

DroidExec: Root Exploit Malware Recognition Against Wide Variability via Folding Redundant Function-relation Graph

Te-En Wei

National Taiwan University of Science and Technology
Taipei 106, Taiwan
Email: d9807501@mail.ntust.edu.tw

Hahn-Ming Lee

National Taiwan University of Science and Technology
Taipei 106, Taiwan
Academia Sinica,
Nankang, Taipei 11529, Taiwan
Email: hmlee@mail.ntust.edu.tw

Hsiao-Rong Tyan

Chung Yuan Christian University,
Chung Li District, Taoyuan City 32023, Taiwan
Email: tyan@ice.cycu.edu.tw

Hong-Yuan Mark Liao

Academia Sinica,
Nankang, Taipei 11529, Taiwan
Email: liao@iis.sinica.edu.tw

Albert B. Jeng

National Taiwan University of Science and Technology
Taipei 106, Taiwan
Email: albertjeng@hotmail.com

Jiunn-Chin Wang

National Taiwan University of Science and Technology
Taipei 106, Taiwan
Email: jcwang@just.edu.tw

Abstract—DroidExec is a novel root exploit recognition to reduce the influence of wide variability, which usually affects the Android malware detection rate, because of Android applications' various properties. In Android, a specific malware family (e.g., root exploit malware), and thus its implementation may be influenced by the campaign it is serving, and thus producing wide variability, leading its samples to appear to match a wider range of potential families. In this paper, we propose a similarity recognition named as DroidExec, reducing wide variability via folding redundant function-relation graph based on Bipartite Graph Conceptual Matching of graph edit distance. We compute the multiple square roots for each 2×2 block in the cost matrix to conceptually cripple the wide variability. In the experiments, we measure the applications' opcode structural similarity for clustering Android malware. Empirical validation shows that DroidExec can effectively filter surplus and various behaviors, which can improve the precision/recall rate from 82%/95% to 83%/97%, respectively.

I. INTRODUCTION

In recent years, Android smartphones have grown rapidly more popular; consequently, the development of Android applications and the repackaging of Android apps have continued to accelerate. Attackers can easily repackage an application under their own names or embed advertisements to earn pecuniary profits. Thus, analysis of Android applications becomes a complicated task due to the nature of the interaction among the various entities in its component-based framework. Due to the enormous number of malicious Android applications per day, researchers have proposed several malware detection methods based on static analysis such as learning-based and

signature-based approaches. A learning-based approach extracts features from an application's behavior (e.g., permission requests and critical API calls) and applies standard machine learning algorithms to perform binary classification [34] [8] [7] [11]. However, because the extracted features are associated with application syntax rather than program semantics, these detectors are also susceptible to evasion [21]. A signature-based approach looks for specific patterns in the bytecode and API calls [19] [32] [24], which is easily evaded by bytecode-level Transformation Attacks [30]. To impersonate a benign application, a malware developer can exploit Android applications' various and event-driven characteristics, which necessarily causes widespread behaviors, to evade the malware detection system based on a serving campaign.

Fortunately, the graph edit distance has attracted particular interest because it is error-tolerant to noise and distortion, which is defined as the cost of the least expensive sequence of edit operations needed to transform one graph into another. According to Xu et al., graph similarity is widely used in many applications, particularly in the identification of obfuscated malware using the similarities of function-call graphs [20]. The graph edit distance is also an important measure of the similarity between two malwares [15]. Therefore, the graph edit distance is a feasible way to resolve the wide variability problem.

In Android, a specific malware family (e.g., root exploit), and thus its implementation may be influenced by the campaign it is serving, and thus producing wide variability, leading

its samples to appear to match a wider range of potential families [5]. Ho et al. [29] shows that the attacker performs root exploits from the Java code, which may create false negatives and requires access to low-level APIs in Java. Therefore, root exploit malware can achieve the API access goal through a wide and varied campaign that causes many wide variability problems. Based on the reports of Zhou et al. [35], Asroot is a root exploit malware class that utilizes privilege escalation to secretly steal sensitive data from a mobile.

Hence, we propose a novel recognition called DroidExec to defeat the influence of wide variability via folding redundant function-relation graph. Moreover, we propose a bipartite graph conceptual matching to decrease the redundant function-relation graph when decomposing the cost matrix based on graph edit distance, and we then acquire a smaller matrix scope by computing the multiple square roots for each 2×2 block in the cost matrix. We illustrate this idea concept in Figure 3: bipartite graph matching focuses on randomly matching a pair of vertices, allowing the wide variability (e.g., A and F in Figure 3) to affect the edit distance calculation. Thus, we can resolve this problem through smoothing the influence by computing the multiple square roots for each pair of groups of vertices, as in the bipartite graph conceptual matching in Figure 3. Empirical validation shows that DroidExec can effectively filter surplus and various behaviors, which can improve the precision/recall rate.

In summary, this paper provides the following contributions:

- We propose a novel similarity recognition DroidExec to reduce the influence of wide variability, which usually affect the detection rate because of the various properties of Android applications. DroidExec includes several modules to address the reduction of wide variability. We first construct the function-relation graph to connect all methods using smali codes. Then, we tag the opcode components of a method to generate the whole opcode-component graph in each Android application. Finally, we use the graph edit distance to compute the similarity between arbitrary Android malicious samples.
- To reduce the influence of wide variability in DroidExec, we propose bipartite graph conceptual matching in graph edit distance. In this concept, we first combine two vertices into one group and then compute the multiple square roots for each 2×2 block in the cost matrix that mentions bipartite matching, one pair of groups of vertices at a time. The influence of wide variability could be smoothed via folding redundant behaviors based on conceptual matching process.
- To steal private data, an attacker would design a root exploit malware class with a wide and varied serving campaign, which utilizes privilege escalation to secretly steal sensitive data from a mobile. Thus, wide variability caused by root exploits can effectively disturb the classification results. Fortunately, DroidExec can effectively smooth these widespread characteristics and help the learning algorithm cluster them into the correct class.

In this paper, section 2 surveys related work, and we describe the computation of the graph edit distance and present its notation in Section 3. Next, we propose DroidExec, which reduces the influence of wide variability, in Section 4. We then present the experimental results of applying DroidExec to malicious Android datasets in Section 5. We also present the empirical analysis in Section 6 to explain how DroidExec can resolve the wide variability problem. Finally, we draw conclusions and present limitations in Section 7.

II. RELATED WORK

Android is a popular system in mobile, but several security problems still exist until now. For example, system relies on virtualization (e.g., sandbox) or emulation, and have recently started to be available to process mobile malware. Therefore, there are some papers proposed to resolve the dynamic problems in sandbox such as monitoring behavior, performance, hardware and software components [31] and In-App Billing [6]. Existing tools for automatically detecting and classifying malware have proliferated over the last years, which basically fall into two general categories: (1) signature-based and (2) machine learning-based approached. Signature-based looks for specific patterns in the bytecode and API calls [19] [32] [24] [18], which is easily evaded by bytecode-level Transformation Attacks [30]. Learning-based approach extracts features from an application's behavior (E.g., permission requests and critical API calls) and applies standard machine learning algorithms to perform binary classification [34] [8] [7] [11]. However, the extracted features are associated with application syntax, rather than program semantics, these detectors are also susceptible to evasion [21].

Currently, the trends of malware engineering suggest that malicious software will continue to evolve its sophistication [10]. Furthermore, the availability of reuse-oriented development methodologies are popular for attackers, which is enabled by reverse engineering tools such as APKTool[2] and Androguard [1]. This is particularly important for variant malware generation with wide self-functionality of Android operation system. In this regard, one of the most common distribution strategy for smartphone malware consists of repackaging popular applications and distributing them through alternative markets with additional malicious code attached such as piggybacked [35] [32]. Variant malware such as piggybacked can pretend a benign application, which necessarily contain widespread behaviors to evade the malware detection system based on serving campaign. Fortunately, graph similarity computation is a popular method to defeat against the various features and malicious behaviors. Xu et. al proposed a method to identify the variant malware based on the function-call graph [20]. Hu et. al used the graph edit distance that aims to capture the similarity among variants within the same malware family [33]. In recent years, some authors have generated the malicious behavioral sub-graph as features to detect the obfuscation malware based on learning approach [5] [21]. Additionally, the graph edit distance is also widely used in malware detection, which has attracted particular interest

because it is error-tolerant to noise and distortion. Kinable et. al proposed a method to compute pairwise graph similarity scores via graph matching which approximately minimize the graph edit distance [15] [23]. However, the graph edit distance isn't widely used in Android application repackaging detection. The graph edit distance is only used in Android obfuscated application detection, Zhabg et. al. proposed a user interface based approach to Android application repackaging detection using feature view graph and tested the Google Play's applications using their tool [9]. Unfortunately, they only focus on benign application in Google Play. Due to the Android application's various and event-driven characteristics, in this paper, we try to plant the graph edit distance to Android variant malware detection and defeat against the influence of wide variability.

III. DESCRIPTION OF DROIDEXEC

In this section, we propose a novel root exploit recognition named DroidExec to reduce the influences of wide variability, which utilizes a smaller cost matrix by performing bipartite graph conceptual matching using graph edit distance.

A. DroidExec in a Nutshell

We denote the graph edit distance between two opcode-component graphs g_1 and g_2 as $D(g_1, g_2)$, which is the sum of VertexCost and EdgeCost. In which, VertexCost and EdgeCost indicate the numbers of insertions, deletions and substitutions of vertices and edges, respectively, where determine the minimum amount of distortion for transforming one graph into another graph [28]. Let $g_1 = (V_1, E_1)$ and $g_2 = (V_2, E_2)$: g_1 is the source graph of malware, and g_2 is the target graph of malware. The graph edit distance between g_1 and g_2 is defined as follows:

$$D(g_1, g_2) = \text{VertexCost} + \text{EdgeCost} \quad (1)$$

$$\sigma(g_1, g_2) = \frac{D(g_1, g_2)}{|V(g_1)| + |V(g_2)| + |E(g_1)| + |E(g_2)|} \quad (2)$$

Finally, the similarity $\sigma(g_1, g_2)$ of two graphs g_1 and g_2 is obtained from the graph edit distance $D(g_1, g_2)$. The details are described in Equation 2, which is a real value on the interval [0,1], where 0 indicates that graphs g_1 and g_2 are identical, whereas a value near 1 implies that the pair is highly dissimilar. As mentioned before, finding the minimum graph edit distance, i.e., $\min_{D(g_1, g_2)}$, is an NP-hard problem but can be approximated. Riesen et al. and Francesc Serratos introduced an approximation algorithm with a good trade-off between accuracy and speed [26] [27]. Given two graphs G_1 and G_2 , where $|G_1| = n$ and $|G_2| = m$, their algorithms, respectively, use a $(n + m) \times (n + m)$ cost matrix C_1 and a $n \times m$ cost matrix C_2 , which both give the cost of mapping a vertex $v \in V(G_1)$ to a vertex $v \in V(G_2)$. Next, Munkres algorithm [22], also known as the Hungarian algorithm, which runs in polynomial time, is applied to find an exact one-to-one vertex assignment that minimizes the total mapping cost. Each

entry in the cost matrix represents the cost of matching vertex $v \in V(G_1)$ to a vertex $u \in V(G_2)$. The cost of matching a pair of nodes c_{ij} could equal the transformed cost as defined for the graph edit distance in the following subsection.

DroidExec consists of three components: (1) Structural Graph Constructor, (2) Bijective Similarity Measurer and (3) Confusion Reduction Module. The details are shown in Figure 1. The Structural Graph Constructor consists of two subcomponents, the Function-relation Graph Extraction and the Opcode-component Graph Constructor. The main purpose is to attempt to extract the malicious behavioral patterns from smali codes disassembled from Android malicious sample. These patterns are used to construct the relations between functions and opcode-components for each function with extracted relation edges. The Bijective Similarity Measurer attempts to bijectively generate a cost matrix between two malicious samples, where each one's element is the similarity of bipartite mapping from a vertex of predecessor sample to a vertex of the successor sample. The Confusion Reduction Module locally decomposes the scope of the cost matrix, which uses bipartite graph conceptual matching to reduce the wide variability. It preserves the key malicious behavior patterns to discover the structural relation to smooth these widespread characteristics and help the learning algorithm clustering them into the correct class.

B. Function-relation Graph Extraction

To cluster all collected Android malware, we applied our system DroidExec to real-world deployment using a public dataset with input samples collected from the Contagio Mobile [3] site and Zhou et al. [32]. We use public tool Androguard [1] to extract the vertex/edge relationships in smali method-call graph. It uses a disassembling tool such as APKTool [2] to obtain the smali codes and draw the function-relation graph as shown in Figure 2.

C. Opcode-component Graph Constructor

In this subsection, we expect to label vertices according to the type of the opcodes contained in each smali method, and the extracted features are similar to Gascon et al. [13]. However, we do not assume any category to label the opcodes. On the contrary, we have reviewed the dalvik opcode specification assuming all 218 distinct opcodes based on their functionality as described in website [4]. Each vertex can thus be labeled using a 218-bit field, where each bit is associated with one of the opcodes.

D. Bijective Similarity Measurer

In this subsection, more accurate cost estimation allows the discovery of better graph matchings and hence more accurate edit distances. The cost of matching a pair of vertices, element c_{ij} of the matrix, could equal the transformed cost as defined for the graph edit distance, which consists three components: Host jaccard similarity, In-degree, jaccard similarity and Out-degree jaccard similarity can be observed developing between

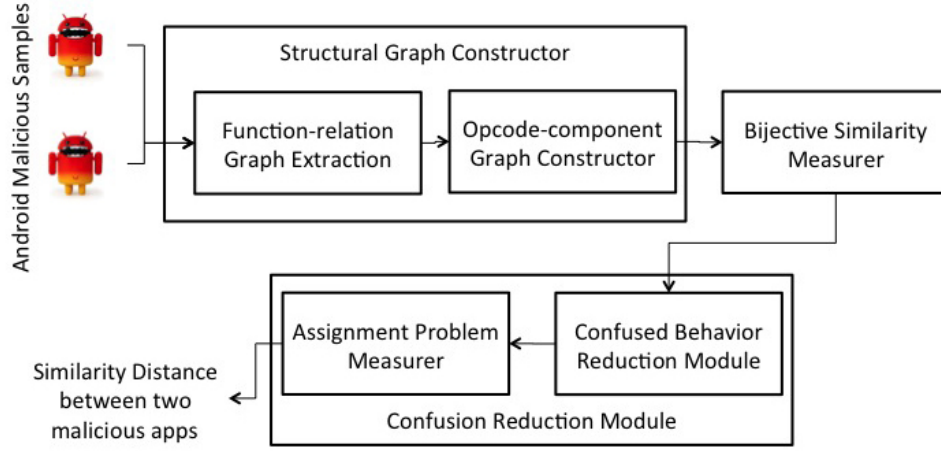


Fig. 1. The system architecture of DroidExec. Use Android malware behavioral graph as input, and produces similarity results of behavioral models as an output after decomposing the wide variability.

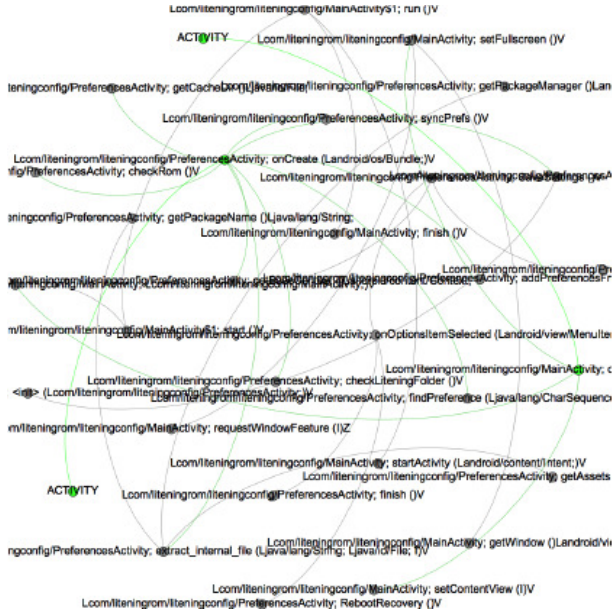


Fig. 2. An Example of a function-call graph generated from a real malware sample.

vertices v and u but can be presented as $\delta(v, u)$, $\delta^-(v, u)$ and $\delta^+(v, u)$, and their details can be defined as follows:

$$\delta(v, u) = \left| 1 - \frac{Bit_{op}(v) \cap Bit_{op}(u)}{Bit_{op}(v) \cup Bit_{op}(u)} \right| \quad (3)$$

$$\delta^-(v, u) = \left| 1 - \frac{Neighbor^-(v) \cap Neighbor^-(u)}{Neighbor^-(v) \cup Neighbor^-(u)} \right| \quad (4)$$

$$\delta^+(v, u) = \left| 1 - \frac{Neighbor^+(v) \cap Neighbor^+(u)}{Neighbor^+(v) \cup Neighbor^+(u)} \right| \quad (5)$$

where $Bit_{op}(v)$ indicates the extraction of vertex v 's opcode components, presented by 218 binary bits defined

as the above section. Additionally, $Neighbor^-(v)$ and $Neighbor^+(v)$, respectively, present the Neighbor vertices of vertex v , which are defined as follows:

$$Neighbor^-(v) = \{z \mid e_{z,v} \in E\} \quad (6)$$

$$Neighbor^+(v) = \{z \mid e_{v,z} \in E\} \quad (7)$$

Clearly, each element c_{ij} of the cost matrix indicates the bijective mapping of the i -th vertex to the j -th vertex that can be defined as follows:

$$c_{ij} = \delta(i, j) + \delta^-(i, j) + \delta^+(i, j) \quad (8)$$

E. wide variability Reduction Module

Computing the graph edit distance is an assignment problem that consists of finding an optimal assignment from the elements of graph g_1 to the elements of graph g_2 , where g_1 and g_2 have the same cardinality. Assuming that the numerical costs are given for each assignment pair, an optimal assignment is one that minimizes the sum of the assignment costs. Based on previous work [26], resolving the assignment problem by bipartite graph matching can be shown in Figure 3 (a). In our proposed DroidExec, we denote a novel cost matrix C' , whose scope is smaller than original cost matrix C after bipartite graph conceptual matching. We can reduce the influences of wide variability during conceptual matching. Therefore, the assignment problem of bipartite graph conceptual matching can be shown in Figure 3 (b). In our proposed bipartite graph conceptual matching, there are two graphs g_1 and g_2 and an $n \times n$ cost matrix C with real numbers given, where $|g_1| = |g_2| = n$. The conceptual matched matrix element c'_{xy} correspond to the decomposed c_{ij} , and the cost is that for assigning elements of the x -th group in g_1 to elements of the y -th group in g_2 . The assignment problem can be stated as finding a permutation p'_x that minimizes $\sum_{x=1}^{n/2^l} d_{xp'_x}$, and l refers to the number of bipartite graph conceptual matching.

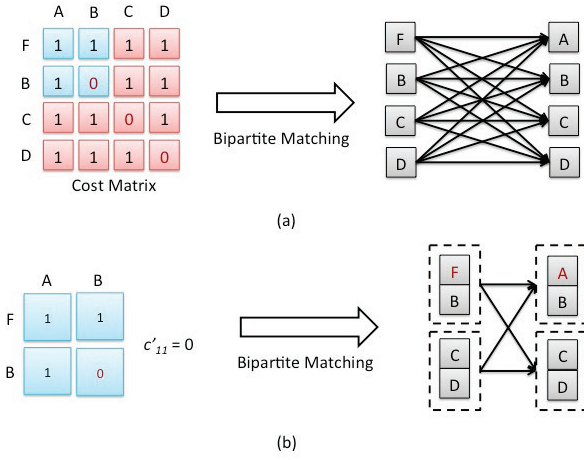


Fig. 3. An example of a comparison between bipartite graph matching and bipartite graph conceptual matching. (a) An original cost matrix indicates the bipartite graph matching (b) A conceptual matched matrix indicates the bipartite graph conceptual matching

Therefore, we denote the conceptual matched matrix C' and its elements c'_{xy} that are shown as Equation 9 and Equation 10. In Equation 10, we compute the multiple square roots of each 2×2 block rather than the arithmetic mean. Because the geometric mean is less influenced by variant data than the arithmetic mean, it can flexibly acquire a distance value. This characteristic not only preserves the optimal permutation from the Munkres algorithm but also enables filtering of surplus and various behaviors. The concept is described in Figure 3. We assume the vertices F and A are both wide variability, and then smooth them into an acquired matrix using bipartite graph conceptual matching to eliminate the side-effects of each wide variability.

$$C' = \begin{bmatrix} c'_{11} & c'_{12} & \cdots & c'_{1\frac{n}{2}} \\ c'_{21} & c'_{22} & \cdots & c'_{2\frac{n}{2}} \\ \vdots & \vdots & \ddots & \vdots \\ c'_{\frac{n}{2}1} & c'_{\frac{n}{2}2} & \cdots & c'_{\frac{n}{2}\frac{n}{2}} \end{bmatrix} \quad (9)$$

$$c'_{ij} = (c_{2a-1,2b-1} \times c_{2a-1,2b})^{0.25} \times (c_{2a,2b-1} \times c_{2a,2b})^{0.25} \quad (10)$$

Furthermore, we define a mapping equation, as shown in Equation 11, that performs decomposition from position (i, j) of the cost matrix C to position (i, j) of the conceptual matched matrix C' .

$$(i, j) = \left\{ (a, b) \mid a = 1, 2, \dots, \left\lfloor \frac{n}{2} \right\rfloor, b = 1, 2, \dots, \left\lfloor \frac{n}{2} \right\rfloor \right\} \quad (11)$$

In the conceptual matched matrix, we define a geometric mean computational function $f_{gm}(X)$ (shown as Equation 12) that implements bipartite graph conceptual matching, where X is an input cost matrix.

$$f_{gm}(X) = C' \quad (12)$$

F. Assignment Problem Measurer

The Munkres algorithm [22] [17] is a known algorithm that solves the bipartite matching problem in polynomial time. After obtaining the conceptual matched matrix, we implement the Munkres algorithm to find the optimal permutation that minimizes the cost as the similarity distance between the two Android malicious samples' graphs g_1 and g_2 . In Algorithm 1, the first three steps determine whether the cost matrix has the specific scenario which means the optimization bipartite match from source sample's vertex to target sample's vertex already exists. Then, we can decide whether the scope of the cost matrix needs to be reduced. Furthermore, we acquire a conceptual match matrix C' from the function $f_{gm}(C)$, which can effectively eliminate the side-effects of each wide variability.

Algorithm 1 The Bipartite Graph Conceptual Matching algorithm

Require: Cost Matrix C

Ensure: Conceptual Matched Matrix C'

- 1: For each row of C , find the smallest element and subtract it from each element in its row
 - 2: Find a zero in the resulting C . If there is no starred zero in its row or column, star that zero. Repeat for each zero in C
 - 3: Cover each column containing a starred zero, and x columns are covered
 - 4: **if** $x < n$, $C = n \times n$ **then**
 - 5: Return $C' = f_{gm}(C)$
 - 6: **else**
 - 7: Return $C' = C$
 - 8: **end if**
-

IV. EXPERIMENTAL RESULTS

In this section, we will demonstrate how the DroidExec can be utilized to perform distance computations to reduce the influences of wide variability in conceptual matching way. We collected Android malware from the Contagio Mobile [3] site and Zhou et al. [32]. The total Android malicious samples we have is 1247. In this paper, we purpose to decrease the existence of the wide variability that indicate the functions or methods of smali codes in each Android application. Zhang et. al [9] mention repackaged applications that have close number of functions (smali methods) with each other. Therefore, we use a part of collected malicious samples whose number of functions are close with each other. Moreover, to increase the complexity of opcode components, and we then obfuscate these malicious samples using transformation attack that proposed by Rastogi et. al [30].

A. Evaluation Methods

In this subsection, we consider the similar opcode structures of function-call graphs, and we analyze the models using a graph similarity methodology. This method is helpful for discovering similarity correlations between two graphs. Given

two graphs g_i and g_j , we use a measurement of the graph similarity between the two graphs that depends on how well one graph is described by the model for the other graph. $d_{i,j}$ is the distance between the two graphs g_i and g_j , as follows. In particular, we have pairwise similarities for all pairs of graphs. We use all pairs of graphs to calculate all distances, and we construct a similarity matrix for all Android malware samples.

$$d_{i,j} = \sigma(g_i, g_j) \quad (13)$$

$$d_{i,j} = \begin{bmatrix} d_{1,1} & d_{1,2} & \cdots & \cdots \\ d_{2,1} & d_{2,2} & \cdots & \cdots \\ \vdots & & \ddots & \\ \vdots & & & \ddots \end{bmatrix} \quad (14)$$

Then, we present the evaluation methods for the experiments. Because the graph edit distance is a popular approach for the detection of malicious samples, we consider the feature selection from Gascon et al. [13] in our evaluation, which detected the Android malware based on structural similarity. The value of each element in the cost matrix is calculated by the similarity between the i -th function of one sample and the j -th function of another sample. The similarity consists of the sum of host-similarity and neighbor-similarity as considered by Kinable et al. [15], who used the graph edit distance for traditional malware classification. Here, host-similarity indicates the cost to transform the i -th function to the j -th function, and neighbor-similarity refers to the similarities between the neighbor functions of the i -th function and the j -th function. Moreover, these similarities are all calculated by the Jaccard Similarity of the opcode compositions.

The evaluation methods we used are from Wang et al. [12] and Rieck et al. [16], which involve precision, recall rate, and F-measure to evaluate our experiments. These methods were used to analyze the reference dataset described above. The variable n is the number of malware samples; we use 11 malware families of the reference dataset to evaluate our performance. The goal of precision is to find the maximum number of instances of the same malware family in each group. It measures how well individual groups agree with the malware families. Precision assigns samples of different variants to different groups. Y is the number of groups, and y is each group. Precision is defined as follows:

$$Precision = \frac{1}{n} \sum_{y \in Y} \max |\#y| \quad (15)$$

The goal of the recall rate is to find the maximum number of instances of the same group in each malware family. It measures the extent to which malware families are distributed over groups. The recall rate assigns samples of the same variants to the same group. M is the number of malware families, and m is each malware family such that the reference dataset has 11 malware families. The recall rate is defined as follows:

$$Recall = \frac{1}{n} \sum_{m \in M} \max |\#m| \quad (16)$$

According to the evaluation method described above, if each report belongs to a group, we will obtain a precision = 1 but the worst recall rate, and if all reports are in the same group, we will obtain a recall rate = 1 but the worst precision. To obtain both high precision and a high recall rate when evaluating our system, we attempt to have each group contain all samples of one malware class. Therefore, we use an F-measure score to combine both precision and the recall rate. The F-measure is defined as follows:

$$F - measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (17)$$

To evaluate our performance and to compare the methodology whether using bipartite conceptual matching or not in the referenced dataset, we need to decide the number of clusters. We used the method proposed by Ray and Turi [25], and we measured the distance of the samples from their cluster centroid. This method is used to iteratively find the different number of clusters and to select the one that results in compact clusters. Therefore, we used both intra-cluster and inter-cluster distances to measure the number of clusters. We attempted to find the number of clusters with better results. First, we used the intra-cluster distance to calculate the distance between a sample and its cluster centroid in each cluster. The intra-cluster distance is defined as follows:

$$IntraCluster = \frac{1}{n} \sum_{i=1}^K \sum_{x \in Y_i} \|x - z_i\|^2 \quad (18)$$

where n is the number of Android malicious samples, K is the number of clusters, and z_i is the cluster centroid of cluster Y_i . Second, we used the inter-cluster distance to calculate the distance between different cluster centroids and to find the minimum of these values. The inter-cluster distance is defined as follows:

$$InterCluster = \text{minimum}(\|z_i - z_j\|^2) \quad (19)$$

where z_i and z_j are the different cluster centroids. Finally, we hope to find the minimum intra-cluster distance and maximum inter-cluster distance to find the number of clusters with better results. We used the validity ratio to combine both the intra-cluster and inter-cluster distances. The validity ratio is defined as follows:

$$Validity = \frac{IntraCluster}{InterCluster} \quad (20)$$

Therefore, we want to find the number of clusters with better results to minimize the validity ratio. The validity ratio in the different numbers of clusters is shown in Figure 4 based on using the reference dataset. We almost found the best number of clusters in the reference dataset. For the experiments, we determined that 7 clusters minimized the validity ratio, and

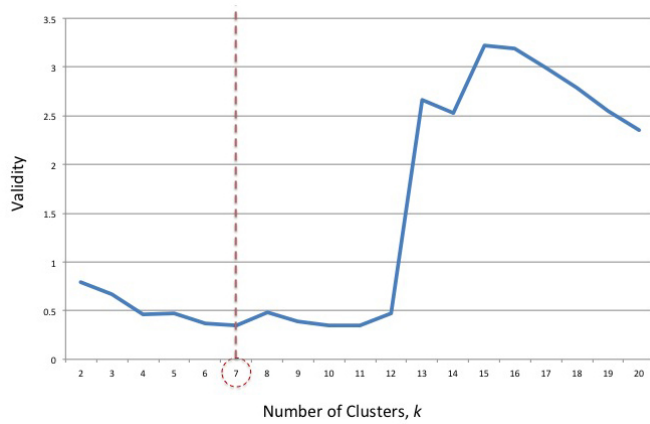


Fig. 4. Validity ratio in the different numbers of clusters

we then fixed the parameter of the better number of clusters, k , to be equal to 7.

B. Empirical Results

The experimental results are summarized in Table I. We considered the bipartite matching method as the baseline for comparing the effectiveness of our approach, bipartite conceptual matching in DroidExec. For both our method and bipartite matching, we focused on the similarity of opcode-component graphs of Android malware to using the Munkres algorithm.

In DroidExec, we have a higher F-measure (89%), as shown in Figure 6. In our experiments, we used 7 clusters to cluster all malicious samples, whereas in bipartite matching, the number of clusters is selected based on validity. Our precision (83%) is also greater than that of bipartite matching, as shown in Figure 5. Additionally, we also use the same method to evaluate the transformations. In Figure 6 and Figure 5, we acquire higher F-measure (89%) and precision (83%) using our bipartite conceptual matching than bipartite matching.

Because we precisely divided the dataset into 11 Android malware classes, portions of the malware classes were distributed over different clusters. The goal of precision is to find a large amount of biased data in all the malware classes to discover the variants of malware classes. For example, the Asroot malware class is distributed over two clusters with maximum numbers in each cluster. Asroot spreads over many different clusters because it is a privilege escalation malware with widespread and various behaviors [19] [14].

The goal of recall is for samples of the same class to be grouped in the same cluster in almost all cases. When many clusters are produced, the recall rate should be reduced in Equation 16. In contrast, our recall rate (97%) was also better than the recall rate of bipartite matching (95%), as shown in Figure 5, because we used our method to group the same malware class in the same cluster. The same evaluated results of recall rate were obtained in transformed samples. The recall rate of transformations using bipartite conceptual matching is 97%, better than bipartite matching (95%). The F-measure was

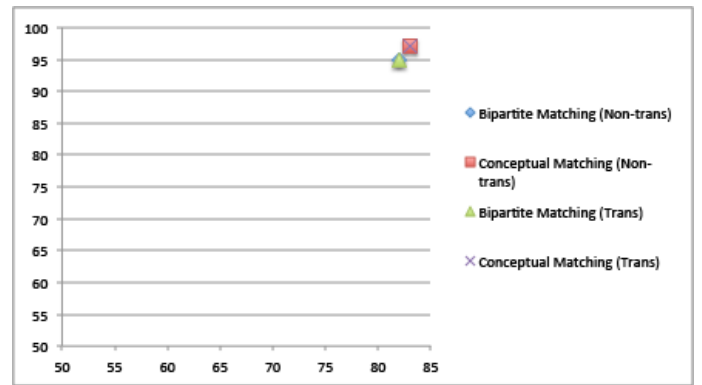


Fig. 5. Precision rate and recall rate of Bipartite Conceptual Matching and Bipartite Matching

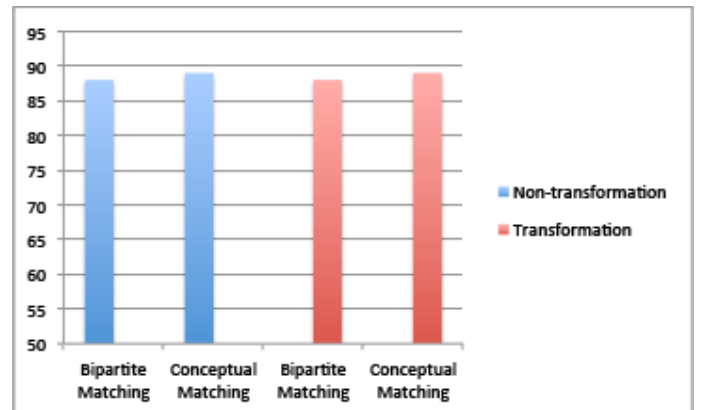


Fig. 6. F-measures of Bipartite Conceptual Matching and Bipartite Matching

used to evaluate both the precision and recall rate. Therefore, our method groups variants of malware classes better than bipartite matching, and the transformation attack would not allow our method to reduce the wide variability in DroidExec. In the next section, we discuss certain malware classes for which our method and the bipartite matching outperform each other in different occasions.

V. DISCUSSION

In this case, the clustering results for bipartite matching lead to a 82%/95% precision/recall rate, and it performs effectively on malware classes such as FakePlayer, jSMShider, SndApps, GPSSMSSpy, Gone60 and FakeInstaller. In our bipartite conceptual matching, we obtain the same clustering results for malware classes as obtained with bipartite matching. However, there are certain classes (e.g., Asroot, BaseBridge, DroidDreamLight, GGTracker and Tapsnake) that share the most similar opcodes in their opcode-component graphs; thus, k-means clusters them into one class. We discovered that this situation occurred with bipartite matching, which is affected by the wide variability. As shown in Table I, we acquired better clustering results using bipartite conceptual matching whether the sample is transformed or not. Clearly, Asroot class is the key malware in our method, which increases the

TABLE I
THE EXPERIMENTAL RESULTS.

Method	Transformation [30]	Number of Clusters	Precision (%)	Recall (%)	F-Measure (%)
Bipartite Conceptual Matching	No	7	83	97	89
Bipartite Matching	No	7	82	95	88
Bipartite Conceptual Matching	Yes	7	83	97	89
Bipartite Matching	Yes	7	82	95	88

precision/recall rate to 83%/97%. Moreover, Asroot is also a root exploit malware class that utilizes privilege escalation to secretly steal sensitive data from mobiles. Therefore, to achieve its access goal, Asroot needs a wide and varied campaign that causes many wide variability.

In our experiments, bipartite matching clusters several samples of Asroot into the GPSSMSSpy class. We denote an identification named *asroot_{miss}* to present these mis-clustered samples. The Asroot class is privilege escalation malware according to the reports of [19] and [14]; it only shares a small portion of behaviors embedded with various functionalities, and these various functionalities are largely shared with GPSSMSSpy and observed as wide variability. In particular, *asroot_{miss}* are tiny malwares whose serving campaign possesses only the key behaviors (e.g., `extract_internal_file()` function) and has completely different functionalities compared with others in the Asroot class. Our analysis shows the opcode compositions of the `extract_internal_file()` function that all Asroot samples have, and the others are various functionalities comparing *asroot_{miss}* to other Asroot samples. Therefore, bipartite matching clusters this tiny and variant sample *asroot_{miss}* into the wrong class. Fortunately, our bipartite conceptual matching can amend this problem: the multiple square roots of each 2×2 block are computed once, and the lowest cost is retained, which can effectively smooth the wide variability. Empirical validation shows that we obtain better precision and a better recall rate. Moreover, we cluster Asroot samples into the right class rather than the wrong class GPSSMSSpy using bipartite conceptual matching in DroidExec.

• Limitations and Future Work.

Like any learning-based approach, DroidExec requires an accurate training dataset and features to cluster malicious behaviors into the correct classes. The effectiveness of our approach depends on the quality of the given feature selections, e.g., the opcode components we use, referenced by [13]. According to our observations, malware classes such as BaseBridge, DroidDreamLight,

GGTracker and Tapsnake allow DroidExec to exhibit a lower precision/recall rate regardless of using bipartite matching or bipartite conceptual matching. These classes have a smaller number of functions, and their opcodes overlap completely, which influences the clustering results. Therefore, our further works will attempt to construct the behavioral graph of API-calls or Android Components (e.g., activity, service, broadcast receiver and content provider) to address the real operations for each application and difficulty of the modification even an attack performs in a different serving campaign. Fortunately, our bipartite conceptual matching can handle the Asroot class, as detailed above.

VI. CONCLUSION

In this paper, we propose a novel recognition, DroidExec, to decrease the influence of wide variability via folding redundant function-relation graph based on bipartite graph conceptual matching of graph edit distance. We used public datasets of Android malicious samples and obfuscated samples by transformation attack [30] to prove our idea concept. To determine the structural similarity between different function-relation graphs in variants of Android malware classes, we used the graph edit distance to measure the similarity between two malicious samples based on each opcode component of functions and the correlation between functions. Thus, each sample can be represented as an Opcode-components Graph. Additionally, we used bipartite graph conceptual matching to reduce the scope of the cost matrix, which can reduce the widespread contents when computing the multiple square roots of each 2×2 block. In our experiments, we improve the precision/recall rate from 82%/95% to 83%/97%. Second, empirical validation also shows that DroidExec cannot be affected by a transformation attack when identifying the wide variability.

ACKNOWLEDGMENT

This research is supported in part by the National Science Council of Taiwan under grants number NSC 102-2218-E-011-010 MY3 and 102-2218-E-011-011-MY3. The authors also

gratefully acknowledge the helpful comments and suggestions of the reviewers, which have improved the presentation.

REFERENCES

- [1] Androguard. <https://code.google.com/p/androguard/>.
- [2] Apktool. <http://code.google.com/p/android-apktool/>.
- [3] Contagio mobile. <http://contagiomindump.blogspot.com/>.
- [4] Dalvik opcodes. http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html.
- [5] G. G. V. Y. Chao Yang, Zhaoyan Xu and P. Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *Computer Security-ESORICS 2014*. Springer International, 2014.
- [6] W. R. Collin Mulliner and E. Kirda. Virtualswindle: An automated attack against in-app billing on android. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, 2014.
- [7] M. H. H. G. Daniel Arp, Michael Spreitzenbarth and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.
- [8] T.-E. W. H.-M. L. Dong-Jie Wu, Ching-Hao Mao and K.-P. Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Seventh Asia Joint Conference on Information Security (Asia JCIS)*, pages 62–69, 2012.
- [9] S. Z. D. W. Fangfang Zhang, Heqing Huang and P. Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 2014)*, 2014.
- [10] P. P.-L. Guillermo Suarez-Tangil, Juan E. Tapiador and A. Ribagorda. Evolution, detection and analysis of malware for smart devices. In *IEEE Communications Surveys & Tutorials*, 2013.
- [11] B. S. N. L.-Y. Q. Hao Peng, Chris Gates and R. Potharaju. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 241–252. ACM, 2012.
- [12] T.-E. W. Horng-Tzer Wang, Ching-Hao Mao and H.-M. Lee. Clustering of similar malware behavior via structural host-sequence comparison. In *Proceeding COMPSAC '13 Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference*, pages 349–358, 2013.
- [13] D. A. Hugo Gascon, Fabian Yamaguchi and K. Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54, 2013.
- [14] X. Jiang and Y. Zhou. A survey of android malware. In *Android Malware*, NY. USA., 2013. Springer.
- [15] J. Kinable and O. Kostakis. Malware classification based on call graph clustering. In *Journal in computer virology*, 7(4), pages 233–245, 2011.
- [16] C. W. Konrad Rieck, Philipp Trinius and T. Holz. Automatic analysis of malware behavior using machine learning. In *Journal of Computer Security*, 19(4), pages 639–668, 2011.
- [17] H. W. Kuhn. Variants of the hungarian method for assignment problems. In *Naval Research Logistics Quarterly*, 3(4), pages 253–258, 1956.
- [18] V. N. Luke Deshotels and A. Lakhota. Droidlegacy: Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*. ACM, 2014.
- [19] Q. Z. S. Z. Michael Grace, Yajin Zhou and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 281–294. ACM, 2012.
- [20] S. Q. J. X. H. Z.-Y. R. Ming Xu, Lingfei Wu and N. Zheng. A similarity metric method of obfuscated malware using function-call graph. In *Journal of Computer Virology and Hacking Techniques*, 9(1), pages 35–47, 2013.
- [21] H. Y. Mu Zhang, Yue Duan and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *21st ACM Conference on Computer and Communications Security (CCS'14)*. ACM, 2014.
- [22] J. Munkres. Algorithms for the assignment and transportation problems. In *Journal of the Society for Industrial & Applied Mathematics*, 5(1), pages 32–38, 1957.
- [23] H. M. Orestis Kostakis, Joris Kinable and K. Mustonen. Improved call graph comparison using simulated annealing. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC 2011)*, 2011.
- [24] C. N.-R. Rahul Potharaju, Andrew Newell and X. Zhang. Plagiarizing smartphone applications: Attack strategies and defense techniques. In *Engineering Secure Software and Systems*. Springer Berlin Heidelberg, 2012.
- [25] S. Ray and R. H. Turi. Determination of number of clusters in k-means clustering and application in colour image segmentation. In *Proceedings of the 4th international conference on advances in pattern recognition and digital techniques*, 1999.
- [26] K. Riesen and H. Bunke. Approximate graph edit distance computation by means of bipartite graph matching. In *Image and Vision Computing*, 27(7), pages 950–959, 2009.
- [27] F. Serratos. Fast computation of bipartite graph matching. In *Pattern Recognition Letters*, 14, pages 244–250, 2014.
- [28] K. R. Stefan Fankhauser and H. Bunke. Speeding up graph edit distance computation through fast bipartite matching. In *Graph-based representations in pattern recognition*, pages 102–111. Springer Berlin Heidelberg, 2011.
- [29] X. G. Tsung-Hsuan Ho, Daniel Dean and W. Enck. Prec: practical root exploit containment for android devices. In *Proceedings of the 4th ACM conference on Data and application security and privacy (CODASPY '14)*, pages 187–198. ACM, 2014.
- [30] Y. C. Vaibhav Rastogi and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 2013.
- [31] T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014.
- [32] X. J. P. N. Wu Zhou, Yajin Zhou. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, number 317-326. ACM, 2012.
- [33] T.-c. C. Xin Hu and K. G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 611–620, 2009.
- [34] W. D. Yousra Aafer and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *Security and Privacy in Communication Networks*, pages 86–103. Springer International Publishing, 2013.
- [35] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy (SP)*, pages 95–109, 2012.