A Method to Evaluate CFG Comparison Algorithms

Patrick P.F. Chan and Christian Collberg
Department of Computer Science
University of Arizona, USA
{pingfaichan,collberg}@gmail.com

***Abstract*—Control-Flow Graph (CFG) similarity is a core technique in many areas, including malware detection and software plagiarism detection. While many algorithms have been proposed in the literature, their relative strengths and weaknesses have not been previously studied. Moreover, it is not even clear *how* to perform such an evaluation. In this paper we therefore propose the first methodology for evaluating CFG similarity algorithms with respect to accuracy and efficiency. At the heart of our methodology is a technique to automatically generate benchmark graphs, CFGs of known edit distances.**

We show the result of applying our methodology to four popular algorithms. Our results show that an algorithm proposed by Hu et al. is most efficient both in terms of running time and accuracy.

I. Introduction

Detecting similarities between two pieces of software has found application in many, often security-related, areas. For example, to detect or classify malware, we need to compare a code sample against a database of known malware instances [1]–[7]; to detect software theft we need to find code-segments from a suspicious program that occur in published code [8]; to automatically grade programming assignments, we need to compare assignment submissions to the code provided by the instructor [9]; to detect code clones we need to find multiple code-segments in a program similar enough to be generalized into a macro or procedure [10]; and to detect changes between different versions of a program we need to find the locations where these versions are the same [11]. While similarity analysis algorithms have been proposed that work on all kinds of program representations (including raw binary instruction, textual assembly code, source code, and abstract syntax trees), a common approach is to compare the Control-Flow Graphs (CFG) of the programs of interest. This seems intuitively attractive since a CFG represents all possible execution paths of a function, and thus conveniently encodes its behavior. We define a Control-Flow Graph as follows:

Definition 1 (Control-Flow Graph)**.** *A Control-Flow Graph is a directed graph* $\mathrm{CFG}(V_{\mathrm{CFG}}, E_{\mathrm{CFG}})$ *that represents the control flow of a function.* V_{CFG} *is the set of nodes (known as* basic blocks*), each having a single entry and a single exit point. An edge in* E_{CFG} *represents a possible flow of control from the end of one block to the beginning of the other.* V_{CFG} *has a unique entry node which dominates all nodes and a unique exit node which post-dominates all nodes. We use* IE_i *and* OE_i *to represent the sets of incoming and outgoing edges of a node* i*, respectively, and* IN_i *and* ON_i *to represent the sets of nodes connected to incoming and outgoing edges of a node* i*, respectively.*

While, in principle, a CFG can be arbitrarily complex, graphs found in real programs tend to have some very specific properties: the outdegree of a node is often upper bounded by two (exceptions include nodes that represent switch statements, exception handling, and computed gotos); CFGs often resemble series-parallel graphs [12]; CFGs are often *reducible* [13]; and basic blocks tend to be small, on the order of 4-7 instructions [14].

In spite of the plethora of CFG-based code similarity algorithms that have been proposed in the literature [1]–[7], [9], [11], [15], [16], there seems to have been little effort into evaluating them relative to each other. In other words, we have no way of knowing under what circumstances we should prefer one algorithm over another, or whether a newly proposed algorithm improves on the state of the art. This is problematic since these algorithms are often at the heart of—and seriously affecting the accuracy and performance of—the applications they support. What is worse, there seem to exist no *methodologies* for carrying out such evaluations. In this paper we propose the first methodology for evaluating software similarity algorithms that are based on CFG topology.

A. CFG Similarity Algorithms

Since general graph similarity testing is expensive and CFGs have some very special properties, we can get significant benefits (both with respect to performance and accuracy) from developing specialized CFG similarity algorithms. These are defined as follows:

Definition 2 (CFG Similarity Algorithm)**.** *A CFG similarity algorithm* $\mathcal{A}$ *is a function* $\Delta_{\mathcal{A}} : (G_1, G_2) \to [0, 1]$ *that computes a real-valued similarity score between*

two graphs G_1 and G_2. A result of 1 indicates that the two graphs are identical. Δ may use a function $\mathrm{NS}_{\mathcal{A}} : (n_1, n_2) \to [0, 1]$ *to compute the* node similarity *between two basic blocks n_1 and n_2.*

Unlike in many related areas, there does not exist a predefined *gold standard* for CFG similarity evaluation. In this paper, we approach this problem by automatically generating such a standard: a set of CFGs with known *edit distances* from a seed CFG. This allows us to evaluate a CFG similarity algorithm by looking at how accurately and efficiently it can recover the relative proximity between randomly chosen CFGs and the seed CFG.

A CFG similarity algorithm may rely on graph topology or basic block contents (instructions), or both, to compute the similarity score. Taking block contents into account can be very helpful. Consider, for example, matching two functions f and g, both of which contain exactly one `CALL` instruction, contained in basic blocks BB_i^f an BB_j^g, respectively. In this case, a similarity algorithm would be wise to directly match BB_i^f to BB_j^g. In the work presented here we concentrate on *topology* only, i.e. in Definition 2 we set $\mathrm{NS}_{\mathcal{A}} \equiv 1$. It is important future work to extend our methodology to content-based CFG similarity algorithms.

B. Contributions and Organization

In this paper we make the following contributions:

- We present a methodology to evaluate and compare CFG similarity algorithms. In particular, this methodology allows us to automatically generate CFGs with known edit distances with respect to a seed CFG. The evaluation system and benchmark graphs are provided under an open source license allowing the research community to evaluate new CFG similarity algorithms and extend our methodology.[1]
- Using our evaluation methodology, we evaluate four CFG similarity algorithms from the literature [3], [9], [15], [17]. We consider CFG topology only. Our results show that the algorithm presented by Hu et al. is superior to the other three algorithms in terms of both accuracy and efficiency.
- We survey existing general graph similarity algorithms and CFG similarity algorithms and discuss their application areas. This provides insights into how common properties of CFGs lead to specially crafted similarity algorithms and why CFG similarity algorithms matter.

The rest of the paper is organized as follows. Section II presents the evaluation methodology for CFG similarity algorithms. Section III describes the details of four CFG similarity algorithms. Section IV presents the evaluation results of the four CFG similarity algorithms. Section V surveys related work. Section VI concludes.

[1]The source code of the evaluation system, the benchmark graphs, and the source code of the re-implemented CFG similarity algorithms can be found at cfgsim.cs.arizona.edu.

II. Evaluation Methodology

Fundamental to our investigation is the concept of *Graph Edit Distance*:

Definition 3 (Graph Edit Distance)**.** *The Graph Edit Distance is a function* $\mathrm{ED} : (G_i, G_j) \to \mathbb{N}$ *that computes the smallest number of* edit operations *needed to transform G_i into G_j. An edit operation is one of*

- *Add a zero-degree node.*
- *Delete a zero-degree node.*
- *Add an edge between two existing nodes.*
- *Delete an existing edge.*

A cost function $\mathrm{cost} : (e) \to \mathbb{N}$ *gives the cost of an edit operation e.*

A *node substitute* operation is sometimes included in the set of edit operations. For the purposes of this investigation we will assume uniform costs, i.e. $\mathrm{cost} \equiv 1$.

Note that two similarity algorithms $\mathcal{A}$ and $\mathcal{B}$ may produce totally different distribution of similarity scores when applied to the same graphs. For this reason, evaluating algorithms by directly comparing their similarity scores is not feasible. We instead use *rankings*.

Specifically, to evaluate a CFG similarity algorithm $\mathcal{A}$ we start by generating CFGs $G_1, G_2, \ldots$ with increasing edit distances with respect to a seed CFG G_0. That is, for each G_i, $\mathrm{ED}(G_0, G_i) = i$. We then rank $G_1, G_2, \ldots$ according to the similarity scores given by $\mathcal{A}$ such that the higher the similarity score $\Delta_{\mathcal{A}}(G_0, G_i)$ the higher G_i is ranked. We get a "goodness score" for algorithm $\mathcal{A}$ by comparing the ranking it produces to the ground truth $\langle G_1, G_2, G_3, \ldots \rangle$, using ranking correlation algorithms such as *sortedness* [18] or *Pearson correlation* [19].

A. Evaluation Algorithm

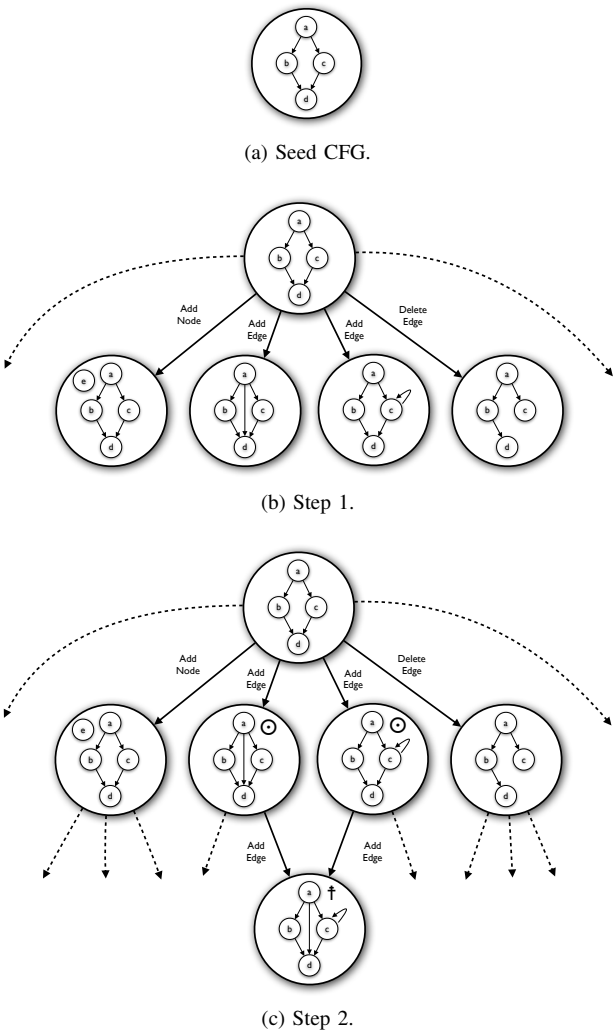
Algorithm 1 gives an overview of the evaluation process for a set of CFG similarity algorithms $\langle \mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k \rangle$. The process starts by generating test cases, where each test case is a tuple of CFGs with known edit distances from a seed CFG:

Definition 4 (Test Case)**.** *Given a seed CFG G_0 and a depth d, a* test case *is a d-tuple $(G_1, G_2, \ldots, G_d)$ of CFGs, such that* $\forall 1 \leq j \leq d, \mathrm{ED}(G_0, G_j) = j$.

The exact generation process is described in the next section. We next compare each CFG in each test case with the seed CFG, rank the results, and compute a goodness score using the Pearson ranking correlation method.

Algorithm 1 Evaluation algorithm. $\langle \mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k \rangle$ is the list of similarity algorithms under study, G_0 is the seed graph, d is a parameter of the test case generation process describing the length of each test case, n is the number of test cases. The output is a ranking of the $\mathcal{A}_i$, best first.

procedure EVALUATE($\langle \mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_k \rangle, G_0, d, n$)
  for $a \leftarrow 1, k$ **do**
    for $t \leftarrow 1, n$ **do**
      for $g \leftarrow 1, d$ **do**
        $\text{scores}_g \leftarrow \Delta_{\mathcal{A}_a}(G_0, \text{TestCases}_{t,g})$
      end for
      $\text{rank} \leftarrow \textbf{rank}(\text{scores})$
      $\text{corr}_{\mathcal{A}_a,t} \leftarrow \textbf{pearson}((G_{t,1}, G_{t,2}, \ldots, G_{t,d}), \text{rank})$
    end for
    $\text{avg}_{\text{corr}_{\mathcal{A}_a}} = \textbf{average}(\text{corr}_{\mathcal{A}_a,1}, \text{corr}_{\mathcal{A}_a,2}, \ldots, \text{corr}_{\mathcal{A}_a,t})$
  end for
  return rank(avg_{corr})
end procedure



(a) Seed CFG.

(b) Step 1.

(c) Step 2.

Fig. 1. Construction of EDG.

B. Benchmark Graph Generation

In order to generate CFGs with known edit distances with respect to a seed CFG G_0, we need to repeatedly apply edit operations to G_0. For example, if we remove two edges from G_0 to form G_2, $\text{ED}(G_0, G_2)$ should be two. Additionally, we need to make sure that the edit distances are the shortest possible; that is, if we can generate the same CFG by applying different different sequences of edit operations, we always choose the sequence with the fewest operations. Finally, since we allow add node and add edge operations, an infinite number of CFGs can be generated, and we therefore need to set an upper bound on edit sequence length.

To support the CFG generation process, we propose an *Edit Distance Graph* (EDG). Intuitively, an EDG represents all the possible CFGs that can be generated from G_0, given a set of possible edit operations. In general, the EDG will be infinite, and in practice we therefore bound it up to some depth d.

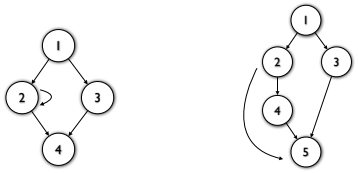
Definition 5 (EDG)**.** *An Edit Distance Graph (EDG) is a directed graph* $\text{EDG}(V_{\text{EDG}}, E_{\text{EDG}})$ *where each vertex* $v \in V_{\text{EDG}}$ *represents a CFG. An edge* $(v_1, v_2) \in E_{\text{EDG}}$ *exists iff the CFG represented by* v_2 *can be obtained from the CFG represented by* v_1 *through a single edit operation. The CFGs represented by the vertices are unique.*

In Definition 3, we only allow adding and deleting zero-degree nodes because we want to ensure that the operations have unit costs. In other words, if we want to delete a node we must first delete its incident edges and if we want to add a node with some incoming or outgoing edges, we must first create the node.

To construct the EDG, a CFG is used as the seed. Every possible edit operation on the seed is used to generate a new graph which may or may not be a CFG (according to Definition 1). For each of the newly generated graphs, an EDG node is created (if it does not already exist) to represent it and an edge from the seed node to the new EDG node is created. This process is repeated for each of the newly generated EDG nodes until a given depth d is reached. This creates levels of EDG nodes. More specifically, an EDG node is at level i if it is created with i edit operation(s).

Figure 1 shows the construction of an example EDG of depth two. The EDG node representing the seed CFG is shown in Figure 1(a). Figure 1(b) shows four graphs generated after step 1 of the EDG construction, i.e. nodes constructed by editing the seed CFG. The four newly generated graphs are the results of performing various kinds of edit operations on the seed CFG. Since the seed CFG has no zero-degree node, no node deletion can be performed. In Figure 1(c) the graph in the node marked by † can be the obtained by modifying either one of the two graphs whose EDG nodes are marked with ⊙. For this reason, only one node is created.

1) Test Case Generation: To generate a test case, we randomly pick a CFG G_i from each level i of the EDG to form a tuple $(G_1, G_2, \ldots, G_d)$. We ignore



(a) CFG A. (b) CFG B.

Fig. 2. Example CFGs used as a running example in Section III.

any graphs which are not legitimate CFGs according to Definition 1.[2] Since G_i has been selected from level i of the EDG, it has been constructed by performing i edit operations, starting from the seed CFG G_0. Therefore, $\mathrm{ED}(G_0, G_i) = i$, and the test case has the property specified in Definition 4.

III. CFG Similarity Algorithms

To exercise the evaluation methodology of Section II we applied it to a selection of four representative CFG similarity algorithms. We give an overview of these algorithms here and present the results of the study in Section IV.

Algorithm $\mathcal{A}_{\text{Kruegel}}$ (Section III-A) is designed to detect polymorphic network worms. The basic idea is to construct fingerprints from k-subgraphs extracted from CFGs found in executables collected from network streams.

Algorithm $\mathcal{A}_{\text{Hu}}$ (Section III-B) is designed to perform fast approximate matching in malware databases, allowing anti-virus companies to determine if a new piece of malware is similar to any known instances. The graph similarity algorithm uses graph edit distance to compare the function-call graphs of a new malware sample to existing examples. Although this algorithm was designed to compare function-call graphs, here we use it to compare CFGs.

Algorithm $\mathcal{A}_{\text{Vujošević-Janičić}}$ (Section III-C) is designed to automatically grade students' programming assignment submissions. The graph similarity algorithm is based on the intuition that two nodes i and j of graphs A and B are considered to be similar if i's neighborhood of nodes can be matched to a neighborhood of j.

Algorithm $\mathcal{A}_{\text{Sokolsky}}$ (Section III-D) is designed to quantify the similarity between viruses of the same family. The algorithm formalizes CFGs as *Labeled Transition Systems* (LTS). The basic idea is that a node s_1 in an LTS G_1 simulates a node t_1 in another LTS G_2 if any outgoing edge of $t_1 \to t_2$ labeled by, say, a can be matched by an edge $s_1 \to s_2$, also labeled by a, in such a way that s_2 simulates t_2.

The two CFGs shown in Figure 2 will be used as a running example to illustrate how these four algorithms compute the topological similarity between two CFGs.

A. $\mathcal{A}_{\text{Kruegel}}$: An Algorithm Based on k-subgraphs Mining

Kruegel et al. [3]'s algorithm extracts subgraphs that have k nodes (k-subgraphs) from CFGs and turns them into feature vectors.

1) Step 1. Generating k-subgraphs.: First, the algorithm performs a depth-first traversal starting from each node N and generates a spanning tree rooted at N such that the outdegree of every node is ≤ 2. From these spanning trees the algorithm generates k-subgraphs recursively by considering all possible allocations of $k-1$ nodes among the left and right subtrees under the node N. The result of this process is a set of planar graphs such that the maximum outdegree is two and the maximum indegree is one.

2) Step 2. Graph Fingerprinting.: Each k-subgraph is next mapped to an integer (a fingerprint) such that two subgraphs have the same fingerprint if and only if they are isomorphic. The k-subgraph is first canonicalized (using Nauty [20]), and then transformed into a $k \times k$ adjacency matrix. The rows of the adjacency matrix are then concatenated to form the fingerprint, a bit vector of length k^2.

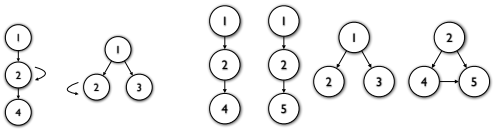
3) Step 3. Graph Coloring.: The original algorithm takes the contents of basic blocks into account by assigning a 14-bit bit vector to each block, and concatenating these vectors onto the rows of the adjacency matrix. A bit is set to 1 iff an instruction belonging to one of 14 instruction classes appears in that basic block. Since we only consider topology in this study, we omit this step.

4) Step 4. Computing Similarity.: For their application, $\mathcal{A}_{\text{Kruegel}}$ does not need to compute an actual similarity score. In our experiments, we compute the similarity between two graphs A and B using the Jaccard index of their corresponding sets of fingerprints [21]:

$$\frac{|\mathrm{fingerprint}(A) \cap \mathrm{fingerprint}(B)|}{|\mathrm{fingerprint}(A) \cup \mathrm{fingerprint}(B)|}$$

5) Running Example.: Figure 3 shows the two sets of subgraphs extracted from CFG A and CFG B in Figure 2 respectively where parameter k is set to 3. Due to the self-loop at Node 2 of CFG A, the subgraphs extracted from CFG A cannot be matched to any subgraphs extracted from CFG B. In other words, the similarity between the two graphs is 0.

[2] During the generation process we might have produced a graph which is disconnected, or has no unique entry or exit node, and which therefore is not a legitimate CFG. However, from such an illegitimate CFG further operations could generate a legitimate one, and it therefore remains part of the EDG. An illegitimate CFG will never be part of test cases, however.



(a) Subgraphs extracted from CFG A.

(b) Subgraphs extracted from CFG B.

Fig. 3. Subgraphs extracted From CFG A and CFG B for Algorithm $\mathcal{A}_{\text{Kruegel}}$ in Section III-A.

B. $\mathcal{A}_{Hu}$: An Algorithm Based on Edit Distance

An algorithm based on graph edit distance is presented by Hu et al. [17]. The algorithm can approximate the minimum number of edit operations needed to transform one graph into another graph. The basic idea is to build a cost matrix that represents the costs of mapping the different nodes in the two graphs. After that, the Hungarian algorithm [22] is used to find a matching between the nodes such that the total cost (edit distance) is minimized.

1) Step 1. Building The Cost Matrix: In order to calculate the edit distance between two graphs, G_1 and G_2, a cost matrix needs to be constructed first. Let V_1 and V_2 denote the sets of vertices for G_1 and G_2 respectively. $|V_2|$ dummy nodes are added to V_1 and $|V_1|$ dummy nodes are added to V_2. The cost matrix is thus a $(|V_1|+|V_2|)\times(|V_1|+|V_2|)$ square matrix (see Figure 4 as an example). It represents the cost of matching each of the nodes in G_1 to any node in G_2. Denote the entries in the cost matrix by a_{ij} where $i, j \in \{1 \dots |V_1|+|V_2|\}$. The cost matrix can be divided into four submatrices. The first submatrix is a $|V_1| \times |V_2|$ matrix at the top left corner. It denotes the cost of matching a real node in G_1 to a real node in $|G_2|$. The values of the entries in it are given by:

$$\begin{aligned} a_{ij} =& \text{relabeling cost} + (|\text{ON}_i| + |\text{ON}_j| - 2\times \\ & |\text{ON}_i \cap \text{ON}_j|)+ \\ & (|\text{IN}_i| + |\text{IN}_j| - 2 \times |\text{IN}_i \cap \text{IN}_j|) \\ & \text{where} \\ & \text{ON}_i \text{ and } \text{IN}_i \text{ denote the outgoing neighbors} \\ & \text{and incoming neighbors of the } i\text{-th node in } G_1 \\ & \text{respectively.} \\ & \text{ON}_j \text{ and } \text{IN}_j \text{ are defined similarly for } G_2. \end{aligned} \tag{1}$$

In the above formula, relabeling cost is the cost of editing the instructions in a node to make them the same as those in the node that is matched to it.

The second submatrix is a $|V_2| \times |V_1|$ matrix at the bottom right corner. It is a zero matrix as it represents matching a dummy node to a dummy node which costs nothing. The third submatrix is a $|V_2| \times |V_2|$ submatrix at the top right corner. It represents the matching of a real node in G_1 to a dummy node which essentially means a node deletion. The values of the entries on the diagonal are given by:

$$\begin{aligned} a_{ij} =& 1 + |\text{OE}_i| + |\text{IE}_i| \\ & \text{where} \\ & \text{OE}_i \text{ and } \text{IE}_i \text{ denote the outgoing edges} \\ & \text{and incoming edges of the } i\text{-th node of } G_1. \end{aligned} \tag{2}$$

The entries not at the diagonal are set to ∞. The forth submatrix at the bottom left corner is defined similarly to the third submatrix.

2) Step 2. Finding Best Match of the Nodes: After the cost matrix has been defined, the aim is to find a matching between the nodes in G_1 and the nodes in G_2 with the lowest cost. The cost of a matching is given by the sum of the costs of the pairs in the matching according to the cost matrix. This is an instance of the *assignment problem*:

Definition 6 (Assignment Problem)**.** *Let A, B be two sets of elements. A matching of them is a set of pairs $M = \{(i,j)|i \in A, j \in B\}$ such that no elements are matched to more than one element of the other set. The enumeration functions $f : \{1,2,...,k\} \to A$ and $g : \{1,2,...,k\} \to B$ are defined such that $M = \{(f(l), g(l))|l = 1,2,...,k\}$ where $k = |M|$. The weight function $w(a,b)$ is a function assigning weights to any pairs of elements (a,b) where $a \in A$ and $b \in B$. The weight of a matching is the sum of the weights of the pairs in it. A matching is optimal if the pairs in it are formed in such a way that the weight of the matching is minimized.*

The Hungarian algorithm [22] finds an optimal solution to the assignment problem in $\mathcal{O}(n^3)$ time. Finally, the edit distance between the two graphs is given by the cost of the matching obtained.

3) Step 3. Computing Similarity.: $\mathcal{A}_{\text{Hu}}$ only computes the edit distance but not the similarity score between the two input graphs. In our experiments, we compute the similarity scores using the formula below. The rationale behind it is that for two completely different graphs G_1 and G_2, to transform G_1 into G_2, we need to delete all the nodes and edges of G_1 and add all the nodes and edges of G_2.

$$\text{sim}(G_1, G_2) = 1 - \frac{\text{edit distance}}{|V_1| + |E_1| + |V_2| + |E_2|}$$

4) Running Example.: The cost matrix built to compute the matching of the nodes between the two CFGs, CFG A and CFG B, in our running example is shown in Figure 4. The four submatrices are enclosed by dotted rectangles.

This cost matrix is built as described in Step 1. As shown in the figure, the top left entry represents the cost of matching node 1 of CFG A to node 1 of CFG B. It is

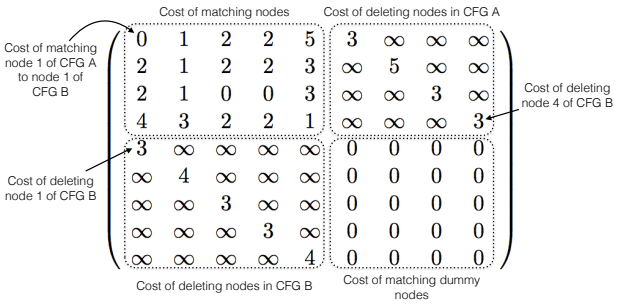


Fig. 4. The cost matrix of match CFG A to CFG B in Algorithm $\mathcal{A}_{\text{Hu}}$ in Section III-B.

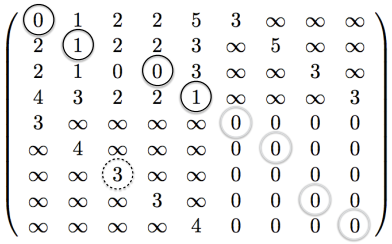


Fig. 5. The entries chosen by Hungarian algorithm on the cost matrix in Algorithm $\mathcal{A}_{\text{Hu}}$ in Section III-B.

given by equation 1. Since we just focus the topology similarity, we ignore the relabeling cost. The first entry of the 4th row represents the cost of deleting Node 1 of CFG B. It is given by equation 2. The Hungarian algorithm is then used to find the best match between the nodes of the two CFGs.

In Figure 5, the entries circled by black circles represent matching between nodes in the two graphs. The entry circled by dotted circle represents the deletion of a node. The entries circled by grey circles represent the matching between dummy nodes and they cost nothing. Adding up the costs of these entries, we get the total cost of these edit operations which is 5.

C. $\mathcal{A}_{\text{Vujošević-Janičić}}$: An Algorithm Based on Neighbor Matching

Vujošević-Janičić et al. [9] proposed a CFG similarity algorithm based on neighbor matching. The basic idea is to iteratively build a similarity matrix between the nodes of the two CFGs, based on the similarity of their neighbors. Given this similarity matrix, using the Hungarian algorithm [22], a matching between the nodes is found such that the resulting similarity score is the highest.

1) Step 1. Topological Similarity.: Given two graphs G_1, G_2, with m and n nodes, respectively, we iteratively create an $m \times n$ similarity matrix. In the k^{th} step, we denote the matrix elements by a_{ij}^k. When $k = 0$, $a_{ij}^0 = 1$. In each subsequent step, the element values are given by:

$$a_{ij}^{k+1} = \frac{s_{\text{in}}^{k+1}(i,j) + s_{\text{out}}^{k+1}(i,j)}{2} \tag{3}$$

where

$$s_{\text{in}}^{k+1}(i,j) = \frac{1}{m_{\text{in}}} \sum_{l=1}^{n_{\text{in}}} a_{f_{ij}^{\text{in}}(l) g_{ij}^{\text{in}}(l)}^{k} \tag{4}$$

$$s_{\text{out}}^{k+1}(i,j) = \frac{1}{m_{\text{out}}} \sum_{l=1}^{n_{\text{out}}} a_{f_{ij}^{\text{out}}(l) g_{ij}^{\text{out}}(l)}^{k} \tag{5}$$

$$\begin{aligned} m_{\text{in}} &= \max(|\text{IE}_i|, |\text{IE}_j|) \\ n_{\text{in}} &= \min(|\text{IE}_i|, |\text{IE}_j|) \\ m_{\text{out}} &= \max(|\text{OE}_i|, |\text{OE}_j|) \\ n_{\text{out}} &= \min(|\text{OE}_i|, |\text{OE}_j|) \end{aligned}$$

f_{ij}^{in} and g_{ij}^{in} are the enumeration functions of the optimal matching of in-neighbors for node i and j with weight function $w(a,b) = a_{ab}^k$, f_{ij}^{out} and g_{ij}^{out} are defined similarly. $\frac{0}{0}$ is defined to be 1 (which occurs when $m_{\text{in}} = n_{\text{in}} = 0$ or $m_{\text{out}} = n_{\text{out}} = 0$). The iterating ends when $\max_{ij} |x_{ij}^k - x_{ij}^{k-1}| < \epsilon$ for some chosen precision ϵ. After the algorithm terminates, the similarity of the two graphs is given by the weight of the optimal matching based on the similarity matrix.

2) Step 2. Node Similarity.: Given a pair of instruction sequences s_1 and s_2, the edit distance between s_1 and s_2 is given by the minimum number of insert, replace, or delete operations to transform s_1 into s_2. The similarity between a pair of basic blocks is given by $\frac{1-d(s_1,s_2)}{M}$ where s_1 and s_2 are the sequences of instructions in the basic blocks, $d(s_1, s_2)$ is the edit distance between s_1 and s_2, and M is the maximum edit distance between s_1 and s_2.

To take into account the node similarity, the topological similarity algorithm described needs to be modified in two ways. Let y_{ij} denote the similarity between node i in G_1 and node j in G_2. First, when $k = 0$, all the entries a_{ij} of the similarity matrix are set to y_{ij}. Second, the values of the entries are now given by:

$$a_{ij}^{k+1} = \sqrt{y_{ij} \cdot \frac{s_{\text{in}}^{k+1}(i,j) + s_{\text{out}}^{k+1}(i,j)}{2}} \tag{6}$$

3) Running Example.: Initially, the similarity matrix is a 4 by 5 matrix with all entries set to 1. At each iteration, the entries in the similarity matrix is updated one by one. In this running example, we are interested in how the entry at the 2nd row and 2nd column is updated in the first iteration. That entry represents the similarity between Node 2 of CFG A and Node 2 of CFG B. The costs of matchings between these two sets of in-neighbors are represented by a 2-by-1 zero matrix.

This cost matrix is calculated based on the initial cost matrix (the one before the first iteration). The first element in this vector represents the cost of matching Node 1 in CFG A to Node 1 of CFG B. The second element of this vector represents the cost of matching Node 2 in CFG A to Node 1 of CFG B. They are both

zero since the initial similarity matrix consists of all ones. The similarity between in-neighbors of Node 2 of CFG A and Node 2 of CFG B is then given by equation 4. The similarity for out-neighbors of Node 2 of CFG A and Node 2 of CFG B is calculated similarly and the value is 1. Therefore, the similarity between Node 2 of CFG A and Node 2 of CFG B is given by equation 3.

After the similarity matrix stabilizes, the calculation of the similarity of the two CFGs is obtained by using the Hungarian algorithm (see Definition 6) based on the latest similarity matrix. The similarity score for the two CFGs in the running example is 0.90.

D. $\mathcal{A}_{Sokolsky}$: An Algorithm Based on Simulation

Sokolsky et al. [15] proposed a simulation-based graph similarity algorithm. It models the control flow graphs using *Labeled Transition Systems* (LTS). Given two CFGs, the basic idea is to recursively match the most similar outgoing nodes starting from the entry nodes and sum up the similarity of the matched nodes and edges. The similarity of two CFGs with entry nodes n_1 and n_2 is given by the following recursive formula:

$$S(n_1, n_2) = \begin{cases} N(n_1, n_2) & \text{, if } \mathrm{OD}_{n_1} = 0 \\ (1-p) \cdot N(n_1, n_2) + & \\ p \cdot max(W_1, W_2) & \text{, otherwise,} \end{cases} \quad (7)$$

where

$$p \in (0, 1)$$

$$W_1 = max_{n_2 \xrightarrow{b} n_2'} L(b, \epsilon) \cdot S(n_1, n_2') \quad (8)$$

$$W_2 = \frac{1}{\mathrm{OD}_{n_1}} \cdot \sum_{n_1 \xrightarrow{a} n_1'} max(max_{n_2 \xrightarrow{b} n_2'} (L(a, b) \cdot S(n_1', n_2')), L(a, \epsilon) \cdot S(n_1', n_2)) \quad (9)$$

In the above equations, N, L are functions that calculate node similarity and label similarity respectively. p is a parameter. OD_{n_1} denotes the outdegree of the node n_1. In our experiments, we chose p to be 0.5 as suggested in the paper to strike a balance between local similarity and "step" similarity.

1) Running Example.: In this running example, we demonstrate how we compute the similarity between CFG A and CFG B. The computation starts with with computing the similarity between node 1 of CFG A and node 1 of CFG B. We skip the recursive calls to make our presentation simple and easy to understand. We pick p to be 0.5. Additionally, we ignore label similarity because the edges of the CFGs are not labeled. We assume all the nodes are the same since we want to focus on structural similarity in this running example.

Since the outdegree of node 1 of CFG A is not zero, according to equation 7, we need to use the expression $(1-p) \cdot N(n_1, n_2) + p \cdot max(W_1, W_2)$ to compute the similarity. In this expression, W_1 and W_2 are given by equation 8 and equation 9 respectively. Figure 6a illustrates the computation of W_1. It involves the computation of the similarity between node 1 of CFG A and the two children of node 1 of CFG B. W_1 is then given by the maximum of these two values (i.e. $max(0.9219, 0.9648)$) and the result is 0.9648. The calculation of W_2 is divided into two parts. The first part is illustrated in Figure 6b. It involves three similarity computations as shown in the figure. The maximum similarity scores among them is 0.9961. The second part is illustrated in Figure 6c. It again involves three similarity computations as shown in the figure. The maximum similarity scores among them is 0.9375. W_2 is then given by $\frac{1}{\mathrm{OD}_{n_1}}$ multiplied by the summation of the maximum similarity scores given by these two parts (i.e. 0.9961 and 0.9375) and the result is $0.5 \times (0.9961 + 0.9375) = 0.9668$. After getting W_1 and W_2, we can calculate the similarity between node 1 of CFG A and node 1 of CFG B which is given by $(1-p) \cdot N(n_1, n_2) + p \cdot max(W_1, W_2) = 0.5 + 0.5 \times max(0.9648, 0.9668) = 0.9834$.

TABLE I. EDG METADATA.

Level	No. of Newly Added CFGs	Time used (sec)	File Size (M)
1	82	3	0.05
2	3342	3	1.02
3	90242	78	23
4	1817182	2134	304
5	29140543	58825	5700

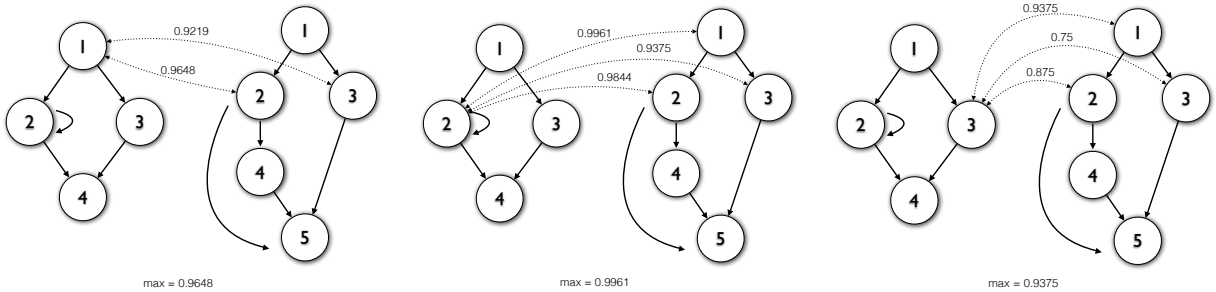
IV. EVALUATION

To ensure uniform implementations we re-coded the four CFG similarity algorithms in Section III in Python. We then evaluated them on a large number of test cases generated using the methodology described in Section II-B. In this section, we give details on the generated benchmarks and present the evaluation results.

A. Test Set

In order to generate the test set we built an EDG of depth five. We used the CFG of the routine `Scramble_ByteSequence` from the automatic decompilation of the Stuxnet virus [23] as the seed CFG G_0. We built the EDG in a breadth-first fashion. Table I shows the number of graphs (including graphs that are not legal CFGs) at each level. The generation process took approximately 17 hours on a machine with two Intel Xeon E5640 CPUs and 94 GB of RAM running Linux 3.2.0. It generated more than 31 million graphs.

From the generated EDG we randomly selected 100 test cases. Each test case is a five-tuple $(G_1, G_2, G_3, G_4, G_5)$ where G_i is selected randomly from level i of the EDG. We then use the test cases to evaluate the four CFG similarity algorithms as described in Algorithm 1.



(a) The calculation of W_1. (b) The calculation of W_2 (part 1). (c) The calculation of W_2 (part 2).

Fig. 6. Running example of $\mathcal{A}_{\text{Sokolsky}}$.

TABLE II. SUMMARY OF FIGURE 7.

Algorithm	Average	Max(Best)	Min(Worst)
$\mathcal{A}_{\text{Hu}}$	0.885	1	-0.3
$\mathcal{A}_{\text{Kruegel}}$	0.486	1	-0.9
$\mathcal{A}_{\text{Vujošević-Janičić}}$	0.805	1	-0.4
$\mathcal{A}_{\text{Sokolsky}}$	0.409	1	-0.8

TABLE III. TOTAL TIME USED BY THE ALGORITHMS TO PROCESS THE TEST SET.

Algorithm	Total time used (sec)	Relative time
$\mathcal{A}_{\text{Hu}}$	1.996	1.1
$\mathcal{A}_{\text{Kruegel}}$	1.815	1.0
$\mathcal{A}_{\text{Vujošević-Janičić}}$	6.179	3.4
$\mathcal{A}_{\text{Sokolsky}}$	2.315	1.28

B. Results

The results of the evaluation are shown in Figure 7. The value given by Pearson correlation [19] is in the range of [-1,1] with 1 indicating total positive correlation, 0 indicating no correlation, and -1 indicating negative correlation. Therefore, the higher the scores, the more accurate is the corresponding algorithm. The results given in this figure are summarized in Table II. The average score given by $\mathcal{A}_{\text{Hu}}$ is the highest among the four algorithms. Also, the min(worst) score given by $\mathcal{A}_{\text{Hu}}$ is the highest. This means that the worst score given by $\mathcal{A}_{\text{Hu}}$ is still better than the worst scores given by the three other algorithms.

Table III shows the total time used by the four algorithms to process the test set. Where $\mathcal{A}_{\text{Kruegel}}$ is the best performing algorithm, the most accurate algorithm, $\mathcal{A}_{\text{Hu}}$, is only 10% slower.

To ensure that the choice of ranking correlation method did not affect our results, we re-ran our evaluation using the sortedness [18] method instead of Pearson, and saw no difference in the final analysis.

From our evaluation results, we can see that the two algorithms that are based on the Hungarian algorithm ($\mathcal{A}_{\text{Hu}}$ and $\mathcal{A}_{\text{Vujošević-Janičić}}$) outperformed the other two in terms of accuracy. The basic idea behind these two algorithms are similar. Both of them try to build a matrix that represents the costs of matching the nodes between the two CFGs and find the optimal matching using the Hungarian algorithm. $\mathcal{A}_{\text{Kruegel}}$ relies on exact matching between extracted k-subgraphs. Since not all possible k-subgraphs are extracted, some subgraphs that match may be missed. On the other hand, since it requires exact match between the subgraphs, the algorithm is sensitive to even small changes to the CFGs. $\mathcal{A}_{\text{Sokolsky}}$ compares the nodes between the two graphs starting at the entry nodes and recursively compares their children. This intrinsically imposes an ordering of comparison and limits the possible matchings between the nodes. Additionally, it requires a parameter p to be carefully chosen. In the original paper, the authors mentioned that they are considering machine learning approaches for determining parameter values but details of that have not been reported.

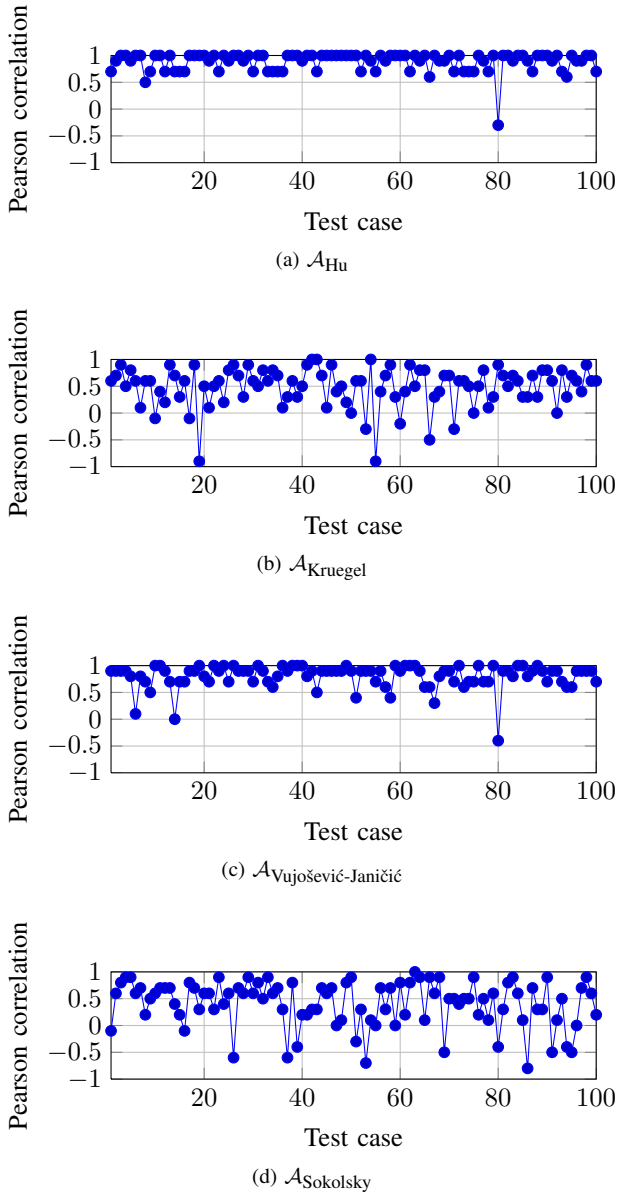
V. RELATED WORK

A. Graph Similarity

Generic graph similarity is a known hard problem (graph edit distance problem is NP-hard [24] and subgraph isomorphism problem is NP-complete [25]). Common approaches include graph edit distance [26], graph isomorphism [27], [28], subgraph matching [29], and iterative comparison [30]. However, The unique properties of CFGs makes it feasible to develop specialized algorithms that perform well for graphs that occur naturally.

B. Plagiarism Detection Algorithm Evaluation

Potthast et al. proposed an evaluation framework for plagiarism detection [31]. In particular, they proposed a methodology to generate artificial plagiarism cases by shuffling, removing, inserting, or replacing words or short phrases at random. However, their method is applied to generate text plagiarism cases only. Moreover, the edit distances between the generated plagiarism cases and the source passage are not known.



(a) $\mathcal{A}_{\text{Hu}}$

(b) $\mathcal{A}_{\text{Kruegel}}$

(c) $\mathcal{A}_{\text{Vujošević-Janičić}}$

(d) $\mathcal{A}_{\text{Sokolsky}}$

Fig. 7. Evaluation result of the four CFG similarity algorithms by Pearson correlation. (Lower is better)

Roy et al. proposed a framework to evaluate code clone detection tools [32]. The first part of their framework generates code clones automatically. The basic idea is to inject mutated code fragments into the code base. To mutate a code fragment, various editing operations are performed. They include changing whitespace, changing comments, changing formatting, renaming identifiers, replacing identifiers with expressions, reordering declarations, etc. Again, the edit distances between the generated code clones and the source code base are not known.

C. CFG Similarity Algorithms

In addittion to the four algorithms in Section III, many more have been proposed in the literature. We briefely review some of them here.

Apiwattanapong et al. [11] proposed a CFG similarity algorithm based on recursive structural matching to identify changes between different versions of a program. CFGs are extended to represent object-oriented constructs and model their behavior. The comparison of two extended CFGs starts with identifying single-entry, single-exit subgraphs (called hammocks) and replacing these with hammock nodes. The matching is then done recursively on nodes and hammock nodes based on labels. The algorithm proposed by Gheorghescu [1] only compares basic blocks of the CFGs. The algorithm is used to classify viruses. Bruschi et al. [4] proposed a malware detection system based on graph isomorphism tests on the CFGs of the malware. For the algorithm proposed by AI-Daoub et al. [33], comparison is based on the alignment of the node sequence and trying to find if the their adjacency matrices are identical after removing the mismatched nodes. Tsai et al. [16] proposed an algorithm to evaluate control-flow obfuscating transformations. It is based on counting the sizes of the common subgraphs found in the two CFGs under comparison. A CFG similarity algorithm based on graph kernel is proposed by Anderson et al. [5] to detect malware. A kernel, $K(x, x)$, is a generalized inner product and can be thought of as a measure of similarity between two objects. Zhao [6] proposed a scheme based on comparing features extracted from CFGs for virus detection purposes. Another scheme to detect polymorphic malware variants was proposed by Cesare et al. [7]. It uses a structuring algorithm similar to the algorithm used in the DCC decompiler [34] to generate the strings for CFGs being compared. After that, q-grams are extracted from the strings to form features. Kinable et al. [2] proposed a graph edit distance based algorithm to classify malware, based on their call graphs.

VI. Conclusion

In this paper, we proposed a methodology to evaluate CFG similarity algorithms. In the preprocessing phase, we generate CFGs with known edit distances with respect to a reference CFG. Then, we use the CFG similarity algorithm under evaluation to measure the similarity between some of these CFGs and the reference CFG. After obtaining such CFG similarity scores, we rank the CFGs according to the scores and see how close the given rank is to the standard rank.

The current EDG construction does not take instructions in the nodes into account when generating new graphs since doing so will tremendously increase the number of possible edit operations on each graph and, hence, the size of the resulting EDG. One direction for future work is to take instructions into account and at the same time control the size of the resulting EDG.

ACKNOWLEDGMENT

We would like to extend our appreciation for the valuable comments from Saumya Debray. This work is supported by the NSF (grant no. 1145913).

REFERENCES

[1] M. Gheorghescu, "An automated virus classification system," in *Virus Bulletin Conference*, 2005, pp. 294–300.

[2] J. Kinable and O. Kostakis, "Malware classification based on call graph clustering," *Journal in Computer Virology*, vol. 7, no. 4, pp. 233–245, 2011.

[3] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *Proceedings of the 8th international conference on Recent Advances in Intrusion Detection*, ser. RAID'05. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 207–226.

[4] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control-flow graph matching," in *Proceedings of the Third international conference on Detection of Intrusions and Malware & Vulnerability Assessment*, ser. DIMVA'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 129–143.

[5] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane, "Graph-based malware detection using dynamic analysis," *J. Comput. Virol.*, vol. 7, no. 4, pp. 247–258, Nov. 2011.

[6] Z. Zhao, "A virus detection scheme based on features of control flow graph," in *Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC), 2011 2nd International Conference on*, 2011, pp. 943–947.

[7] S. Cesare and Y. Xiang, "Malware variant detection using similarity search over sets of control flow graphs," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, 2011, pp. 181–189.

[8] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, ser. Addison-Wesley Software Security Series. Addison-Wesley, Jul. 2009, iSBN 9780321549259, Editor: Gary McGraw.

[9] M. Vujošević-Janičić, M. Nikolić, D. Tošić, and V. Kuncak, "Software verification and graph similarity for automated evaluation of students' assignments," *Inf. Softw. Technol.*, vol. 55, no. 6, pp. 1004–1016, Jun. 2013.

[10] W. S. Evans, C. W. Fraser, , and F. Ma, "Clone detection via structural abstraction," in *14th Working Conference on Reverse Engineering (WCRE)*, 2007, pp. 150–159.

[11] T. Apiwattanapong, A. Orso, and M. J. Harrold, "A differencing algorithm for object-oriented programs," in *Proceedings of the 19th IEEE international conference on Automated software engineering*, ser. ASE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 2–13.

[12] S. Kannan and T. A. Proebsting, "Register allocation in structured programs," in *SODA*, K. L. Clarkson, Ed. ACM/SIAM, 1995, pp. 360–368.

[13] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[14] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[15] O. Sokolsky, S. Kannan, and I. Lee, "Simulation-based graph similarity," in *Proceedings of the 12th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 426–440.

[16] H.-Y. Tsai, Y.-L. Huang, and D. Wagner, "A graph approach to quantitative analysis of control-flow obfuscating transformations," *Information Forensics and Security, IEEE Transactions on*, vol. 4, no. 2, pp. 257–267, 2009.

[17] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 16th ACM conference on Computer and communications security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 611–620.

[18] "Measures of sortedness," http://www.cs.rutgers.edu/~mlittman/courses/Archive/cps130-97/lectures/lect05/node16.html, 1997.

[19] K. Pearson, "Mathematical Contributions to the Theory of Evolution. III. Regression, Heredity, and Panmixia," *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, vol. 187, pp. 253–318, 1896.

[20] B. D. McKay and A. Piperno, "Practical graph isomorphism, {II}," *Journal of Symbolic Computation*, no. 0, pp. –, 2013.

[21] Jaccard, "The distribution of the flora of the alpine zone," in *New Phytologist*, vol. 11, 1912, pp. 37–50.

[22] H. W. Kuhn and B. Yaw, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, pp. 83–97, 1955.

[23] "Partial stuxnet source decompiled with hexrays," https://github.com/Laurelai/decompile-dump/blob/master/output/016169EBEBF1CEC2AAD6C7F0D0EE9026/016169EBEBF1CEC2AAD6C7F0D0EE9026.c, 2010.

[24] Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou, "Comparing stars: On approximating graph edit distance," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 25–36, Aug. 2009.

[25] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, ser. STOC '71. New York, NY, USA: ACM, 1971, pp. 151–158.

[26] X. Gao, B. Xiao, D. Tao, and X. Li, "A survey of graph edit distance," *Pattern Anal. Appl.*, vol. 13, no. 1, pp. 113–129, Jan. 2010.

[27] S. Fortin, "The graph isomorphism problem," Tech. Rep., 1996.

[28] M. Pelillo, "Replicator equations, maximal cliques, and graph isomorphism," 1999.

[29] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *Proc. VLDB Endow.*, vol. 5, no. 9, pp. 788–799, May 2012.

[30] L. A. Zager and G. C. Verghese, "Graph similarity scoring and matching," *Applied Mathematics Letters*, vol. 21, no. 1, pp. 86 – 94, 2008.

[31] M. Potthast, B. Stein, A. Barrón-Cedeño, and P. Rosso, "An evaluation framework for plagiarism detection," in *Proceedings of the 23rd International Conference on Computational Linguistics: Posters*, ser. COLING '10. Stroudsburg, PA, USA: Association for Computational Linguistics, 2010, pp. 997–1005.

[32] C. K. Roy and J. R. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," *Software Testing Verification and Validation Workshop, IEEE International Conference on*, vol. 0, pp. 157–166, 2009.

[33] E. AI-Daoub, A. Al-Shbail, and A. M. Al-Smadi, "Detecting metamorphic viruses by using arbitrary length of control flow graphs and nodes alignment," *UbiCC Journal*, vol. 4, no. 3, pp. 628–633, 2009.

[34] C. Cifuentes, "Reverse compilation techniques," Ph.D. dissertation, Queensland University of Technology, 1994, presented to the School of Computing Science, Queensland University of Technology.