Mining CFG as *API Call-grams* to Detect Portable Executable Malware

Parvez Faruki  
MNIT, Jaipur, India.  
parvezfaruki.kg@gmail.com

Vijay Laxmi  
MNIT, Jaipur, India.  
vlaxmi@mnit.ac.in

M. S. Gaur  
MNIT, Jaipur, India.  
gaurms@mnit.ac.in

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

ABSTRACT
Malware writers use evasion techniques like code obfuscation, packing, compression to conceal from Anti-Virus (AV) scanners as AV use syntactic signature to detect a known malware. Our detection approach is based on semantic aspect of PE executable that analyzes API Call-grams to detect unknown malicious code. Static analysis covers all the paths of code which is not possible with dynamic behavioral methods as latter does not guarantee execution of sample being analyzed. Modern malicious samples also detect controlled virtual and emulated environments and stop the functioning. Samples are analyzed by generating API Call graph from Control Flow Graph (CFG) of executables. Call graph is represented as *Call-grams* to detect vicious files.

1. **INTRODUCTION**
Malware infects a computer to either delete important content, compromise integrity of a machine or act as software robot being a part of ransomware. Our focus of research is Windows Portable Executable (PE) files as they are prominent target of malware writers [7]. Anti Virus programs primarily use signature based detection to identify a known malware by extracting a unique hex-pattern known as signature. Application Programming Interface (API) in Windows represent the abstract functionality of programs. We aim to improve detection of malware instances and minimize false positives by generating sequence of API calls known as *API Call-grams*. Novelty of our approach is use of semantic invariant Call sequence to detect unseen malware.

2. **PROPOSED METHODOLOGY**

1. As shown in Figure 1 the first step is to collect nearly equal number of benign and malware PE binaries. Malware samples were checked for Packer signature. Packed samples are unpacked with ETHER [3].

2. Executables are disassembled with IDA-Pro [2]. Automated script disassembles the benign and malware samples. Once the assembler files are generated, we preprocess the files to remove unwanted elements.

3. Generate Control Flow Graph for each disassembled file. Abstraction of an executable is represented by the API calls. Then, we reduce the CFG to API Call graphs that stores the calls made by an executable into an output file.

4. API calls are converted to *Call-grams* for n = 1, 2, 3 and 4. The *Call-grams* are our features for input to machine learning methods. They are normalized to train with WEKA [1]. K-fold Cross-Validation is applied to differentiate vicious and benign executables.

Benign program are collected from fresh installation of Windows XP SP2 and system utilities. Samples were scanned with Kaspersky AV 2011 [4] to acknowledge their clean behavior. Malware dataset is taken from well known malware repositories like VX Heavens [8] and allied user agencies.

2.1 API Call Graph
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 displays a typical Call-graph showing the relationship of API’s made at various events. Here, Node is represented by “CALL” which directs towards an API. API’s of Call graph is stored for every sample for conversion to *Call-grams*.

2.2 API Call-grams
Instruction n-gram is a well known non-signatured approach to detect malicious executables. Instruction n-gram is a well known non-signatured approach to detect malicious executables. The API call-grams are overlapping sub-strings of length n generated in a sliding window fashion. n-grams have found importance in text categorization and identification of files [6].

3. 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. Table 1 shows typical call gram sequence. File1 and File2 has record of call-graph. 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. 

Analysis based on frequency of API calls has high false positives as API calls are also used by benign programs. Once the Call-grams are generated, we find common calls among benign and malware samples. The generated API call-gram displays frequency of individual API calls and their global occurrence with unique API grams across the files.

4. EXPERIMENTAL SETUP, RESULTS

The experiments were performed on Intel Core i3 2.40 GHz machine, 4 GB RAM, Ubuntu 10.10 host operating system. Evaluation metric is True positives (TP)–correct malware instances, True Negative (TN)–correct benign instances, False Positive (FP)–False Negative (FN) [5].

Table 1: API Call-gram Sequence

<table>
<thead>
<tr>
<th>File</th>
<th>API Call-gram</th>
<th>Call-gram Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>File1</td>
<td>1 API Call-gram</td>
<td>A1, A2, A3, A4, A5</td>
</tr>
<tr>
<td>File2</td>
<td>1 API Call-gram</td>
<td>A1, A2, A3, A4, A5</td>
</tr>
<tr>
<td>File3</td>
<td>2 API Call-gram</td>
<td>A1, A2, A3, A4, A5, A6, A7</td>
</tr>
<tr>
<td>File4</td>
<td>2 API Call-gram</td>
<td>A1, A2, A3, A4, A5, A6, A7</td>
</tr>
<tr>
<td>File1</td>
<td>3 API Call-gram</td>
<td>A1, A2, A3, A4, A5, A6, A7</td>
</tr>
<tr>
<td>File2</td>
<td>3 API Call-gram</td>
<td>A1, A2, A3, A4, A5, A6, A7</td>
</tr>
<tr>
<td>File3</td>
<td>4 API Call-gram</td>
<td>A1, A2, A3, A4, A5, A6, A7</td>
</tr>
<tr>
<td>File4</td>
<td>4 API Call-gram</td>
<td>A1, A2, A3, A4, A5, A6, A7</td>
</tr>
</tbody>
</table>

Table 2: Detection Accuracy

<table>
<thead>
<tr>
<th>Classifier</th>
<th>1 API-gram</th>
<th>2 API-gram</th>
<th>3 API-gram</th>
</tr>
</thead>
<tbody>
<tr>
<td>RandomForest</td>
<td>94.2</td>
<td>99.2</td>
<td>98.4</td>
</tr>
<tr>
<td>SVM</td>
<td>94.2</td>
<td>97.9</td>
<td>98.1</td>
</tr>
<tr>
<td>Voted Perceptron</td>
<td>92.9</td>
<td>96.1</td>
<td>97.9</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>81.2</td>
<td>93.8</td>
<td>90.2</td>
</tr>
</tbody>
</table>

Table 2 displays experimental results for our API Call-grams approach. The results of 1 API-gram displays an accuracy of 84%. API are primarily developed for fast and convenient use of system resources, false positives increase due to their presence in sane and malcode. Two API Call-grams show an improvement over single API Call-grams. 3 API call-grams show a significant increase in accuracy from 99.5 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.

5. CONCLUSIONS

This poster discusses Call Graph mined as API Call-grams to analyst code obfuscated malware to minimize false positives. Abstract representation of a program is obtained from the sequence of API calls made by a file during static CFG extraction. Novelty of our approach is considering API Call-grams. API-grams consider the sequence in which calls are made leading to high detection rate.

6. REFERENCES