Paranoid Android:
Android Malware Classification Using Supervised Learning on Call Graphs

Mark Mendoza, Michael Zhu

December 11, 2017

1 Introduction

Malware design and detection is an eternal arms race of increasing sophistication. A new front has been recently expanded in the discipline of malware obfuscation and self-modification, seeking to fool the signature-based approaches dominant in commercial anti-virus software. In response, security researchers have been seeking to design methods to classify executables based on their semantic function rather than their syntactic contents. The most interesting of these approaches has been to apply the powerful tools of network analysis to the graph defined by the state transitions of an application: its control flow graph.

In this paper, we present our novel approach of Android malware classification based on supervised learning on the network properties of control flow graphs. Our program extracts the control flow of a given `.apk` using FlowDroid, then extracts a collection of network properties of the control flow graph using Snap.py, then retrieves an identification of that malware into a specific "family" from a XGBoost classifier, which we have trained on a labeled dataset. Using this approach, we were able to achieve upwards of 74% classification accuracy into specific families (different versions of a single exploit), and upwards of 98% classification accuracy into malware categories (e.g. Trojans, Adware, Ransomware).

2 Related Work

Because control flow is an inherently hard to obfuscate element of any program, there has been significant research attention paid to its application to malware detection.

2.1 Detecting Metamorphic Malwares using Code Graphs (Lee et al. 2010)

The first of these significant analyses came from Lee et al. The primary challenge addressed in this paper is that control flow graphs can get arbitrarily large, especially in highly obfuscated binaries, which absolutely kills the performance of the linear comparison methodology they were using to compare these graphs. Their approach to making these comparisons more computationally tractable took two steps.

The first of these was to only consider transitions between system calls, as most malware's actual payload impact must come from executing malicious system calls. However, there are still lots of possible system calls on modern systems, leading to still fairly large graphs. To address this, Lee et al. made a classification of most of the relevant system/API calls as being acting upon one of 32 different objects (like files, sockets, handles, registry entries etc.), with one of four different behaviors (open, read, write, close). Then the previously intractably large graphs can be easily reduced by unifying together all of the nodes in a given category (e.g. all of the various file writing system calls in the program would be collapsed to one node), keeping all of the connectivity. In this way, no matter how large the program, the "code graph" would never have more than $32 \times 4 = 128$ nodes. The corresponding adjacency matrix to this graph is the "semantic signature" they would go on to use for identification.

Identification was accomplished by calculating similarity scores between a new unknown binary and each of the existing "semantic signatures" in the library. This kind of matching is very simple to implement and very fast on these graphs of fixed size, but was surprisingly effective, defeating

the code insertion, code reordering and code replacement obfuscation strategies in their test set, unlike Norton, BitDefender and Mcafee.

We see this paper as a great proof of concept of the effectiveness of graphs as identifiers for obfuscated malware, but their reduction strategy is far too aggressive. By using a trained classifier instead of linearly comparing against the entire training corpus, we could afford to use larger feature sets. Furthermore, we used graph properties that far more comprehensively summarize the properties of a control flow graph without simply dropping the vast majority of nodes.

2.2 Malware Variant Detection Using Similarity Search over Sets of Control Flow Graphs (Cesare et al. 2011)

Cesare et al. approach the problem of malware detection from a significantly different angle. Although the classification algorithm used is similar to that used by Lee et al., the features are extracted from the control flow graph instead of the function call graph – the control flow graph is more granular than the function call graph in that its nodes is a basic block of code, rather than an entire function.

The paper also introduces two different ways of representing the graph as a feature vector. The first decomposes the graph into k-subgraphs, while the second takes the novel approach of converting the graph into a structured string and decomposing the string into q-grams.

By combining Lee et al.'s idea of node unification graph reduction with this paper's notion of the relevancy of k-subgraphs, we came up with the novel features of the 3 and 4-subgraph frequencies of the "API-Graph". We expand on what exactly this feature entails in our section on feature extraction.

2.3 Mal-Netminer: Malware Classification based on Social Network Analysis of Call Graph (Jang et al. 2016)

In this paper, Jang et al. demonstrated the applicability of social network analysis properties of control flow graphs as features for trained classifiers. Rather than calculate these graphs statically, Jang et al. instead inferred them by tracing the data dependencies between system calls captured by systracing the executions of their malware in a quarantined environment.

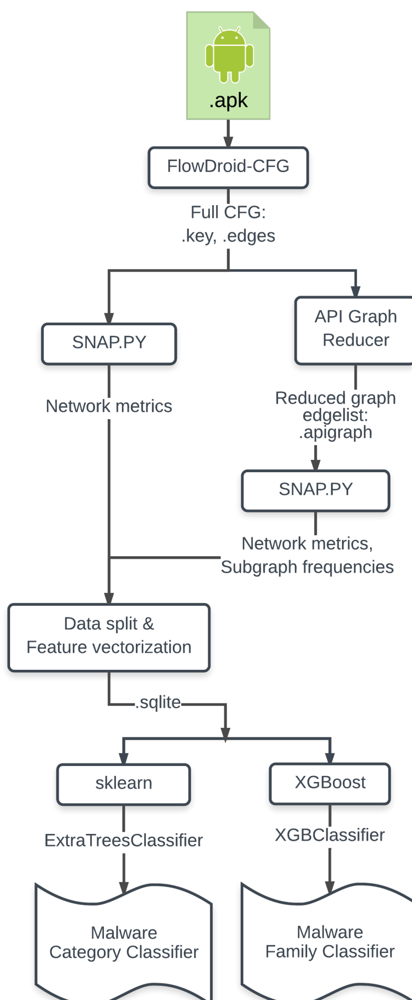
They found that malware binaries in the same family (i.e. obfuscated versions of the same piece of malware) shared similar degree distributions (primarily formulated as the proportion of nodes with degree greater than 1). Furthermore, the same extended to degree centrality/reachability and average distance metrics, but not to betweenness centrality, clustering coefficient, network density or component ratio. They then fed these successfully correlated metrics as features into a variety of classifiers, and had the most success with using the C4.5 decision tree algorithm, boosted with AdaBoost. Using this classification scheme, they were able to achieve above 96% accuracy.

These results were very exciting in both their absolute degree as well as how many opportunities for improvement and extension there still with this general approach. We saw the use of only system call graphs as an unnecessary limitation, as well as their reliance on dynamically recorded behavior. By using the statically analyzed full call graph, we got a much more complete picture of a potential piece of malware, allowing us more detail in our network analysis metrics. Furthermore, by using SNAP instead of GUI only Ucinet 6, we achieved much better feature extraction throughput and portability.

2.4 An Android Malware Detection Approach Using Community Structures of Weighted Function Call Graphs (Du et al. 2017)

The most technically similar paper to our approach is the results released by Du et al. this year, as they also used a graph based approach to evaluate Android malware. Instead of using the kind of network properties that we and Jang et al. used, they chose to base their analysis on the community structure surrounding instances of system calls that they had *a priori* categorized as being suspicious. They turned the function call graph into a weighted graph based on that suspiciousness quantity, and then used the Girvan-Newman algorithm to partition that graph into communities. From these communities, they calculated features like the number of community structures that contain malicious nodes, the maximum malicious behavior value for each community, and the sum of all behavior values for each community.

Figure 1: Experiment pipeline



These features are then fed into a supervised learning algorithm (BayesNet) to perform the binary classification task of identifying a file as malware or benign. The paper's classifier outperformed the antivirus softwares on most malware families, and had the highest average detection accuracy.

Our efforts expand on the achievements of Du et al. in a number of significant ways. Most notably, they did not try to apply their classification to the more complex problem of disambiguating malware into specific families, which is critical for developing threat intelligence as well as correctly mitigating the impact of an already contracted piece of malware. Furthermore, the accuracy of their analysis was conditioned on the accuracy of their *a priori* suspiciousness categorizations of system calls. This leaves any scanner implementing their approach vulnerable to malware variations adapting to use previously "innocent" system calls in a way that the creator of the suspiciousness list hadn't expected.

Table 1: Feature List

Features	Full CFGs	API Graphs
Number of Nodes	✓	✓
Number of Edges	✓	✓
Number of Self-Loops	✓	✓
Number of Unique Edges	✓	✓
Maximum In-Degree	✓	✓
Maximum Out-Degree	✓	✓
Number of Strongly Connected Components	✓	✓
Effective Directed Diameter	✓	✓
Effective Undirected Diameter	✓	✓
Ratio of Nodes With In-Degree 1	✓	✓
Ratio of Nodes With Out-Degree 1	✓	✓
Average Clustering Coefficient	✓	✓
Ratio of Edges In Triads	✓	✓
Density	✓	✓
Average In-Degree Centrality	✓	✓
Average Out-Degree Centrality	✓	✓
Average Closeness Centrality	✓	✓
Frequency of 3-Subgraphs		✓
Weighted Frequencies of 4-Subgraphs		✓

3 Method

3.1 Dataset

For our supervised learning, we needed a large set of live Android malware labeled into specific families. We found just that in the Android Malware Dataset available from Argus Labs, which contains over twenty thousand samples of malware classified into 71 different distinct families [WLR+]. There aren't any off-the-shelf control flow graph extractors available for Android, though there are a variety of data flow analysis libraries that can be retrofitted to extract the control graph for our purposes. We tried using both FlowDroid and Argus-SAF, from Paderborn University and Argus Labs, respectively [ARF+14][WRO+14]. Though Argus-SAF offers a wider variety of traversal strategies and more detailed inter-component data-flow tracking, their processing was orders of magnitude slower than that performed by Flowdroid, and also failed to extract a graph from a significant portion of our samples due to internal class resolution bugs [1].

We ran our Flowdroid-based graph extractor with a simple Python `multiprocessing` wrapper script to parallelize the processing of all of the samples in the AMD set, all hosted on a 16 CPU AWS EC2 instance. After 17 hours of disassembly and traversal, we were left with a pile of categorized control flow graphs ready for SNAP feature extraction. These graphs consisted of an edge list defining the full connectivity of the CFG, along with a key mapping those node id's to the prototype of the function that corresponds to the node.

3.2 Feature Extraction

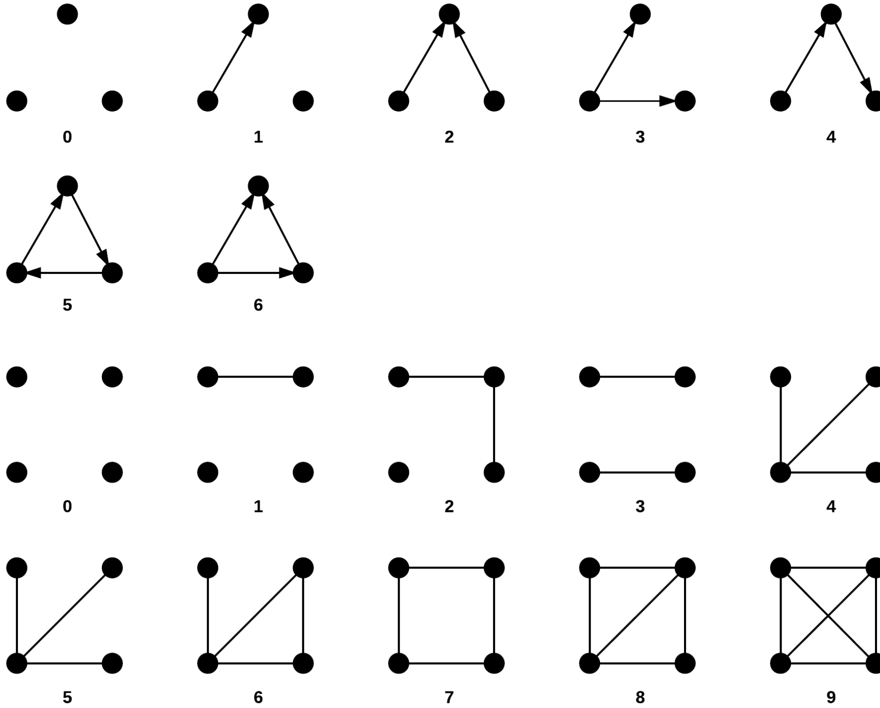
For our first type of feature extraction we used Snap.py's network analysis toolbox to calculate a range of different scalar properties of the control flow graphs we had extracted. We were able to get these using the network properties directly offered by Snap.py, except for averages that we had to calculate ourselves. The full list of these features are included in Table 1.

3.2.1 API Graph

From Cesare et al.'s paper, we got the idea that the frequencies of the various k-subgraphs for some k would be predictive for identifying malware [CX11]. However, as Lee et al. discovered, the

[1] Our Flowdroid graph extraction solution can be found in our attached code as the `flowdroid-cfg` Java project, and the Argus-SAF one as `Argus-SAF-CFG-Extractor` Scala project. Both of these are additionally accompanied by the `gen_graphs.py` Python wrapper scripts for parallel execution

Figure 2: Directed 3-subgraphs and undirected 4-subgraphs



only way to tractably calculate these heavy-duty metrics is by first performing some type of graph reduction [LJL10]. In order to perform our reduction, we exploited the information we gathered about each node in our CFGs: their prototypes. In Java executables, the originating class of a function call is permanently recorded, thus we can see from our extracted data which nodes originated from the same developer-written classes, or from the Android defined API packages. By collapsing all such nodes into one, we are able to greatly decrease the number of nodes (typically by around 10×), and reasonably decreases the number of edges (due to de-duplication). Both of these effects greatly improve our ability to enumerate the instances of k-subgraphs. Accordingly, after extracting all of the "API graphs" from our full CFGs, we used random sampling to estimate the frequencies of all of the directed 3-subgraphs, along with those of the undirected 4-subgraphs (see Figure 2).

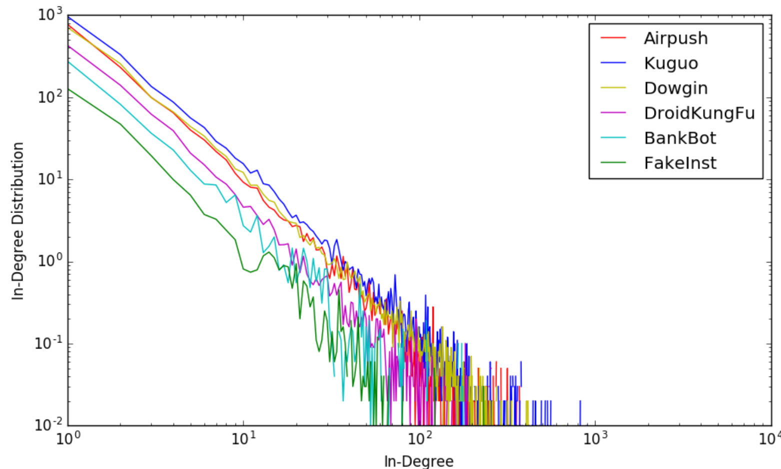
For 3-subgraph frequencies, we used 50,000 random samples. Due to the sparsity of the API graph, the majority of 4-subgraphs sampled are disconnected. To address this, we first took 50,000 random samples to estimate the frequencies of the disconnected 4-subgraphs. Then to get more accurate estimates of the connected 4-subgraph frequencies, we took another 50,000 samples of only connected subgraphs.

These two frequency vectors, along with the same network properties we calculated for the full CFG, were calculated for each of our extracted API graphs, and all of those calculated features were placed into a SQLite database for our learning pipeline.

3.3 Learning

Using the processed data described in the aforementioned section, we trained models for two different classification tasks. The first task was to classify files into one of five malware categories, as labeled on the AMD dataset website: Adware, Backdoor, HackerTool, Ransom, and Trojan. The second task was to classify files into one of the six malware families with the most samples: Airpush, Andup, Boxer, Dowgin, FakeInst, Koler. Note that malware family is a strictly more granular category than malware type, so we expected it to be the harder learning task. We used

Figure 3: Mean in-degree distributions of 6 malware families (averaged across 500 samples)



sklearn to split data into train and test for each classification task and vectorize the feature objects.

As a first pass, we tried several different models out-of-box from sklearn and found that nearest-neighbors and tree-based ensemble methods performed the best. As such, we also tested XGB-Classifier from the XGBoost Python API, which also uses tree-based ensemble methods. We used sklearn's built-in grid search with 3-fold cross-validation to tune parameters.

4 Results & Analysis

4.1 Graph Properties

When examining our data, we can see some interesting information emerge directly from the CFGs. In figures 3 and 4, we have plotted the mean in-degree and out-degree distributions for 100 random samples (respectively) of the 6 largest malware families (by memory size). From the plots, we observe that both in-degree and out-degree distributions appear to follow a power law for all malware families. We can also see that while the different families have similar slopes, they significantly differ in the number of nodes with in/out-degree 1. This is notable because Jang et al. used similar metrics (portion of calls with in/out-degree of 1) as features in their analysis of Windows executables [JWM+16]. This similarity suggested some degree of relationship between malware on different platforms, which is further discussed in 4.3.

4.2 Feature Results

We also discovered some interesting results by examining statistical measures on our calculated features. Some notable features significantly different means across families and relatively small standard deviations (across 500 random samples) are listed in Tables 2 and 3. One interesting finding is that although Jang et al. chose to use average in-degree centrality over the average out-degree centrality, our tests showed that average out-degree centrality had lower standard deviation (see Table 3).

Two other notable features explored were effective diameter and triangle participation ratio. We used effective diameter instead of average path length (used by [JWM+16]) since the SNAP Python API does not support average path length out-of-box. Effective Diameter was computed as the approximation of the 90th percentile of the distribution of shortest path lengths, by performing BFS from 100 random starting nodes. Another notable feature is triangle participation ratio (Table 3). The average ratios are fairly different between families, and the standard deviation is low for all 6 families tested. It was this surprising fact about triangle participation ratio that drove us to look deeper at k-subgraph frequency features, as triangle participation ratio is related to 3-subgraph frequencies.

Figure 4: Mean out-degree distributions of 6 malware families (averaged across 500 samples)

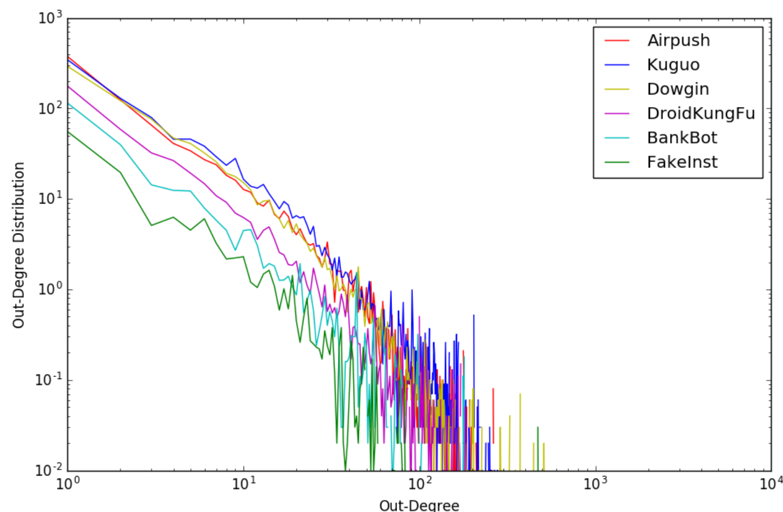


Table 2: Averages and Standard Deviations of Notable Features (500 samples per family)

	# Nodes		Density		Effective Diameter	
Family	Avg	Std Dev	Avg	Std Dev	Avg	Std Dev
Airpush	1305.40	911.15	0.01094	0.00944	10.236	4.170
BankBot	456.01	423.19	0.05292	0.09328	5.824	3.766
Dowgin	1435.72	1057.79	0.00921	0.00865	8.545	3.314
DroidKungFu	710.34	520.26	0.01569	0.03058	6.425	2.725
FakeInst	236.47	69.80	0.03313	0.02829	4.393	0.949
Kuguo	1803.15	818.85	0.00595	0.00187	10.440	2.774

Table 3: Averages and Standard Deviations of Notable Features, cont.

	In-degree Centrality		Out-degree Centrality		Triangle Participation Ratio	
Family	Avg	Std Dev	Avg	Std Dev	Avg	Std Dev
Airpush	0.01808	0.01806	0.02374	0.00740	0.5095	0.07665
BankBot	0.04034	0.02043	0.02969	0.01366	0.4928	0.15013
Dowgin	0.02363	0.01879	0.02323	0.01392	0.4324	0.09489
DroidKungFu	0.03202	0.04112	0.02184	0.01288	0.3682	0.09308
FakeInst	0.05982	0.05193	0.04145	0.02385	0.3194	0.11607
Kuguo	0.01082	0.00675	0.01958	0.00650	0.4188	0.05630

Figure 5: Normalized confusion matrix for malware family classification

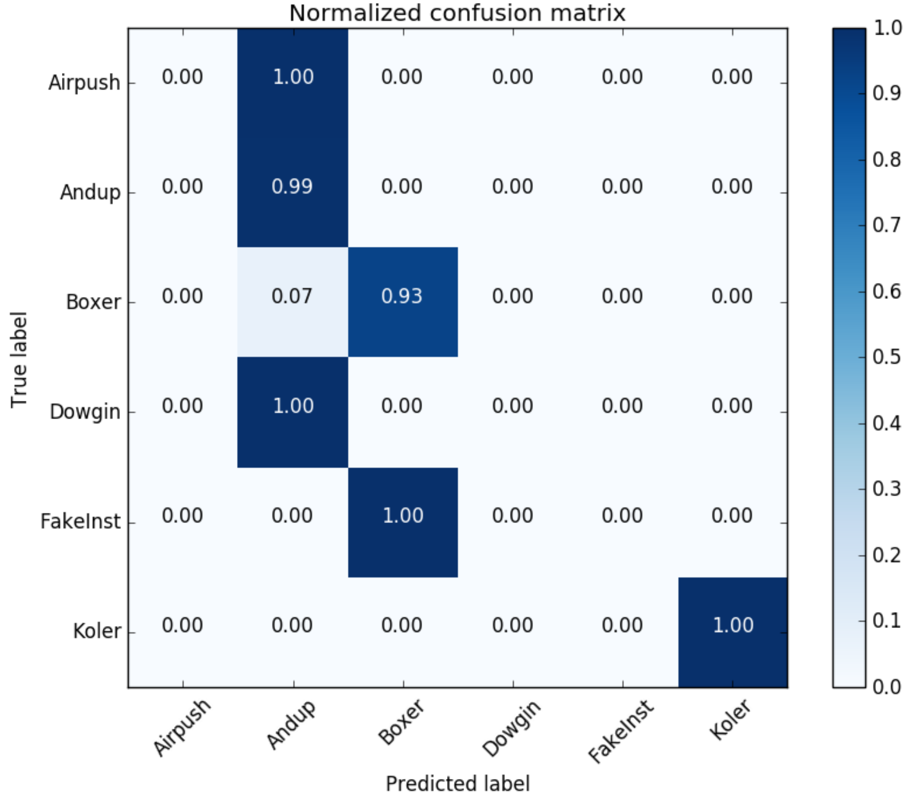


Table 4: Test results for malware category classification

	Precision	Recall	F1 score	Support
Adware	0.98	1.00	0.99	8471
Backdoor	1.00	0.93	0.96	539
HackerTool	1.00	0.82	0.90	95
Ransom	1.00	1.00	1.00	1233
Trojan	1.00	0.96	0.98	2819

4.3 Classification Results

For the malware category classification task, we were able to achieve 98.4% training accuracy (5-fold cross-validation) and 98.7% test accuracy using an Extra-Trees model (`sklearn.ensemble.ExtraTreesClassifier`). Full results are shown in Table 5.

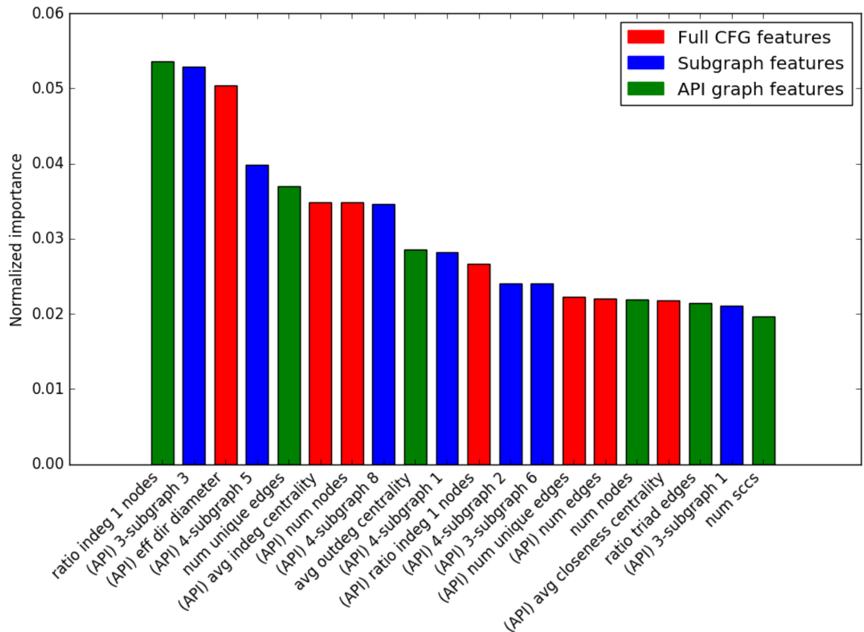
For the malware family classification task, we achieved 74.0% cross-validation accuracy and 74.1% test accuracy. From the confusion matrix (Figure 5), we see that almost all of the confusion is between families of the same category: Airpush, Andup, and Dowgin are adware, Boxer and FakeInst are Trojans, and Koler is ransomware. This suggests that the features we extracted, while very effective for distinguishing different malware categories, are not enough to distinguish between different families in the same category.

We also retrieved the normalized importance values for our successful malware category classification, the top 20 of which are plotted in Figure 6. From these values, we can see that our API graph reduction not only didn't lose us information, but seemed to highlight the important connectivity that actually distinguished different malware categories. In particular, many of the subgraph frequency features appear in the top 20, while only 6 features directly generated from the full CFG made it. Furthermore, our subgraph frequency features also had a strong showing with 7 entries in the top 20. Refer to Figure 2 to map indices to subgraphs. One potential area of future research is the semantic differences between the different 3- and 4-subgraphs as they relate to program functionality. More generally, a better understanding of the semantic significance of different graph metrics would likely improve results in both classification tasks.

Table 5: Test results for malware family classification

	Precision	Recall	F1 score	Support
Airpush	0.00	0.00	0.00	2353
Andup	0.70	0.99	0.82	8471
Boxer	0.80	0.93	0.86	2819
Dowgin	0.04	0.00	0.00	1015
FakeInst	0.00	0.00	0.00	651
Koler	1.00	1.00	1.00	1233

Figure 6: Normalized feature importances in malware category classification



Also, we see many of the important features identified by Jang et al. for classifying Windows malware are also important in our task [JWM+16]. Of particular note is the ratio of indegree 1 nodes, which we found to be the most important feature for malware category classification. This supports the hypothesis that there are language-agnostic network features that can be extracted from function call graphs to classify malware. This could be useful for the development of cross-platform antivirus systems, and an interesting direction of future work would be to compare call graphs between malware samples from the same family (e.g. Trojan) but different platforms (e.g. Android and Windows).

5 Conclusion & Future Work

The clearest area of potential improvement for this general methodology has clearly been demonstrated to be in the discipline of categorizing malware into specific families. Our confusion matrix results clearly demonstrated that our classifier had no discriminatory ability within a given malware category. This suggests that any future improvements on this general method will need to augment the network property features with more specific information about the particular interaction a piece of malware does with the Android API in order to achieve its exploit, as that is really what makes a given family unique. Our API graph concept is a first step towards that kind of information but evidently does not recover enough detail in order to fully characterize malware behavior.

Even so, this research shows that this approach of performing supervised learning on network properties of Android malware control flow graphs does identify malware into categories with high accuracy. The next steps for truly testing the value of this approach would be to challenge this

system with more aggressively obfuscated malware, along with attempting real-world deployment of such a system as an active Android anti-virus program.

References

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[CX11] S. Cesare and Y. Xiang. Malware variant detection using similarity search over sets of control flow graphs. In *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 181–189, Nov 2011.

[DWL17] Y. Du, J. Wang, and Q. Li. An android malware detection approach using community structures of weighted function call graphs. *IEEE Access*, 5:17478–17486, 2017.

[JWM+16] Jae-wook Jang, Jiyoung Woo, Aziz Mohaisen, Jaesung Yun, and Huy Kang Kim. Malnetminer: Malware classification approach based on social network analysis of system call graph. *CoRR*, abs/1606.01971, 2016.

[LJL10] Jusuk Lee, Kyoochang Jeong, and Heejo Lee. Detecting metamorphic malwares using code graphs. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1970–1977, New York, NY, USA, 2010. ACM.

[WLR+] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware.

[WRO+14] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1329–1341. ACM, 2014.