IMAD: In-Execution Malware Analysis and Detection

Syed Bilal Mehdi
nexGIN RC
NUCES-FAST
Islamabad, Pakistan
bilal.mehdi@nexginrc.org

Ajay Kumar Tanwani
nexGIN RC
NUCES-FAST
Islamabad, Pakistan
ajay.tanwani@nexginrc.org

Muddassar Farooq
nexGIN RC
NUCES-FAST
Islamabad, Pakistan
muddassar.farooq@nexginrc.org

ABSTRACT

The sophistication of computer malware is becoming a serious threat to the information technology infrastructure, which is the backbone of modern e-commerce systems. We, therefore, argue the need for developing sophisticated, efficient, and accurate malware classification techniques that can detect a malware on the first day of its launch–commonly known as "zero-day malware detection". To this end, we present a new technique, IMAD, that can not only identify zero-day malware without any apriori knowledge but can also detect malicious process in-execution. The capability to recognize a malware while it is executing, empowers an operating system to immediately kill it before it can cause any significant damage. IMAD is a realtime, dynamic, efficient, in-execution zero-day malware detection scheme, which analyzes the system call sequence of a process to classify it as malicious or benign. We use Genetic Algorithm to optimize system parameters of our scheme. The evolutionary algorithm is evaluated on real world synthetic data collected from a Linux system. The results of our experiments show that IMAD achieves more than 90% accuracy in classifying running processes as benign or malicious. Moreover, our scheme can classify approximately 50% of malware within first 20% of their system calls.

Categories and Subject Descriptors

D.4.6 [**Software**]: Security and Protection—*Invasive software*; C.2.0 [**Computer Systems Organization**]: Computer Communication Networks—*Security and protection*

General Terms

Algorithms, Experimentation, Security

Keywords

System Call, Malware, Classification

GECCO '09 Montreal, Canada

1. INTRODUCTION

The increasing number and dangerous consequences of malware attacks is posing a serious threat to the security challenges of new millennium. The imposters or intruders are now focusing their attention towards stealthy malware which reaches vulnerable hosts and then stays undetected on the hosts. "The longer a threat remains undiscovered in the wild, the more opportunity it has to compromise computers before measures can be taken to protect against it. Furthermore, its ability to steal information increases the longer it remains undetected on a compromised computer" [1]. A recent outbreak of well-known Conficker worm in Jan 2009 which affected more than 15 million computers all around the world further validates the claim [3].

The true shortcoming of existing malware anti-virus products is their legacy of sticking to the signature-based technology. This paradigm follows the following workflow: (1) malware causes intended damage, (2) forensic experts of anti-virus companies get its samples to study its behavior, (3) they assign it a signature which is effectively a sequence of its instructions, (4) the signature is added to the database, (5) customers are notified to update their signatures' database, (6) customers update their database and are finally protected against this malware. A careful reader can easily conclude that this signature-based technology suffers from two main shortcomings: (1) inability to detect an attack at the day of its launch–known as "zero-day malware detection", (2) inability to cope with an exponential growth in new malware [2].

Security experts are now focusing their attention on non-signature based zero-day malware detection techniques and it is becoming an active area of research. To this end, they have proposed two types of techniques: (1) static, and (2) dynamic. The static techniques take an executable file as an input and apply different data mining techniques to classify it as malicious or benign. In contrast, dynamic techniques analyze the runtime behavior of a process and then raise an alarm if the process is detected as malicious. The real challenges in the design of dynamic malware detection techniques are to achieve: (1) high detection accuracy, (2) low false alarm rate, (3) low processing overhead, and (4) in-execution detection before malware causes any significant damage. To the best of our knowledge, no existing dynamic techniques meet the requirements (3) and (4); therefore, they are mostly considered "not suitable for deployment on real-world computer systems". Security experts believe that if a dynamic technique that meets the aforementioned requirements can be developed then it will be resilient to

code obfuscation, self-encryption and polymorphic transformations of a stealthy malware because a malware finally has to do the desired malicious activity while it is executing. However, these transformations can easily defeat well-known static techniques.

The major contribution of this paper is our dynamic scheme IMAD that is not only efficient and accurate but also has the ability to detect in-execution zero-day malware. The novel dimension of our scheme is that it maintains an 'impression' of a running process by analyzing its sequence of system calls. The 'impression' of a process is a constantly varying measure of the malicious activity level of the process; the moment it exceeds above a threshold value, the process is declared as malicious and is immediately killed to limit any significant damage. We have evaluated our scheme on a real-world collection of Linux viruses from VX Heavens [12]. The results of our experiments show that our scheme achieves more than 90% classification accuracy with 0% false alarm rate. It is light on processor and can detect 50% of malware within first 20% of system calls. Therefore, we can safely conclude that we are able to meet almost all of the outlined requirements of a next generation malware security solution. Our system can be easily realized inside the kernel of Linux and therefore becomes our last line of defence–once the malware has successfully penetrated to all anti-malware shields and now is ready to execute.

The rest of the paper is organized as follows. We explain the nature of our dataset in Section 2. The proposed IMAD framework is presented in Section 3 followed by its evaluation in Section 4. Strengths and limitations of the technique are summarized in Section 5. We describe some of the important related work in this context in Section 6. Finally, the paper is concluded in Section 7 with an outlook towards our future work.

2. DATASET

We use our customized system calls logger module (will be shortly introduced) to log system calls of benign and malicious processes. We have collected 500 Linux Malware from publicly available VX Heavens collection that include different types of malware like Backdoors, Flooders, Hacktools, Rootkits, Trojans, Viruses and Worms repository [12]. We manually executed them on a virtual Linux machine so that it can be easily restored to its original uninfected state after each successful malware run. The logged data of system calls is stored on an attached USB drive which ensures that the data remains intact if we restore the virtual machines. This process, however, cannot be automated because it requires user intervention in running the application with the malware, unmounting the USB and then restoring the virtual machine.

During the logging process, we encountered some unexpected problems in collected Linux Malware. To our surprise, some of them are not even ELF files – *ELF or Executable Linkable File is a standard binary file format for UNIX and UNIX-like systems* – and most of the remaining ELF files result in either a segmentation fault or a syntax error. A few require some other applications–which are not easily available–for their execution. Another interesting observation is that some of them even detect that they are being traced by giving the message "debugging detected". In this scenario, we lost confidence in their system calls as we do not know if they perform malicious activity while knowing that they are being traced. Therefore, we have removed these files from our dataset. After doing this interesting laborious study, we have finally identified 100 malware files that successfully executed.

For benign processes, we have selected benign applications that are available in `/bin`, `/sbin` and `/usr/bin` folder of Linux. We used the above-mentioned process to log the sequence of system calls of 180 benign processes as well.

3. ARCHITECTURE OF IMAD

In this section, we present the architecture of our IMAD scheme. The important components of our scheme are: (1) system calls logger, (2) n-grams generator, (2) n-grams analyzer, (3) goodness evaluator, (4) ngc786 in-execution classifier, (4) genetic optimizer, and (6) alarm generator (See Figure 1). We now discuss each component in detail.

3.1 System Calls Logger

The success of our scheme is directly dependent on the sequence of system calls of a process; therefore, it is extremely important to collect real dataset of 'system calls' sequence of processes running on Linux computer. We have developed a customized application–similar to existing Linux application 'strace'–to log the trace of system calls of a process. Our application uses ptrace system call that logs all system call of a process–by rehashing their name to a number $(0 - 324)$–running on Intel Core 2 Duo 1.8GHz architecture with Linux 2.6.23.1 kernel. Our application starts the test process as a child and then records its system calls.

3.2 n-grams Generator

Once we have collected raw sequence system calls of malicious and benign files, we represent them in a sequence of n-grams. *n-gram is a fixed sized window of a sequence of system calls where 'n' represents the size of the window.* The representation of a sequence of system calls by n-grams has been widely used by a number of researchers to detect malware [6][7][8]. For example, the sequence '1 2 3 4 5 6 7 8' can be transformed into '1 2 3 4', '2 3 4 5', '3 4 5 6', '4 5 6 7' and '5 6 7 8' 4-grams or '1 2 3 4 5 6', '2 3 4 5 6 7' and '3 4 5 6 7 8' 6-grams. .

The size of a window plays a crucial role in the representation of any sequence. If the size of the window is too small, n-grams fail to represent any useful information. We use an example to elaborate this point: if the size of the window is 2, then the sequence '1 2 3 2 1 2' transforms to unique n-grams: '1 2', '2 3', '3 2' and '2 1'. Similarly, the trace '3 2 3 2 1 2' transforms to '3 2', '2 3', '2 1' and '1 2' n-grams which are the same as of the previous trace. As a result, it is not possible to differentiate two traces from one another. But if the size of window is 4, the sequence '3 2 3 2' is only present in the later sequence; consequently, we can differentiate between the two traces. However, the greater size of the window leads to larger number of unique n-grams and thus increases the complexity of the corresponding algorithm.

We have done some pilot studies to choose an appropriate n-grams size. The outcome of our pilot studies is that 4 and 6 grams provide good classification accuracy at a reasonable processing overhead. Therefore, in rest of the paper we only show results for 4 and 6 grams.

3.3 n-grams Analyzer

The n-grams generator creates two different datasets for 4 and 6 grams respectively. The role of n-grams analyzer is to categorize unique n-grams as malicious, benign and neutral. n-grams that are found only in the traces of malicious processes are called 'malicious n-grams' in the rest of the paper. Similarly, the n-grams that are present only in the traces of benign processes are termed as 'benign n-grams'. The n-grams that are found in the traces of both malicious and benign processes are categorized as 'neutral n-grams'.

Our 4-grams dataset contains 7045 unique n-grams. The distribution of these unique n-grams is: 1401 malicious n-grams, 4692 benign n-grams and the remaining 952 are neutral n-grams. In comparison, for 6-grams dataset, we have $10,005$ n-grams–approximately 50% more compared with 4-grams dataset. These unique n-grams contain 2356 malicious n-grams, 6279 benign n-grams while the rest of 1370 are neutral n-grams.

3.4 Goodness Evaluator

This component assigns a 'goodness value' to each unique n-gram. The goodness values are then used to calculate an overall impression value of a process. A higher impression value of a process results in a higher probability of declaring a process as benign. Use of 4-grams and 6-grams gives us large number of unique n-grams. Therefore, it makes perfect sense to cluster n-grams on the basis of their category determined by n-grams analyzer. The goodness evaluator assigns highest goodness value of +1 to benign n-grams and lowest goodness value of -1 to malicious n-grams. The evaluator uses a genetic optimizer to assign goodness values to neutral n-grams. The method of assigning goodness values is explained in detail in Section 3.6.

3.5 ngc786 Classifier

We now discuss our ngc786 (next generation classifier). The motivation for developing our own classifier is that we want to classify time series data with minimum number of possible attributes (n-grams). To the best of our knowledge, all existing classifiers take a fixed length feature vector. The length is simply the number of unique n-grams in our case. We need to run a given process from start to end to know the presence or absence of different n-grams and then accordingly assign values to the feature vector table. This classification paradigm does not allow us to do in-execution classification with only partial information about the feature table. In order to accomplish this requirement of in-execution classification with variable length feature vector, we have developed ngc786.

To this end, we provide a growing length sequence of n-grams generated by n-grams generator. The classifier has some goodness values of all of unique n-grams provided by the goodness evaluator and uses these goodness values to incrementally calculate a coefficient for an executing process–which we call 'impression coefficient'. This impression coefficient is updated for every new system call or an n-gram using the following formula:

$$I \leftarrow G + \alpha I \tag{1}$$

where I stands for the impression coefficient of the process, G is the goodness value of the last n-gram and α represents the smoothing factor. We now discuss the significance of three parameters–goodness of n-grams, upper and lower bounds of impression coefficients (explained in Section 3.5.1) and the smoothing factor–given as an input to our ngc786 classifier.

We have already discussed in the goodness evaluator component that how we assign goodness to n-grams (G). The upper and lower bounds (B_u & B_l) respectively of the impression coefficient should be tuned for achieving higher classification accuracy. The important constraint that these values should be within certain limit of zero. If smoothing factor is 0.5, the impression coefficient can have a maximum value of 2. Similarly if the bounds are set near to zero, then premature classification–correct or wrong–will result because it is very difficult to keep the impression in between two close bounds. Finally, the value of smoothing factor α determines the number of last α n-grams in calculating the smoothing factor. If it is kept too low, the impact of history is reduced and if we keep it too high then the current values do not play a significant role in calculating the impression coefficient.

3.5.1 Alarm Generator

The value of impression coefficient is checked at its each update that happens after every system call. The moment the impression coefficient of a process goes below its lower bound, it is declared malicious and an alarm is raised. Similarly, if the impression coefficient increases above an upper bound, the process is declared as benign. The pseudo-code of ngc786 and the alarm generator is provided in Algorithm 1.

If the impression coefficient lies in between the upper and the lower bound, our classifier leaves the process unclassified at this moment because it does not have enough information. This approach simply delays the decision of classification with the hope that at next decision point the alarm generator might have sufficient information to raise an alarm. In the worst case scenario, the complete process might execute and we are unable to classify it. Since we are developing an in-execution classifier, it has no way of knowing a priori the length of feature vector.

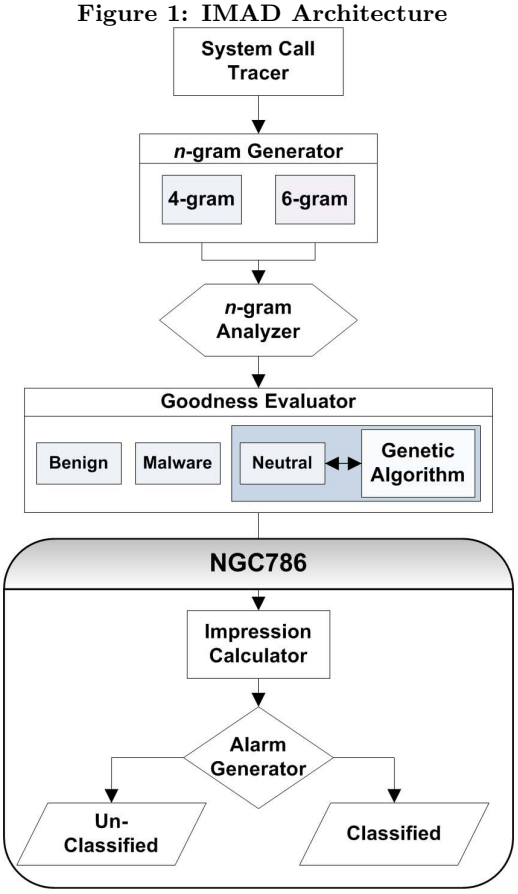
Algorithm 1 *ngc786*

```
procedure Classification algorithm for ngc786
while Process is unclassified and Process is in execution do
    Wait for next system call
    n_g ⇐n-gram of last n system calls in Sequence;
    I ⇐ n_g + α * I {α is smoothing coefficient}
    if I > B_u {B_u is upper bound} then
        Hypothesis ⇐ Benign;
    else if I < B_l {B_l is lower bound} then
        Hypothesis ⇐ Malicious;
    end if
end while
```

3.6 Genetic Optimizer

It is obvious that the classification accuracy of IMAD is dependent on a number of parameters. These parameters should be tuned/optimized for best performance. Our classifier needs to tune goodness of 952 and 1370 neutral n-grams for 4-grams and 6-grams respectively. Moreover, we need to optimize three additional parameters: smoothing factor, upper bound and lower bound of the impression coefficient. As a result, we have to optimize (952+3) or (1370+3) variables for 4-grams and 6-grams respectively. The realtime dynamic optimizer, therefore, has to tune a large number

Figure 1: IMAD Architecture



of parameters by taking into account their mutual complex dependencies.

We use a standard Genetic Algorithm (GA) for this purpose because GAs are known to give good results in real-time dynamic environments. In GA, our chromosome for 4-grams case has 955 genes, one for each corresponding variable. Similarly for 6-grams, our chromosome has 1373 genes. Each chromosome actually represents one complete training instance of our ngc786 classifier. We now assign the fitness value to each instance by using the following formula:

$$F = \frac{P}{TP} + \frac{N}{TN} + \frac{FP}{P} + \frac{FN}{N} \tag{2}$$

where,
TP = Number of true positives,
TN = Number of true negatives,
FP = Number of false positives,
FN = Number of false negatives,
P = Total number of positives,
N = Total number of negatives,
F = Fitness value.

A chromosome with small fitness value is more desirable than a chromosome with high fitness value. If a classifier randomly decides to classify all processes as benign or malicious, then this would lead to 50% classification accuracy– which of course is not acceptable. However, our fitness formula heavily penalizes this situation because it leads to an infinite fitness value–that represents the worst possible classifier.

Algorithm 2 *IMAD algorithm*

procedure Fitness Calculation for Training/Optimization of IMAD (Training instances, goodness values of all n-grams, α, B_u, B_l)
$TP \Leftarrow 0$
$TN \Leftarrow 0$
$FP \Leftarrow 0$
$FN \Leftarrow 0$
$P \Leftarrow 0$
$N \Leftarrow 0$
while More training instances available **do**
  New Instance = Next Instance in Training Instances
  Sequence = System Call Sequence of New Instance
  if New Instance is Benign **then**
    $P \Leftarrow P + 1$
    $Actual \Leftarrow Benign$
  else
    $N \Leftarrow N + 1$
    $Actual \Leftarrow Malicious$
  end if
  $Hypothesis \Leftarrow Unknown$;
  while $Hypothesis == Unknown$ and Process is in execution **do**
    $n_g \Leftarrow$goodness value of last n-gram in Sequence;
    $I \Leftarrow n_g + \alpha * I$ {α is smoothing coefficient}
    if $I > B_u$ {B_u is upper bound} **then**
      $Hypothesis \Leftarrow Benign$;
    else if $I < B_l$ {B_l is lower bound} **then**
      $Hypothesis \Leftarrow Malicious$;
    end if
  end while
  if Hypothesis == Malicious AND Actual == Malicious **then**
    $TP \Leftarrow TP + 1$
  else if Hypothesis == Benign AND Actual == Benign **then**
    $TN \Leftarrow TN + 1$
  else if Hypothesis == Malicious AND Actual == Benign **then**
    $FP \Leftarrow FP + 1$
  else
    $FN \Leftarrow FN + 1$
  end if
end while
$Fitness \Leftarrow P/TP + N/TN + FP/P + FN/N$

The fitness formula also caters for unclassified instances. For example a malicious file declared as benign will not only decrease TP but also increases FN, whereas the same malicious file left unclassified will just decrease TP only. Consequently, an individual that leaves an instance unclassified is more fitter compared with the one that incorrectly classifies it. Moreover the fitness tries to strike a balance in the percentage of correctly classified benign processes and correctly classified malicious processes. For example, if we keep incorrectly classified instances constant, then an individual that correctly classifies 90% malicious and 100% benign processes is inferior in terms of fitness compared with the one that correctly classified 95% of both types of processes.

In our training sessions, we have used a population of 1000 for 50 generations with mutation and crossover fractions of 0.2 and 0.8 respectively. This configuration of GA is finalized after doing more than dozen of pilot studies but their details have been skipped for brevity.

The complete fitness function criteria for Genetic Algorithm is described in Algorithm 2

4. RESULTS AND DISCUSSION

We now provide results obtained from our experiments. We first focus on the accuracy of our IMAD scheme compared with other well-known classifiers. The purpose of this analysis is to understand the accuracy behavior of our scheme. However, we must emphasize that the comparison

is slightly unfair as no other compared classifier has the ability to do in-execution detection. Rather all of them–except our IMAD–create a fixed length feature vector once the malware has completely executed. Therefore, detecting a malware once it has executed and caused the damage is no more than an academic research having virtually no value in becoming an integral part of existing anti-virus products. We also provide an insight about how much executing malware is detected in how many system calls.

4.1 Detection Accuracy

To provide a systematic analysis, we use 10-fold cross validation technique to determine classification accuracies of all algorithms. In this technique, the dataset is split into 10 random parts. Nine parts are used for training and 1 part is used for testing. This process is repeated 10 times to ensure that each part is tested at least once.

Remember that if IMAD does not have enough information then it leaves the process unclassified in the hope that it might be able to classify it in future. Tables 1 and 2 show this behavior of IMAD. In case of 4-grams, IMAD leaves 36 processes as unclassified which is approximately 12% of total number of files. This number increases to 18.21% in case of 6-grams. One can conclude from both tables that the true benefit of going from 4-grams to 6-grams is that the number of misclassifications decreases. The number of misclassified benign files goes from 7 to 0 and similarly for the misclassified malicious files it goes from 3 to 1. We also see an increase in the number of correctly classified malware processes from 75 to 77. The downside of leaving processes unclassified is about $12 - 18\%$ of total processes at the end of execution. For practical purposes, the effect is the same as if the process would have been benign; therefore, we ultimately classify unclassified processes as benign at the end of their execution.

Table 1: Performance of IMAD on 4-gram dataset

	Classified as Benign	Classified as Malicious	Left as Unclassified	Total
Benign	159	7	14	180
Malicious	3	75	22	100
Total	162	82	36	280

Table 2: Performance of IMAD on 6-gram dataset

	Classified as Benign	Classified as Malicious	Left as Unclassified	Total
Benign	151	0	29	180
Malicious	1	77	22	100
Total	152	77	51	280

4.2 Comparison

We now compare accuracy of our IMAD scheme with four well-known classification techniques namely Support Vector Mahcine(SVM), RIPPER, C4.5 and Naive Bayes. Remember that the comparison is made to analyze detection accuracy of IMAD with benchmark classifiers; otherwise–as mentioned before–none of them has in-execution malware detection capability.

Support Vector Machines: The concept of Support Vector Machine (SVM), invented by Vladimir Vapnik, in its simplest form aims to develop a hyperplane that separates a set of positive samples from a set of negative samples with a maximum margin [16]. For linear separability of the problem space, SVMs use a kernel function for mapping training data to a higher-dimensional space. We are using SMO (Sequential Minimal Optimization) which is a fast and efficient SVM training algorithm implemented in WEKA [16].

C4.5: Decision trees are usually used to map observations about an item to draw conclusions about the item's target value using some predictive model[13]. They are very easy to understand and are efficient in terms of time especially on large datasets. They can be applied on both numerical and categorical data, and statistical validation of the results is also possible. We use C4.5 decision tree (J48) that is implemented in WEKA. We use the default parameters for J48.

RIPPER: We also use a propositional rule learner, Repeated Incremental Pruning to Produce Error Reduction (RIPPER), proposed by William W. Cohen as an optimization of IREP [15]. RIPPER, performs quite efficiently on large noisy datasets with hundreds of thousands of examples. It caters for missing attributes, numerical variables and multiple classes. The algorithm works by initially making a detection model composed of rules which are improved iteratively using different heuristic techniques. The constructed rule set is used to classify the test cases. We use default parameters for RIPPER in WEKA.

Naive Bayes: Naive Bayes is a simple probabilistic classifier that assumes independence among the features i.e. the presence or absence of a feature does not affect any other feature [14]. The algorithm works effectively and efficiently when trained in a supervised learning environment. Due to its inherent simple structure it often gives very good performance in complex real world scenarios. The maximum likelihood technique is used for parameter estimation of Naive Bayes models.

Table 3: Performance of classifiers on 4-gram dataset

Classifier	TP	TN	FP	FN	Detection Accuracy	False Alarm Rate
IMAD	75	173	7	25	75.00%	3.89%
SVM	64	167	13	36	64.00%	7.22%
C4.5	85	150	30	15	85.00%	16.67%
RIPPER	79	153	27	21	79.00%	15.00%
Naive Bayes	64	162	18	36	64.00%	10.00%

Table 4: Performance of classifiers on 6-gram dataset

Classifier	TP	TN	FP	FN	Detection Accuracy	False Alarm Rate
IMAD	77	180	0	23	77.00%	0.00%
SVM	62	163	17	38	62.00%	9.44%
C4.5	70	153	27	30	70.00%	15.00%
RIPPER	77	149	31	23	77.00%	17.22%
Naive Bayes	58	154	26	42	58.00%	14.44%

We now tabulate the results of our comparative study obtained from 4-grams and 6-grams in Table 3 and Table 4 respectively. It is clear from both tables that IMAD benefits from using 6-grams because its detection accuracy goes from 75% to 77% and the false alarm rate goes down to zero

from 4%. The true benefit of using IMAD is its relatively comparable accuracy but with almost 0% alarm rate.

We believe that IMAD is able to significantly outperform SMO because SMO classifies an instance by looking at the complete feature vector. This global view of the feature vector significantly degrades the accuracy because it directly depends on the dominated values. For example, if trace of a malicious process contains 100 neutral n-grams and 2 malicious n-grams, then SMO's result are dominated by 100 neutral n-grams rather than 2 malicious n-grams; consequently it can lead to a misclassification. In comparison, IMAD–because of the goodness evaluation component–has a better probability of classifying this malicious process.

We have analyzed the number of rules in case of JRIP and the number of leaves in case of J48. Our observation is that JRIP has limited number of attributes in the rules and similarly J48 has small number of nodes in decision trees. Consequently, both JRIP and J48 are unable to use all benign and malicious n-grams; rather, use only a limited number of n-grams.

4.3 Analysis of In-Execution Detection Time

Now we focus our attention to the in-execution malware analysis and detection feature of IMAD. Specifically, we are interested in analyzing the number of n-grams needed by IMAD to detect malware. We normalize this number with respect to the total number of n-grams of that process. The outcome of this study will provide an insight that how much percent on average of malware has executed before it is detected. It might be of direct interest to know that how many malware are detected within 20% of their execution trace from the beginning. Similarly it will provide little value, if a malware is detected after it has executed 95% of its trace.

We present the results of our technique in Figure 2 and Figure 3 for 4-grams and 6-grams dataset. In these figures, detected malware are grouped together on the basis of the percentage of n-grams required before detection–smaller percentage is desirable because it implies earlier detection. The area covered by pie depicts the fraction of malware identified in a particular range. We see that in case of 6-gram dataset, IMAD is able to detect almost 50% of malware within their first 20% of n-grams.

We, however, emphasize that a late detection towards the end of its execution may not necessarily mean that it has caused its intended damage to the computer system. It is our observation that most of stealthy malware tries to hide

Figure 2: 4-grams dataset: Pie-Chart showing identified malware according to percentage of execution before identification

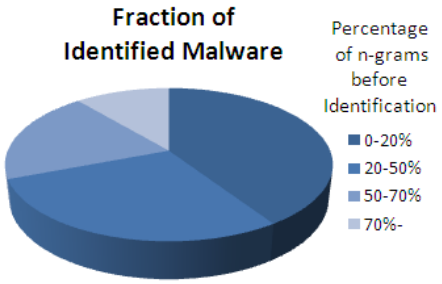
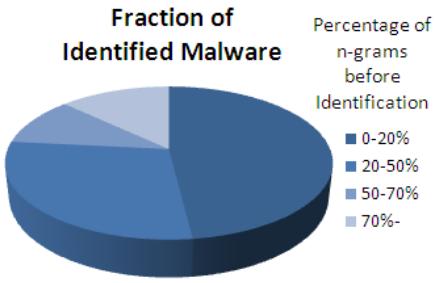


Figure 3: 6-grams dataset: Pie-Chart showing identified malware according to percentage of execution before identification



itself in the beginning with an aim to cause the real damage towards the end. In this scenario, even late detection is of significant value compared with no detection. On the other side, it is also possible that malware performs malicious activities at the beginning of its execution and afterwards gets detected in the middle of its execution. In such a case, IMAD may not completely achieve its desired objective.

We now show a detailed histogram that shows the fraction of n-grams needed by IMAD before detecting in Figure 4 and Figure 5 for 4-grams and 6-grams.

Figure 4: The Histogram shows percent of 4-grams needed before detection of malware

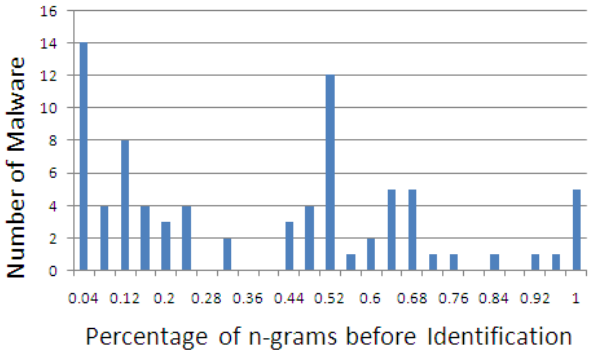
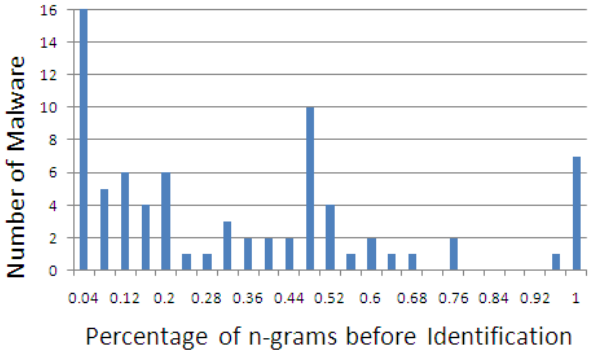


Figure 5: The Histogram shows percent of 6-grams needed before detection of malware



We can conclude that 4-grams histogram and 6-grams his-

togram are significantly similar to one another. Both of them first have a large chunk in the beginning and then in the middle. This similarity provides a useful insight about the behavior of ngc786 classifier in IMAD: it can identify any type of malware after analyzing the same percentage of n-grams irrespective of the size of window i.e 4-grams form or 6-grams.

We argue that this behavior of detecting a particular type of malware after a fixed length of trace irrespective of 4- or 6-grams indicate that the malware must have either done or at least tried to do malicious activity at this particular point in time. This strengthens our belief that our IMAD scheme is able to successfully detect a malware (in most the cases) before they can do any significant damage.

5. STRENGTHS AND LIMITATIONS

We now summarize strengths and limitations of IMAD in this section to provide a reader a useful insight of our scheme.

Our technique has following merits:

- The main strength of our IMAD is its in-execution malware detection capability.
- IMAD–even with in-execution detection–provides comparable accuracy compared with existing benchmark classifiers. However, it provides significantly better false alarm rate.
- IMAD–even with 6-grams–requires approximately 1 KB of memory. Recall that we need to maintain a table in which we associate each n-gram a goodness value. In case of 6-grams, one n-gram approximately requires 6 bytes of memory and its goodness being a real number requires 4 bytes. Recall that for 6-grams we had $10,005$ unique n-grams. Since each n-gram takes 10 bytes; therefore, total size is just $100,050$ bytes which is reasonably acceptable. For 4-grams size of memory is just $7045 * 8 = 56,360$ bytes.
- We also need to evaluate the processing overhead because IMAD achieves best performance with 6-grams. Our approach requires a processor to do–during testing phase–one floating-point multiplication, one floating-point addition and one floating-point comparison only. On a Core 2 45nm processor, floating-point multiplication, addition and comparison takes just 5, 3 and 3 CPU cycles. With a few other move instructions the calculation of goodness should take less than 20 CPU cycles. If we add n-gram generation and analysis cycles of few hundred microseconds, it is still reasonably acceptable.

The important limitations of IMAD is as follows:

- Our analysis is based on just 100 malware samples, which is relatively a small number to do any generalization of IMAD.
- The parameters such as arguments of the system calls are ignored in the experiments. If they are also considered, we can further enhance the accuracy of IMAD. However, this experimentation is our main subject of future work.
- A few simpler parameters such as the frequency of a particular system call of a particular process or the frequency of a combination of system calls of a process might have been also helpful in further improving the accuracy of the our scheme.

6. RELATED WORK

S. Forrest et al. proposed an intrusion detection system using system call sequences [6]. The system focuses on analyzing the processes that execute with higher privileges. They used one such process 'sendmail' for testing and created a database, consisting of normal system call sequences for sendmail. Later they attacked the computer with malware and compared the sequences generated by sendmail with the database. They found that sequences generated under attack are considerably different from the ones that are generated without an attack. This technique has the limited ability to classify a limited number of known process in-execution. For example, if the system is trained for sendmail, then it can not be used for any other process other than sendmail. On the other hand, we train our system on one set of processes and test it on another set that does not contain any process from the training set. G. Helmer et al. extended the work on sendmail data using a feature vector approach and used Genetic Algorithm for feature selection [10].

W. Lee. et al. trained JRIP classifier with dataset containing normal sequences and sequences generated under attack [7]. However, their work is also based on analyzing limited number of known processes.

The authors in [8] used feature vector approach for differentiating worms from benign processes. They developed a feature vector that shows the short sequences of system calls present or absent in a process's trace. The classifiers used are SVM and Naive Bayes. The main objective of their research focused on classifying unknown processes; however, they cannot block a malicious process in-execution because their analysis starts once the process has finished.

The author in [9] also collected system call sequences for various processes and divided them into different groups of system calls that belong to benign processes, malicious processes or processes under attack or both kinds of processes. However, the author did not do any empirical results to study the efficacy of immune systems for malware detection.

7. CONCLUSIONS

In this paper, we have proposed a novel in-execution realtime efficient malware detection scheme–IMAD. We argue that existing classifiers do not have the ability to operate on variable length feature vector; therefore, they cannot be adapted for in-execution requirement. Consequently, we have developed our own ngc786 classifier which can classify on the basis of variable length feature vector. This is an important contribution of the work because the classifier can be useful in many realtime critical control systems.

We have also empirically evaluated our technique with SMO, J48, JRIP and Naive Bayes classifier. Our results show that IMAD has achieved better or comparable accuracies compared with most of them, albeit with significantly smaller false alarm rate. We have also analyzed the fraction of n-grams required by IMAD to classify an executing malware. Our results show that approximately 50% of malware

are classified within first 20% of their execution. This shows that IMAD has the potential to be integrated into existing ani-virus products.

In future, we plan to increase the number of benign and malicious traces that would give us a better understanding of IMAD. Moreover, we plan to identify system calls that are more important in deciding the behavior of a process. In this way, we can significantly reduce the number unique n-grams required for classification. To conclude that in-execution malware detection provides exciting avenues to bio-inspired community for doing novel research that can offer solutions to real world security threats.

8. REFERENCES

[1] Symantec Internet Security Threat Report XI: Trends for July – December 2007, September 2007.

[2] CI Security, The Centre for Counter Intelligence and Security Studies, `http://www.cicentre.com/news/cyber_security.html`(accessed Jan 20, 2009)

[3] United Press International UPI, `http://www.upi.com/Top_News/2009/01/25/Virus_strikes_15_million_PCs/UPI-19421232924206/`(accessed Jan 26, 2009)

[4] N. Idika, A. P. Mathur, "A Survey of Malware Detection Techniques", Tehnical Report, Department of Computer Science, Purdue University, 2007.

[5] S. Kumar, "Classification and Detection of Computer Intrusion", PhD Thesis, Computer Sciences Department, Purdue University, 1995.

[6] S. Forrest, S. A. Hofmeyr, A. Somayaji, T. A. Longstaff, "A Sense of Self for Unix Processes", In Proceedings of IEEE Symposium on Security and Privacy, pp. 120-128, CA, 1996.

[7] W. Lee, S. J. Stolfo, "Data Mining Approaches to Intrusion Detection", In Proceedings of 7th USENIX Security Symposium, San Antonio, 1998.

[8] X. Wang, W. Yu, A. Champion, X. Fu, D. Xuan, "Detecting Worms via Mining Dynamic Program Execution", In Proceedings of Third International Conference on Security and Privacy in Communication Networks and the Workshops, SecureComm, pp. 412-421, Nice, 2007.

[9] A. Iqbal, "Danger Theory Mataphor in Artificial Immune System for System Call Data", PhD Thesis, Universiti Teknologi Malaysia, 2006.

[10] G. Helmer, J. Wong, V. Honavar, L. Miller, "Feature Selection Using a Genetic Algorithm for Intrusion Detection", In Proceedings of the Genetic and Evolutionary Computation Conference, GECCO, pp. 13-17, Orlando, 1999.

[11] J. H. Holland, "Adaption in Natural and Artificial Systems", University of Michigan Press, Ann Harbor, 1975.

[12] VX Heavens Virus Collection, VX Heavens website, `http://hvx.netlux.org`

[13] J.R. Quinlan, "C4.5: Programs for machine learning", Morgan Kaufmann, USA, 1993.

[14] M.E. Maron, J.L. Kuhns, "On relevance, probabilistic indexing and information retrieval", Journal of the Association of Computing Machinery, 7(3), pp. 216-244, 1960.

[15] W.W. Cohen, "Fast effective rule induction", International Conference on Machine Learning, ICML, pp. 115-123, USA, 1995.

[16] J. Platt, "Fast training of support vector machines using sequential minimal optimization", Advances in Kernel Methods Support Vector Learning, pp. 185-208, MIT Press, USA, 1998.