Mining Control Flow Graph as *API Call-grams* to Detect Portable Executable Malware

Parvez Faruki[*]
Govt. Polytechnic,
Ahmedabad, India.
parvezfaruki.kg@gmail.com

Vijay Laxmi[†]
Malaviya National Institute of
Technology, Jaipur, India.
vlaxmi@mnit.ac.in

M. S. Gaur[‡]
Malaviya National Institute of
Technology, Jaipur, India.
gaurms@mnit.ac.in

Vinod P.[§]
SCMS School of Engineering, Kerala, India.
pvinod21@gmail.com

ABSTRACT

Present day malware shows stealthy and dynamic capability and avails administrative rights to control the victim computers. Malware writers depend on evasion techniques like code obfuscation, packing, compression, encryption or polymorphism to avoid detection by Anti-Virus (AV) scanners as AV primarily use syntactic signature to detect a known malware. Our approach is based on semantic aspect of PE exectable that analyses API Call-grams to detect unknown malicious code. As in–exact source code is analysed, the machine is not infected by the executable. Moreover, static analysis covers all the paths of code which is not possible with dynamic behavioural methods as latter does not gurantee the execution of sample being analysed. Modern malicious samples also detect controlled virtual and emulated environments and stop the functioning. Semantic invariant approach is important as signature of known samples are changed by code obfuscation tools. Static analysis is performed by generating an API Call graph from control flow of an executable, then mining the Call graph as *API Call-gram* to detect malicious files.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection–Invasive Software(Virus, Rootkits etc.); I.5 [**Computing Methods**]: Pattern Recognition

[*]Assistant Professor, Information Technology Department.
[†]Corresponding Author, Associate Professor and Head, Computer Engineering Department.
[‡]Professor, Computer Engineering Department.
[§]Associate Professor, Computer Engineering Department.

General Terms

Experimentation and Security

Keywords

Control Flow Graph, API Call-graphs, *API Call-gram*, Pattern recognition, Malware

1. INTRODUCTION

1.1 Overview

A malicious file also known as malware/malcode infects a computer to either delete important content, compromise integrity of a machine, act as software robot being a part of crimeware, or unknowingly be a part of ransomware syndicate as a conduit in crime. Malcode is a commonly used term for harmful programs like Virus, Worms, Trojans, Backdoor, Spyware, Pay per Install programs, FakeAV, Rootkits [20] to name a few. Our focus of research is limited to Windows Portable Executable (PE) files as Microsoft Windows operating system binaries are prominent targets of malware writers [33].

AV programs performs policing act, identify, protect the machine and mitigate harmful software and their malicious intent. AV programs primarily use signature based detection to identify a cognized malware by extracting a unique hex-pattern known as signature. Signature based detection schemes are built around a large database containing byte sequence which characterize individual files. Unknown, zero-day malware detection fails as the signature for the same is not available in the database of AV [1]. Easy availability of encryption kits, obfuscation tools and packing techniques create a serious problem for AV detection. Dynamic Analysis is easily circumvented as the malware stops functioning or stays dormant on detecting virtual or emulated controlled environment. Dynamic detection approach is necessary but not sufficient in the changing scenario.

Obfuscation tools like packers hinder static analysis. This leads to analysing the obfuscated code instead of original malware. Powerful unpacking hardware virtualization tools such as ETHER [10] are coarse grain tools being used as generic upackers to obtain the original dump of malicious samples. Application Programming Interface (API) calls of Windows represent the abstract functionality of the programs. We aim to improve the classification of malware instances correctly reaching near 1 and minimize the false positives by reducing incorrect classification of benign as malware samples towards zero respectively. Instead of global frequency of API calls we generate sequence of API calls in order as instruction N-grams, known as ***API Call-grams*** in malware and saneware. API's are primarily written to interact with hardware and use different system service of Microsoft Windows. Novelity of our approach is use of semantic invariant API call sequence known as ***API Call-grams*** that is similar to instruction N–Grams to detect unknown malware.

1.2 Motivation

Exponential increase of malicious samples due to availability of code obfuscation tools have led to to look for alternative to syntactic signatured approach and reduce burden on signature based AV detection. Malicious programs misuse API strings as many of them are still undocumented due to proprietary constraints of Microsoft [8]. Malware writers misuse API Calls and keep the size of executables compact due to easy availability of functionality provided by the propriety vendor to evade detection.

Paper is organized as follows. Related work is covered in section 2. Section 3 discusses data–set preparation and brief overview of our proposed approach. We discuss ***API Call-gram*** generation along with classification tools and techniques. Section 4 discusses experimental setup, results, and classification approach with pattern recognition tool WEKA [32]. Comparison of our proposed method is discussed in section 5. Conclusion and future scope is covered in section 6.

2. RELATED WORK

Malware detection approach is either static, dynamic or hybrid. Dynamic analysis is performed by analysing the traces and strains of executable sample under controlled environment running on virtual or emulated platform. Problem with the approach is, all the paths may not be covered and appropriate condition may not occur for the sample to execute self. Moreover, dynamic analysis is time consuming as it needs to be executed for a particular time period for observation. Malicious software may also harm the machine once executed and gets out of control. Static detection methods analyse structural information of an executable with static or semantic signatures using reverse engineering techniques. As analysis is performed without executing malafide code, this mechnism covers all the paths.

In [28], critical API calls were extracted statically using IDA-Pro [7]. Thus, all the late–bounded API calls that are made using `GetProcAddress`, `LoadLibraryEx`, etc. are not taken into account. Moreover this approach did not work for packed malware.

Johanese Kineable et el [16] employ pattern recognition techniques by Clustering generated API Call-graph. The authors generate a semantic invariant graph known as API Call-graph where API calls are modeled as vertices and the calls between them are represented by directional edges. This graph represents the abstraction of an executable. Pairwise graph similarity is used to find distance among the samples. Call-graph is clustered with DBSCAN and K-medeoids to generate clusters of variants of families.

In [14] the authors employ semantic invariant approach by analysing the abstraction of an executable without disassembling it. Jump and call sequences are considered prime elements due to their non sequential decision making approach, to create a topological graph known as Code graph to differentiate the malicious executables from benign. The authors report an accuracy of 67% for real malware.

Jeong Lee et el [13] have proposed an extended approach based on code graph to detect metamorphic malware by using the abstraction of a program representing API call sequences. They extract the system-call sequence to generate a topological graph known as Code graph. Authors use similarity measurements to find out variants of code obfuscated families. The authors report an accuracy of 91% for detecting metamorphic malware.

In their proposed work, bonfante et el [12] create a rewriting engine to detect morphed malware variants. Analysis of variants of known malware is based on syntactic as well as semantic structure of a program. Signatures of malware are represented by a Control Flow Graph. Signature matching technique is based on tree automaton. Krugel *et al* [4] proposed a method based code analysis to identify structural similarity between malicious code (worms). The proposed method is based on the CFG generated for worms described as fingerprint for worms. Their system is found to be resilient against common code transformation techniques.

In their proposed work [24], authors proposed a semantic approach to detect variants of malware. This method is based on the functionality of system call executed by malware samples. Their main focus is to identify all instructions and its parameters which are used for calling a system call. They propose a pattern matching technique which is able to identify semantically equivalent parts of code. The method is capable of identifying programs that are related to each other and the ones that are totally dissimilar.

Rachit *et al* [26] created a malware normalizer making use of term rewriting rules. The method was applied on virus named as *Win32.Evol*. The main objective of the proposed work was to convert program variants into smaller number of variants.

In Hunting for metamorphic engines [35], Hidden Markov Models (HMMs) were used to represent statistical properties of a set of metamorphic virus variants. The metamorphic virus data set was generated from metamorphic engines like G2, Next Generation Virus Construction Kit (NGVCK), Virus Creation Lab for Win32 (VCL32) and Mass Code Generator (MPCGEN). HMM is trained on a family of metamorphic viruses to determine if a given program is similar to the viruses the HMM represents.

Authors in [9] proposed a *"phylogeny"* model. The n–gram feature extraction technique was proposed and fixed permutation was applied on the code to generate new sequences, called n-perms. Since new variants of malware evolve by incorporating permutations, the proposed n–perm model was developed to capture instruction and block permutations. The experiment was conducted on a limited data set consisting of 9 benign samples and 141 worms collected from VX Heavens [34]. The proposed method showed that similar variants appeared closer in the *phylogenetic tree* where each node represented a malware variant.

In [29] the authors generate a control flow graph of the disassembled executable file. Basic blocks are generated to create a flow graph and relation among the blocks is created. The authors employ graph isomorphism to find similarity among the executables.

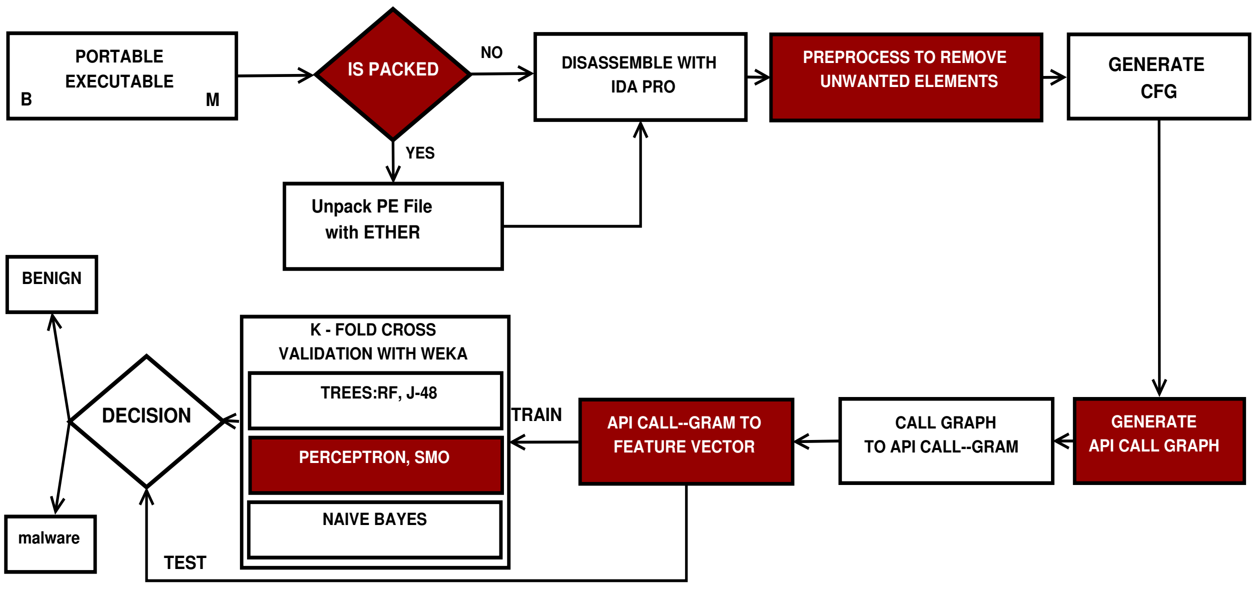

Figure 1: Proposed Approach for Malicious File Detection

Cesare et el [3] generate abstract representation of the executable using Control Flow graph. Graph isomorphism is used to determine similarity among the executables using sub–graph matching with data–mining approach to detect variants of polymorphic malware.

Ronghua et el [31] proposed a behavioral system to differentiate malware and clean files by analyzing the traces API calls of Portable Executable (PE) files by executing the samples on virtual machine with Microsoft Detours [15] and HookMe as tracing tools. They extract the API calls from the generated trace reports. The authors have used imbalanced data set that consists of huge number of benign and a very small set of clean files. The API call report classifies the malware and clean files based on global frequency and occurance of API strings in individual files. The authors report accuracy of above 97% with low false positive. The imbalanced data set tends to be biased towards the malware API calls of malware files as authors use imbalanced dataset. Moreover, authors have not specified the categories of clean files, as different categories of benign programs also use varied API calls in software develpment.

In [27], authors have used reverse engineering technique to extract API Calls from the Import Address Table (IAT) of Windows PE files. The authors then, look for global occurance of API calls in benign and malware samples. They have classified the malware and benign samples with data mining techniques. This method can be easily evaded by use of obfuscation tools. API Calls extracted from IAT of obfuscated file belong to the encypted layer instead of the actual file. Moreover, unpacking the code before analysis is time-consuming and does not guarantee accurate unpacked code.

In [22] the authors extract the API calls on a dynamic environment using QEMU. The authors detect metamorphic malware samples based on API call frequency, by looking for calls prominently used by malware and benign programs. They classify the samples using pattern recognition techniques.

3. PROPOSED METHODOLOGY

Here we discuss the outline of proposed work for `API Call-gram` along with data–set preparation as depicted in Figure 1.

1. As shown in Figure 1 the first step is to collect nearly equal number of benign and malware PE files. The malware samples checked for Packer signature. If packed, the samples are unpacked with ETHER [10] a hardware virtualization unpacker.
2. The samples are diassembled with IDA–Pro [7] disassembler. A batch script disassembles the benign and malware samples. Once the assembler files are generated, we preprocess the files to remove elements not needed for generating Control Flow Graph.
3. Generate CFG from the disassembled files. The abstraction of the executable is represented by the API calls made with the API call statements. We generate a Call-Graph from the CFG to store the API calls made by an executable into an output file.
4. The API calls are our features to be converted as Feature Vectors (FV). API calls are converted as `API Call-grams` for n = 1, 2, 3 and so on.
5. The `Call-grams` are our feature vectors for input to machine learning methods. They are normalized to train with WEKA [32], data–mining tools.
6. 10 fold Cross–Validation with Algorithms like Random–Forest [2], J-48 [30], SMO-SVM [23] and Voted Perceptron [11] is applied on the normalized CSV to classify the malicious and benign executables.

3.1 Dataset Preparation

The data set consists of different categories of programs (refer Tables 1 and 2) that includes varied category of API calls such as performing network activity, file operations,

Table 1: Benign (B) Data set

Category	Number	Size(min.)	Size(max)
System-software	1230	800 KB.	150 MB.
Application-software	400	4 MB.	150 MB.
Utility-software	567	1 MB.	48 MB.
Downloaders	50	1 MB.	150 MB.
Media-Players	100	3 MB.	100 MB.
Device Drivers	200	1 MB.	230 MB.
Virtualization s/w	3	130 MB.	250 MB.
Anti-Virus	10	200 MB.	400 MB.
Network-Activity	35	2 MB.	43 MB.

graphic functionality, registry operations, and memory related tasks.

Windows API is a core functionality for easy interaction with the system and ensures fast and compact source code development. Misuse of known as well as undocumented API by the malware writers to fulfill the malafide intention needs proper analysis by *Call-gram* sequencing.

3.2 API Call Graph

An API Call Graph is a directed graph which represents abstract functionality of API calls made by different procedures in the program. CFG previews flow of control among procedures.

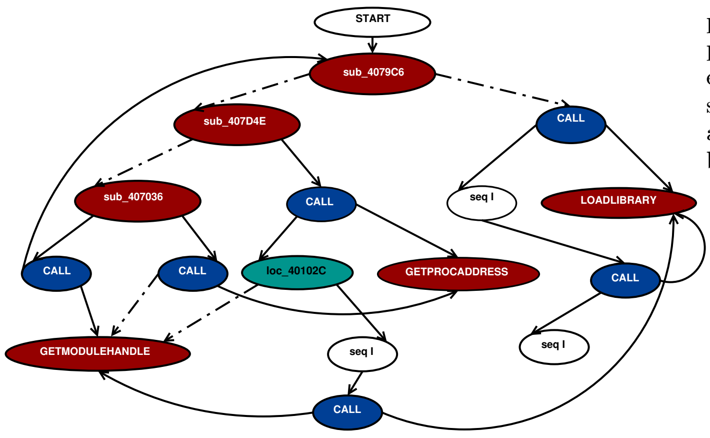

Figure 2: API Call-Graph

API Call-graph represents the abstraction of a program with reference to the sequence of API calls made by an executable. Figure 2 displays a typical Call-graph showing the relationships of API's made at various events in a file. The node is represented by "CALL" which directs towards an API being used to perform certain functionality. The call graph records API calls made into an output file to convert API calls as *API Call-grams*.

The benign program are collected from fresh installation of Windows XP SP2, system utilities and resource like cnet [5]. The downloaded benign samples were scanned with Kaspesky AV 2011 [17] with latest database updates to acknowledge their clean behaviour before their use in the experiment. Malware dataset are gathered from well known malware repositories such as VX Heavens [34], OffensiveComputing [6], and Honey–pot contents of allied user agencies.

First the Control Flow Graph of the disassembled executable is generated. This CFG is converted into a reduced API Call-Graph. The Call-Graph represents the general abstraction of a file. This Call-Graph is converted to *API Call-grams*. The Call-gram files are input to generate a single, two, three and four *API Call-grams*. Unique *API Call-grams* with their global frequency and occurance across the benign and malware samples is generated respectively. Their occurance in individual files is generated for unigram, bigram, trigram etc. for benign and malicious samples.

Table 2: Malware (M) Data set

Category	Number	Size(min)	Size(max)
Virus	732	900 KB.	1.3 MB.
Worm	632	800 KB.	1.4 MB.
Trojan	437	1.1 MB.	2.3 MB.
Backdoor	446	900 KB.	27 MB.
RootKits	75	1 MB.	1.6 MB.
Peer-Peer Bot	63	700 KB.	2.6 MB.
New Malware	479	1.3 MB	3.6 MB.
FakeAV	84	15 MB.	102 MB.
Hoax	321	1 MB.	3.5 MB.
User agencies	370	703 KB.	2.2 MB.

3.3 *API Call-grams*

An imbalanced data set has large number of files for a particular class may be biased towards the API calls of that particular class. Moreover, we also need to perform hypothesis testing like t-test when one particular class has very few samples compared to another. Motivated by the work of authors in [21] we have considered nearly equal number of benign and malware files in our experiments.

Table 4: Number of API attributes in call grams

Call-gram	B	M	$B \cap M$	$B \setminus M$	$M \setminus B$	$B \cup M$
1 API-gram	36	35	35	1	0	36
2 API gram	493	576	420	73	156	649
3 API gram	1052	1173	664	388	509	1511
4 API-gram	6303	8084	1602	4699	6482	12783

Initially the executable is dis–assembled and pre–processed to remove unwanted elements. The preprocessing step extracts the API calls ignoring the parameters of each call. Control Flow Graph of the file is generated. API Call graph is extracted from the CFG. It is then converted as *API Call-grams*. We generate one *API Call-gram* along with a unique single API call list that consists of global count of API and its presence in number of files in benign and malware class respectively. Subsequently we generate API string for two, three, four grams and so on. The example shown below explains the process of generating *API Call-grams*.

4. CALL-GRAPH AS *API CALL-GRAMS*

API Call-graph represents the semantics of an executable. Investigation of frequency of call in a single file and global frequency among multiple files has been investigated extensively. The issue is, API calls are written for fast and compact program development, for benign purpose but are being misused by the malware writers to fulfill their evil intent and reduce size of their code.

Table 3: `API Call-gram` Sequence

File	`API Call-gram`	`Call-gram Sequence`
File1	*1 `API Call-gram`*	$A_1, A_2, A_3, A_4, A_5, A_6, A_7$
File2	*1 `API Call-gram`*	$A_1, A_2, A_7, A_4, A_5, A_6, A_8$
File1	*2 `API Call-gram`*	$A_1A_2, A_2A_3, A_3A_4, A_4A_5, A_5A_6, A_6A_7$
File2	*2 `API Call-gram`*	$A_1A_2, A_2A_7, A_7A_4, A_4A_5, A_5A_6, A_6A_8$
File1	*3 `API Call-gram`*	$A_1A_2A_3, A_2A_3A_4, A_4A_5A_6, A_5A_6A_7$
File2	*3 `API Call-gram`*	$A_1A_2A_7, A_2A_7A_4, A_7A_4A_5, A_4A_5A_6, A_5A_6A_8$
File1	*4 `API Call-gram`*	$A_1A_2A_3A_4, A_2A_3A_4A_5, A_3A_4A_5A_6, A_4A_5A_6A_7$
File2	*4 `API Call-gram`*	$A_1A_2A_7A_4, A_2A_7A_4A_5, A_7A_4A_5A_6, A_4A_5A_6A_8$

API Call-grams

No.	File 1.		File 2.	
	API Call	Code	API Call	Code
1.	Loadlibrary	A1	Loadlibrary	A1
2.	CreateFile	A2	CreateFile	A2
3.	ReadFile	A3	VirtualAlloc	A7
4.	Loadlibrary	A4	Loadlibrary	A4
5.	CreateThread	A5	CreateThread	A5
6.	RegOpenkey	A6	RegOpenkey	A6
7.	VirtualAlloc	A7	CreateMutex	A8

Figure 3: `API Call-Graph to Call-gram` **generation.**

Malware writers use them for their own purposes to generate compact harmful code. Mere frequency of API calls leads to higher false positives as the same API are also used by benign programs to perform certain functionality. The sequence of multiple calls is an important investigation point. Instruction *n-grams* is a very well known non–signatured approach to detect malicious code from benign. Novelity of our approach is generation of *`API Call-grams`* from the API Call-Graph. It can look for multiple API call made to fulfill one single operation.

Once the *`API Call-grams`* are generated, we find common calls among *`API Call-grams`* of benign and malware samples (refer Table 4). The generated API gram displays frequency of individual API calls and their global occurance with unique API grams across the files. This list consists of

- Common API calls of benign and malware samples ($B \cap M$)
- Union of API calls among benign and malware ($B \cup M$)
- Discriminant calls present benign samples ($B \setminus M$)
- Discriminant calls present in malware samples ($M \setminus B$)

Similarly common and prominent API call are generated for 2, 3 and 4 *`API Call-grams`*. These prominent API strings of different classes are our parameters for input to pattern recognition algorithms.

4.1 Classification Technique

Classification is performed with WEKA [32] on the feature vectors generated for one to four *`API Call-grams`*. Feature vector table is normalized before submission to classifier. Weka library has implemented set of pattern recognition, machine learning and clustering algorithms to perform binary classification. We classify the feature vector attributes by executing feature vector (FV) tables on Random-Forest (RF) [2], Sequential Minimal Optimization (SMO) [23], J-48 decision tree [25] , Naïve–Bayes [30] and Voted Perceptron [11] [30]. We perform 10–fold cross–validation [18] as it is the most effective k–fold classification scheme.

Random Forest is an ensembled classifier. It aggregates results of multiple trees generated for the dataset. J-48 algorithm creates a decision tree that is based on the attribute of training data. It continues the process until it finds the attribute that can distinctly identify among the given instance based on minimum entropy value or maximum information gain.

Sequential Minimal Optimization (SMO) algorithm is implemented in WEKA library to train support vector classification. SVM rudimentary model is a classifier that performs learning of a decision boundary among two classes (eg. M and B) in an input space. This algorithm globally identifies missing values and replaces missing values, and transforms nominal attributes into binary ones. It also normalizes all attributes by default.

Voted Perceptron (VP) is a classifier where the input x is mapped to an output function. It is based on perceptron algorithm of Rosenblatt and Frank [11]. This algorithm advantages the data linearly separable and has large margins. VP is efficient with respect to computation time in comparison to Vapnik's SVM [11].

Naïve–Bayes classification is based on Bayes conditional probability [30]. It identifies all the attributes of data and analyses them independent of each other assuming that they are equally important and disjoint. This leads to low accuracy.

4.2 Evaluation Metrics

Evaluation metric is computed on basis of True positives (TP)–correctly classified malware instances, True Negative (TN)–correctly identified benign instances, False Positive (FP)–benign instances mis–classified as malware and False Negative (FN)–malicious samples mis–classified as benign [19]. *True Positive rate (TPR)* and *True Negative Rate (TNR)* are known as *sensitivity* and *specificity* respectively. Accuracy (ACC) of malware detection systems is measured with respect to high TPR and low FPR [30].

- `TPR = TP/(TP + FN)`
- `FPR = FP/(FP + TN)`

Table 5: **Detection Accuracy**

Classifier	1 API-gram	2 API-gram	3 API-gram	4 API-gram
RandomForest	84.2	89.3	**98.3**	**98.4**
SMO-SVM	84.2	**87.9**	**98.1**	**98.2**
Voted Perceptron	82.9	**86.1**	**97.9**	**97.7**
J-48	81.2	86.4	96.9	96.6
Naïve–Bayes	76.2	81.8	90.2	90.3

- `TNR = TN/(TN + FP)`
- `FNR = FN/(FN + TP)`
- `ACC = (TP + TN)/(TP + TN + FP +FN)`

5. EXPERIMENTAL SETUP, RESULTS

5.1 Experimental Setup

The experiments were performed on Intel Core i3 2.40 GHz machine, 4 GB RAM, Ubuntu 10.10 host operating system. The samples were unpacked by Hardware Virtualization using Ether.

5.2 Results

Table 5 displays experimental results for our *API Call-grams* approach. The results of 1 API-gram displays an accuracy of 84%. This result of 1 gram is analogous to the global and file frequency of API calls discussed by Aksan sami et el [27] and Vinod P. et el [22]. API are primarily developed for fast and convenient use of system resources, false positives increase due to their presence in sane and malcode. High FP increase leads to lower detection rate.

The 2 *API Call-grams* results show an improvement over single *API Call-grams*. 3 API call-gram show a significant increase in accuracy from 89.3 to 98.1%. This is due to the frequency of sequence of API's made by malicious programs to perform a task that a normal program would make but have lower frequency of calling the sequence. For example, creating a file, writing to it and sending it also performed by benign code. But the frequency of calling such *API Call-grams* differ significantly in both types. This leads to higher detection rate (DR).

5.3 Discussions

It is important to detect a malicious file from a sane file. We have compared our method against the most relavant work either implementing static analysis or behavioural methods for classifying malicious files with high accuracy. *API Call-gram* approach gives better accuracy compared to other approaches for malware detection as in general, the *Call-gram* model captures temporal snapshot of a program which reflects the behavioral patterns of an executable.

In case of smaller value of *API Call-gram* model for n=1 or 2, fewer number of features exist in the feature space and the frequency of features is also similar in both classes (i.e. malware and benign), which leads to mis–classification. In case of higher value of *API Call-gram* for n=4, 5, 6 and above large number of features exist in the feature space. These features are either absent in one of the classes, or they appear with very small frequency in both the classes. Thus, these features are referred by us as redundant features and they degrade classification accuracy. The figure below depicts TPR and FPR for different *API Call-gram*. This figure depicts that the 3 *API Call-gram* model produces better classification accuracy compared to other models.

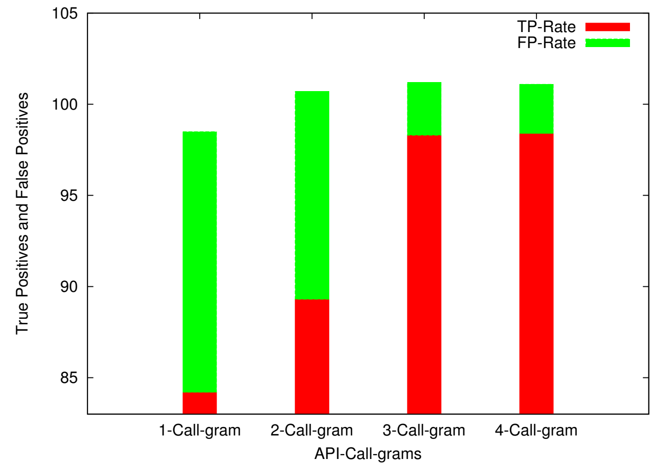

Figure 4: Comparison of `API Call-gram` TPR, FPR

6. COMPARISON WITH PREVIOUS WORK

Comparison of our proposed detection model with existing techniques is summarized in Table 6. Ronghua Tian et el [31] used global frequency and occurance of API calls across the sane and malcode files for malware detection. The authors used an imbalanced data–set for analysis, but did not use any normalization mechanism to remove the bias for a small set of benign files. The authors have reported accuracy 97.3% for detecting malicious code.

Heejo lee et el [14] generate code graph from an executable file and compare them to identify real malware. The code graph is a the semantic signature generated to detect malicious code. Detection rate of their method is 67%.

Faraz Ahmed et el [1] consider detection of malware based on API categories like as Network activity, Memory management, File manipulation, graphic control, registry activity. They create a core API set of 237 malware known as critical API's to detect malicious code. Classification accuracy of their method is reported 97%. The authors have not discussed the false positive in their approach.

In [3] the authors generate a Control flow based Polymorphic malware detection mechanism. The proposed approach works well against the code obfuscated malware. Authors have not discussed the detection or accuracy of their proposed method.

Table 6: Comparison of `API Call-Graph` approach with existing work

Author	**Dataset size** Benign-B, Malware-M	**Method**	**DR, ACC, TPR**
Jusuk Lee [13] *et al*	3200–M	Metamorphic Malware Detection with Code-graph	91%
Heejo Lee [14] *et al*	2300-M	Malware Detection with with Code-graph	67%
Dullian Roles [29] *et al*	not specified	Semantic Invariant CFG approach	not specified
J. Kinable [16] *et al*	not specified	Call Graph Clustering	Not specified
Cesare [3] *et al*	Not specified	Flow Graph based Polymorphic Malware Detection	
`Proposed Method`	**3234-B, 3256-M**	**Call Graph mined** as `Call-gram`	**98.4**% ACC 2.7% FP

Dullian roles et el [29] implemented a semantic invariant approach by generating a CFG of the existing malware sample to extract semantic signatures. The authors use tree automata for detection. The paper does not discuss the detection accuracy. In [13] the authors implement the code graph for detection of metamorphic malware. They claim an accuracy of 91% in detecting real code obfuscated malware.

7. CONCLUSIONS AND FUTURE SCOPE

In this work we discuss API Call Graph mined as *`API Call-grams`* to analyse code obfuscated malware and achieve low false positive. Abstract representation of a program is obtained from the sequence of API calls made by a file during static CFG extraction. Novelity of our approach is considering *`API Call-grams`*. API-grams consider the sequence in which calls are made leading to high detection rate. Our approach reduces false positives. Mere frequency of API strings leads to high false positives as API strings are originally written for windows SDK. API-grams consider the sequence in which calls are made one after another which leads to high detection and low false positives. Our contribution includes extracting API calls from unpacked executables as a Control Flow graph. It is converted as a reduced API Call-Graph. Call-Graph is converted as API-grams to generate individual and common API call strings. API grams are features differentiating benign and malware with an accuracy of 98.4% on Random forest and False positives of 2.8%. Our approach performs better than the recent models on a comparatively varied and big database. In future we would like to go for higher Call-gram sequencing and apply feature reduction methods to reduce false positives close to theoretically 0%.

8. REFERENCES

[1] F. Ahmed, H. Hameed, M. Z. Shafiq, and M. Farooq. Using Spatio-Temporal Information in API Calls with Machine Learning Algorithms for Malware Detection. In *Proceedings of the 2nd ACM workshop on Security and artificial intelligence*, pages 55–62. ACM, March 2009.

[2] L. Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[3] S. Cesare and Y. Xiang. A Fast Flowgraph based Classification System for Packed and Polymorphic Malware. In *Proceedings of the 2010 24th IEEE International Conference on Advanced Information Networking and applications*, AINA '10, pages 721–728. IEEE, 2010.

[4] K. Christopher, K. Engin, M. Darren, and R. William. Polymorphic Worm Detection using Structural Information of Executables. In *Proceedings of RAID 2005*, pages 207–226. Springer-Verlag, 2005.

[5] CNET. Free Software Downloads and Software Reviews. http://download.cnet.com/windows/?tag=hdr;brandnav, Last Accessed March 2012.

[6] Danny-Quist. Offensive Computing. http://offensivecomputing.net/, Last Accessed March 2012.

[7] Datarescue. "IDA–PRO Disassembler. http://www.datarescue.com/idabase.

[8] K. Dunham. Malcode context of api abuse. Technical report, SANS Institute, 2011.

[9] K. Enamul, W. Andrew, and L. Arun. Malware Phylogeny Generation using Permutations of Code. In *Journal in Computer Virology*, pages 13–23. ACM, 2010.

[10] ETHERXEN. "Ether-Xen Hypervisor. http://ether.gtisc.gatech.edu/source.html.

[11] Y. Freund and R. E. Schapire. Large margin classification using the perceptron algorithm. *Mach. Learn.*, 37(3):277–296, dec 1999.

[12] B. Guillaume, K. Matthieu, and Y. M. Jean. Architecture of a Morphological Malware Detector. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, Journal in Computer Virology, pages 263–270, 2010.

[13] L. Heejo, Kyoochang, and L. Jeong. Code Graph for Metamorphic Malware Detection. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1970–1977. ACM, 2010.

[14] L. Heejo and L. Kyoochang, Jeong. Code Graph for Malware Detection. In *International Conference on Information Networking*, ICOIN '08, pages 1–5. IEEE, 2008.

[15] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd conference on USENIX Windows NT Symposium -*

Volume 3, WINSYM'99, pages 1–9. ACM, 1999.

[16] K. Johanese and K. Ostrasis. Malware Classification based on Call Graph Clustering. In *Journal of Computer Virology*, pages 233–245, 2011.

[17] Kaspersky. Kaspersky Anti–Virus 2011. http://www.kaspersky.com/.

[18] R. Kohavi. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 2*, IJCAI'95, pages 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[19] V. Laxmi, M. S. Gaur, P. Faruki, and S. Naval. Peal-Packed Executable AnaLysis. In *Proceedings of International Conference on Advanced Computing Networking and Security*, ADCONS '11. SPRINGER LNCS, 2011.

[20] E. Manuel, S. Theodoor, K. Engin, and K. Christopher. A Survey on Automated Dynamic Malware-Analysis Techniques and Tools. *ACM Computer Survey*, 44(2):6, 2012.

[21] R. Moskovitch, D. Stopel, C. Feher, N. Nissim, and Y. Elovici. Unknown Malcode detection via Text Categorization and the Imbalance Problem. In *ISI*, pages 156–161, 2008.

[22] V. P. Nair, H. Jain, Y. K. Golecha, M. S. Gaur, and V. Laxmi. MEDUSA: MEtamorphic malware Dynamic analysis Using Signature from Api. In *5th International Conference on Malicious and Unwanted Software*, MALWARE 2010, pages 263–269. ACM, 2010.

[23] J. Platt. Fast Training of Support Vector Machines using Sequential Minimal Optimization. In B. Schoelkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods – Support Vector Learning*. MIT Press, 1998.

[24] Z. Qinghua and R. Douglas. MetaAware: Identifying Metamorphic Malware. In *Computer Security Applications Conference, Annual*, pages 411–420, 2007.

[25] R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[26] M. Rachit, M. Antony, R. Douglas, P. C., and Z. Qinghua. Normalizing Metamorphic Malware Using Term Rewriting. In *International Workshop on Source Code Analysis and Manipulation SCAM '06*, pages 75–84. IEEE, 2006.

[27] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamze. Malware Detection based on Mining API Calls. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1020–1025. ACM, 2010.

[28] V. Sathyanarayan, P. Kohli, and B. Bezawada. Signature Generation and Detection of Malware Families. In *Proceedings of the 13th Australasian conference on Information Security and Privacy*, ACISP '08, pages 336–349, Berlin, Heidelberg, 2008. Springer-Verlag.

[29] R. T. Dullien and B. Universitaet. Graph based Comparison of Executable Objects. pages 1970–1977. University of Technology in Florida, 2005.

[30] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Pearson Education Inc. South Asia, Noida, India, 1 edition, MAY 2006.

[31] R. Tian, R. Islam, and L. Batten. Differentiating Malware from Cleanware Using Behavioural Analysis. In *Proceedings of the 3rd international conference on Security of information and networks*, SIN '10, pages 23–30. IEEE, March 2010.

[32] UniversityOfWaikato. WEKA:Data Mining with Open Source Machine Learning Software. http://www.cs.waikato.ac.nz/ml/weka, Last Accessed March 2012.

[33] VirusTotal. VirusTotal - Free Online Virus, Malware and URL Scanner. https://www.virustotal.com/, Last Accessed January 2012.

[34] VX-Heavens. Virus Collections (VXheavens). http://vl.netlux.org/vl.php/, Last Accessed February 2012.

[35] W. Wing and S. Mark. Hunting for Metamorphic Engines. In *Journal in Computer Virology*, pages 211–229, 2006.