Topology-Aware Hashing for Effective Control Flow Graph Similarity Analysis

Yuping Li[1(✉)], Jiyong Jang[2], and Xinming Ou[3]

[1] Pinterest, San Francisco, USA
yupingli@mail.usf.edu
[2] IBM Research, Yorktown Heights, USA
[3] University of South Florida, Tampa, USA

Abstract. Control Flow Graph (CFG) similarity analysis is an essential technique for a variety of security analysis tasks, including malware detection and malware clustering. Even though various algorithms have been developed, existing CFG similarity analysis methods still suffer from limited efficiency, accuracy, and usability. In this paper, we propose a novel fuzzy hashing scheme called topology-aware hashing (TAH) for effective and efficient CFG similarity analysis. Given the CFGs constructed from program binaries, we extract blended n-gram graphical features of the CFGs, encode the graphical features into numeric vectors (called graph signatures), and then measure the graph similarity by comparing the graph signatures. We further employ a fuzzy hashing technique to convert the numeric graph signatures into smaller fixed-size fuzzy hash signatures for efficient similarity calculation. Our comprehensive evaluation demonstrates that TAH is more effective and efficient compared to existing CFG comparison techniques. To demonstrate the applicability of TAH to real-world security analysis tasks, we develop a binary similarity analysis tool based on TAH, and show that it outperforms existing similarity analysis tools while conducting malware clustering.

Keywords: CFG comparison · Binary similarity · Malware analysis

1 Introduction

Control flow graph (CFG) similarity analysis has played an essential role in malware analysis, *e.g.*, detecting the variants of known malware samples [3,6,11,21,27], evaluating the relationship between different malware families, studying the evolution of different malware families, and triaging large-scale newly collected malicious samples to prioritize the new threats. Research also demonstrated that it is the most fundamental component for effective binary bug search [12], which tried to create signatures from known vulnerable version CFGs and identify the vulnerable version of binaries. Therefore, effective

S. Chen et al. (Eds.): SecureComm 2019, LNICST 304, pp. 278–298, 2019.
https://doi.org/10.1007/978-3-030-37228-6_14

and efficient CFG similarity analysis is much desired for operational security analysis in practice.

Despite the numerous efforts towards effective CFG similarity comparison, we found it is still challenging to apply existing CFG comparison approaches for real-world analysis (*e.g.*, structural based binary similarity analysis). Graph matching is known to be computationally expensive. Even though several approximate graph isomorphism algorithms [21,27,30,44] have been developed and applied to CFG comparison over the past several decades, it is still a time-consuming procedure to compare a large number of CFGs at the same time. For instance, comparing binary `A` with m functions and binary `B` with n functions would result in $m * n$ pairwise CFG comparisons. In addition, we notice that the majority of existing CFG similarity comparison algorithms work with "raw" CFG structures, and rely on inefficient CFG representations for comparison. Last but not least, the evaluation of existing CFG comparison algorithms mainly focus on recognizing the similarity of CFGs. However, performing well with regard to recognizing CFG similarities does not guarantee also doing good in identifying CFG differences. And the later capability is equally important especially if the information of how "similar" of two CFGs are also used in practical applications, *e.g.*, whether the algorithm will generate comparatively low similarity scores if the input CFGs are significantly different.

In this paper, we hypothesize that the original CFG representation is not required to measure the CFG similarity if a CFG can be effectively encoded with certain representative graph features, which could result in a universal and compact graph format and make the overall comparison more efficient at the same time. Therefore, based on the insight that the n-gram concept is applicable to represent CFGs to assess graph similarity, we design a blended n-gram graphical feature based CFG comparison method, called topology-aware hashing (TAH). The n-gram concept has been extensively applied for measuring document similarity where contiguous sequences of n items are extracted from an input stream. We apply it to CFGs in a similar manner, except working with multiple input paths. Extracting n-gram graphical features from CFG structures enables us to effectively encode arbitrary CFGs to the same format. To facilitate the comparison between graph signatures, we further employ a fuzzy hashing technique to convert the numeric graph signatures into smaller fixed-size fuzzy hash signatures. In this way, we achieve high accuracy through the n-gram graphical feature representation, and high efficiency through the compact fuzzy hash comparison. Compared to the state-of-the-art CFG comparison algorithms, our approach achieves the highest accuracy for hierarchical clustering and takes the least amount of time to complete all pairwise comparisons. To demonstrate the effectiveness of the structural comparison approach, we further implement a binary similarity analysis tool based on TAH. When compared with the state-of-the-art binary similarity tools, it achieves the highest accuracy at the F-score of 0.929 for singe-linkage malware clustering tasks.

In summary, we have the following major contributions:

– We propose a blended n-gram graphical feature based CFG comparison method called TAH. It extracts the n-gram graphical features from the topology of CFGs, and measures the similarity of CFGs by comparing the graphical features encoded in fuzzy hash signatures.
– We design a clustering analysis based evaluation framework to comprehensively assess various CFG comparison techniques, and show that TAH is more stable, faster, and generates more accurate results compared to state-of-the-art CFG comparison techniques.
– We design and implement a TAH-based binary similarity analysis tool, and demonstrate that it effectively performs malware clustering tasks with 2865 carefully labeled malware samples in an efficient manner.

2 Related Work

2.1 CFG Similarity Analysis

Control flow graph (CFG) similarity analysis is the core technical component of many existing security analysis systems, and various techniques have been proposed for approximate CFG similarity computation.

1. **Min-cost bipartite graph matching:** Hu *et al.* [21] developed an edit distance based graph isomorphism algorithm by building a cost matrix that represents the costs of mapping the nodes in two graphs, and using the Hungarian algorithm [28] to find an optimal mapping between the nodes such that the total cost (*i.e.*, edit distance) is minimized. Vujošević *et al.* [44] iteratively built a similarity matrix between the nodes of two CFGs based on the similarity of their neighbors, and adopted the Hungarian algorithm to find the matching between the nodes in two graphs such that the resulting similarity score is the highest.
2. **Maximal common subgraph matching:** McGregor [30] designed a backtrack search algorithm to find the maximal common subgraph of two graphs. This idea has been used to design efficient CFG comparison algorithms, and adopted for binary semantic difference analysis [15] and binary code search [12] scenarios. Given the maximal common subgraph output, a graph similarity score was calculated as the maximal number of common subgraph nodes divided by the number of available nodes between two graphs.
3. ***k*-subgraph matching:** Kruegel *et al.* [27] designed an algorithm based on k-subgraph mining. They generated a spanning tree for each node in the graph such that the out-degree of every node was less than or equal to 2, then recursively generated k-subgraphs from the spanning trees by considering all possible allocations of $k-1$ nodes under the root node. Each k-subgraph was then canonicalized and converted into a fingerprint by concatenating the rows of its adjacency matrix.

4. **Simulation-based graph similarity:** Sokolsky *et al.* [42] modeled the control flow graphs using Labeled Transition Systems. Given two CFGs, they recursively matched the most similar outgoing nodes starting from the entry nodes, and summed up the similarity of the matched nodes and edges. The overall similarity of two CFGs was then defined by a recursive formula.
5. **Graph embedding:** Genius [13] was designed to learn high-level feature representations from an attributed CFG (ACFG) and encode the graphs into numerical vectors using a codebook-based graph matching approach. It used 6 block-level attributes (*e.g.*, string constants and the number of instructions) and 2 inter-block level attributes (*e.g.*, the number of offspring and betweenness). Using the same features, Gemini [46] proposed a neural network-based approach to compute the graph embedding for an ACFG, and achieved better accuracy and efficiency. CFG similarity is then measured by comparing the embedded graph representation.

Our TAH algorithm belongs to the graph embedding category, which is also known as topological descriptors [14,20] in other domains. We notice that lots of recent graph embedding techniques [18] were mainly designed to represent the individual graph nodes [1,34] in vector spaces. They were often applied for network (*i.e.*, undirected graph) structure analysis [45] and require additional training processes [5,24,45]. TAH is different from Genius and Gemini in that TAH is basic block content-agnostic, and its graph embedding is always deterministic and requires no separate training process. Graph kernels are widely adopted for comparing graph similarities. However, we note that the majority of the graph kernels are designed and used for analyzing undirected graphs or networks [16,41,43], and require label [41] and weight [25] information. Therefore, they are not directly applicable to analyzing CFGs, which are directed, unlabeled, and unweighted graphs. Nevertheless, we notice that our n-gram concept resonates some of the structural properties used in graph kernel algorithms, such as graphlets [41] (e.g., the subgraphs with k nodes where $k \in 3, 4, 5$).

2.2 Binary Similarity Analysis

We discuss the previous binary similarity analysis methods that are most relevant to our approach.

BitShred [23] was a system designed for large scale malware similarity analysis and clustering, and it extracted n-gram features from the machine code sequences of the executable sections and applied feature hashing to encode the features into a bit-vector. nextGen-hash [29] was a concretized fuzzy hashing approach based on the core ideas developed in BitShred, and achieved more accurate results than other fuzzy hash algorithms; however, its significant fingerprint size made it hard to use in practice. Myles and Collberg [33] proposed to use opcode-level n-grams as software birthmarks and applied it to prove the copyright of software. Opcode level n-gram representation of a binary has also been explored to detect similar malicious code patterns [4,22,32,39]. Alazab *et al.* [2] proposed to detect malware using n-gram features from API call sequences.

SSdeep [26] was a representative fuzzy hashing algorithm that was used to detect homologous files using context triggered piecewise hashes.

CFG based analysis was also used for binary code comparison. BinDiff [11] was a binary comparison tool that assisted vulnerability researchers and engineers to quickly find the differences and similarities using function and basic block level attributes. BinSlayer [3] modeled a binary diffing problem as a bipartite graph matching problem. It assigned a distance metric between the basic block in one function and the basic block in another function that minimized the total distance, and found that graph isomorphism based algorithms were less accurate when the changes between two binaries were large. BinHunt [15] and iBinHunt [31] relied on symbolic execution and a theorem prover to check semantic differences between basic blocks. Although they might yield better accuracy, it was hard to use in practice due to an expensive operation cost. Kruegel *et al.* [27] used the previously mentioned k-subgraph matching algorithm to detect polymorphic worms, which was also based on CFG structural analysis. Cesare and Xiang [6] also extracted fixed-size k-subgraph features and n-gram features from the string representation of a CFG for malware variant detection.

To some extent, our n-gram graphical features are close to k-subgraphs, but they are different concepts. Unlike existing work that tried to find an optimal matching between functions in two binaries, TAH compares the CFGs of the entire binary using the overall n-gram graphical features. Furthermore, n-gram features from a CFG string [6] were still derived in a traditional n-gram usage manner while our proposed n-gram graphical features are directly extracted from CFG structures.

3 Approach Overview

We illustrate the workflow of TAH in Fig. 1. Given two sets of CFGs, we extract the blended n-gram graphical features from the input CFGs and encode them as numeric vectors, called graph signatures. To make it more efficient to use and compare, we subsequently convert the graph signatures into fixed-size bit-vectors, called fuzzy hash outputs. Finally, we compare the corresponding fuzzy hash outputs to calculate the similarity of input CFGs.

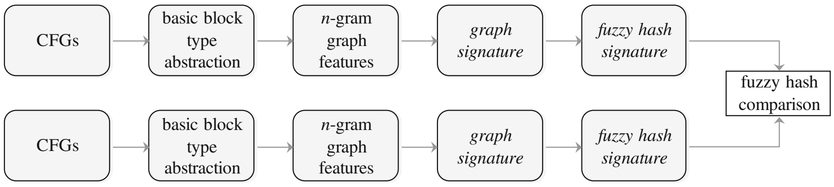


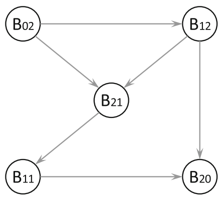
Fig. 1. The workflow of TAH

3.1 Basic Block Type Abstraction

In order to extract representative graphical features, we abstract the basic blocks of CFGs using categorization. The main objective of the abstraction is to categorize the nodes of CFGs into different types, which are then used to denote the "content" of the nodes as used in traditional n-gram application scenarios.

We explore a simple yet effective abstraction of basic block types which captures the topology of a CFG, and demonstrate that such a simple type abstraction approach produces reliable results. In particular, we define the basic block types based on the number of parents (*i.e.*, node in-degree) and the number of children (*i.e.*, node out-degree). To study the CFGs of real-world applications including goodware and malware, we experimentally analyzed a total of 93,470 binaries that were obtained from newly installed Android and Windows operation systems, and malware sharing websites like VirusShare [38].

		outdegree			
		0	1	2	≥ 3
indegree	0	B_{00}	B_{01}	B_{02}	B_{03}
	1	B_{10}	B_{11}	B_{12}	B_{13}
	2	B_{20}	B_{21}	B_{22}	B_{23}
	≥ 3	B_{30}	B_{31}	B_{32}	B_{33}



n	n-gram feature set
1	B_{02}, B_{12}, B_{21}, B_{11}, B_{20}
2	$B_{02}B_{12}$, $B_{02}B_{21}$, $B_{12}B_{21}$, $B_{12}B_{20}$ $B_{21}B_{11}$, $B_{11}B_{20}$
3	$B_{02}B_{12}B_{21}$, $B_{02}B_{21}B_{11}$, $B_{02}B_{12}B_{20}$ $B_{12}B_{21}B_{11}$, $B_{21}B_{11}B_{20}$
4	$B_{02}B_{12}B_{21}B_{11}$, $B_{02}B_{21}B_{11}B_{20}$
5	$B_{02}B_{12}B_{21}B_{11}B_{20}$

Fig. 2. A sample CFG and its blended n-gram features

We noted that the majority (97%) of indegree and outdegree values were between 0 and 3, and mainly focused on in-degree values ranging from 0 to 3, and out-degree values ranging from 0 to 3 when abstracting CFGs. This abstraction approach results in a total of 16 different basic block types as shown in the left table of Fig. 2 where each entry in the table denotes a specific basic block type annotated with its indegree and outdegree values. Basic blocks whose indegree is larger than or equal to 3 are considered as the same type, and basic blocks whose outdegree is larger than or equal to 3 are considered as the same type. During our experiments, we noticed that this approach did not cause significant feature collision as only 1.27% of basic blocks from real-world applications had larger than 3 outdegree and only 3.67% of basic blocks had larger than 3 indegree.

3.2 Blended *n*-gram Graphical Feature Extraction

Similar to the traditional n-gram analysis, we consider a node (*i.e.*, basic block) in a CFG as a single *item*, and define an *n-gram graphical feature* to be the *consecutive n basic blocks* from an input CFG. In order to encapsulate the structural properties, the connectivity among nodes, and the contextual information of the input CFG, we include all k-gram ($k \in [1, n]$) features as the complete graphical feature set. This k-gram model considering all possible sequences from length 1 to n was previously referred to as *blended* n-gram features [36, 40].

Let us take the sample CFG as shown in the middle of Fig. 2 to explain the blended n-gram graphical features in more details. Each basic block is represented in the abstracted basic block type as discussed in Sect. 3.1, e.g., $\texttt{B}_{21}$ with 2 parent nodes and 1 child node. For a given n, we extract all possible blended n-gram graphical features at every node in the CFG. For example, at node $\texttt{B}_{02}$, the 1-gram feature is $\texttt{B}_{02}$ itself, the 2-gram features are $\texttt{B}_{02}\texttt{B}_{12}$ and $\texttt{B}_{02}\texttt{B}_{21}$, and the 3-gram features are $\texttt{B}_{02}\texttt{B}_{12}\texttt{B}_{21}$, $\texttt{B}_{02}\texttt{B}_{12}\texttt{B}_{20}$, and $\texttt{B}_{02}\texttt{B}_{21}\texttt{B}_{11}$. This procedure is called *visiting* node $\texttt{B}_{02}$, and visiting a node reaches descendant nodes at up to $n-1$ levels away. We apply this procedure for all nodes in the CFG and obtain the resulting blended n-gram graphical feature sets. The complete 5-gram graphical features for the sample CFG are presented in the right table of Fig. 2. Note that cycles in a CFG will not be an issue since each node in a CFG is visited only once and the visiting order makes no difference.

Larger n can result in a larger feature space and provide more distinguishing capabilities. On the other hand, a larger feature space also requires more storage and computing resources to extract all n-gram graphical features and perform subsequent operations, e.g., comparisons. We comprehensively assessed the impact of different n-gram sizes, and empirically chose blended 5-gram as the default n-gram size balancing the accuracy and the efficiency. A naïve implementation of the blended 5-gram feature set would result in a feature space of 1,118,480[1]. However, there are certain n-gram graphical features that are invalid by definition. For example, k-gram ($k \geq 2$) features that contain 0 indegree of basic block types (*i.e.*, $\texttt{B}_{00}$, $\texttt{B}_{01}$, $\texttt{B}_{02}$, $\texttt{B}_{03}$) but do not start with them are invalid. Similarly, k-gram ($k \geq 2$) features that contain 0 outdegree of basic block types (*i.e.*, $\texttt{B}_{00}$, $\texttt{B}_{10}$, $\texttt{B}_{20}$, $\texttt{B}_{30}$) but do not end with them are also invalid. After removing such invalid features, the blended 5-gram feature set has smaller 118,096 legitimate entries.

3.3 Graph Signature Generation and Comparison

We generate a *graph signature* by encoding the blended n-gram graphical features into a numeric vector. Each entry in the vector represents a specific feature, and the value of the entry denotes the number of appearances of its corresponding feature in the graph. In this way, both the content and the frequency of features are taken into consideration when building graph signatures. All the graph signatures are in the same size which is determined by the n-gram graphical feature space. We describe a feature entry as a 32-bit unsigned integer type, which we empirically validate that 2^{32} feature space is large enough for practical usage.

We employ the following cosine similarity measure ($\mathbb{C}$) to compute the similarity between two graph signatures.

$$\mathbb{C}(\mathcal{G}_a, \mathcal{G}_b) = \frac{\mathcal{G}_a \cdot \mathcal{G}_b}{|\mathcal{G}_a| \cdot |\mathcal{G}_b|} \tag{1}$$

[1] 16 of 1-gram features, 16^2 of 2-gram features, 16^3 of 3-gram features, 16^4 of 4-gram features, and 16^5 of 5-gram features.

The rationale behind the use of the cosine similarity measure is that it provides an ideal foundation for effective signature size compression, which is often desired for large scale analysis. We describe how we generate a more compact fuzzy hash signature from it in Sect. 3.4.

In practice, the feature counts for different binary programs vary significantly, and the cosine similarity may yield less accurate results as it only assesses orientation of vectors rather than magnitude of vectors. For example, two vectors (1, 2, 3, 4) and (2, 4, 6, 8) yield the cosine similarity score of 1.0, however the magnitude of vectors are significantly different. This is due to the cosine similarity only measuring the angle between the input vectors. We denote the total number of the graphical features contained in a CFG as $\mathbb{N}$, and define the size rectification factor ($\mathbb{R}$) between two CFGs as follows:

$$\mathbb{R}(\mathcal{G}_a, \mathcal{G}_b) = \frac{\mathtt{min}(\mathbb{N}(\mathcal{G}_a), \mathbb{N}(\mathcal{G}_b))}{\mathtt{max}(\mathbb{N}(\mathcal{G}_a), \mathbb{N}(\mathcal{G}_b))} \tag{2}$$

We then compute the final similarity ($\mathbb{S}$) between two graph signatures by multiplying the rectification factor to the cosine similarity, $\mathbb{S}(\mathcal{G}_a, \mathcal{G}_b) = \mathbb{R}(\mathcal{G}_a, \mathcal{G}_b) \times \mathbb{C}(\mathcal{G}_a, \mathcal{G}_b)$. According to the definition, the final similarity of two graph signatures is 1.0 when they are exactly the same, and the size rectification factor is to regulate the similarity only when input graph vectors are significantly different.

3.4 Fuzzy Hash Signature Generation and Comparison

The above graph signature representation provides an effective similarity comparison mechanism; however, the signature size increases exponentially when larger n-gram sizes are used. For example, the size of the graph signature is about 461 KB (*i.e.*, $4B \cdot 118096$) with blended 5-gram graphical features, which is challenging to store and compare at a large scale. To make the technique easier to use and further facilitate the graph signature storage and comparison process, we compress the "raw" graph signature into k-bit vector representation using fuzzy hashing principles.

Specifically, we pre-define a unique seed number and prepare k independent vectors with elements that are selected randomly from Gaussian distribution, and configure each random vector to be the same dimension as the raw graph signatures. We denote the i-th random vector and the raw graph signature as $\mathcal{V}_i$ and $\mathcal{G}$, respectively; and then define the following function $\mathbb{B}$ to compare each random vector against the graph signature.

$$\mathbb{B}_i(\mathcal{V}_i, \mathcal{G}) = \begin{cases} 1 & \text{if } \mathcal{V}_i \cdot \mathcal{G} \geq 0 \\ 0 & \text{if } \mathcal{V}_i \cdot \mathcal{G} < 0 \end{cases} \tag{3}$$

In this way, each random vector is used to create one projection for the raw graph signature based on the dot product between the random vector and the graph signature, and the output of k times of projections becomes a k-bit vector.

The overall random projection procedure is formally known as hyperplane locality sensitive hashing (LSH) [8,10]. Since all the graph signatures are projected into {0, 1} space through the same hashing process, graph signatures with close "locality" will be projected to similar k-bit vectors. We consider the generated k-bit hash value as a *fuzzy hash signature*. Given two fuzzy hash signatures $\mathcal{F}_a$ and $\mathcal{F}_b$, we compute Hamming similarity ($\mathbb{H}$) to measure the similarity between the projected hash outputs as follows:

$$\mathbb{H}(\mathcal{F}_a, \mathcal{F}_b) = 1 - \frac{|\mathcal{F}_a \oplus \mathcal{F}_b|}{k} \tag{4}$$

LSH is commonly used as an efficient technique to conduct the approximate nearest neighbor search in high-dimension objects. Previous research in LSH domain [8,17] showed that (1) cosine similarity (as used for graph signature comparison) is one type of similarity measurement that admits LSH families; and (2) for any similarity function $sim(x, y)$ that admits LSH projection, we can always obtain an LSH family that maps the original objects to {0, 1} space and has the property that the similarity between projected objects (*e.g.*, Hamming similarity) is proved to correspond to the original similarity function at $\frac{1+sim(x,y)}{2}$. Therefore, we leverage $\mathbb{H}(\mathcal{F}_a, \mathcal{F}_b)$ to estimate the original cosine similarity between graph signatures by:

$$\tilde{\mathbb{C}}(\mathcal{G}_a, \mathcal{G}_b) = 2 \times \mathbb{H}(\mathcal{F}_a, \mathcal{F}_b) - 1 \tag{5}$$

Increasing the number of random projection vectors makes the similarity estimation more accurate. To obtain the optimal n-gram size and fingerprint size, we conduct grid search against the same ground truth dataset with a set of potential parameters, and record the optimal clustering result considering both the efficiency and accuracy[2]. According to the empirical process, we set 256 as the optimal k value. The total number of graphical features $\mathbb{N}$ is represented as a 32-bit integer, and the final fuzzy hash output is 288 bits by default. We compute the final fuzzy hash similarity by multiplying the rectification factor to the estimated hash similarity: $\tilde{\mathbb{S}}(\mathcal{G}_a, \mathcal{G}_b) = \mathbb{R}(\mathcal{G}_a, \mathcal{G}_b) \times \tilde{\mathbb{C}}(\mathcal{G}_a, \mathcal{G}_b)$.

4 Evaluation of CFG Similarity Analysis Algorithms

In this section, we evaluate the effectiveness and accuracy of different CFG comparison algorithms. Our evaluation mainly focuses on the capability of the algorithms to differentiate CFG structures (*i.e.*, the topology) *without* relying on basic block content. The auxiliary information provided by basic block content could further improve CFG comparison. For example, the abstraction process discussed in Sect. 3.1 can be extended to incorporate such information.

We compared TAH to representative CFG similarity analysis algorithms discussed in Sect. 2. For min-cost bipartite graph matching algorithm [21,44], k-subgraph matching [27], and simulation-based graph comparison [42], we used

[2] The parameter selection process is not included in the paper due to space limitation.

the implementations provided by Chan [7]. We implemented McGreger's maximum common subgraph matching algorithm [30] and our TAH. The graph embedding based CFG comparison algorithms [13,46] are not evaluated since they rely upon a separate training process and require six types of specific features derived from concrete basic block content. The final outputs of the algorithms were normalized ranging from 0 to 1. To facilitate evaluating arbitrary CFG comparison algorithms, we plan to release our evaluation framework and the corresponding dataset. A new CFG comparison algorithm can be easily evaluated in this framework by providing a plugin that takes two CFGs as input and outputs a normalized similarity score.

4.1 Algorithm Evaluation Strategy

To the best of our knowledge, the only prior work that formally evaluated different CFG similarity algorithms was that of Chan *et al.* [7]. They created a ground truth CFG dataset by applying different levels of edit operations to a seed CFG, and checked if the algorithms could identify a similar level of similarity differences between the generated testing CFGs and the seed CFG. However, we observe that Chan *et al.*'s methodology is problematic from the following perspectives: (1) the ground truth dataset and evaluating strategy were inherently biased towards edit-distance based CFG comparison algorithms; (2) different edit operations (*e.g.*, adding/deleting a node, and adding/deleting an edge) might have different costs. For example, editing a node will not impact any existing edges; on the contrary, editing an edge will affect two nodes. Therefore, the testing CFGs generated by the same number of edit operations may present different similarity levels.

We propose a new evaluation strategy where we employ a CFG comparison algorithm in a *hierarchical agglomerative clustering (HAC) system* as a custom distance function, then use the custom distance function to conduct clustering analysis for the same ground-truth dataset, and use the overall clustering result as a performance indicator of the corresponding CFG comparison algorithm. HAC is a bottom-up version of the hierarchical clustering methods, in which all input items are initially considered as singleton clusters, and then for a specified distance threshold t the algorithm iteratively merges the clusters with the minimum distance as long as the corresponding cluster distance d is less than t. The distance between two clusters is often referred to as "linkage" and the following three linkage criteria are commonly used: *single linkage* considering the cluster distance as the minimum distance between all the entries of two clusters, *average linkage* considering the cluster distance as the average distance between all the entries of two clusters, and *complete linkage* considering the cluster distance as the maximum distance between all the entries of two clusters.

The rationale for evaluating different CFG comparison algorithms through HAC are that: (1) the fundamental component of a HAC system is the similarity measurement between all input items, which can be pre-calculated as a distance matrix using each CFG comparison algorithm; (2) when analyzing the same ground truth dataset, the only parameter that will impact the final clustering result is the distance matrix which is controlled by each CFG comparison algorithm; (3) the clustering analysis procedure assesses the capability to group similar items and separate different items at the same time.

To measure the clustering results, we adopt the measurement of precision and recall. Precision and recall measure two competing criteria of a clustering algorithm: the ability to separate items from different clusters, and the ability to group together items belonging to the same cluster. We consider the intersection point (or the nearest point) between precision and recall to be the optimal clustering output. For simplicity, all the clustering results are subsequently measured with F-score, which is the harmonic mean of the optimal precision and recall.

4.2 Experiment Data Preparation

To create a ground-truth CFG dataset, we compiled the latest version of the Android Open Source Project code and obtained 588 ELF ARM64 binaries. We analyzed the compiled binaries, collected all the function level CFGs that had 20 nodes, and then randomly selected 5 seed CFGs from all available 20-node CFGs. We applied *one* edit operation (*e.g.*, adding a node, deleting a node[3], adding an edge, and deleting an edge) for each seed CFG. Since all the edges and nodes can be edited in single operation and any of the existing two nodes can be added with an additional edge, each of the seed CFG can be used to create about 500 artificial and structurally similar CFGs. In the end, we created a total of 1,934 CFGs from the 5 seed CFGs. We selected CFGs with 20 nodes since they typically provided enough varieties between different CFGs, and the individual CFG comparison did not take too long to complete for all the evaluated algorithms.

4.3 Evaluation Results

To avoid the bias towards a particular linkage strategy, we report the clustering results with three linkage approaches for all CFG comparison algorithms. We summarize the optimal clustering results for different algorithms in Table 1, and present the detailed single-linkage CFG clustering results in Fig. 3. The fuzzy hash signature based CFG comparison approach is labeled as TAH and the graph signature based CFG comparison approach is labeled as TAH$'$. We included TAH$'$ to verify that fuzzy hashing based TAH closely approximated TAH$'$ with little impact on accuracy.

To further dissect the clustering results, we separated all CFG pairs into two categories: same group CFG pairs and different group CFG pairs. Ideally, the distance of the same group CFG pair is expected to be small, and the distance of the different group CFG pair is expected to be large. We present the minimum and the maximum distances of each group in Table 2, and plot the cumulative distance distribution for each algorithm in Fig. 4.

[3] A graph node can be deleted only if it is isolated.

Table 1. Optimal clustering results for different CFG comparison algorithms

Algorithm	Single linkage	Average linkage	Complete linkage	Avg F-score
Hu [21]	0.847	0.872	0.879	0.866
Vujošević [44]	0.749	0.869	0.876	0.831
Sokolsky [42]	0.367	0.456	0.501	0.441
Kruegel [27]	0.530	0.530	0.530	0.530
McGreger [30]	0.597	0.588	0.324	0.503
TAH′	0.816	0.926	1.000	0.914
TAH	0.817	0.926	0.864	0.869

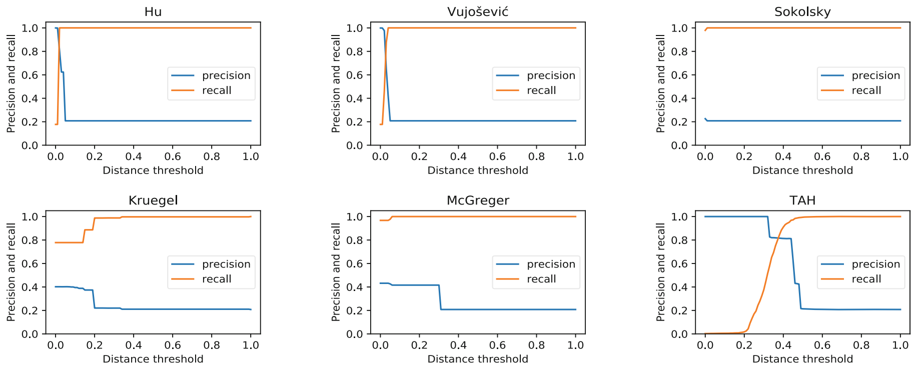
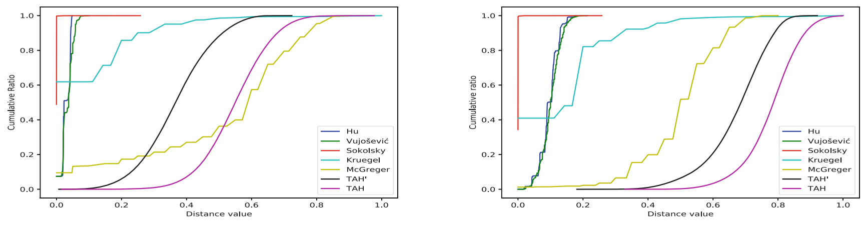


Fig. 3. Single-linkage clustering results for different CFG algorithms

Table 2. Distance ranges for different CFG pairs

Algorithm	Same-group		Diff-group		All pairs		Same-group		Diff-group		All groups	
	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
Hu	0.000	0.049	0.000	0.182	0.000	0.182	0.000	0.032	0.029	0.165	0.000	0.165
Vujošević	0.000	0.100	0.000	0.213	0.000	0.213	0.000	0.067	0.028	0.203	0.000	0.203
Sokolsky	0.000	0.258	0.000	0.258	0.000	0.258	0.000	0.250	0.000	0.250	0.000	0.250
Kruegel	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000	0.000	1.000
McGreger	0.000	0.905	0.000	0.800	0.000	0.905	0.000	0.935	0.000	0.833	0.000	0.935
TAH′	0.007	0.724	0.181	0.921	0.007	0.921	0.000	0.777	0.294	0.917	0.000	0.917
TAH	0.015	0.978	0.328	1.000	0.015	1.000	0.000	0.964	0.398	1.000	0.000	1.000

Combining the distance range information with CFG clustering results, we can see that: (1) the algorithms proposed by Hu, Vujošević, and Sokolsky had very narrow overall distance ranges for all CFG pairs, thus CFGs were quickly merged into one group during the clustering process, which resulted in high recall and low precision for the majority of provided distance thresholds. However, the edit distance based CFG comparison algorithms generated relatively lower distance outputs for same group CFG pairs and higher distance outputs for different



(a) Same-group CFG pairs (b) Different-group CFG pairs

Fig. 4. Cumulative distance distribution for all comparison algorithms

group CFG pairs, therefore they still generated overall good F-score outputs. (2) the algorithms proposed by Kruegel and McGreger provided a broader distance range, but for both same group CFG pairs and different group CFG pairs at the same time. This made precision and recall from different clustering thresholds slowly intersected with each other or never intersected at all, which resulted in overall poor F-score. (3) TAH′ and TAH both provided very good distance ranges. Figure 4 also demonstrates that they clearly separated majority of same group CFG pairs and different group CFG pairs, *e.g.*, when choosing the distance threshold around 0.50 for TAH′ and choosing the distance threshold around 0.70 for TAH. Therefore, they both generated very good F-score outputs.

As mentioned earlier, all the algorithm implementations evaluated in this paper only considered the topology of the CFGs and ignored the content of basic blocks, *i.e.*, the content similarity between all basic block pairs were considered as 1. Therefore, these implementations may not faithfully represent the full capability of the original designs, and the evaluation results presented in this paper only reflected the algorithms' capability of measuring the similarity of CFG structures, without considering the basic block content. Even though our evaluation framework supports experiments with real-world CFGs and the TAH can be extended to incorporate boolean node attributes[4], it is practically challenging to conduct large scale experiments with real-world CFGs as it is nontrivial to prepare a large-scale ground truth dataset using real-world CFGs that present *controlled* and *known* similarity levels. Firstly, the source code level similarities are commonly not proportionally preserved after the complicated compilation procedure, *i.e.*, it is difficult to control the granularity of CFG similarities through source code updates. Secondly, when analyzing large amount malicious real-world binary samples that were labeled as the same malware family label, we noticed that samples often either shared the exact same functions or had significantly changed functions, while few CFGs were closely similar.

Our evaluation strategy highlighted the strengths and the weaknesses of existing CFG comparison algorithms when comparing graph topologies. In summary, TAH′ showed the best separation capabilities between similar CFG pairs and

[4] *E.g.,* adding 1 bit to record whether the node contains string constants during basic block type abstraction as described in Sect. 3.1.

different CFG pairs, and TAH was a faster approximate method yet still quite comparable to the accuracy of TAH$'$, whereas existing CFG comparison algorithms either had limited distance output ranges or had almost the same distance ranges between same group CFG pairs and different group CFG pairs. This demonstrates that TAH$'$ and TAH are more reliable and have more balanced capability to recognize similar CFGs and identify different CFGs at the same time. High F-scores of TAH$'$ and TAH are achieved mainly because the differences between n-gram graphical features are always proportional to the structural differences of CFGs. For example, the same group CFGs will have more similar n-gram graphical features, and the different group CFGs will have more different graphical features, which subsequently leads to the desired distinguishing capabilities.

4.4 Overall Performance

For each algorithm, we measured the time taken (in seconds) to finish the similarity calculation for all CFG pairs. Since the hierarchical clustering algorithm has n^2 complexity, the prepared 1,934 CFGs would result in 1,869,211 pairwise CFG comparisons. Note that all the algorithm implementations took two CFGs as input and produced a similarity score, which means the graph signatures for TAH$'$ and fuzzy hash signatures for TAH were generated 1,869,211 times during the evaluation. And since TAH fuzzy hash outputs are generated from TAH$'$ graph signatures, this made the TAH approach dramatically slower than other approaches. However, if we cache the graph embedding process, *e.g.*, precalculating the graph signature for TAH$'$ and the fuzzy hash output for TAH for each CFG then directly loading the generated signatures/hashes within each pairwise CFG comparison routine, TAH$'$ only took 125.5 s to finish the graph signature generation and all pairwise comparison, and TAH only took 23.8 s to finish the fuzzy hash generation and all pairwise comparison. This was much more efficient than other existing approaches. For the same dataset, Hu's algorithm took 755.6 s, Vujošević's algorithm took 1788.0 s, Sokolsky's algorithm took 483.1 s, Kruegel's algorithm took 321.8 s, and McGreger's algorithm took 2542.1s to finish. Note that for the same CFG input, the fuzzy hash output is the same if we apply the algorithm and generate it again, so it makes sense to only generate it once and caching the intermediate and compact CFG representation when used in real-world. However, this won't be applicable for the existing algorithms since there is no intermediate CFG representation for them and they need to repeated compare the "raw" CFG inputs. This evaluation indicates that TAH is particularly suitable for large scale dataset analysis.

5 Evaluation of Binary Similarity Analysis Tools

Based on TAH, we implemented a binary similarity analysis tool[5] to evaluate the effectiveness of the structural comparison approach. We assessed the effectiveness of TAH to conduct binary similarity analysis, and compared it with the

[5] For simplicity, we also refer our binary similarity analysis tool as TAH.

following existing binary similarity comparison solutions: SSdeep v2.14.1 [26] and BinDiff v4.3.0 [11]. Other previously proposed binary similarity analysis tools were not evaluated because they were neither maintained [3] (*i.e.*, not working with a majority of the collected binaries) nor publicly available [15]. In order to exclude the potential impact of the different disassemblers on the CFG construction accuracy, TAH was implemented as an IDA Pro plugin. The current implementation of TAH used IDA Pro v6.8 to process the target binaries and constructed the corresponding CFGs. The same version of IDA Pro was also used by BinDiff.

We embedded all the binary similarity analysis tools (*e.g.*, TAH, BinDiff, and SSdeep) into the hierarchical clustering system, and used the clustering outputs to evaluate the similarity measurement accuracy of the tools. We prepared the ground truth dataset by collecting desktop malware samples labeled with malware family names, and considered that the binaries with the same family name were more similar to each other than binaries from different families were. Because the labeled malware datasets used in previous research were either discontinued or only contained a list of file hashes, we prepared our own labeled ground truth malware dataset. In the end, we collected 2,865 recent desktop malware samples from VirusShare [38], and every sample was consistently labeled by at least 25 antivirus products listed on VirusTotal [9]. The resulting ground truth malware dataset containing 8 different malware families is shown in Table 3.

Table 3. Ground truth malware dataset

Malware family	Size	Malware family	Size	Malware family	Size
InstalleRex	1115	OutBrowse	615	MultiPlug	384
DomaIQ	184	LoadMoney	173	Linkular	164
InstallCore	127	DownloadAdmin	103		

We also measured precision and recall to evaluate the clustering outputs. For all tools, we summarized the optimal clustering results with regard to different clustering strategies in Table 4, and depicted the overall clustering results with different binary similarity analysis tools in Fig. 5. We can see that TAH generated the highest F-score of 0.929 for single linkage clustering analysis. BinDiff produced similar results with F-score of 0.883, while SSdeep only achieved overall F-score of 0.690. Because SSdeep operated at the binary stream level, it was not able to identify a significant number of semantically similar binaries. BinDiff and TAH both operated at the CFG level, and effectively identified a larger number of similar binaries. Different similarity calculation logic of the tools led to the F-score differences between BinDiff and TAH.

We also evaluated the time taken to conduct the experiments with each tool. Since BinDiff did not have an intermediate "signature" representation for binary CFGs, we only calculated the time it took to finish all the pairwise computations

Table 4. Optimal clustering results for different binary similarity analysis tools

Tool	Single linkage	Average linkage	Complete linkage	Time taken
SSdeep	0.690	0.690	0.689	2.7 min
BinDiff	0.883	0.883	0.883	166.4 min
TAH	0.929	0.903	0.909	0.9 min

by converting all the input binaries into `BinExport` format. The time to conduct pairwise comparisons with each tool is shown in the last column of Table 4. We can see that TAH outperformed SSdeep and BinDiff in terms of efficiency, and BinDiff was dramatically slower than other tools. The main reason for the higher efficiency of TAH was that a fuzzy hash signature was essentially a bit-vector which was more CPU friendly than SSdeep hash representation was.

It is worth to mention that various previous malware clustering systems already demonstrated very promising results. For example, using different datasets, Malheur [37] reported F-score of 0.95, BitShred [23] showed F-score of 0.932, and FIRMA [35] claimed F-score of 0.988. As a reference, we chose to conduct experiments with BitShred, which was a state-of-the-art malware clustering tool using static and dynamic binary features. Since BitShred only adopted the single-linkage clustering strategy, we plot the single-linkage clustering results for all the tools in Fig. 5. From the graph, we can see that BitShred reached the optimal F-score of 0.885. Overall, TAH generated the best clustering results (*e.g.*, F-score of 0.929) with single-linkage clustering. Figure 5 also shows that recall for TAH and BitShred at the distance threshold of 0 are above

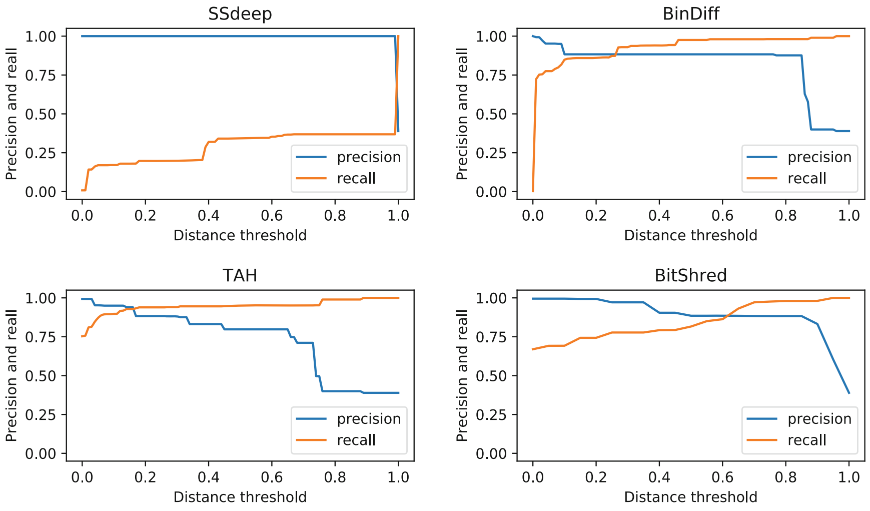


Fig. 5. Single-linkage clustering results of different similarity analysis tools

0.650, while recall of SSdeep and BinDiff at this threshold was 0. This is because TAH and BitShred correctly identified a significant number of binary pairs as similarity of 1.000, while both SSdeep and BinDiff could not identify any of such binary pairs. We further notice that the majority of precision values (*e.g.*, with distance thresholds of [0.000, 0.995]) for SSdeep were 1.000, which means all the binary pairs that were identified as similar (*i.e.*, similarity score larger than 0.005) were indeed similar. However, at the same time, the corresponding recall values for SSdeep were less than 0.365, which indicates that SSdeep failed to recognize a significant number of binaries that were known to be similar regardless of the distance thresholds. This is in line with our practical usage experience with SSdeep.

6 Limitation

6.1 Feature and Signature Collision

We consider the potential attack scenario by generating similar graph features for different CFGs. According to the design in Sect. 3.1, feature collision happens when: (1) nodes have indegree or outdegree larger than 3; (2) nodes with different content are in the same topology. The first type of collision is rare in real-world binary programs (*e.g.*, 3.67% of nodes in our real-world datasets), and the second type of collision is largely alleviated by recording the node context (*i.e.*, n-gram features) and graphical feature counts. n-gram feature extraction allows the resulting feature differences to be proportional to the structural differences. The proposed CFG similarity analysis algorithm TAH compares CFG graph signatures by measuring the overall cosine similarity of graph signatures and the relative CFG sizes. Since the graph signature is mainly a summary of all features contained in a CFG, it is theoretically possible for binaries with different CFGs to generate similar graph signatures. However, random signature collision for overall binary CFGs is rare in practice. To further reduce the possibility of collision, we can increase the graphical feature space by incorporating certain basic block content information into the type abstraction process, such as the presence of a string or numeric constant, and the number of instructions.

6.2 Obfuscation and Evasion Techniques

It is well-known that malware samples are often packed in recent years to evade signature-based malware analysis tools [19]. Even worse, malware authors can apply multiple layers of packing, or employ advanced packers that dynamically decrypt original code on-the-fly or interpret instructions in a virtualized environment. We believe the TAH-based binary similarity analysis is still useful in practice because of the following reasons. (1) Lots of real-world binaries are still unpacked, especially for adware or PUP programs. TAH can be used to quickly filter out similar binaries that have been processed before, or used for triaging a large number of unprocessed binaries (even packed ones) by grouping similar instances together. For this purpose, traditional cryptographic hash and existing similarity analysis solutions are less effective. (2) CFG level analysis makes it possible to provide a binary similarity analysis solution that has a good balance between accuracy and efficiency. For example, dynamic analysis based approach can defeat obfuscation, but is almost impossible to efficiently analyze a large scale of dataset. (3) Comparing to low-level binary sequences, it will be more difficult to add randomness to the CFG structure for the packed binaries. For instance, the dead code can be removed during the CFG construction procedure. Thus the packed binaries would often share certain deobfuscation routines, which can be viewed as the signature of the packers.

7 Conclusion

In this paper, we proposed an effective CFG comparison algorithm TAH, which compares CFGs using n-gram graphical features. In order to compare with existing CFG comparison solutions, we designed a clustering analysis based evaluation framework, and systematically showed that TAH was more accurate and efficient compared to state-of-the-art CFG comparison techniques. Based on TAH, we also developed a graphical comparison based fuzzy hash tool for binary similarity analysis. We empirically demonstrated that TAH outperformed existing binary similarity analysis tools while conducting malware clustering analysis.

Acknowledgment. This research was partially supported by the U.S. National Science Foundation under grant No. 1622402 and 1717862. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Ahmed, A., Shervashidze, N., Narayanamurthy, S., Josifovski, V., Smola, A.J.: Distributed large-scale natural graph factorization. In: Proceedings of the 22nd International Conference on World Wide Web, pp. 37–48. ACM (2013)
2. Alazab, M., Venkataraman, S., Watters, P.: Towards understanding malware behaviour by the extraction of API calls. In: 2010 Second Cybercrime and Trustworthy Computing Workshop (CTC), pp. 52–59. IEEE (2010)
3. Bourquin, M., King, A., Robbins, E.: BinSlayer: accurate comparison of binary executables. In: Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, Rome, Italy, p. 4. ACM (2013)
4. Canfora, G., De Lorenzo, A., Medvet, E., Mercaldo, F., Visaggio, C.A.: Effectiveness of Opcode ngrams for detection of multi family android malware. In: 2015 10th International Conference on Availability, Reliability and Security (ARES), Toulouse, France, pp. 333–340. IEEE, IEEE Computer Society (2015)
5. Cao, S., Lu, W., Xu, Q.: Deep neural networks for learning graph representations. In: AAAI, pp. 1145–1152 (2016)
6. Cesare, S., Xiang, Y.: Malware variant detection using similarity search over sets of control flow graphs. In: 2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications, pp. 181–189. IEEE, IEEE Computer Society (2011)

7. Chan, P.P., Collberg, C.: A method to evaluate CFG comparison algorithms. In: 2014 14th International Conference on Quality Software (QSIC), Washington, DC, USA, pp. 95–104. IEEE, IEEE Computer Society (2014)
8. Charikar, M.S.: Similarity estimation techniques from rounding algorithms. In: Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing, pp. 380–388. ACM (2002)
9. Virustotal (2018). https://www.virustotal.com
10. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: Proceedings of the Twentieth Annual Symposium on Computational Geometry, pp. 253–262. ACM (2004)
11. Dullien, T., Rolles, R.: Graph-based comparison of executable objects (English version). SSTIC **5**, 1–3 (2005)
12. Eschweiler, S., Yakdan, K., Gerhards-Padilla, E.: discovRE: efficient cross-architecture identification of bugs in binary code. In: Proceedings of the 2016 Network and Distributed System Security (NDSS) Symposium, San Diego, CA, USA. The Internet Society (2016)
13. Feng, Q., Zhou, R., Xu, C., Cheng, Y., Testa, B., Yin, H.: Scalable graph-based bug search for firmware images. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, pp. 480–491. ACM (2016)
14. Galvez, J., Garcia, R., Salabert, M., Soler, R.: Charge indexes. New topological descriptors. J. Chem. Inf. Comput. Sci. **34**(3), 520–525 (1994)
15. Gao, D., Reiter, M.K., Song, D.: BinHunt: automatically finding semantic differences in binary programs. In: Chen, L., Ryan, M.D., Wang, G. (eds.) ICICS 2008. LNCS, vol. 5308, pp. 238–255. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88625-9_16
16. Gärtner, T.: A survey of kernels for structured data. ACM SIGKDD Explor. Newsl. **5**(1), 49–58 (2003)
17. Gionis, A., Gunopulos, D., Koudas, N.: Efficient and tumble similar set retrieval. ACM SIGMOD Rec. **30**(2), 247–258 (2001)
18. Goyal, P., Ferrara, E.: Graph embedding techniques, applications, and performance: a survey. Knowl. Based Syst. **151**, 78–94 (2018)
19. Guo, F., Ferrie, P., Chiueh, T.: A study of the packer problem and its solutions. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 98–115. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87403-4_6
20. Hert, J., et al.: Comparison of topological descriptors for similarity-based virtual screening using multiple bioactive reference structures. Org. Biomol. Chem. **2**(22), 3256–3266 (2004)
21. Hu, X., Chiueh, T.-C., Shin, K.G.: Large-scale malware indexing using function-call graphs. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, Chicago, Illinois, USA, pp. 611–620. ACM (2009)
22. Hu, X., Shin, K.G.: DUET: integration of dynamic and static analyses for malware clustering with cluster ensembles. In: Proceedings of the 29th Annual Computer Security Applications Conference, New Orleans, Louisiana, USA, pp. 79–88. ACM (2013)
23. Jang, J., Brumley, D., Venkataraman, S.: BitShred: feature hashing malware for scalable triage and semantic analysis. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, Chicago, Illinois, USA, pp. 309–320. ACM (2011)
24. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 (2016)

25. Kondor, R., Pan, H.: The multiscale Laplacian graph kernel. In: Advances in Neural Information Processing Systems, pp. 2990–2998 (2016)
26. Kornblum, J.: Identifying almost identical files using context triggered piecewise hashing. Digit. Investig. **3**, 91–97 (2006)
27. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic worm detection using structural information of executables. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 207–226. Springer, Heidelberg (2006). https://doi.org/10.1007/11663812_11
28. Kuhn, H.W.: The Hungarian method for the assignment problem. Nav. Res. Logist. (NRL) **2**(1–2), 83–97 (1955)
29. Li, Y., et al.: Experimental study of fuzzy hashing in malware clustering analysis. In: 8th Workshop on Cyber Security Experimentation and Test, CSET 2015, vol. 5, p. 52. USENIX Association (2015)
30. McGregor, J.J.: Backtrack search algorithms and the maximal common subgraph problem. Softw. Pract. Exp. **12**(1), 23–34 (1982)
31. Ming, J., Pan, M., Gao, D.: iBinHunt: binary hunting with inter-procedural control flow. In: Kwon, T., Lee, M.-K., Kwon, D. (eds.) ICISC 2012. LNCS, vol. 7839, pp. 92–109. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37682-5_8
32. Moskovitch, R., et al.: Unknown malcode detection using OPCODE representation. In: Ortiz-Arroyo, D., Larsen, H.L., Zeng, D.D., Hicks, D., Wagner, G. (eds.) EuroIsI 2008. LNCS, vol. 5376, pp. 204–215. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89900-6_21
33. Myles, G., Collberg, C.: K-gram based software birthmarks. In: Proceedings of the 2005 ACM Symposium on Applied Computing, Santa Fe, New Mexico, USA, pp. 314–318. ACM (2005)
34. Ou, M., Cui, P., Pei, J., Zhang, Z., Zhu, W.: Asymmetric transitivity preserving graph embedding. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1105–1114. ACM (2016)
35. Rafique, M.Z., Caballero, J.: FIRMA: malware clustering and network signature generation with mixed network behaviors. In: Stolfo, S.J., Stavrou, A., Wright, C.V. (eds.) RAID 2013. LNCS, vol. 8145, pp. 144–163. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41284-4_8
36. Rieck, K., Laskov, P.: Linear-time computation of similarity measures for sequential data. J. Mach. Learn. Res. **9**, 23–48 (2008)
37. Rieck, K., Trinius, P., Willems, C., Holz, T.: Automatic analysis of malware behavior using machine learning. J. Comput. Secur. **19**(4), 639–668 (2011)
38. Roberts, J.: VirusShare.com (2015). http://virusshare.com/
39. Shabtai, A., Moskovitch, R., Elovici, Y., Glezer, C.: Detection of malicious code by applying machine learning classifiers on static features: a state-of-the-art survey. Inf. Secur. Tech. Rep. **14**(1), 16–29 (2009)
40. Shawe-Taylor, J., Cristianini, N.: Kernel Methods for Pattern Analysis. Cambridge University Press, New York (2004)
41. Shervashidze, N., Schweitzer, P., van Leeuwen, E.J., Mehlhorn, K., Borgwardt, K.M.: Weisfeiler-Lehman graph kernels. J. Mach. Learn. Res. **12**, 2539–2561 (2011)
42. Sokolsky, O., Kannan, S., Lee, I.: Simulation-based graph similarity. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 426–440. Springer, Heidelberg (2006). https://doi.org/10.1007/11691372_28
43. Vishwanathan, S.V.N., Schraudolph, N.N., Kondor, R., Borgwardt, K.M.: Graph kernels. J. Mach. Learn. Res. **11**, 1201–1242 (2010)

44. Vujošević-Janičić, M., Nikolić, M., Tošić, D., Kuncak, V.: Software verification and graph similarity for automated evaluation of students' assignments. Inf. Softw. Technol. **55**(6), 1004–1016 (2013)
45. Wang, D., Cui, P., Zhu, W.: Structural deep network embedding. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1225–1234. ACM (2016)
46. Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., Song, D.: Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 363–376. ACM (2017)