BinSlayer

Accurate Comparison of Binary Executables

Martial Bourquin Andy King Edward Robbins
University of Kent
bourquin.martial@gmail.com a.m.king@kent.ac.uk er209@kent.ac.uk

Abstract

As the volume of malware inexorably rises, comparison of binary code is of increasing importance to security analysts as a method of automatically classifying new malware samples; purportedly new examples of malware are frequently a simple evolution of existing code, whose differences stem only from a need to avoid detection. This paper presents a polynomial algorithm for calculating the differences between two binaries, obtained by fusing the well-known BinDiff algorithm with the Hungarian algorithm for bi-partite graph matching. This significantly improves the matching accuracy. Additionally a meaningful metric of similarity is calculated, based on graph edit distance, from which an informed comparison of the binaries can be made. The accuracy of this method over the standard approach is demonstrated.

1. Introduction

Automated auditing of binary code is a topic of increasing interest to both academia and industry. The pace of software development is such that governments and other organisations must deal with software updates on an almost daily basis that must be vetted for vulnerabilities. Meanwhile the anti-virus (AV) industry is attempting to cope with an almost constant proliferation of new malware, most of which are simple evolutions of known code, and needs to identify, classify and protect against each. For these tasks binary differencing is of great necessity; for vulnerability discovery in software updates the security analyst wishes to examine only the new code and so must identify what is old and what is new quickly, while in malware identification/classification the key questions are whether the code is a mutation of a known piece of malware, and if so, what has changed that needs to be accounted for? In contrast to various nefarious applications, such as reversing security patches, binary differencing has also found application in legal settings as a litmus test for infringement against open source licensing and copyright agreements.

The most obvious cause of changes between different versions of the same application is simple addition and removal of code. However there are at least two other sources of differences which result in some binaries appearing different, while in fact being functionally identical. The first is so-called 'server-side polymorphism', typically a result of using different compiler optimisations, a different compiler version, or a different compiler all together. The second is wilful obfuscation in order to bypass detection by signature engines of anti-virus software. Regardless of the source of the changes, they render simple comparison techniques such as byte-for-byte file differencing almost completely useless. Some examples of such changes are:

- Sequences of opcodes may remain identical but register allocation can change depending on availability at a compile time, i.e. 'ecx' may become 'edx' and so on.
- If instructions do not depend on other instructions they may be reordered due to pipeline optimisations without affecting operational semantics.
- Junk/do-nothing code is inserted between instructions in order to bypass detection by signature engines of anti-virus software.
- Obfuscating [7] or diversifying [2, 8] transforms are applied to a binary to replace sets of instructions by equivalent ones, thus preserving the action of the code but changing its representation at the instruction level. In the former case the new code aims to be more cryptic; in the latter case it aims to be unique.

This paper seeks to address these issues in binary comparison by bringing together two existing approaches to the problem. Both techniques perform structural matching [16]. In the context of binaries this means that they seek to identify/recover key structural components from each binary to compare and match, establishing an isomorphism between the two binaries.

The first technique was developed by Thomas Dullien (AKA Halvar Flake) and his colleagues in their tool BinDiff [15]. BinDiff attempts to reconstruct the Control Flow Graph (CFG) of each binary, as it uses the functions and basic blocks as the units for comparison.

Note that the Control Flow Graph can be understood as a graph of graphs; at the top level it consists of the Call Graph (CG), which links functions together via function calls and returns, while each function is itself a CFG linking basic blocks via simple branches/jumps. To clearly disambiguate whether the term CFG refers to the CFG as a graph of graphs of a whole program, or the CFG of a single function, henceforth when used in this paper it can be assumed that it refers to the graph of a function alone, unless otherwise stated.

The BinDiff algorithm compares functions and basic blocks based on graph-centric properties derived as identifiers for them, such as number of out-going edges, in-coming edges etc. These properties make up a tuple for each function or basic block, but it is important to note that the tuples are *not* necessarily unique. BinDiff first creates an initial set of matches consisting only of uniquely identical functions from each binary. This set is then expanded upon by taking each matched pair and searching for more unique matches, but only amongst their unmatched neighbours, thus limiting the search space and increasing the likelihood of finding unique matches. This is then repeated with the new matches exhaustively

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPREW '13 Jan 26, 2013 Rome, Italy.
Copyright © 2013 ACM 978-1-4503-1857-0/13/01…$15.00

WIZ, Inc. EXHIBIT - 1047
WIZ, Inc. v. Orca Security LTD.

Algorithm 1: Initial match discovery

```
function initialMatches(S_A, S_B);
M ← ∅;
foreach vertex a_i ∈ S_A do
    foreach Selector ε do
        if (a_i, b_j) ← ε(a_i, S_B) then
            M ← M ∪ {a_i ↦ b_j};
            S_A ← S_A \ {a_i};
            S_B ← S_B \ {b_j};
            break;
return (M, S_A, S_B);
```

until no new unique matches can be found. The treatment of children and parent nodes are discussed subsequently. The BinDiff algorithm can be applied to match blocks in an analogous way to matching functions.

The second technique is also graph-centric [14], and is based on a thread of work in Graph-Edit-Distance (GED) computation [17, 23]. GED, from graphing theory, is the minimal number of edit operations required to transform a graph $\mathcal{G}_A$ into a graph $\mathcal{G}_B$ [30]. A series of edit operations is called an edit path. With a directed labelled graph, an edit operation is either vertex substitution, vertex insertion or vertex deletion. A vertex substitution occurs when a vertex in graph $\mathcal{G}_A$ is substituted for one from graph $\mathcal{G}_B$. Edge substitutions are defined analogously. Vertex insertion occurs when a vertex from graph $\mathcal{G}_B$ is added to graph $\mathcal{G}_A$. Edge insertion is defined in a corresponding manner. Vertex deletion and edge deletion are defined similarly.

There are multiple edit paths from one graph to another, as one would expect, but the desired path is the one with the lowest total cost of edit operations. GED also provides a convenient and meaningful metric for measuring graph, and hence binary, similarity. Unfortunately computation of the GED is an NP-hard problem [33]. However, it has been shown [26, 30] that an approximation can be obtained by transforming the problem into a weighted bipartite graph matching problem, which can be solved in cubic time with the Hungarian algorithm [24]. Furthermore, Hu et Al. have shown that performing automatic analysis and classification of malware samples into families using an approximation of the GED is indeed possible [17].

BinDiff is fast and moreover the matches it makes are usually accurate. The problem is that the matches it derives are often incomplete: It frequently fails to match a pair of functions that are conspicuously similar. This issue stems from the fact that a function can exist in one binary whose tuple does not uniquely match the tuple of a function in the other. The real problem is that basing the matching of functions purely on tuples is brittle, as they abstract the structural features of a function. This is not a problem if the desire is to only match functions which are identical as deemed by the abstraction, rather than those that have undergone modification and are similar but not identical. It is arguable whether this is an issue when two related executables are scrutinised so as to diagnose the effect of a patch. However, for the problem of malware classification, namely quantifying the degree of similarity between two arbitrary binaries, it is important to also match similar functions, a concept which is crystalised with the notion of shortest edit path. BinDiff does provide a metric for the similarity of the graphs, but it is not particularly meaningful as the developers of BinDiff themselves concede [22]. On the other hand GED is one of, perhaps the, most natural graph metric [11], and its solution always provides a suggested match for remaining unmatched nodes.

Algorithm 2: Match propagation

```
function propagateMatches(M, S_A, S_B);
foreach {a_i ↦ b_j} ∈ M do
    foreach Property π do
        S'_A ← π(a_i, S_A);
        S'_B ← π(b_j, S_B);
        if S'_A ≠ ∅ ∧ S'_B ≠ ∅ then
            foreach vertex a'_i ∈ S'_A do
                foreach Selector ε do
                    if (a'_i, a'_j) ← ε(a'_i, S'_B) then
                        M ← M ∪ {a'_i ↦ b'_j};
                        S'_A ← S'_A \ {a'_i};
                        S'_B ← S'_B \ {b'_j};
                        S_A ← S_A \ {a'_i};
                        S_B ← S_B \ {b'_j};
                        break;
return (M, S_A, S_B);
```

The work presented here performs automated static comparison of binary executable files by combining the two approaches outlined. The BinDiff algorithm for matching both functions and basic blocks between two binaries is improved by augmentation with the Hungarian algorithm, which finds matches with the minimum possible GED over all functions and basic blocks. The net effect of overlaying the Hungarian algorithm on top of the BinDiff algorithm is a more robust matching procedure that mops up and matches functions that would otherwise remain unmatched. We consider this to be the main contribution of the work. Moreover, the mop up phase is cubic, which is pleasing considering the computational complexity of GED.

GED is also adapted to provide an accurate metric for the edit operations required to transform one binary into another. The system is implemented in a novel way that provides an accurate final edit path to the analyst at the end of the diffing process, and quantitative accuracy results are presented for the number of functions matched that conclusively demonstrate the value of this method. As far as we know, our work also represents a step change in systematic evaluation; previously accuracy testing has been carried out on a few examples, at best.

2. BinDiff

As briefly explained in section 1, BinDiff [16] associates each basic block/function with a tuple that describes some of its properties. For example, the tuple for a function is (α, β, γ) where:

- α is the number of *basic blocks* in the CFG of the function.
- β is the number of *edges* in the CFG of the function.
- γ is the number of *function calls* in the CFG of the function.

In fact, in the later paper by Dullien [9], the concept of matching is generalised using the idea of a selector. A selector is a function that takes a vertex a_i from a graph G_A, and a set $\mathcal{S}_B$ of vertices from another graph G_B and returns a vertex from S_B that uniquely matches a_i if precisely one exists. Thus matching based on the tuple defined above can be thought of as a selector that simply compares the elements of the tuples for equality and uniqueness. Another example of a selector is a checksum selector based on a cyclic redundancy check (CRC32) of the machine code that constitutes a block. For unstripped binaries the symbolic names of the functions

Algorithm 3: BinDiff

function binDiff($\mathcal{G}_A$, $\mathcal{G}_B$);
$\mathcal{S}_A \leftarrow \mathcal{G}_A$;
$\mathcal{S}_B \leftarrow \mathcal{G}_B$;
$\mathcal{M}' \leftarrow \emptyset$;
$(\mathcal{M}, \mathcal{S}_A, \mathcal{S}_B) \leftarrow$ initialMatches($\mathcal{S}_A$, $\mathcal{S}_B$);
while $\mathcal{M}' \neq \mathcal{M}$ **do**
  $\mathcal{M}' \leftarrow \mathcal{M}$;
  $(\mathcal{M}, \mathcal{S}_A, \mathcal{S}_B) \leftarrow$ propagateMatches($\mathcal{M}$, $\mathcal{S}_A$, $\mathcal{S}_B$);
return $(\mathcal{M}, \mathcal{S}_A, \mathcal{S}_B)$;

provide a natural selector [9], however this selector should be applied with caution as very different functions can share the same name, for instance main.

The BinDiff algorithm finds an initial set of matches by comparing (using selectors) all vertices $\mathcal{S}_A$ in the first graph with all the vertices $\mathcal{S}_B$ in the second graph. In Dullien's first publications [14, 15] these initial matches were then expanded upon by taking all neighbours of a pair of matched vertices, and searching for matches amongst those. This limited the vertices that were searched, and in so doing increased the likelihood of finding new matches. However, Dullien later [9] generalises neighbours to more abstract concepts that he refers to as properties. Consider a function that takes a vertex and returns all neighbours of that vertex; it is in fact taking a vertex and returning a subset of the vertices in the graph for which the neighbour relationship holds. Properties abstract the construction of subsets to arbitrary relationships, for example, the parent property can be used to return all parents of a vertex, or the child property to return all children.

2.1 BinDiff Algorithm

Initial matches are found by using all selectors across all the vertices in the first graph $\mathcal{G}_A$. If a selector finds a uniquely matching vertex in the second graph $\mathcal{G}_B$ it returns a match (a_i, b_j). The returned matches then create an initial mapping $\mathcal{M}$, containing all matching vertices. The initial match discovery algorithm is shown in Algorithm 1.

The next step of the BinDiff algorithm is to find more matches based on those found initially. The propagateMatches algorithm listed in Algorithm 2 does exactly this. For each initial match properties are used to create subsets $\mathcal{S}'_A$ and $\mathcal{S}'_B$ of the remaining unmatched graph vertices, consisting of all the neighbours of the vertices in the match, or all the parents, etc., depending on which property is used. Selectors are then used on the vertices in the subsets to find new matches. Note that, as in initial match discovery, after a match is discovered the vertices must be removed from the sets so that the algorithm only ever searches amongst unmatched vertices.

Finally the main program loop (Algorithm 3) brings all this together by first creating the initial matches, and then by calling propagate until no new matches are discovered. Like many iterative algorithms BinDiff can be reformulated in terms of so-called delta sets so that propogateMatches is not called on all of $\mathcal{M}$, but merely on those pairs that have been most recently added.

2.2 Illustrated Example

We will now consider an example execution of the BinDiff algorithm by comparing a simple binary with itself, using only the standard 3-tuple selector given previously. Listing 1 is a listing of the C source code of the program, while Figure 1 shows the recovered call graph (CG). Notice that only 8 functions can be seen in the source code of the program, while the CG contains 18 nodes. This

Listing 1. A basic C program

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void D() {
  printf("\n"); return;
}

void C() {
  D();
  printf("\n");
}

void B() {
  C();
  printf("\n");
}

void A() {
  B();
  printf("\n");
}

int main() {
  srand(45);
  A();
  B();
  C();
  D();
  rand();
}
```

is due to code inserted by the compiler for program initialisation, and support functions such as srand. Since we are comparing the binary with itself we would expect that all vertices will be matched.

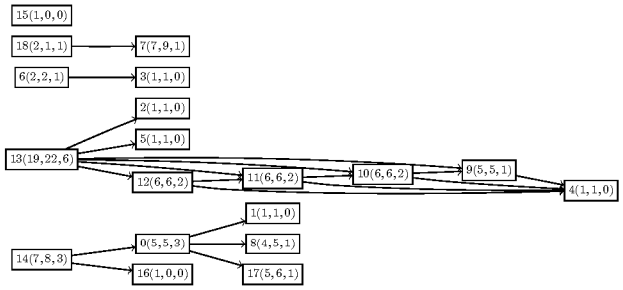

Figure 1. Extracted Call-Graph from the binary 'test'

Searching for initial matches selects 9 vertices: 0, 6, 7, 8, 9, 13, 14, 17 and 18. The matches represent those vertices that are unique. For example, note that vertex 7 has been matched since the tuple (7, 9, 1) is unique, while vertex 10 is unmatched since the tuple (6, 6, 2) also characterises vertex 11.

BinDiff now proceeds with propagation to find more matches. On the first iteration of propagateMatches, vertices 1, 3, 4, 10 and 16 are matched. To demonstrate how, consider propagation on vertex 9 using the child property. We can see that vertex 4, which is not unique in the graph, is unique in the subset of children of vertex 9, being its only child, and so it is immediately selected. Similarly, vertex 10 is matched in this iteration when applying the parent property to vertex 9, as it is both unique in the parent subset

and the only remaining unmatched parent (vertex 13 having already been removed in initial matching).

The final iteration will match vertices 11 and 12 in a similar manner, and after this the algorithm stops since it cannot propagate any more matches. Thus, although the compared binaries in this example are identical, the algorithm failed to match three vertices; 2, 5, and 15. Indeed, vertices 2 and 5 have the same tuple, so they cannot be matched because the selector will always return the empty set.

However, vertex 15 has a unique label in the unmatched vertex sets. We could run the the BinDiff algorithm again, using the unmatched sets of vertices from each binary as inputs, and in this second run vertex 15 will be matched by the initial match function immediately. However a better tactic is to simply modify the BinDiff algorithm to call the initial matches function again after propagation has finished.

2.3 Similarity Metric

Once BinDiff has computed its matches it presents statistics that purport to measure similarity and confidence for each match and for matching across the binary as a whole. The BinDiff manual explains these values as follows:

> "The confidence value displayed by the differ is the average algorithm confidence (match quality) used to find a particular match weighted by a sigmoid squashing function ... The final similarity value is multiplied by confidence - even a seemingly good match is not trustworthy if produced by weak algorithms."
>
> `www.zynamics.com/bindiff/manual/#N2049A`

Similarity is constructed from a weighted sum, though it is not clear how confidence values are derived.

3. Bipartite Graph Matching, GED and the Hungarian Algorithm

The Hungarian algorithm [24] was designed to solve the assignment problem, which is well known in the combinatorial optimisation literature, and is concerned with finding an optimal assignment between two input sets of the same size. However, the algorithm can be reinterpreted as finding an optimal matching in a bipartite graph, as first demonstrated by Jonker and Volgenant [18]. A bipartite graph matching is defined as a bijective function $\phi : V_{\mathcal{G}_A} \rightarrow V_{\mathcal{G}_B}$, where $V_{\mathcal{G}_A}$ is the vertex set of graph $\mathcal{G}_B$ and $V_{\mathcal{G}_A}$ the vertex set of graph $\mathcal{G}_B$. To fully explain the relationship between the two problems let us first consider a simple example of the assignment problem:

Problem: We have a list of tasks to do and a list of workers, and a matrix that shows the cost of a worker undertaking a certain task. We want to assign a task to each worker so that the final assignment minimises the total cost required to perform all tasks.

$$\begin{array}{c} \\ Worker1 \\ Worker2 \\ Worker3 \end{array} \begin{array}{c} \begin{array}{ccc} TaskA & TaskB & TaskC \end{array} \\ \begin{bmatrix} \boxed{3} & 7 & 4 \\ 15 & 8 & \boxed{7} \\ 3 & \boxed{4} & 9 \end{bmatrix} \end{array}$$

Finding an optimal assignment does not necessary imply that each worker will be assigned to the task for which their cost is minimal. In this example, the optimal assignment is made by all grey boxes and its total cost is 14.

The problem of bipartite graph matching is analogous to the assignment problem, in that the aim is to uniquely assign each vertex of a graph $\mathcal{G}_A$ to a vertex in another graph $\mathcal{G}_B$ so that the assignment minimises the cost (i.e. edit distance) required to transform $\mathcal{G}_A$ into $\mathcal{G}_B$. The minimum total cost is then considered the GED between the graphs. However, there are some problems:

1. There is no clear way to assign costs to the edit operations required to calculate GED.
2. There are three different edit operations possible between any two vertices, and each has a separate cost. Therefore determining a single cost to be used as an element in the cost matrix is not possible.
3. The CG/CFG of different binaries can have different sizes, and the standard form of the Hungarian algorithm only works over equal size graphs.

Fortunately, the problems outlined above have already been addressed by several researchers, most recently Riesen and Bunke [29]. They define the cost matrix as follows:

Definition 3.1 (Cost Matrix for Bipartite Graph Matching)**.** Let $\mathcal{G}_A = (V_A, E_A)$ be the source graph and $\mathcal{G}_B = (V_B, E_B)$ be the target graph, where $V_A = \{a_1, \ldots, a_n\}$ and $V_B = \{b_1, \ldots, b_m\}$. The cost matrix is:

$$\mathbf{C} = \left[\begin{array}{cccc|cccc} c_{1,1} & c_{1,2} & \cdots & c_{1,m} & c_{1,\varepsilon} & \infty & \cdots & \infty \\ c_{2,1} & c_{2,2} & \cdots & c_{2,m} & \infty & c_{2,\varepsilon} & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots & \ldots & \ddots & \ddots & \infty \\ c_{n,1} & c_{n,2} & \cdots & c_{n,m} & \infty & \ldots & \infty & c_{n,\varepsilon} \\ \hline c_{\varepsilon,1} & \infty & \ldots & \infty & 0 & 0 & \cdots & 0 \\ \infty & c_{\varepsilon,2} & \ddots & \vdots & 0 & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \infty & \vdots & \ddots & \ddots & 0 \\ \infty & \cdots & \infty & c_{\varepsilon,m} & 0 & \ldots & 0 & 0 \end{array}\right]$$

where

- the upper-left quarter contains all costs $c_{i,j}$ to substitute a vertex a_i from graph $\mathcal{G}_A$ with a vertex b_j from graph $\mathcal{G}_B$;
- the diagonal of the upper-right quarter contains all costs $c_{i,\varepsilon}$ to delete a vertex a_i from $\mathcal{G}_A$;
- the diagonal of the bottom-left quarter contains all costs $c_{\varepsilon,j}$ to insert a vertex b_j from graph $\mathcal{G}_B$ into graph $\mathcal{G}_A$;
- the bottom-right quarter is zero.

The dimensions of the cost matrix are $(n + m) \times (m + n)$, and it is thus square. Costs can be set in such a way that the algorithm computes a lower bound of the true edit distance (the minimum number of edits) [17], though we define costs using the selectors in the BinDiff algorithm to quantify the degree of distortion required to transform a vertex from $\mathcal{G}_A$ to a vertex in $\mathcal{G}_B$.

4. BinSlayer

BinDiff is able to match CG/CFG nodes with a high degree of accuracy, however, it leaves it to the analyst to deal with unmatched sets of functions from compared executables, and it may be a long and fastidious process to manually sort them and decide which have been deleted, inserted or modified. To address this problem, BinSlayer applies the Hungarian algorithm to the unmatched vertices in $\mathcal{G}_A$ and $\mathcal{G}_B$ with costs assigned as follows:

$$\begin{aligned} c_{i,j} &= |\alpha_i - \alpha'_j| + |\beta_i - \beta'_j| + |\gamma_i - \gamma'_j| \\ c_{i,\epsilon} &= \alpha_i + \beta_i + \gamma_i \\ c_{\epsilon,j} &= \alpha'_j + \beta'_j + \gamma'_j \end{aligned}$$

where α_i, β_i and γ_i are, as previously stated, the number of basic blocks in the function at vertex a_i of $\mathcal{G}_A$, the number of edges in a_i and the number of function calls in a_i. Likewise α'_i, β'_i and γ'_i represent the corresponding values for the vertex b_j of $\mathcal{G}_B$. These costs are designed to reflect whether or not an edit operation represents a strong modification of the graph. The cost $c_{i,j}$ quantifies the structural difference between vertices a_i and b_j, as characterised by differences in their α_i, β_i and γ_i values. Observe that $c_{i,j} < c_{i,\epsilon}$ and $c_{i,j} < c_{\epsilon,j}$ reflecting that deletion and insertion are both stronger than a substitution operation.

BinSlayer also applies the Hungarian algorithm in the same way to the problem of matching basic blocks. This can be achieved merely by replacing α_i, β_i and γ_i with structural measures appropriate for blocks, again using the selector used for matching blocks in BinDiff [9]. In this case, α_i is assigned to the number of blocks on the shortest path from the given block to a function exit; β_i is set to the number of blocks on the shortest path from the entry point into a function to the given block; and γ_i is defined to be the number of function calls within the block. There is no reason why these measures could not be augmented with others [9].

Assigning non-uniform costs to the edit operations of deletion, insertion and substitution leads to a generalised notion of GED [29] where the cost of an edit path is defined to be the sum of the costs of the component edit operations. Moreover this new notion of GED can be normalised, to give a metric between 0 and 1 for the similarity (or, in fact, dissimilarity) of two binaries:

Definition 4.1 (Graph Dissimilarity)**.** The dissimilarity $\delta(\mathcal{G}_A, \mathcal{G}_B)$ between two graphs $\mathcal{G}_A$ and $\mathcal{G}_B$ is a real value on the interval $[0, 1]$, where 0 indicates they are identical whereas a value near 1 implies that they are highly dissimilar:

$$\delta(\mathcal{G}_A, \mathcal{G}_B) = \frac{ged(\mathcal{G}_A, \mathcal{G}_B)}{(\sum_{i=1}^{n} \alpha_i + \beta_i + \gamma_i) + (\sum_{j=1}^{m} \alpha'_j + \beta'_j + \gamma'_j)}$$

where $E_{\mathcal{G}_A}$ and $E_{\mathcal{G}_B}$ are the edges of graphs $\mathcal{G}_A$ and $\mathcal{G}_B$ respectively and $ged(\mathcal{G}_A, \mathcal{G}_B)$ is the GED between graphs $\mathcal{G}_A$ and $\mathcal{G}_B$.

Observe that the numerator never exceeds the denominator, though it can equal it, and hence dissimilarity is guaranteed to give a variable in the interval $[0, 1]$.

Therefore our approach to binary comparison is quite straightforward; first use the BinDiff algorithm to match as many nodes in the executables as possible with a high degree of confidence, and second use the Hungarian algorithm to match the remaining unmatched nodes. Due to the inaccuracies of the Hungarian algorithm, we have written a validation algorithm to attempt to correct the edit path it produces. The validator uses both BinDiff and the Hungarian algorithm to improve the accuracy of the matches. Finally, a normalisation of the GED is calculated as a metric of graph similarity to present to the user.

4.1 Validator

The Hungarian algorithm is used to find an edit path between two binaries that has the minimum overall GED. However, the resultant edit path will typically contain errors, where structurally dissimilar functions have been substituted one for the other because the edit cost for substitution is lower than that of insertion or deletion. The validator, Algorithm 4, attempts to correct these errors by adjusting the costs of edit operations for functions to better reflect structural differences.

Since it is not clear exactly how BinDiff forms its similarity measurement (see section 2.3), we created a simple metric for our implementation of BinDiff to measure similarity, based on the number of matches it found as a fraction of the maximum number of vertices in the compared binaries:

Definition 4.2 (Match Similarity)**.** For a given match $\mathcal{M}$ between two graphs, $\mathcal{G}_A$ and $\mathcal{G}_B$, the match similarity is defined as:

$$\delta_{\mathcal{M}}(\mathcal{G}_A, \mathcal{G}_B) = |\mathcal{M}| / \max(|V_{\mathcal{G}_A}|, |V_{\mathcal{G}_B}|)$$

The validator checks the assignments made for functions (and only functions) against BinDiff, using this similarity measurement to decide how good the matches made by the Hungarian algorithm actually were. It takes a list of vertex substitutions made by the Hungarian algorithm, $\mathcal{P}$, and a threshold t. For each substitution $\{a_i \mapsto b_j\} \in \mathcal{P}$, it compares the structural similarity of the function at a_i with the function at b_j. The CFGs of these functions are denoted $\mathcal{G}_{a_i}$ and $\mathcal{G}_{b_j}$ respectively. Note, the vertices a_i and b_j are actually functions. If a_i is sufficiently similar to b_j, as determined by $\delta_{\mathcal{M}}$ and the similarity threshold t, then the substitution $\{a_i \mapsto b_j\}$ is added to $\mathcal{P}'$. Otherwise, the cost, $c_{i,j}$ of the substitution in the cost matrix is doubled, so as to decrease the likelihood of this substitution being generated again by the Hungarian algorithm. In addition a_i and b_j are added to the sets of unmatched vertices in readiness for the next application of the Hungarian algorithm. When all input substitutions have been considered, the Hungarian algorithm is reapplied to the adjusted cost matrix, yielding a new set of substitutions. This should increase the accuracy of the matching.

Algorithm 4: Validator

```
function validator(P, t);
U1 ← ∅;
U2 ← ∅;
P' ← ∅;
foreach {a_i ↦ b_j} ∈ P do
    (M, S_i, S_j) ← binDiff(G_{a_i}, G_{b_j});
    if δ_M(G_{a_i}, G_{b_j}) ≤ t then
        U1 ← U1 ∪ {a_i};
        U2 ← U2 ∪ {b_j};
        c_{i,j} ← 2c_{i,j};
    else
        P' ← P' ∪ {a_i ↦ b_j};
return P' ∪ hungarian(U1, U2);
```

4.2 Implementation Details

BinSlayer is implemented in a highly configurable C++ program with a Qt based GUI. The user is presented with a clear list of matched and unmatched nodes, including the source of the match (BinDiff/Hungarian algorithm/validator), is able to correct matches manually and to view the CFGs and binary code itself.

As far as we can tell from published sources, most other binary comparison implementations use the industry standard IDA Pro disassembler to recover the programs CFG, however, IDA Pro is known to struggle in this task, especially with obfuscated binaries. We decided instead to use DynInst [1], which excels at CFG reconstruction, able to recover the full CFG for many obfuscated and stripped binaries, and even succeeding at determining indirect jump targets.

5. Experimental Results

To investigate the accuracy of composing BinDiff with the Hungarian algorithm we have compared the number of matched functions against the number found by the vanilla BinDiff algorithm. Our evaluation focused on various versions of the coreutils suite of programs. Coreutils consists of various Unix shell utilities which

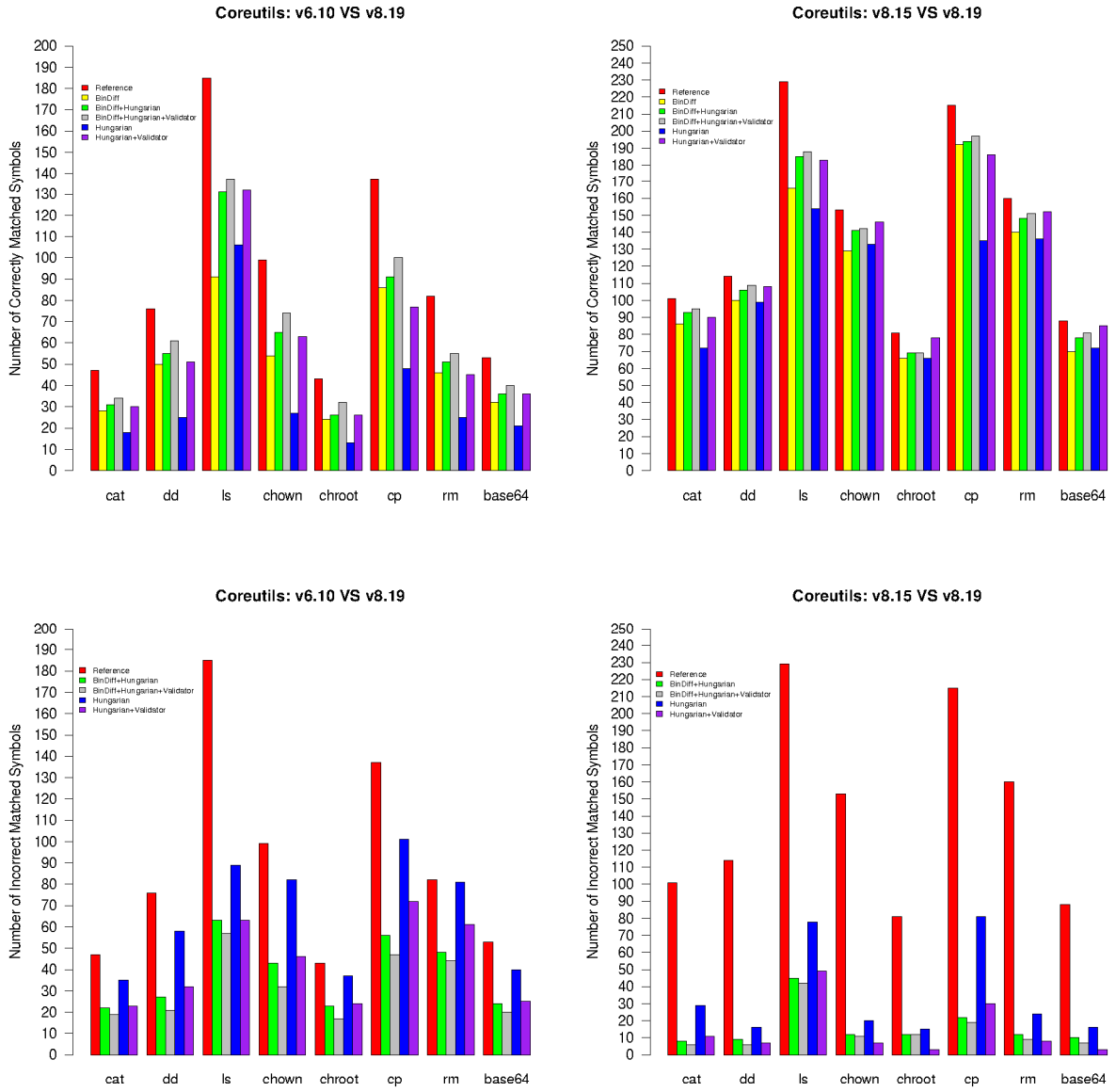

Figure 2. Number of Correctly Matched Functions

are attractive for the purposes of analysis because many versions are readily available. Moreover the relatively small size of the utilities facilitates hand checking. For the purposes of assessing performance we chose to compare eight different versions of the libxml2 dynamically linked library for XML file handling. This binary is sufficiently large to enable timing differences to be observed. We have investigated scalability by measuring the running time of both the BinDiff algorithm and the Hungarian algorithm against the size of the executable. We chose 12 executables with sizes between 200 kilobytes and 22 megabytes.

To assess accuracy and more specifically to measure how many of the matches were actually correct or incorrect, we ran BinDiff using symbols as the only selector. This is guaranteed to match all named functions and therefore provides an upper bound on the best possible matching, hereafter referred to as the reference. Note that we cannot assess correct matching of basic blocks or unnamed functions because there is no automatic way to establish if a match is correct or not. We thus used named functions because names suggest that they share provenance and should be matched. As noted in section 1, we consider matches between different versions

of the same function as correct, even if they are not structurally identical as is required by BinDiff.

The graphs of Figure 2 present the reference as the leftmost column for each of the eight binaries. For the top panes, which show correct matches, the second column gives the number of correct matches found by the vanilla BinDiff algorithm without using the symbolic selector, but using the other standard selectors, namely, child, parent and CRC32. Henceforth these selectors, and only these selectors, are used in subsequent experiments. The following columns of each graph detail the number of correct or incorrect matches found with:

- BinDiff followed by the Hungarian algorithm;
- BinDiff and the Hungarian algorithm followed by the validator;
- the Hungarian algorithm alone; and
- the Hungarian algorithm followed by the validator.

We compared two different versions of the utilities, v6.10 and v8.15, against the latest version, v8.19. The rational behind these choices was to compare versions which were similar, as well versions that were dissimilar, in an attempt to explore how sensitive BinSlayer is to the similarities of the graphs under test. The results clearly show that BinDiff combined with the Hungarian algorithm finds more correct matches than BinDiff alone. Moreover often the number of extra matches is truly significant. Adding the validator to this generally finds further matches, with the exception of a single outlier in the case of version 8.15 of chroot. It is interesting to see that the Hungarian algorithm alone sometimes, though not with any consistency, finds more correct matches than BinDiff. This is surprising because BinDiff is reliable in those matches it makes, though it falls short in the number of matches that it finds. Conversely the Hungarian algorithm finds more matches, but with a higher degree of error. The validator again improves the degree of matching with the Hungarian algorithm. It is interesting to observe that the proportional increase in matches over BinDiff is greater when there is a greater disparity between version numbers. This suggests that the Hungarian algorithm is particularly important when there is a significant difference between the structure of the executables under test. As alluded to earlier, the Hungarian algorithm matches a number of functions incorrectly. This is because it seeks the lowest overall GED, which can mean matching functions with a high substitution cost even when they are structurally dissimilar. However, the validator does reduce the number of incorrect matches to some extent. The overall message is that a combination of BinDiff and the Hungarian algorithm consistently outperforms either algorithm alone in making correct matches, and therefore appears to be a good candidate for binary matching.

In order to measure performance eight successive versions of libxml were selected, all of which exceed 3.6MB in size. All comparisons were made against v2.8.0, the latest version. The leftmost pane of Figure 3 indicates the overall runtime of the comparison. The test environment used to perform the benchmarking is a virtualised (with VMware) 32-bit Ubuntu Linux machine, with an Intel i5 2500k (4 cores at 3.33GHz) CPU and 4GB of RAM. The Linux implementation of the POSIX.1 clock_gettime was used to measure execution time. The BinSlayer executable was compiled with gcc optimisation level 2. Running times were variable but never exceeded 90 seconds, which is encouraging for such large binaries. Not surprisingly, the running time grows with the size of the cost matrix, which suggests that other (less redundant) representations of the cost matrix could improve performance. However, Figure 4 shows that, perhaps surprisingly, there is no clearcorrelation between the size of the executable and the running time of either BinDiff or the Hungarian algorithm. This is presumably because BinDiff is an iterative algorithm whose running time is dependent not only on how many matches can be found, but also on the dependencies between them (the fact that one match can lead another being discovered in a subsequent iteration). Furthermore, the size of the cost matrix and therefore the running time of the Hungarian algorithm depends on how many matches remain undiscovered by BinDiff. The only only correlation we have noticed is that the running time of the Hungarian algorithm increase as the GED between the binaries increases. This is illustrated in the third pane of Figure 4.

Finally, it is encouraging to see that the GED increases as the difference between the version numbers increases. Although it is not monotonically increasing, the centre graph of Figure 3 supports the hypothesis that GED is truly a measure of semantic similarity. We conjecture that v2.7.5 included one-off features that were subsequently scrapped.

6. Related Work

Several approaches have been developed for automatically comparing the structural similarity of executables, and they can broadly be divided into four categories: BinDiff-like, fingerprint/string hashing, bipartite graph matching/GED, and other graph based methods. (Further afield, model checking has been proposed for malware detection [21] though, as far as we know, not for measuring similarity. Somewhat surprisingly, binary matching also arises in the problem of migrating profile information from an older, extensively profiled build to a newer build [32].)

6.1 BinDiff-inspired algorithms

Between 2002 and 2005, Dullien and his colleagues developed a binary comparison algorithm based on graph matching [9, 14–16] that has come to be known as BinDiff. This work has been influential, inspiring a number of researchers, including ourselves. Based on these ideas, Carrera and Erdlyi built an automated system for classifying malware samples into families [5]. Later, Briones and Gomez refined this approach applying matching based on the CRC32 of opcodes, which speeds up the matching and allows the techniques to scale smoothly to large malware databases [4]. They also filtered samples by such characteristics as the number of functions, size, compiler and instruction entropy to avoid comparing graphs of clearly unrelated samples. However, all graph-based approaches, including our own work, ultimately depend on the quality of CG and CFG recovery. CG and CFGs recovery is difficult due to indirect calls, though advances are gradually being made on this problem [3, 20].

6.2 Fingerprint/string hashing Algorithms

Due to CFG recovery issues, other research has focused on an approach called fingerprint hashing [27]. The core idea is to generate a unique fingerprint for each basic block and use the generated signature to match basic blocks as realised in the DarunGrim2 [28] tool. This approach has the advantage of coping better with function splitting and inlining, which impede CFG matching. An analog of graph edit distance, string edit distance, appears in this work [13] and is applied on the block signatures to compare basic blocks. String edit distance has also been applied to measure the similarity of CFG graphs by encoding them as strings [6]. Weighted n-grams have also been applied on basic blocks for the purposes of comparison [31] as have Bloom filters [19].

6.3 Bipartite Graph Matching/GED Algorithms

Graph edit distance is arguably the most reliable graph similarity metric [11], and consequently has been studied for some time as a method by which to compare binaries. However it suffers from a major drawback - its complexity [33]. Although algorithms exist

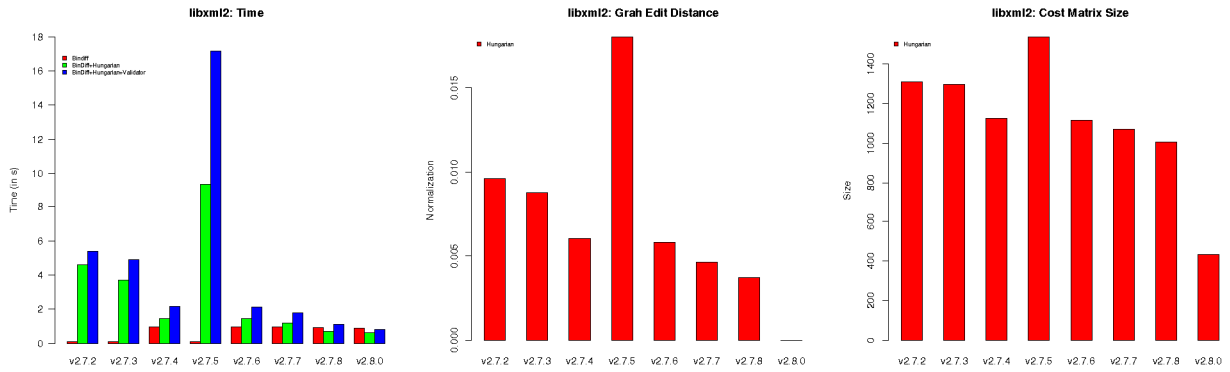

Figure 3. Computation Time, Normalised GED and Cost Matrix Size

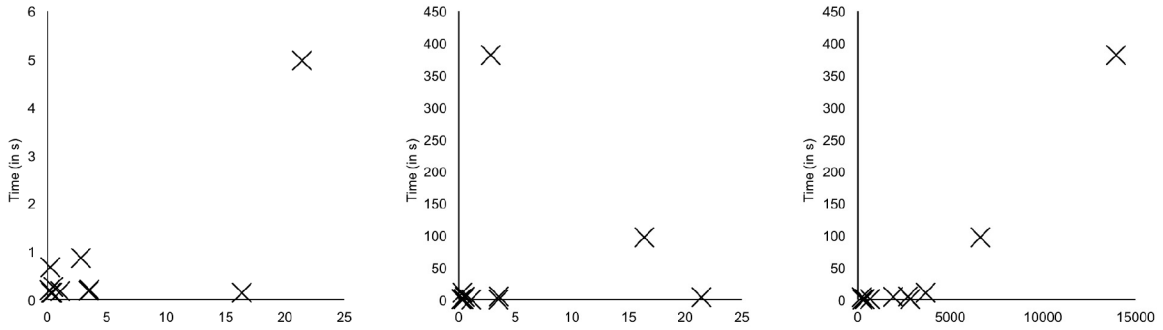

Figure 4. Execution times: BinDiff algorithm vs size (MB), Hungarian algorithm vs size (MB) and Hungarian algorithm vs GED

for GED computation [26], they are not efficient enough for binary comparison.

An important advance was made by Riesen and Bunke who transformed the problem of computing the GED into a weighted bipartite graph-matching problem which can be solved in polynomial time with the Hungarian algorithm [30], which leads an approximation of the GED. They have proved that this approximation results in a sub-optimal edit distance, even though the Hungarian algorithm is optimal, because of the lack of embedded structural information for the two graphs under comparison. This approximation is also proven to be an upper bound of the true edit distance. Further work on the subject leads to performance improvements by taking into account the edge cost as well [29].

A complete system called *SMIT* which performs automatic classification of malware samples into families using the graph edit distance metric has been realised by Hu et al [17]. Their comparison engine, based on the work of Riesen and Bunk, makes some improvements: they take into account the edit cost of adding or deleting edges in the cost matrix each time a node is either substituted, inserted or deleted, and they use a so-called neighbour-biased Hungarian algorithm which tends to match neighbours of already matched functions. One thing to note is that they assigned a unit cost to all edit operations, and so two functions can be matched even if their number of basic blocks is very different.

Malware classification has also been conducted by Kostakis et al in 2011 [23], but their approach is based on simulated annealing. They also proposed a normalisation of the GED based on the total number of functions and function calls in both executables, providing a metric for binary differencing.

6.4 Other Graph Based Methods

BinHunt [10] uses the maximum common subgraphs metric to measure the similarity between two compared executables, using a backtracking algorithm. The comparison at the basic block level is performed by a symbolic execution combined with theorem proving. While this approach leads to good results in terms of detection of code changes the computation time needed is not affordable for larger binaries, in part because it is a dynamic analysis. Whole program traces have also been applied in matching algorithms [25] designed to aid the comprehension of programs which have been obfuscated by aggressive control flow transformations such as control-flow flattening or function inlining and outlining.

7. Discussion

BinSlayer is not a finished tool; it is more an on going research project. The work reported in this paper raises issues in both theory and implementation. An interesting theoretical question is how should the cost matrix be set up so as to best reflect structural properties of the binaries under test? This paper has used costs which reflect the number of basic blocks and edges in the CFG, though this has not been substantiated either by experimentation or systematic analysis. GED itself requires examination since the cur-

rent algorithm tends to prefer substitutions to deletion or insertion, which suggests that a more refined measure might be more appropriate for binary matching. The BinDiff algorithm as implemented by Dullien includes a third tier of matching, that of the instruction level. Adding this layer to BinSlayer should further increase accuracy.

Our emphasis on GED reflects our desire to quantify the degree of structural similarity between two malware objects. This objective naturally led to a matching algorithm that matches structurally similar but not identical functions. It should be noted that this differs in philiosophy from BinDiff, which aspires to only match functions which are structurally identical (in practice two functions are deemed to be structurally identical if their selectors map to the same values). These differing aims mean that BinDiff is well suited to finding differences in two executables that are known to be related, whereas BinSlayer is capable of detecting the level of similarity between two arbitrary executables.

In terms of implementation it would be interesting to see if the regularity in the Hungarian algorithm can be exploited by hardware such as multicore processors or GPUs. The Hungarian algorithm itself warrants close scrutiny as it presents the computational bottleneck in the current implementation. With an eye towards user interaction, it would be interesting to see how one can allow the user to suggest matches or corrections whilst making interaction as unintrusive as possible.

BinSlayer, like other tools based on structural comparison, is susceptible to function in-lining and obfuscation [27]. The rationale for building on top of DynInst is that when its so-called defensive mode becomes available it will be able to deactivate defensive checks, and capture obfuscated control flow such as those based on return address manipulation, exceptions, unpacking and instruction overwriting [22]. DynInst, with its defensive mode features, represents state-of-the-art in control flow reconstruction, and BinSlayer will inherit this functionality. Nevertheless, obfuscation techniques such as opaque predicates [7] and control flow flattening [12] may impede matching, which begs the question of how structural comparison can be generalised to become more obfuscation resistant. When the defensive mode of DynInst is released, we intend to empirically investigate how this affects matching.

8. Conclusion

Motivated by the problems of classifying malware, litigation against copyright infringement, and discovering security vulnerabilities, we have developed a new technique for comparing the structure of binary executables. We have shown that the Hungarian algorithm is not at odds with the BinDiff algorithm, but rather can be fused with it to achieve a more robust matching algorithm that can successfully match more functions than either technique alone.

As a by-product of the construction we compute the Graph Edit Distance (GED), which broadly increases as the difference between version numbers increases. This suggests that GED reflects semantic commonality between binaries. Although the technique constructs a large cost matrix the matching algorithm is scalable enough to compare large executables in an acceptable time frame. Furthermore the approach of using cost functions with embedded structural information regarding the binary appears to be novel in itself.

Acknowledgments

We would like to thank Thomas Dullien for his fascinating seminar on BinDiff at Dagstuhl seminar 12051 [22] that inspired this study. This work was funded, in part, by a Royal Society joint project grant number JP101405.

References

[1] DynInstAPI. http://www.dyninst.org/dyninst.

[2] B. Anckaert, B. De Sutter, and K. De Bosschere. Software Piracy Prevention through Diversity. In *ACM Workshop on Digital Rights Management*, pages 63–71. ACM, 2004.

[3] S. Bardin, P. Herrmann, and F. Védrine. Refinement-Based CFG Reconstruction from Unstructured Programs. In *VMCAI*, volume 6538 of *LNCS*, pages 54–69, 2011.

[4] I. Briones and A. Gomez. Graphs, entropy and grid computing: Automatic comparison of malware. In *Proceedings of Virus Bulletin International Conference 2008*, october 2008.

[5] E. Carrera and G. Erdlyi. Digital genome mapping – advanced binary malware analysis. In *Proceedings of Virus Bulletin International Conference 2005*, october 2004.

[6] S. Cesare and Y. Xiang. Classification of malware using structured control flow. In *Australasian Symposium on Parallel and Distributed Computing*, volume 107, pages 61–70. Australian Computer Society, Inc., 2010.

[7] C. Collberg, C. Thomborson, and D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *POPL*, pages 184–196. ACM, 1998.

[8] B. De Sutter, B. Anckaert, J. Geiregat, D. Chanet, and K. De Bosschere. Instruction set limitation in support of software diversity. In *International Conference on Information Security and Cryptology*, volume 5461, pages 152–165, 2008.

[9] T. Dullien and R. Rolles. Graph-based comparison of executable objects. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications*, 2005.

[10] D. Gao, M. Reiter, and D. Song. Binhunt: Automatically finding semantic differences in binary programs. In *Proceedings of the 10th International Conference on Information and Communications Security*, ICICS '08, pages 238–255. Springer-Verlag, 2008.

[11] X. Gao, B. Xiao, D. Tao, and X. Li. A survey of graph edit distance. *Pattern Analysis and Applications*, 13(1):113–129, Jan. 2010.

[12] J. Ge, S. Chaudhuri, and A. Tyagi. Control flow based obfuscation. In *ACM Workshop on Digital Rights Management*, pages 83–92. ACM, 2005.

[13] M. Gheorghescu. An automated virus classification system. In *Proceedings of Virus Bulletin International Conference*, pages 294–300, October 2005.

[14] Halvar Flake. More fun with graphs. *Black Hat Federal*, 2003.

[15] Halvar Flake. Graph-based binary analysis. *Black Hat Europe*, 2003. `www.blackhat.com/presentations/bh-europe-03/bh-europe-03-halvarflake.pdf`.

[16] Halvar Flake. Structural comparison of executable objects. In *Proceedings of the IEEE Conference on Detection of Intrusions and Malware & Vulnerability Assessment - DIMVA*, pages 161–173, 2004.

[17] X. Hu, T. Chiueh, and K. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 611–620. ACM, 2009.

[18] R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38:325–340, 1987.

[19] B. Kang, H. Kim, T. Kim, H. Kwon, and E. Im. Fast malware family detection method using control flow graphs. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, RACS '11, pages 287–292. ACM, 2011.

[20] J. Kinder and D. Kravchenko. Alternating Control Flow Reconstruction. In *VMCAI*, volume 7148 of *LNCS*, pages 267–282, 2012.

[21] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Proactive Detection of Computer Worms Using Model Checking. *IEEE Trans. Dependable Sec. Comput.*, 7(4):424–438, 2010.

[22] A. King, A. Mycroft, T. W. Reps, and A. Simon. Analysis of Executables: Benefits and Challenges (Dagstuhl Seminar 12051). *Dagstuhl Reports*, 2(1):100–116, 2012.

[23] O. Kostakis, J. Kinable, H. Mahmoudi, and K. Mustonen. Improved call graph comparison using simulated annealing. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1516–1523. ACM, 2011.

[24] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[25] V. Nagarajan, X. Zhang, R. Gupta, M. Madou, B. De Sutter, and K. De Bosschere. Matching Control Flow of Program Versions. In *IEEE International Conference on Software Maintenance*, pages 83–94, 2007.

[26] M. Neuhaus, K. Riesen, and H. Bunke. Fast suboptimal algorithms for the computation of graph edit distance. In *Structural, Syntactic, and Statistical Pattern Recognition*, volume 4109 of *Lecture Notes in Computer Science*, pages 163–172. Springer, 2006.

[27] J. Oh. Fight against 1-day exploits: Diffing Binaries vs Anti-diffing Binaries. In *Black Hat USA*, 2009. `http://www.blackhat.com/presentations/bh-usa-09/OH/BHUSA09-Oh-DiffingBinaries-PAPER.pdf`.

[28] J. Oh. ExploitSpotting: Locating Vulnerabilities Out Of Vendor Patches Automatically. In *Black Hat USA*, 2010. `http://www.darungrim.org/Presentations`.

[29] K. Riesen and H. Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision Computing*, 27(7):950 – 959, 2009.

[30] K. Riesen, M. Neuhaus, and H. Bunke. Bipartite graph matching for computing the edit distance of graphs. In *Graph-Based Representations in Pattern Recognition*, volume 4538 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2007.

[31] A. Walenstein, M. Venable, M. Hayes, C. Thompson, and A. Lakhotia. A.: Exploiting similarity between variants to defeat malware: vilo method for comparing and searching binary programs. In *Proceedings of Black Hat DC 2007. https: //blackhat.com/presentations/bh-dc-07/Walenstein/Paper/bh-dc-07-walenstein-WP.pdf*, 2007.

[32] Z. Wang, K. Pierce, and S. McFarling. BMAT – A Binary Matching Tool for Stale Profile Propagation. *The Journal of Instruction-Level Parallelism*, 2:1–20, 2000.

[33] Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou. Comparing Stars: On Approximating Graph Edit Distance. *Proceedings of the Very Large Databases*, 2(1):25–36, Aug. 2009.