a shared library responsible for implementing the browser's core functionality. Developed in C++, *xul.dll* is an ideal sample for evaluating the two tools, as the various high level C++ constructs usually result in complicated series of assembly code instructions that may vary greatly from version to version. Also notice that, according to table I, MSVC seems to split functions in chunks far more often than GCC and, thus, in this experiment, our lumped Markov transition matrix plays an important role. We compared *xul.dll* version 41.0 against version 44.0.2 consisting of 168113 and 178032 functions respectively. Diaphora completed in about 27 hours, found 73638 best matches (42.802%) and 2774 false positives (1.650%). Our tool completed in about 13.5 hours (48834 seconds) and reported 79395 perfect matches (47.227%, 5757 more than Diaphora) and 1966 false positives (1.169%, 808 less than Diaphora).

Last but not least, experiment 5, involves comparing different OpenSSH versions compiled with two different compilers on two different operating systems. The purpose of this experiment is to stress test our tool's fault tolerance. As we can see, our tool detects 2 best matches and 2 false positives while Diaphora 0 and 1 respectively. Both tools perform reliably and, as expected, they detect only minimal similarities. It should be noted that, even though several parts of the programs are semantically equivalent, their assembly instructions differ significantly. This is the reason both tools cannot detect similarities.

VI. Conclusion

We have presented efficient features that can be used to perform function matching in two compared programs. We have empirically shown the claimed effectiveness by comparing our proof-of-concept function matching software with Diaphora, the leading industry standard in binary diffing. More specifically, we have compared the two tools' abilities to discover exact matches between compared executables.

Many questions still remain unanswered however. First and foremost, coming up with equally efficient distance metrics for the proposed features is a very challenging task and the subject of our future research. With the use of distance metrics, algorithm 4 will also be capable of considering inexact, non-perfect matches and will allow for a fair comparison of our approach with other binary diffing software like Zynamics BinDiff, which, according to public domain information, implements a propagation phase.

Furthermore, we believe there's still a lot to be researched regarding the Markov properties of computer programs. Such a study may, in turn, give birth to even more efficient Markov-based features for telling the similarities and the differences between functions in two programs. Additionally, we believe there's a lot of room for improvement on the weight assignment schemes presented in section IV. They need to be more thoroughly studied, evaluated and extended to mirror more distinctive characteristics of the subject CFGs.

Another limitation of our work can also be seen in section IV. The instruction classification feature is based on an existing toolset targeting the Intel X86 architectures. With the (relatively recent) emergence of ARM CPUs in the mobile phone market, security researchers have shifted their focus to these targets. Consequently, it is imperative for a binary diffing tool to be able to handle these architectures as well. For this purpose, executable code of the compared subjects, needs to be translated to a platform-agnostic intermediate representation (IR) first. The resulting IR instructions can then be classified in a similar fashion. Specific implementation details, however, depend on the IR used.

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 [17]. In such an approach, feature vectors can be extracted from the compared programs' data elements to aid in the overall diffing process.

Once our study matures, our prototype system will be released as an open source project, freely available to the public.

References

[1] A. Aho, M. Lam, R. Sethi, J. Ullmanr, *Compilers: Principles, Techniques, and Tools (2nd Edition)*, Addison-Wesley Longman Publishing Co., 2006.

[2] M. Bourquin, A. King, E. Robbins, "BinSlayer: Accurate Comparison of Binary Executables", *2nd ACM SIGPLAN Program Protection and Reverse Engineering*, 2013.

[3] S. Cesare, Y. Xiang, "Classification of Malware Using Structured Control Flow", *Proceedings of the 8th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2010)*, 2010.

[4] S. Cesare, Y. Xiang, W. Zhou, "Control Flow-based Malware Variant Detection", *IEEE Transactions on Dependable and Secure Computing*, 2013.

[5] L. P. Cordella, P. Foggia, C. Sansone, M. Vento, "An improved algorithm for matching large graphs", *3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition*, 2001.

[6] L. P. Cordella, P. Foggia, C. Sansone, M. Vento, "An Efficient Algorithm for the Inexact Matching of ARG Graphs Using a Contextual Transformational Model", *Proceedings of the 13th International Conference on Pattern Recognition*, 1996.

[7] L. P. Cordella, P. Foggia, C. Sansone, M. Vento, "Subgraph Transformations for the Inexact Matching of ARG", *Computing*, suppl. 12, pp. 43-52, 1998.

[8] N. Deo, *Graph Theory with Applications to Engineering and Computer Science*, Prentice-Hall Inc, 1974.

[9] S. Derisavi, H. Hermanns, W. Sanders, "Optimal State-Space Lumping in Markov Chains", *Information Processing Letters*, pp. 309-315, 2003.

[10] J. Koret. *Diaphora: A Free and Open Source Program Diffing Tool* [Online]. Available: http://diaphora.re/

[11] T. Dullien, R. Rolles, "Graph-based comparison of Executable Objects", *Proceedings of the Symposium sur la Securite des Technologies de l'Information et des Communications*, 2005.

[12] T. Dullien, E. Carrera, S. M. Eppler, S. Porst, "Automated Attacker Correlation for Malicious Code", *NATO Information Systems Technology (IST) 091*, 2010.

[13] S. Eschweiler, K. Yakdan, E. Gerhards-Padilla, "discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code", *SP '15 Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2016.

[14] H. Flake, "Structural Comparison of Executable Objects", *Proceedings of the IEEE Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2004.

[15] D. Gao, M. Reiter and D. Song, "BinHunt: Automatically Finding Semantic Differences in Binary Programs", *Information and Communications Security*, pp. 238-255, 2008.

[16] Hex-Rays. *IDA Pro* [Online]. Available: https://www.hex-rays.com/products/ida/

[17] T. A. D. Henderson, A. Podgurski, "Sampling code clones from program dependence graphs with GRAPLE", *SWAN 2016 Proceedings of the 2nd International Workshop on Software Analytics*, 2016.

[18] R. Howard, "Dynamic Probabilistic Systems: volume I: Markov Models", John Wiley & Sons Inc, 1971.

[19] R. Howard, "Dynamic Probabilistic Systems. Volume II: Semi-Markov and Decision Processes", John Wiley & Sons Inc, 1971.

[20] X. Hu, T. Chiueh, and K. G. Shin, "LargeScale Malware Indexing Using Function-Call Graphs", *Computer and Communications Security*, pp. 611-620, 2009.

[21] Intel. *Intel X86 Encoder Decoder Software Library* [Online]. Available: https://software.intel.com/en-us/articles/xed-x86-encoder-decoder-software-library

[22] Intel. *Intel X86 Encoder Decoder* [Online]. Available: https://software.intel.com/sites/landingpage/xed/ref-manual/html/group__IFORM.html

[23] M. Jurczyk. *Using Binary Diffing to Discover Windows Kernel Memory Disclosure Bugs* [Online]. Available: https://googleprojectzero.blogspot.gr/2017/10/using-binary-diffing-to-discover.html

[24] C. Karamitas. *Python bindings for Intel's XED* [Online]. Available: https://github.com/huku-/pyxed

[25] O. Kostakis, J. Kinable, H. Mahmoudi, K. Mustonen, "Improved call graph comparison using simulated annealing", *Proceedings of the 2011 ACM Symposium on Applied Computing*, 2011.

[26] J. Ming, M. Pan and D. Gao, "iBinHunt: Binary Hunting with Interprocedural Control Flow", *Lecture Notes in Computer Science*, pp. 92-109, 2013.

[27] J. Ming, D. Xu, Y. Jiang, D. Wu, "BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking", *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[28] V. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals", *Soviet Physics Doklady*, pp. 707-710, 1966.

[29] McAfee (2017, March). *McAfee Labs Threats Report April 2017* [Online]. Available: https://www.mcafee.com/us/resources/reports/rp-quarterly-threats-mar-2017.pdf

[30] Panda Security (2017, May). *PANDALABS QUARTERLY REPORT Q1 2017* [Online]. Available: http://www.pandasecurity.com/mediacenter/src/uploads/2017/05/Pandalabs-2017-T1-EN.pdf

[31] G. Ramalingam, "On loops, dominators, and dominance frontiers", *PLDI'00 Proceedings of the ACM SIGPLAN 2000 conference on Programming Language Design and Implementation*, pp. 233-241, 2000.

[32] SafeCorp. *Detecting Software IP Theft Using CodeMatch* [Online]. Available: https://www.safe-corp.com/documents/CodeMatch_Whitepaper.pdf

[33] R. Tarjan "Testing flow graph reducibility", *STOC'73 Proceedings of the fifth annual ACM symposium on Theory of computing*, pp. 96-107, 1973.

[34] A. Valmari, G. Franceschinis, "Simple O(mlogn) Time Markov Chain Lumping", *TACAS'10 Proceedings of the 16th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 38-52, 2010.

[35] Z. Wang, K. Pierce, S. McFarling, "BMAT - A Binary Matching Tool", *Second ACM Workshop on Feedback-Directed and Dynamic Optimization*, 1999.

[36] Z. Wang, K. Pierce, S. McFarling, "BMAT - A Binary Matching Tool for Stale Profile Propagation", *The Journal of Instruction-Level Parallelism*, 2002.

[37] Zynamics. *BinDiff* [Online]. Available: https://www.zynamics.com/bindiff.html