Towards a Theory of Generalizing System Call Representation For In-Execution Malware Detection

Bilal Mehdi[1], Faraz Ahmed[1], Syed Ali Khayyam[1,2], Muddassar Farooq[1]
[1]Next Generation Intelligent Networks Research Center (nexGIN RC)(FAST-NUCES)
Islamabad, Pakistan
Email:{bilal.mehdi,faraz.ahmed,muddassar.farooq}@nexginrc.org
[2]School of Electrical Engineering & Computer Science (SEECS)(NUST)
Islamabad, Pakistan
Email:ali.khayam@seecs.edu.pk

***Abstract*—The major contribution of this paper is two-folds: (1) we present our novel variable-length system call representation scheme compared to existing fixed-length sequence schemes, and (2) using this representation, we present our in-execution malware detector that can not only identify zero-day malware without any a priori knowledge but can also detect a malicious process while it is executing. Our representation scheme – a more generalized version of n-gram – can be visualized in a k-dimensional hyperspace in which processes move depending upon their sequence of system calls. The process marks its impact in space by generating *hyper-grams* that are later used to evaluate an unknown process according to their *profile*. The proposed technique is evaluated on a real world dataset extracted from a Linux System. The results of our analysis show that our in-execution malware detector with *hyper-gram* representation achieves low processing overheads and improved detection accuracies as compared to conventional n-grams.**

I. Introduction

There is now a unanimous consensus among the security experts that existing signature based malware detection in Commercial Off the Shelf Anti Virus (COTS AV) products can not cope with exponential growth in new malware. Therefore, researchers advocate the need of non-signature based solutions that can detect a malware on the first day of its launch commonly known as 'zero-day malware'. A number of non-signature based techniques have been proposed recently that use heuristic or behavioral analysis to detect malware [1], [3] and [4]. But unfortunately, most of such existing techniques lack the ability to identify malware during its execution leaving them within the boundaries of forensic analysis. We are interested in developing computer security solutions that can dynamically analyze a process in-execution to classify it as malicious or benign.

The major limitation of many of existing non-signature based techniques is that they use fixed-length byte distribution – commonly known as n-grams – to represent sequence of system calls in a process [6], [7] and [8]. They store all the information which is accompanied by large memory and processing overheads vital for their realtime deployment. Moreover, the intruders' desire to generate stealthy malware in order to gain benefits by reaching vulnerable hosts make conventional fixed length n-gram representation, due to its simplicity, more susceptible.

In this paper, we show that the requirements of the next generation security solutions can be achieved by using better sequence representation schemes. To address the limitations of conventional n-gram representation, we present our more generic hyper-gram representation that has the capability to store variable length generalized information. Our representation scheme reflects short sequence of system calls in a k-dimensional hyperspace where every dimension represents a particular system call. The process while executing marks its impact on a hyper-gram in that hyperspace according to its observed sequence of system calls. The efficacy of hyper-grams is achieved by tuning three transformation parameters for every dimension – addition (α), diminishing (δ) and slope (β). We use genetic algorithm to optimize these parameters for every dimension. Using this representation, we present our in-execution malware detector that can not only detect a zero-day malware but can also kill a process during execution if it shows some significant malicious activity. The proposed in-execution malware detector is evaluated on a real world collection of Linux processes. The results of our analysis show that our proposed in-execution malware detector has improved the detection accuracy as compared to the conventional $n-$gram and has low processing overheads in terms of machine cycles.

Organization of the Paper. The remaining paper is organized as follows: We discuss the limitations of existing system call representations in Section II. The transformation of hyper-grams and tuning parameters are explained in Section III and IV. Then we present our in-execution malware detector followed by experimentation details and results. Finally, we conclude the paper with an outlook to our future work.

II. Limitations of Existing System Call Representation Techniques

During execution some critical system calls occur, it would be better to keep information about their presence for longer periods and for other calls, it might be better to diminish this information faster and earlier. This flexibility is completely lacking in n-gram representation. For efficient representation of the behavior of a process information stored should have the following characteristics: (1) The 'sequence' of system calls is important rather than individual system calls themselves. The

Short Sequence $\xrightarrow{\text{Transformation}}$ A Point in Space

sequence information should be from a little window back in time. Otherwise if we continue to just add information, the chunk of information formed at the end of a process will not only be too large but also too specific to the process. Therefore, the old information has to be lost as we move forward in a process. (2) As every system call has its own functionality, it is necessary to keep information about every system call in its own independent way. This property has been ignored by all of the previously presented techniques and is one of the most important dimension of our novel scheme. (3) The information has to be generalized leading to only a small number of unique information about sequences that describes the characteristic behavior of a benign or malicious process. (4) And most importantly, the information should be kept in a way so that the classification occurs during execution rather than after execution.

To the best of our knowledge, all the system call representation techniques lack our second, third and fourth objective. We now present our novel system call representation technique that can achieve all the aforementioned goals.

III. k-Dimensional Hyperspace

Our system call representation scheme uses a transformation procedure to reflect short sequences of system calls in a $k-$dimensional hyperspace where k is equal to number of unique system calls.

In other words, we declare that every unique point in the space represents some characteristics of a sequence. So we place a process P_r at a position P in the space if the point P represents the same characteristics as observed in the short sequence generated by the process P_r. As the process generates newer system calls, the characteristics of the sequence change and so does its position. This can be visualized as a process moving in a multi-dimensional space. As the process terminates after generating many short sequences of system calls; we see a path followed by it in a k-dimensional space. However, it is to be emphasized that we do not remember the path and consider each point in that path as one independent point. Consider Figure 1, if a process generates System Call 'A', 'B' and 'C', its position moves from point 'A' to point 'B' to point 'C' and then terminates. We just mark 'A', 'B' and 'C' as visited by the process rather than remembering the succession in which these points are reached.

1. We represent information about sequence: As each of these points ('A', 'B' and 'C') represents characteristics of sequence rather than individual system calls, therefore, marking these points satisfies our first objective. Moreover, marking points rather than the whole path enables us to loose information about the complete sequence.

2. Every dimension represents one type of system call: One of the most important premise of our work is the independent representation of every system call. So, in our first transformation step we represent each system call as a separate dimension. In other words, the position along each dimension in the space is affected according to appearance or non-appearance of the corresponding system call only. For example, if we have system call 'A' representing the first dimension in our space, then the position along the first dimension will be affected only by the system call 'A'.

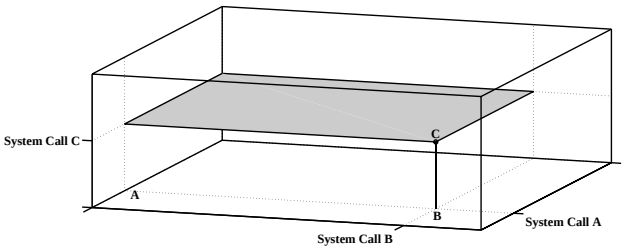

Fig. 1. Hypergram Visualization

3. We diminish, add and generalize information: We follow a three-step methodology to ensure that each point represents information about short sequence rather than the whole sequence: (1) Lose or 'Diminish' information with the passage of sequence by multiplying the position along every dimension by *diminishing factor* (δ) where $\delta \in [0,1]$. (2) 'Add' information with appearance of system call by moving the process forward in that dimension depending upon the current position along the dimension. The increase in the position along the dimension is a function of two parameters – *addition factor* (α) and *sloping factor* (β). (3) 'Generalize' information by truncating position along all dimensions. The resulting point in space is called a **'hyper-gram'**.

4. Mathematical transformation: Consider a process P_r placed at the origin of a k-dimensional hyperspace. Then, the position P of the process P_r is updated with every new system call S_i. As soon as S_i appears, we diminish old information by multiplying position of the process along all dimensions with diminishing factor δ specific to every dimension.

$$P(i) = P(i) * \delta(i) \tag{1}$$

The smaller the value of δ, the faster is the diminishing rate of information along that dimension. After diminishing, we add information of S_i along its dimension k_i using:

$$P(i) = P(i) + \alpha(i) * \beta(i) \,/\, (\beta(i) + P(i)) \tag{2}$$

The addition factor (α) and sloping factor (β) control the significance of adding information along the dimension k_i. Larger α means a larger leap along that dimension, thereby, making appearance of S_i more important. The factor $\beta/(\beta + P)$ ensures that the size leap taken by the process becomes smaller as the current position along the dimension gets higher. This limits the ability of the process to go beyond a certain range forcing generalizing to some extent. A higher value of β limits this effect: if $\beta \to \infty$, then $P(i) = P(i) + \alpha(i)$ irrespective of the current position along the dimension.

5. We truncate position to generate hyper-gram. The hyper-grams are generated by truncating the position of the process to the closest lower integer along all dimensions.

In Section II, we discussed two drawbacks with n-gram representation: (1) equal treatment of all system calls, and (2) less generalized information. The first drawback can be attributed to the same coefficients of α and δ for every dimension. Whereas, the low δ of 0.5 is responsible for lack of generalization.

To provide every dimension with a unique way of keeping information about its system call, we make α and δ for all dimensions independent of one another. For example, a dimension having α of 1 and δ of 0.1 will keep information for not more than one system call. Whereas, a dimension with α of 3 and δ of 0.9 will keep information for more than 10 system calls. However, if the system call appears once and then does not appear for the next few system calls, the position along the dimension will diminish. This conveys a very generalized information of our hyper-grams as compared to conventional n-grams.

IV. Optimization of Tuning Parameters

We showed that tuning parameters can be varied to get a better representation of information, but in order to get the best mapping of information, we must optimize the entire search space of these parameters. We define a good mapping to be one which is: (1) *'more descriptive'* – a mapping that provides necessary information to differentiate between benign and malicious processes, and (2) *'more generalized'* – a mapping that has less number of unique hyper-grams to make testing phase faster.

The first requirement of mapping being more descriptive, forces malware and benign processes to visit different points in space so that the two types of processes can be differentiated. Whereas the second condition forces all the processes to limit their movement within smallest region possible i.e visit same points over and over. Thus for making a mapping that is good by our definition, the whole challenge is to balance these two contradictory conditions such that we have a small visited space but every point has maximum meaning in it.

A. Optimization Technique

Genetic Algorithms have proved to be good for non-linear optimization. We, therefore, have used them to optimize our system parameters (αs, βs and δs). The primary requirement for optimization using Genetic Algorithm is to define a fitness function – the feedback to the algorithm. The algorithm tries to maximize this function. Now we define our fitness function in detail.

B. Fitness Function

To give each hyper-gram its fair share of contribution to the fitness score, we give each hyper-gram a score and then sum all hyper-grams' score to calculate the complete fitness score of the mapping.

First of all we identify two attributes that would describe a hyper-gram's score: its (1) Descriptiveness and (2) Credibility. To find the total score of the point, we multiply these two parameters so that if either the descriptiveness or the credibility of a point is zero, the score of that point drops down making no contribution to the total score of mapping.

$$Score = Descriptiveness * Credibility \tag{3}$$

where,

$$Fitness\ of\ a\ mapping\ = \sum_{visited\ hyper-grams} Score \tag{4}$$

1) Descriptiveness: A good space must be more descriptive and we have defined descriptive space as the space in which benign and malicious processes visit different points. So to give score of descriptiveness to a point in space, we have to define descriptiveness of these points rather than the whole space.

First, we assert that the descriptiveness of a point that was visited equal times by malware and benign processes, should be zero – because this point is showing no separation in the two kinds of processes. Second, the descriptiveness of a point that was visited a few times by one type of processes (let us say malware) but was never visited by the other type of processes (benign) should be maximum – because this point is showing maximum separation in the two kinds of processes. For simplicity we do not go into further details for a point's descriptiveness and define it in terms of percentage representation of the dominant type of process i.e the type of processes that has visited the point more times than the other. The formula for descriptiveness is:

$$Descriptiveness = \begin{cases} \frac{M}{M+B} - 0.5 & \text{if } M > B \\ \frac{B}{M+B} - 0.5 & \text{if } M < B \end{cases} \tag{5}$$

Subtracting 0.5 ensures that a point that was visited equal times by the two kinds of process gets no credit for descriptiveness. Whereas, the value of descriptiveness increases linearly with the percentage of visits of dominant processes.

2) Credibility: In order to make information generalized and compact we give high score to hyper-grams that get more visits by pulling processes towards them and hampering them from visiting newer unique hyper-grams.

Mathematically we define credibility as follows:

$$Credibility = \begin{cases} M - 1 & \text{if } M > B \\ B - 1 & \text{if } M <= B \end{cases} \tag{6}$$

Credibility of a hyper-gram increases with the increase in the number of visits by dominant process.

3) Another way to look at scoring procedure. : We give a score of 1 to every process visit. If the visit is made to a hyper-gram where the process type is dominant then this score is scaled according to the hyper-gram's descriptiveness otherwise it is dropped down to zero (as the visit did not give any advantage). Furthermore, to encourage smaller mappings, every time some visit creates a new hyper-gram, its score is also taken away. We have equal number of visits in all mappings trained on the same dataset (same number of system

calls will lead to same number of visits to hyper-grams). So the only thing that is different in these mappings is where these visits are taking place. If most of them are taking place in more descriptive hyper-grams by dominant process types, the result is a mapping with greater score.

C. In-Execution classifier

Now we present our classifier that can classify processes during execution. We base our classifier on the number of malicious and benign visits for hyper-grams as this is the direct measure of malicious or benign content of the hyper-gram.

During testing phase, with every new system call generated by the process, we calculate the hyper-gram of the process. The profile of the generated hyper-gram is added to the profile of the process. So, whenever a process visits a hyper-gram usually visited by malware, the profile of the process goes more negative. As in this scenario, the profile of the process with more system calls in its trace would likely end up with more negative profile then the process with less number of system calls, we consider the average profile of the process (current profile of the process divided by the number of system calls generated yet). As soon as average profile goes below a certain level, we declare the process as malicious.

V. Experiments

To test our representation, we have used 72 benign and 72 malicious processes. The malware dataset containing Backdoors, Flooders, Rootkits, Worms and Viruses was collected from VX Heavens repository [10]. On the other hand, applications available in `/bin`, `/sbin` and `/usr/bin` folder of Linux were used for benign dataset.

Then we extracted system call sequences of the 144 collected processes by running them on a virtual linux machine. To ensure stability of the system and thus the soundness of collected data, the machine was restored to its uninfected state after every successful malware run. In the case of a process forking, the system calls of the newly formed child process were also collected.

In all the reported experiments we have used standard 2-fold cross validation procedure. Multi-fold cross validation is a well known procedure for accuracy comparison. In this procedure, the data set is divided into several sets (2, in our case). One of the set is used for testing after training on the other sets. This testing is done as many times so that each set is used for testing once.

We have used Receiver Operating Characteristics (ROC) to quantify our results. Area Under the Curve of a ROC plot is a direct measure of classification accuracy. We report AUCs of the ROC plots.

For hyper-gram representation, as discussed in Section IV-A, Genetic Algorithm may end up with different results every time it is run. To ensure accuracy in results, we have used 2-fold cross validation for hyper-gram representation 10 times. The results reported in the paper are the average of those 10 iterations and thus 10 different sequence representations.

We have found that there are only 25 system calls that, on average, occur more than once in a process. System calls appearing less than one time per process may provide an easy yet wrong way of classification for a few processes. Therefore, for comparison purposes, we have excluded such system calls making things harder for classifiers.

VI. Results and Discussions

Now we compare the accuracy of our in-execution classifier using hyper-gram representation with well-known classifiers using n-gram representation. But to select one out of many well-known classifiers we first compare them against each other with our database. We have utilized implementations of five well known classification algorithms, which are available in Waikato Environment for Knowledge Acquisition (WEKA)[14]. We have used: Inductive Rule Learner (JRIP), Decision Tree (J48), Naïve Bayes (NB), Support Vector Machines using Sequential Minimal Optimization (SMO) and Instance Based Learner (IBk). We present the accuracy achieved by these classifiers with varying window sizes on n-gram in Table I. As smaller n-grams do not have enough information about the sequence of system calls so we have considered n-grams with minimum value of $n = 5$. We can observe in Table I that Naïve Bayes has outperformed other classifiers on our dataset.

TABLE I
Mean AUC of 5 well know classifiers against n-grams

N-gram	Classifier				
	JRIP	J48	Naïve Bayes	SMO	Ibk
5-gram	72.20	76.40	**85.48**	81.25	76.20
6-gram	69.65	82.85	**87.30**	83.35	80.15
7-gram	77.45	82.85	**87.28**	82.65	78.35
8-gram	68.05	82.45	**86.03**	82.65	76.35
9-gram	75.85	80.70	**83.93**	82.65	76.50
Mean	73.23	80.38	**82.42**	79.41	77.76

We can see that the maximum AUC is for 6-gram dataset with Naïve Bayes classifier. On the other hand our hyper-gram based in-execution classifier achieves an AUC of 87.85.

A. Processing Overhead

One of the main hurdle in making an in-execution classifier possible can be its processing overhead. To estimate the processing overhead of our classifier, we have developed a C++ application. This application performs the following two steps a million times: (1) calculate the hyper-gram with assumption of appearance of a random system call, (2) emulate searching of hyper-gram in the database of 240 hyper-grams by comparing it to the database. We have compiled this application with optimization disabled to make sure that any of our 'assumptions' in application do not lead to any run-time optimization. The result of the experiment has shown that performing this process a million times without optimization takes about 37 seconds. Thus for every system call, the technique adds a latency of $37/1,000,000$ seconds or 37 microseconds. Considering thousand system calls per second, our technique would repeat the process a thousand times taking

TABLE II
MEAN AUC OF HYPER-GRAM AND N-GRAM WITH NUMBER OF UNIQUE GRAMS

	n of n-gram									hyper-gram
	1	2	3	4	5	6	7	8	9	
AUC	76.22	82.87	83.02	86.92	86.19	86.19	85.84	84.84	84.61	87.85
Number of unique grams	25.0	211.0	495.0	732.5	901.0	1024.5	1123.0	1208.5	1283.5	239.5

$16 * 1000$ microseconds or 37 milliseconds per second. This adds to a 3.7% increase in latency.

Remember that this estimation contains brute force comparison through the database and with optimization disabled. With optimization enabled, it took less than half a second to repeat the process a million times.

B. Hyper-gram vs N-gram Representation

To compare hyper-gram representation with n-gram, we use our in-execution classifier with one simple modification: we use only the last average profile of the process (generated at the end of process' execution). This makes sure that the whole process is considered, thus putting more pressure on representation to be consistent. For example if a malicious process generates sequence similar to benign ones at the end of its execution, our in-execution classifier would not be affected by it as it would have probably classified the process as malicious before further execution. Whereas, our modified classifier would require the representation to count last sequences (similar to benign ones) to be considered as neutral rather than benign.

In Table II, we report the AUCs obtained with different window sizes of n-gram and the average AUC for our representation. The average number of unique grams are also reported. Where the average is calculated from the two different number of grams obtained from the two folds used during training.

Lower number of unique grams ensures that there lesser processing overheads during testing phase. The average number of unique hyper-grams of 239.5 is just above than number of unique 2-grams and much lower than any of $3-9$-grams whereas the accuracy is greater than n-gram of all window sizes. Remember that this AUC and number of unique grams is average of 10 different optimized mappings we achieved. We have observed greater AUC's with smaller number of unique grams in some of these optimizations.

VII. CONCLUSION

We have presented a novel variable-length system call representation scheme `Hyper-grams`, that uses our in-execution classifier for zero-day malware detection. Hyper-gram representation is more light as it represents more sequences in less number of unique hyper-grams. Moreover our in-execution classifier can classify processes during their execution with low processing overheads. Our classifier achieves more accuracy and robustness than well known classifiers. Hyper-gram representation requires malicious sequences along with benign ones rendering it unusable in case of 1-class classification. The optimization time of α, β and δ is high. We have used Genetic Algorithm with population size and number of generations equal to a 100, a total of $10,000$ mappings are created and analyzed before deciding the best one. In our experiments using MATLAB, an average time of 1 hour and 20 minutes was consumed for training of each fold. Other optimization techniques might achieve faster optimization but this analysis is out of scope of this paper. We have not utilized other features of system calls such as system call arguments. Such information in system call arguments can be further utilized for increasing accuracy and robustness.

REFERENCES

[1] 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.

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

[3] 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, 2007.

[4] K. Rieck, T. Holz, C. Willems, P. Düssel, P. Laskov, "Learning and Classification of Malware Behavior', In Detection of Intrusions and Malware and Vulnerability Assessment, pp. 108-125, 2008.

[5] 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.

[6] Q. Qian, M. Xin, "Research on Hidden Markov Model for System Call Anomaly Detection', In Proceedings of Pacific Asia Workshop on Intelligence and Security Informatics (PAISI), 2007, pp. 152-159.

[7] D. Gao, M. K. Reiter, D. Song, "Behavioral Distance Measurement Using Hidden Markov Models", In Proceedings of 9th international symposium, RAID 2006, Hamburg, 2006, vol. 4219, pp. 19-40.

[8] M.Z. Shafiq, S.A. Khayam, M. Farooq, "Embedded Malware Detection using Markov n-grams", Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), pp. 88-107, Springer, France, 2008.

[9] V. S. Sathyanarayan, P. Kohli, B. Bruhadeshwar, "Signature Generation and Detection of Malware Families", In Proceedings of the 13th Australasian conference on Information Security and Privacy, Wollongong, pp. 336 - 349.

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

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

[12] W.W. Cohen, "Fast effective rule induction", In Proceedings of 12th International Conference on Machine Learning, ICML, pp. 115-123, USA, 1995.

[13] 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.

[14] I.H. Witten, E. Frank, Data mining: Practical machine learning tools and techniques, Morgan Kaufmann, 2nd edition, USA, 2005.

[15] D. Mutz, F. Valeur, C. Kruegel, G. Vigna, Anomalous System Call Detection, ACM Transactions on Information and System Security (TISSEC), 9(1), pp. 61-93, ACM Press, 2006

[16] G. Tandon, P. Chan, Learning Rules from System Call Arguments and Sequences for Anomaly Detection, ICDM Workshop on Data Mining for Computer Security (DMSEC), pp. 20-29, IEEE Press, USA, 2003