Scalable Function Call Graph-based Malware Classification

Mehadi Hassen
Florida Institute of Technology
150 W University Blvd
Melbourne, FL 32901, USA
mhassen2005@my.fit.edu

Philip K. Chan
Florida Institute of Technology
150 W University Blvd
Melbourne, FL 32901, USA
pkc@cs.fit.edu

ABSTRACT

In an attempt to preserve the structural information in malware binaries during feature extraction, function call graph-based features have been used in various research works in malware classification. However, the approach usually employed when performing classification on these graphs, is based on computing graph similarity using computationally intensive techniques. Due to this, much of the previous work in this area incurred large performance overhead and does not scale well.

In this paper, we propose a linear time function call graph (FCG) vector representation based on function clustering that has significant performance gains in addition to improved classification accuracy. We also show how this representation can enable using graph features together with other non-graph features.

Keywords

Malware Classification, Graph Classification

1. INTRODUCTION

Anti-malware vendors receive large numbers of files to be examined on a daily basis. For instance, Microsoft's real-time detection anti-malware products generate tens of millions of daily data points that need to be analyzed on a daily basis. The reason behind this huge influx of malware samples is that in an effort to avoid detection, malware authors constantly modify and/or obfuscate what would have been otherwise similar malware samples so that they look like many different files [3]. We refer to these similar samples as belonging to a single malware "family".

The large amount of malware makes it impossible to have human experts analyze all of these files. Hence, the use of machine learning based approaches can be very helpful in combating the malware epidemic.

One of the ways in which automated machine learning based techniques can be used is to group malware samples into groups and identify their respective families. Doing so enables human analysts to focus their attention and analyze fewer representative samples from each family. Hence, machine learning methods for categorizing these samples into groups that contain similar malware samples are a necessity.

Various machine learning based approaches have been proposed in past research works. Many of these approaches rely on features extracted using static analysis of the malware samples. These features range from simple features, such as the ones based on strings found in malware binaries, to more complex features based on function call graph (FCG) representation of the malware binaries. FCG based features preserve the structural information in malware code in the form of functions and the caller-callee relation between them. Past research efforts that used FCG based features incurred performance overheads introduced as a result of evaluating the similarities between call graphs.

In this paper, we propose a malware classification method that groups malware samples into malware families. Our method is based on FCGs, but unlike past works, we overcome the performance overhead associated with the FCG based approach by using a novel technique to convert FCG representation into a vector representation. Our proposed approach has the following advantages:

- It is faster compared to previous works for FCG based malware classification.
- It has a higher classification accuracy compared to previous works.
- The graph feature vector, extracted from FCGs, can be easily combined with other non-graph features.

We will start by first reviewing related research works in Section 2. In Section 3, we will describe the details of our approach and implementation. And finally, we will evaluate our approach and compare its efficiency and effectiveness with previous research works in Section 4.

2. RELATED WORKS

Graph-based features have been used in many research works for malware clustering and classification. The main attraction of graph-based features is that they preserve information on how different parts of the malware code interact.

There are many types of graph information that can be extracted from malware samples: FCGs, control flow graphs and system-call dependency graphs. FCG is a directed graph representation of code where the vertices of the graph correspond to functions (procedures), and the edges represent

CODASPY'17, March 22-24, 2017, Scottsdale, AZ, USA
© 2017 ACM. ISBN 978-1-4503-4523-1/17/03...$15.00
DOI: http://dx.doi.org/10.1145/3029806.3029824

the caller-callee relation between the functions (vertices) [24]. FCGs are usually constructed from disassembled binary code using static analysis. Various research works [14, 17, 27, 19, 18] have used FCGs to extract features for malware classification, indexing and clustering.

When representing malware as FCGs, be it for classification, indexing or clustering, the fundamental question that needs to be addressed is that of measuring graph similarity. Graph edit distance (GED) is one such metric that is used in various domains for measuring the dissimilarity between two graphs by providing a measure which quantifies the minimum amount of edit operations that need to be performed to transform one graph into the other. The appealing aspects of this metric is its customizability (flexibility), which it provides in the form of vertex and edge related edit distance costs that can be defined to incorporate domain knowledge [23]. However, exact computation of GED has exponential time complexity in the number of vertices. Hence, approximations of the metric are used.

As mentioned earlier, one needs to define cost functions to be used for calculating GED. In [17, 19], the authors define GED in terms of three cost function: vertex insertion/deletion cost, edge cost, and vertex relabeling cost. In their approach, before computing GED, a filtering step is applied to remove most similar function pairs from the two graphs, based on the Jaccard Index of the function instruction frequency vector. Then a random bipartite graph mapping of the remaining vertices between the two graphs is constructed. Simulated annealing [26] is then applied to find a bipartite mapping to approximate graph edit distance. Finally, the approximated GED is used in [17] to perform malware clustering.

In [14], a malware database management system is implemented that indexes malware samples based on their FCG similarity using on approximate GED. Similar to previously discussed works, they also approximate GED by finding minimum cost bipartite graph mapping between the vertices of the two graphs. In their case, however, they used the Hungarian Algorithm to find this mapping. The Hungarian algorithm finds the minimum cost of a bipartite mapping based on an input cost matrix. This matrix specifies the cost of mapping a vertex (function) in one graph to a vertex in a second graph. In their implementation this cost is composed of Relabeling Cost, which accounts for the cost of matching functions, and Neighborhood Cost, which takes into consideration matches between the neighboring vertices as well. Because of the time complexity of the Hungarian algorithm, they filter out highly similar functions based on names for external functions, and based on the similarity of instruction mnemonic sequences for local functions.

There are other research works that do not use GED for measuring graph similarity. For example, [27] uses the normalized number of common edges between two graphs as a measure of similarity. The authors begin by first matching external functions based on their function names. Local functions are first matched based on their external function calls, and matching functions (vertices) are removed. The remaining functions are next matched based on their instruction opcode. Then, they match the still remaining functions based on whether the neighboring vertices are matched. Finally, the similarity metric is calculated as the number of common edges between the two graphs and normalized by the sum of number of edges in both graphs over two.

[11] is another example that used a different metric than GED to measure graph similarity. They approximate graph similarity via fixed point propagation. A fixed point between two graphs is defined as a two nodes (one each from graph) that can be determined to represent the same item in both graphs. Their algorithm starts from an initial fixed point and propagates to more fixed points by considering the neighboring nodes.

In [18], the authors use FCGs to extract new features to compute the similarity between two graphs. They start by extracting FCGs, where each function is represented in terms of size types of initial features. They proceed to learning a distance metric for each attribute type and an optimal vertex matching matrix that maximizes between-class distance while minimizing within-class distance. Finally, they train classifiers for each feature type and combine the results in an ensemble classifier.

In [12], the authors propose a way to map function call graphs (FCGs) to vector form inspired by linear-time graph kernel[13]. First, FGSs are extracted from android APK files. To label the graph vertices, instructions are grouped into 15 categories. A 15-bit vector is used to label the vertices, which indicates the presence or absence of these instructions. This label is further changed to the neighborhood hash[13], which is computed as bit-wise XOR of the vertices 15-bit vector and all of it's successors vertices. Finally, the neighborhood hash of the complete graph is obtained by calculating hashes for each node individually and replacing the original labels with the calculated hash values. The graphs are represented as multiset of these hash values. This work presents a way of representing these graphs as feature vectors such that the inner product between the two feature vectors is equal to the multi-set intersection of the two graphs.

The difference between [12] and our work arises primarily from the way we label the FCG. In our case, we label vertices by cluster-id of the function clusters. This allows us to represent functions in more detail because all instructions, as well as the sequence of the instructions, are encoded. It also allows us to control the granularity of this labeling by controlling the number of clusters. Secondly, our vector representation explicitly encodes graph edges, hence preserving more information.

System-call dependency is another type graph representation of a malware. A system-call graph is a graph representation where the vertices correspond to system calls made during the execution of the malware, and edges represent data-flow dependency between system calls, usually determined by dynamic taint analysis. In [20], the authors extract this graph from a malware sample using dynamic taint analysis and then convert it into a smaller graph where the vertices represent a group of system-calls that serve a similar purpose and the edges represent the dependency between these groups based on the dependency between individual system calls. Once this representation is extracted, they define different similarity metrics for detection and classification.

3. APPROACH

There are various ways to extract features through static analysis, for the purpose of classifying malware using machine learning. Previous research works have extracted features from printable strings in malware binaries [25, 16],

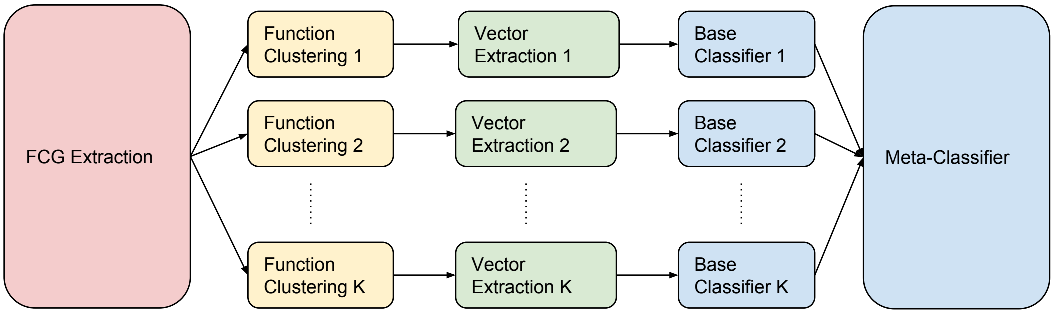


Figure 1: System overview with K pipelines

from the length of the functions in disassembled file [16], from instruction n-grams [15, 28, 21], FCGs [14, 17, 27, 19, 18], or a combination of different features [4].

In this paper we chose to focus on features generated from function call graphs (FCGs). The main reason for this is that FCGs better preserve structural information in binaries, for instance, compared to n-gram features. In addition to containing information about the malware code in the form of functions and their code, they also contain information about the interaction between the functions. The details of FCG extraction are discussed in Subsection 3.1.

One of the challenges with using FCG is that the names of the local functions (the functions written by the program author) are lost during compilation; hence, the vertices of FCG corresponding to these functions are unlabeled. This makes it difficult to compare two functions in different FCGs. Our solution to this problem is to cluster functions based on their instruction sequence and use these cluster-ids as labels for the functions. One of the concern we had about clustering functions was that we would lose some information as multiple functions get hashed to same cluster. Even thought this is true, our results show that classification accuracy is still very high. Function clustering is discussed in Subsection 3.2.

There are multiple ways of comparing the labeled FCGs, such as graph edit distance [17, 14], fixed point propagation [11], etc. In our approach, in addition to being able to efficiently compare FCGs, we also wanted to be able to have the capability for integrating non-graph features when needed. So unlike many past research efforts that classify FCGs, our approach first converts a FCG into a feature vector and then applies machine learning algorithms on these vectors, as discussed Subsection 3.3.

A high level view of our approach is shown in Figure 1. Our system starts by first extracting FCG representations from disassembled malware binaries. Once a FCG is extracted, the functions (which are the graph vertices) are clustered using Locality Sensitive Hashing(LSH) based on the function's instruction opcode sequence. The FCG vertices are then labeled using the cluster-ids. After labeling the FCG, our system extracts a vector representation from the call graph, which will serve as the feature vector of the malware sample.

During function clustering, some functions that are similar might be grouped into different clusters due to the randomized nature of LSH functions. To address this shortcoming, during function clustering, graph labeling and extraction of vector representation is done K times. Each of these vector representations, extracted in parallel, are given as input to a separate classifier, which we refer to in Figure 1 as a base classifier. This is done both during training and test. The predictions of the base classifier are further used as input features for the meta-classifier. We will describe the functionality of each module in the following subsections.

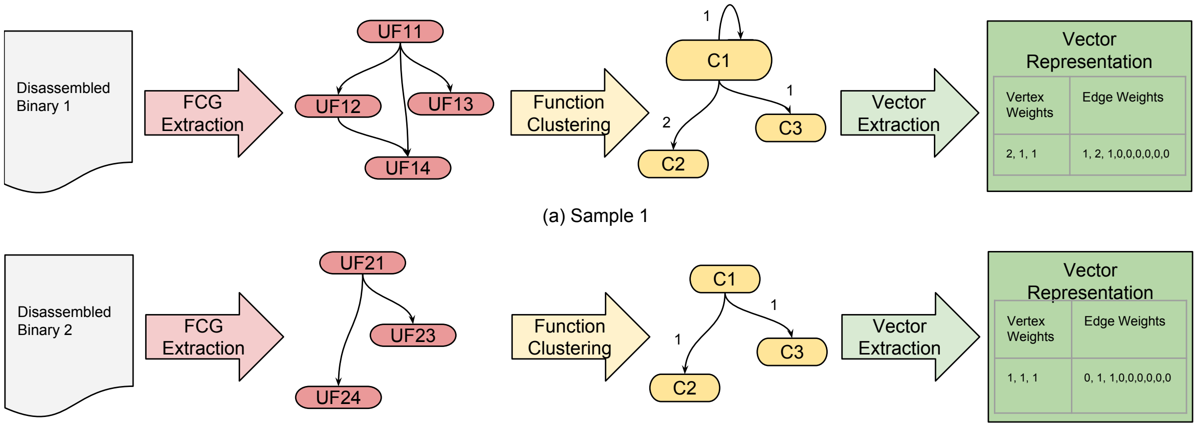
3.1 FCG Extraction

There is much structural information that gets lost when features are extracted from a malware binary. For example, when extracting features, such as instruction n-gram, the organization of malware code into different functions, and the interactions between these functions is not captured in the extracted feature. So in an attempt to preserve the structural information, we use FCGs to represent malware binaries.

A FCG is a directed graph representation of code where the vertices of the graph correspond to functions and the directed edges represent the caller-callee relation between the functions (vertices) [24]. The vertices in this graph can represent local functions defined in the malware code or external functions imported from libraries.

As shown in Figure 1, the first module of our system takes disassembled malware binaries and extracts FCG representations. When extracting FCG, we label vertices of external functions with the function names. The original names of local functions are not preserved during compilation and eventually disassembly. Even if they were, these names may not represent well the instruction sequences that the function implements. Therefore, we leave the vertices corresponding to local functions unlabeled for now. Vertices representing local function also contain the instruction opcode sequence of that function. We represent the caller-callee relation between functions as directed, unweighted edges. After extracting FCG we pass this graph to the next module for clustering the local functions (vertices) and relabeling them with their cluster-id.

To help explain the different modules, we will use a toy example shown in Figure 2. In this example, we assume we have two disassembled binary files. These files are given as an input to the FCG Extraction module, which converts the two samples into a FCG representation. The first sample has FCG consisting of 4 functions: $UF11$, $UF12$, $UF13$ and $UF14$. The second sample has 3 functions $UF21$, $UF23$ and $UF24$. The function names, for the local function, at this point are arbitrary names. Hence, it is difficult to compare



(a) Sample 1

(b) Sample 2

Figure 2: Example

the two graphs. We will discuss our solution to make the comparison easier in the next section.

3.2 Function Clustering

As discussed in the previous section, local functions in FCG are unlabeled. This makes comparing two FCGs very difficult. Our solution to this problem is to cluster the functions (vertices) of FCG and label them with a cluster-id. To perform clustering, we need to have a way of identifying similar functions. In [14], for instance, researchers use the instruction mnemonic sequence edit distance to calculate the similarity between functions which do not have similar names or CRC values of their mnemonic sequence. Even though edit distance might be a good measure of similarity, it has $O(n^2)$ time complexity in the number of instructions. In our work, we try to alleviate this bottleneck using Locality Sensitive Hashing(LSH) for computing an approximate edit distance between the instruction sequences of two functions.

The challenge here is that there are no locally sensitive family of hashes, that we know of, for approximating the edit distance. To address this, we explored the possibility of approximating edit distance with Jaccard Index. We evaluated the effectiveness of Jaccard Index in approximating edit distance by carrying out experiments where we computed the similarity between opcode sequences of functions using edit distance as well as Jaccard Index and computed the Pearson's correlation coefficient between the two. For Jaccard Index, we represent the functions as a set of uni-gram, bi-gram, or trig-ram opcode. Out of the three, Jaccard Index of uni-gram opcodes had the highest Pearson's correlation coefficient with edit distance at 0.957.

Using Jaccard Index to approximate edit distance, however, is still not fast enough. Fortunately, there is a family of locality sensitive hash functions, commonly known as Minhash [7], that can be used to approximate Jaccard Index, allowing us to efficiently approximate the similarity between functions.

Minhash is a LSH technique based on the idea that a hash of a set is the index of the first element under a random permutation. Then the probability that two sets will have the same index value of the first element is equal to the Jaccard similarity between the two sets. However, generating these permutations is computationally expensive. Hence, in practice, the random permutations are approximated by computing the Minhash of a set S as:

$$h(S) = \min_{\forall x \in S}((c_1 * x + c_2) \mod P) \quad (1)$$

where

- x is the index of an element in set S w.r.t. the super set of S;
- c_1 and c_2 are constants and chosen at random for each hash function;
- P is a prime much larger than the size of the universal set of all sets [22].

To improve the precision of the Minhash signature, L hash functions in Equation 1 are used together to generate a single Minhash signature by concatenating the values from the different hash functions. However, it is still possible to have false negatives (i.e. sets that are highly similar but declared to be not similar by Minhash). Multiple runs of Minhash can be used to address this [10].

```
Algorithm 1: Function Clustering
Input  : G: Function call graph. Where functions are
         represented in terms of their instruction
         opcode sequence.
         [$h_1, \ldots, h_L$]: $L$ hash functions, in Equation 1,
for generating Minhash signature.
Output: Function call graph labeled with function
         cluster ids
foreach internal function $v$ in G.vertices do
    signature ←
    minhashSignature($v$.ngramOpcodeSequence,
    [$h_1, \ldots, h_L$]);
    clusterId ← hash(signature);
    $v$.label ← clusterId;
return G;
```

In our systems case, instead of using Minhash to directly calculate the similarity between functions, we use it to cluster functions (vertices) of the given graph. Unlike normal clustering where we would have needed to calculate the similarity between each pair of functions, using LSH we can simply hash the functions into buckets which represent clusters. Logically, we can consider this process as if we are clustering all functions of all FCGs. In reality, however, our use of LSH allows us to cluster the vertices in one FCG without the need to look at other FCGs in an efficient way.

The pseudo-code for our implementation is shown in Algorithm 1. For each vertex, we compute the cluster-id by first generating a Minhash signature for the n-gram opcode sequence. The function *minhashSignature*, in line 2, takes the n-gram opcode sequence and a list of hash functions, defined in Equation 1, as input and generates the Minhash signature. We represent this signature as an array of L hash values for the n-gram opcode sequence computed using the given L hash functions. Note that Minhash operates on set inputs rather than a sequence, as is the case in line 2, where the input to Minhash is an opcode n-gram sequence. One way to convert the n-gram sequence into a set representation is to have a universal set that contains all the n-grams observed in our training samples, and then the individual functions represented as subsets of this universal set. However, to speed up computation in our implementation we take the hash of an opcode n-gram and use this value as the index of the opcode n-gram when calculating the Minhash. The hash function used to compute the index value can be any hashing function with uniform value distribution and a large range of hash values to minimize collisions. In our case, we use murmurhash [5].

Next, these Minhash signatures are further hashed using an ordinary hash function to compute the cluster-id for that vertex (function) and this cluster-id value is used to label the vertex, as shown in lines 3-4. This secondary hashing also allows us to control the number of clusters.

During our implementations, we also experimented with the use of different hash functions for the secondary hash. As mentioned earlier, a hash function that uniformly distributes its keys across buckets is used. However, we thought it might make better sense to have hash collisions when the Minhash hash signatures closely match. So we experimented with using Simhash [9] as the secondary hash function. Our experiments, however, revealed that although using Simhash did improve the classification accuracy of a single base classifier, the hash function which uniformly distributes its keys achieves better accuracy on the overall meta-classifier. Therefore, we decided to use an ordinary hash function for the secondary hash.

At this point the FCGs, labeled by the cluster-ids, can be logically viewed as a graph where the vertices are clusters and the edges are calls made from functions in one cluster to a function in another cluster, or even in the same cluster. This logical representation is shown in Figure 2. However, in actual implementation we simply label the the vertices of input FCG with the cluster-ids.

In our running example in Figure 2, the FCGs extracted by the FCG Extraction module are given as input to the current module. In this module a Minhash signature is generated for each function in the input FCG, a cluster-id is determined using this signature and the function (vertex) is labeled by this id. In case of sample 1, we assume, functions $UF11$ and $UF12$ are hashed to cluster $C1$, $UF13$ is hashed to cluster $C3$, and $UF14$ is hashed to cluster $C2$. In the case of sample 2, we assume functions $UF21$, $UF23$ and $UF24$ are hashed to clusters $C1$, $C3$ and $C2$, respectively.

The resulting graphs shown in Figure 2 are logical view of FCG after labeling. This view shows a graph where the vertices are clusters and the edges are calls made from functions in one cluster to a function in another cluster, or even the same cluster. Now the labeled graphs of sample 1 and 2 are much easier to compare.

3.3 Vector Extraction

In past research works, techniques used for computing graph similarity have been a source of performance bottleneck on FCG based malware classification. Not only were these a performance bottlenecks, but they also made it difficult to integrate non-graph features with graph features. Motivated by these two aspects, we proposed a vector representation of FCGs.

Algorithm 2: Creating Vector Representation

```
Input : G: Function call graph labeled with function
        cluster ids;
Output: Graph vector representation
1 Initialize vertexWeight with zero vector;
2 Initialize edgeWeight with zero vector;
3 foreach v in G.vertices do
4     vertexWeight [v.label ] += 1;
5 foreach e in G.edges do
6     index ← EdgeIndex(e.source.label, e.target.label);
7     edgeWeight [index] += 1;
8 graphVector ← concatenate vertexWeight with
  edgeWeight;
9 return graphVector;
```

We extract vector representation from a FCG labeled using the cluster-ids. This representation consists of two parts, vertex weight and edge weight. The vertex weight specifies the number of times a vertex with a given label (cluster-id) is found in a FCG, or in other words the number of vertices in each cluster for that FCG. The edge weight specifies the number of times an edge is found from a vertex in one cluster to a vertex of another cluster or a vertex within the same cluster.

As shown in Algorithm 2, we start by initializing *vertexWeight* and *edgeWeight* vectors to zero vectors. In lines 3-4, we iterate over all the vertices in the input graph and count the number of vertices labeled with each cluster-id. Next, in lines 5-7 we compute edge weights for the edges between cluster-ids (i.e., an edge from a vertex labeled with one cluster-id to a vertex labeled with another cluster-id) or within a cluster (i.e. an edge between two vertices labeled with the same cluster-id). To do so we iterate over each edge in the input graph and update the frequency of occurrence edges. In our implementation, we use the *EdgeIndex* function to perform a simple lookup for the index representing the edge type. Finally, we concatenate *vertexweight* and *edgeWeight* vectors to form one vector representing the input FCG.

As mentioned in the Section 3.1, vertices of the FCG corresponding to external functions are already labeled by the name of the external functions. While computing the vertex weight we can use a lookup table to map the external function names to index values in the vertex weight vector that correspond to that external function. This requires first identifying all external functions observed in training dataset samples and adds an additional processing. To avoid this, we simply use an arbitrary hash which uniformly distributes its keys to relabel the external functions with this value and use this to as the labels refereed in lines 4 and 6 in Algorithm 2. The same applies for the edge weights. Through our experiment we didn't notice significant decrease in classification accuracy as a result of this hashing. Therefore, we decided to use this approach to avoid processing overhead.

We acknowledge that our vector representation does not preserve every detail of a graph structure. For instance, it is possible to have slightly different graphs with the same vector representation. However, we believe that this can have its own advantage in the field of malware classification by making this representation less susceptible to obfuscation techniques that might change function calling patterns. That is because small variations in the graph structure might not get expressed in the vector representation as long as the edge and vertex frequencies are not changed. Hence, this representation becomes more resilient to changes such as reordering of function calls.

In our running example, in the case of sample 1 for instance, the labeled FCG contains two functions that are labeled $C1$, one $C2$, and one $C3$. The vector representation of the vertex weight part will be $2, 1, 1$. The second part represents edge weights. In sample 1, we have one edge $C1 - C1$, two $C1 - C2$ and one $C1 - C3$. The vector representation of the edge weight part will be $1, 2, 1, 0, 0, 0, 0, 0, 0$. The zeros in the edge weight show that there are no edges, for example, $C2 - C2$. The final vector representation will be a concatenation of these two vectors.

Once the vector representation of all instances is created, the next step is to train models for classification of the malware samples.

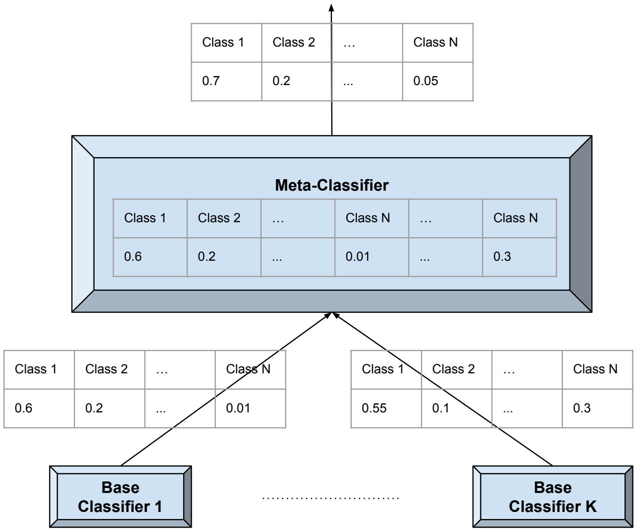
3.4 Base Classifiers

The function clusters generated as discussed in Section 3.2 has high precision but lower recall. That is, each cluster has similar functions, but some similar functions may be hashed into different clusters. Recall can be improved by repeating this clustering step. In our system, K repeated clustering steps are run independently and in parallel. For each run, a separate vector representation is computed. Then separate base classifiers are trained using the output of each run.

Each of our base classifier is a Random Forest classifier [6]. The inputs to a base classifier are the vector representations for the malware FCGs. As shown in Algorithm 3, the input dataset, which is a list of vector representations for the malware FCGs, is segmented into T parts (line 2). Then for each data segment, a classifier is trained on the remaining data segments and used to predict, in the form of a probability distribution of over all classes, for each sample in the segment (lines 3-7). This is used to reconstruct the data set in terms of the predictions of the based classifier.

Algorithm 3: Training base classifier and creating training data for meta-classifier

```
Input  : D: Training set containing FCGs vector
         representation
Output: $D_{new}$: The training set represented in terms
         of class distributions.
1 Initialize $D_{new}$ to empty list;
2 Segment D into T parts;
3 for $t = 1$ to T do
4     Train classifier C on D $-D_t$;
5     foreach Sample d in $D_t$ do
6         $distribution_d \leftarrow$ C.predict(d);
7         Add $distribution_d$ to $D_{new}$;
8 return $D_{new}$;
```


Class 1	Class 2	...	Class N
0.7	0.2	...	0.05

Class 1	Class 2	...	Class N	...	Class N
0.6	0.2	...	0.01	...	0.3

Class 1	Class 2	...	Class N
0.6	0.2	...	0.01

Class 1	Class 2	...	Class N
0.55	0.1	...	0.3

Figure 3: The meta-classifier

3.5 Meta-Classifier

The meta-classifier combines the predictions of the individual base classifiers to output a final prediction. It first receives the predictions for each data instance from the based classifiers. These predictions are in the form of probability distribution over the class labels. Then for each instance, these probability distributions are concatenated to form a single feature vector. In other words, the prediction from the base classifiers serve as an input feature to the meta-classifier. The meta-classifier is then trained on this vector. The output of the meta-classifier is a predicted class label for each data instance as illustrated in Figure 3.

3.6 Enhancements

For a meta-classifier to be effective, we need the base classifiers to be sufficiently different from each other [8]. In the case of our original design presented so far, the difference between the base classifiers comes as a result of the false negatives introduced by function clustering with LSH. Through our experiments, we were able to determine that the different runs of the Function clustering using a same number of Minhash functions(L) were not producing enough variation to take full advantage of ensembling. So to introduce more variation, we used different values for L in Function clustering for the K different runs.

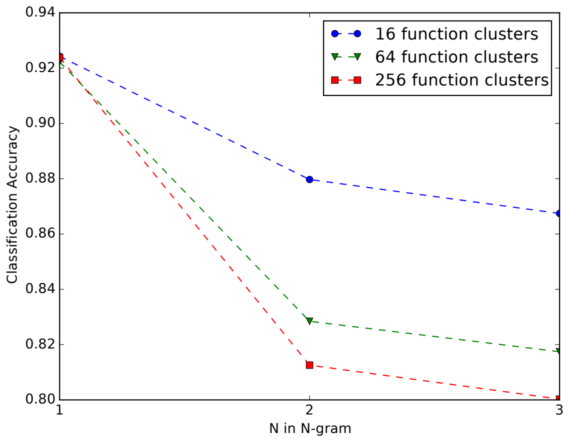


Figure 4: Effect of n-gram length on classification accuracy

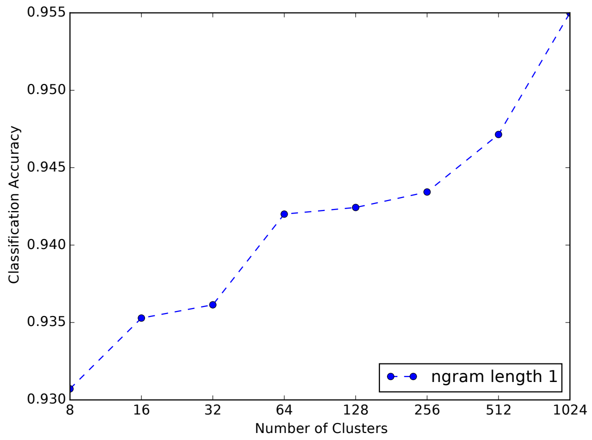


Figure 5: Effect of number of function clusters on classification accuracy

4. EXPERIMENTAL EVALUATION

4.1 Dataset

For the purpose of evaluating our proposed approach for classifying malware into families, we will be using the Microsoft Malware Classification Challenge (BIG 2015) dataset [3]. The original dataset consists of 10,867 labeled malware samples. Our disassembled file parser were able to properly parse 10,260 of the samples. Hence, we will be using these in the following evaluations. The class distribution of these samples are shown in Table 1.

To compare our work with Adagoi [12], we will use a secondary dataset consisting of 1,113 benign android apps and 1,200 malicious android apps. The malicious samples are from the Android Malware Genome Project [2]. A colleague at our university provided us with the benign samples, downloaded from the Google Play Store.

Table 1: Microsoft malware dataset class distribution

Malware Family Name	Number of Samples
Ramnit	1513
Lollipop	2470
Kelihos ver3	2936
Vundo	446
Simda	34
Tracur	294
Kelihos_ver1	387
Obfuscator.ACY	1168
Gatak	1012

4.2 Parameter Selection

The first two parameters of our algorithm that need to be configured experimentally are the length of instruction opcode n-grams, and the number of function (vertex) clusters. N-gram length determines how many instructions are used in each n-gram when representing local functions as sets of n-grams; in other words it determines the n in n-gram. Cluster number determines the number of buckets functions are hashed into.

Figure 4 shows the classification accuracy results when using uni-gram, in both the n-gram length and cluster number experiments, only a single base classifier was used. In the Figure 4, using uni-grams results in better accuracy than both bi-grams and trig-rams and that using bi-grams results in better accuracy than tri-gram. This result can be explained by going back to the distance function (we are trying approximate function similarity in our approach.) As discussed in Section 3.2, we are trying to approximate edit distance using Jaccard Index, which we in turn approximate using Minhash. When computing edit distance, the insertion and deletion operations work on uni-grams. So the longer the n-gram we use when approximating edit distance with Minhash, the less accurate our approximation.

When it comes to the number of function clusters, we expect the classification accuracy to increase as the number of function clusters increases. This expectation can be understood by considering the fact that by hashing function into smaller number of clusters, we increase the likelihood of dissimilar functions being hashed into the same cluster. Obviously, this in turn results in lower classification accuracy. The results in Figure 5 agree with our expectations. The experiments to determine the number of function clusters were carried out using uni-gram opcode sets to represent functions and clustering the functions (vertices) into the specified number of clusters before converting to vector representation.

The other parameters that need to be tuned experimentally are the number of base classifiers(K), and the number of Minhash functions used for computing the Minhash Signature (L). In our experiments, we evaluated various values of K. We observed that in the case the current evaluation dataset increasing the value of K above 6 didn't result in much improvement. As discussed in Section 3.6, using different values for L for the K different runs, results in more variation among the K base classifiers and results in better meta-classifier performance. In our experiments we evaluated a couple of L value combinations and picked the one that resulted in better performance.

4.3 Classification Accuracy Comparison

We evaluate our approaches accuracy by comparing it with our implementation of Kinable et.al. [17, 19] work on FCG classification. In [17], they use an approximate GED to measure the similarity between FCGs. GED distance is approximated using Simulated Annealing (SA) to find a bipartite graph mapping between the vertices of the two graphs that minimizes the GED. Since this is a computationally intensive task, functions in the two graphs that have instruction edit distance greater than some threshold τ are filtered out before creating the bipartite graph mapping. Then the FCGs are clustered based on GED.

Since using SA to approximate GED was computationally intensive, we evaluated both our proposed method and our implementation of Kinable et.al. work on a smaller sub set of the dataset consisting of 1000 malware samples. In our implementation of Kinable et.al. research, we filtered out most similar functions between pairs of graphs by setting the filter threshold τ to 0.9 and then created bipartite mapping on the remaining functions (vertices). We cluster the malware samples using k-medoids and then assigned a class label to each cluster based on the majority class. Hence, we determined every element of a cluster whose class label is different from the majority class as being misclassified. Figure 6 shows the confusion matrix of classifying the thousand samples in this way.

Next we evaluated our approach on the same 1000-malware sample with 10-fold cross validation to predict the labels on the thousand samples. Our system is set-up using six base classifiers(i.e. $K = 6$), which means that there are six parallel pipelines consisting of the Function clustering, Call graph vector extraction, and Base classifier modules. In all the of Function clustering modules, we represented functions (vertices) with uni-gram opcode sequence and clustered them into 64 clusters. To make the base classifiers more decorrelated, the Minhash signatures are generated using varying number of hash functions (i.e. value of L in Algorithm 1). In first two of the six pipelines $L = 1$, in the next two $L = 3$ and in the last two $L = 5$.

As can be seen from the results in Figures 6 and 7, our approach clearly out performs the SA based method with an overall accuracy of 0.979 versus the 0.840 of the previous works on the smaller dataset. Our system has high accuracy for almost all of the malware families, apart from the malware family Simda. The reason behind the poor accuracy in the case of Simda is the very small number of training samples. In the entire evaluation dataset there are only 34 instances, and in the case of the sampler 1000 instance dataset used in this section there were only 4 samples.

Figure 8 shows the classification accuracy of our approach when applied on the entire dataset using a 10-fold cross validation. Again we see that our system gets near perfect classification accuracy for all families except Simda. Which is again due to the smaller number of training samples.

4.4 Speed Comparison

We will now empirically compare the speed of our approach with that of SA based approach. For our approach, we timed the execution in the components starting with Call Graph Extraction to Call graph Vector representation (inclusive). We excluded the time taken by the learning

algorithm. For the SA based approach we timed the process starting with Call Graph Extraction up to the GED com-putation between all pairs of malware samples, using SA. Similarly here, we excluded the time taken by the learning algorithm.

Table 2: Speed comparison

Approach	Average Time (min)
All pair similarity SA and GED	2006
Graph Vector Representation	7

The results in Table 2 show these time measurements, averaged over three runs, on a smaller subset of the evaluation data set consisting of 1000 samples. As we expected, our approach is significantly faster than the SA based technique making FCG classification scalable. These experiments were carried out on a machine with 2.3 GHz quad code CPU with 8 GB memory.

Time Complexity Comparison: To compare the training time complexity of these two approaches we will only consider the feature extraction phase in both methods and exclude the machine learning algorithms.

For the Simulated Annealing based approach, we consider the time complexity of graph similarity computation and exclude the clustering part when looking at the time complexity. As shown in [19], the time complexity of running Simulated Annealing to approximate Graph Edit Distance between two graphs is $O(|V_{max}|^2 \cdot d_{max})$, where $|V_{max}|$ is the number of vertices of the largest graph, and d_{max} is the maximum value of degree for any node. For N graphs then $O(N^2)$ distance (similarity) computations are going to be required. Therefore, for N graphs the worst case time complexity becomes $O(N^2 \cdot |V_{max}|^2 \cdot d_{max})$.

In case of our system we consider time complexity of the Function clustering and Vector extraction modules. When we look at the Function clustering module, it is clear from Algorithm 1 that given an input graph $G = (V, E)$, it has time complexity of $O(|V|)$. For N graphs then the worst case time complexity becomes $O(N \cdot |V_{max}|)$, where $|V_{max}|$ is the number of vertices of the largest graph in the training dataset. For the Vector extraction module, we visit each vertex and each edge only once. Hence, the complexity of the module for a given input graph $G = (V, E)$ is $O(|V| + |E|)$. For N graphs then, the worst case time complexity becomes $O(N \cdot (|V_{max}| + |E_{max}|))$. Finally, the total worst case time complexity of the two modules together for N graphs is $O(N \cdot (|V_{max}| + |E_{max}|))$. So it can be clearly seen here that our approach compares favorably to the one based on Simulated Annealing.

4.5 Classifying benign and malware applications

In this section we will compare our approach with another closely related work, Adagoi [12]. We used the authors implementation of Adagoi [1], which performs classification between benign and malicious Android apps. To work with android apk files, we modified the implementation of the FCG Extraction module, in our system.

The evaluation was performed by randomly splinting the initial dataset to use 80% as training data and the remaining as test data. This was repeated 10 times, and the resulting average ROC curve is shown in Figure 9. As seen in the figure, our approach performs better than Adagoi. For instance, at 0.01 false positive rate, the area under the curve for our approach is 0.0099 where as for Adagoi it is 0.0092.

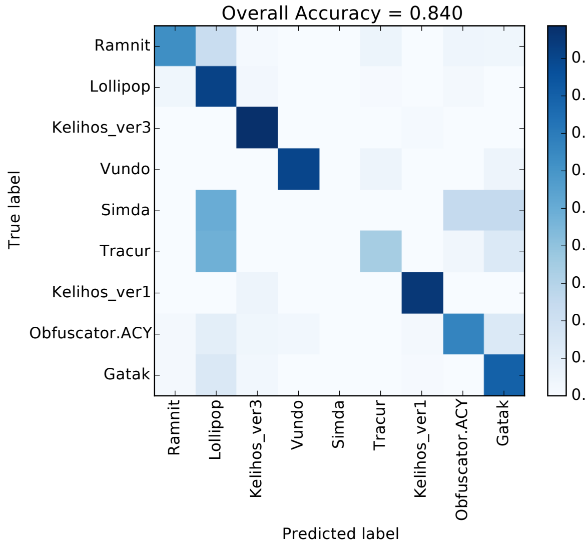


Figure 6: Confusion matrix for using Simulated Annealing on 1000-sample dataset

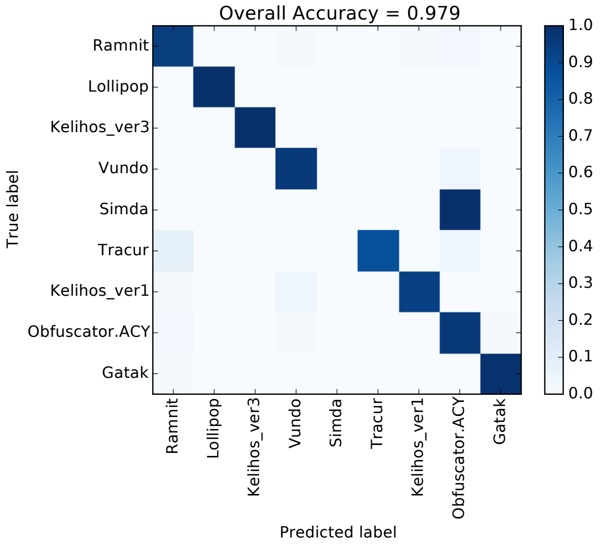


Figure 7: Confusion matrix using meta-classifier on 1000-sample dataset

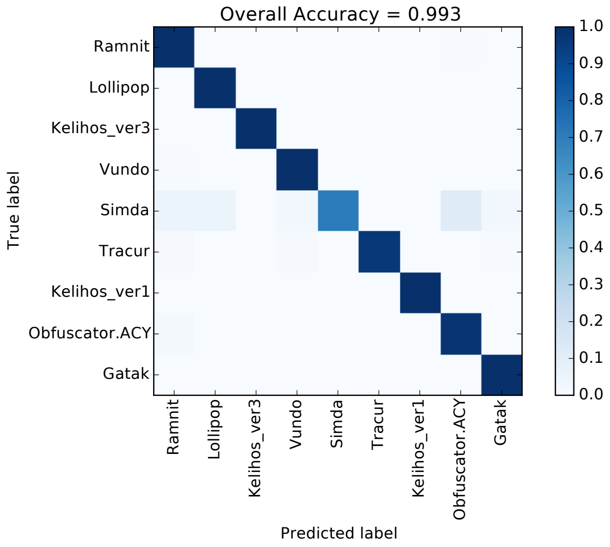


Figure 8: Confusion matrix using meta-classifier on entire dataset

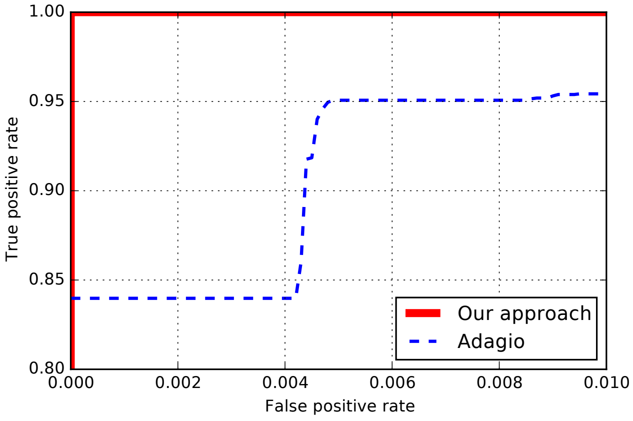


Figure 9: ROC curve

4.6 Combining with Non-graph Features

The former FCG based techniques, which relied solely on graph representation and on measuring graph similarity, were not easy to use together with other non-graph based features. Our proposed approach, on the other hand, extracts feature vectors from FCGs that can be easily used with other features by simply concatenating the feature vectors.

To demonstrate this, we use binary byte bi-gram frequency features. These features are extracted by computing the frequency of byte bi-grams in malware sample raw binary files. We take these feature vectors and concatenate a random subset of them with the graph vectors before passing them to the base classifiers. The reason for taking a random subset of these features is to help make the base classifiers more decorrelated.

Combining these features with the graph features, our overall classification accuracy slightly increased to 0.9933 from 0.993. This can be explained by the fact that both the graph and the n-gram frequency features are have overlaps in that they both are extracted based on the content of malware binary. As a future work we would like to examine combining our FCG features with the various features used in [4].

5. CONCLUSIONS

In this paper we presented a fast and effective malware classification system based on extracting feature vector from FCG representation. Our approach was able to address two bottlenecks in previous malware classification systems that were based on FCG features. First, we were able to speed up the process of measuring similarity between functions using

Minhash, an Locality Sensitive Hashing technique. Second, we avoided the major bottleneck of computing graph similarity by converting the graph representation into vector representation using function clustering based on the Minhash signatures of the functions.

6. REFERENCES

[1] Adagio. https://github.com/hgascon/adagio.

[2] Android malware genome project. http://www.malgenomeproject.org/.

[3] Microsoft malware classification challenge (big 2015). https://www.kaggle.com/c/malware-classification, 2015. [Online; accessed 27-April-2015].

[4] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 183–194. ACM, 2016.

[5] A. Appleby. Murmurhash3. https://github.com/aappleby/smhasher, 2008.

[6] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[7] A. Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29. IEEE, 1997.

[8] P. Chan. *An Extensible Meta-Learning Approach for Scalable and Accurate Inductive Learning*. PhD thesis, Department of Computer Science, Columbia University, New York, NY, 1996.

[9] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM, 2002.

[10] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th international conference on World Wide Web*, pages 271–280. ACM, 2007.

[11] T. Dullien and R. Rolles. Graph-based comparison of executable objects (english version). *SSTIC*, 5:1–3, 2005.

[12] H. Gascon, F. Yamaguchi, D. Arp, 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. ACM, 2013.

[13] S. Hido and H. Kashima. A linear-time graph kernel. In *2009 Ninth IEEE International Conference on Data Mining*, pages 179–188. IEEE, 2009.

[14] X. Hu, T.-c. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 611–620. ACM, 2009.

[15] X. Hu, K. G. Shin, S. Bhatkar, and K. Griffin. Mutantx-s: Scalable malware clustering based on static features. In *USENIX Annual Technical Conference*, pages 187–198, 2013.

[16] R. Islam, R. Tian, L. Batten, and S. Versteeg. Classification of malware based on string and function feature selection. In *Cybercrime and Trustworthy Computing Workshop (CTC), 2010 Second*, pages 9–17. IEEE, 2010.

[17] J. Kinable and O. Kostakis. Malware classification based on call graph clustering. *Journal in computer virology*, 7(4):233–245, 2011.

[18] D. Kong and G. Yan. Discriminant malware distance learning on structural information for automated malware classification. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1357–1365. ACM, 2013.

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

[20] S. D. Nikolopoulos and I. Polenakis. A graph-based model for malware detection and classification using system-call groups. *Journal of Computer Virology and Hacking Techniques*, pages 1–18, 2016.

[21] S. Pai, F. Di Troia, C. A. Visaggio, T. H. Austin, and M. Stamp. Clustering for malware classification. *Journal of Computer Virology and Hacking Techniques*, pages 1–13, 2016.

[22] A. Rajaraman, J. D. Ullman, J. D. Ullman, and J. D. Ullman. *Mining of massive datasets*, volume 1. Cambridge University Press Cambridge, 2012.

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

[24] B. G. Ryder. Constructing the call graph of a program. *Software Engineering, IEEE Transactions on*, (3):216–226, 1979.

[25] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 38–49. IEEE, 2001.

[26] L. Xu and E. Oja. Improved simulated annealing, boltzmann machine, and attributed graph matching. In *Neural Networks*, pages 151–160. Springer, 1990.

[27] M. Xu, L. Wu, S. Qi, J. Xu, H. Zhang, Y. Ren, and N. Zheng. A similarity metric method of obfuscated malware using function-call graph. *Journal of Computer Virology and Hacking Techniques*, 9(1):35–47, 2013.

[28] G. Yan, N. Brown, and D. Kong. Exploring discriminatory features for automated malware classification. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 41–61. Springer, 2013.