In-Execution Malware Detection using Task Structures of Linux Processes

Farrukh Shahzad, Sohail Bhatti, Muhammad Shahzad and Muddassar Farooq
Next Generation Intelligent Networks Research Center (nexGIN RC)
National University of Computer & Emerging Sciences (FAST-NUCES) Islamabad, 44000, Pakistan.
Email:{farrukh.shahzad,muhammad.shahzad,muddassar.farooq}@nexginrc.org, rsmbhatti@yahoo.com

***Abstract*—In this paper, we present a novel framework – it uses the information in kernel structures of a process – to do run-time analysis of the behavior of an executing program. Our analysis shows that classifying a process as malicious or benign – using the information in the kernel structures of a process – is not only accurate but also has low processing overheads; as a result, this lightweight framework can be incorporated within the kernel of an operating system. To provide a proof-of-concept of our thesis, we design and implement our system as a kernel module in Linux. We perform the time series analysis of 118 parameters of Linux task structures and preprocess them to come up with a minimal features' set of 11 features. Our analysis show that these features have remarkably different values for benign and malicious processes; as a result, a number of classifiers operating on these features provide 93% detection accuracy with 0% false alarm rate within 100 milliseconds. Last but not least, we justify that it is very difficult for a crafty attacker to evade these low-level system specific features.**

I. Introduction

The run-time analysis of the behavior of an executing program is successfully used to discriminate between a benign and malicious process. To date, most of these systems utilize two types of techniques: (1) information in the sequence of system calls and their arguments – commonly known as system calls based run-time malware analysis and detection systems [9] and (2) flow graphs that model temporal sequence and dependance of system calls [16]. The run-time analysis is criticized for its two major drawbacks: (1) a crafty attacker can easily evade the system by launching mimicry attacks, [2][3], and (2) significantly high processing overheads of logging and processing run-time information makes the techniques infeasible for real operating systems.

In comparison, we follow a hypothesis in this paper for doing run-time analysis to model behavior of an executing process: *the information in the kernel structures of a process can be used to discriminate between a malicious and a benign process.* The task structure – maintained in the kernel of an operating system – contains records of every action and resource usage of a process; therefore, intuitively speaking the actions and resource usage patterns of a malicious process are expected to be different from that of a benign process. In order to overcome the above-mentioned shortcomings of run-time systems, we set a number of challenging requirements for our malware detection system: (1) high detection accuracy and low false alarm rate, (2) small processing overhead of logging run-time information, (3) efficient training and testing models that enable classification in realtime, (4) the in-execution capability to detect a malware during its execution and subsequently kill it, and (5) it is not easy for a crafty attacker to evade the system.

In order to provide a proof-of-concept of our theory, we take the example of Linux operating system as a case study. Our framework consists of four modules: (1) features logger, (2) features analyzer, (3) features preprocessor, and (4) classifier. The job of features logger is to periodically dump 118 fields of task structure after every millisecond for 15 seconds. The features analyzer utilizes tools of statistical and information theory – to rank the classification potential of all features. The features preprocessor selects a minimal subset features i.e. 11 out of 118 – that can discriminate between a benign and malicious process with high accuracy. Finally, we evaluate a number of classifiers to select the best one. During the training phase of a classifier, we randomly select 10% instances of both benign and malware processes and create a training profile. Once the classifier is trained, we present short listed 11 fields after every 1 millisecond for a decision. The results of our experiments show that after a few milliseconds of execution processes enter into a steady state; as a result, we can take decision on the basis of first 100 instances. The detection accuracy on our dataset – 30 malware and 30 benign processes – is approximately 93% with 0% false alarm rate. This shows the classification power of task structure features for malware detection. Moreover, the overhead of $50 - 70$ microseconds after every millisecond makes the system a suitable choice for embedding our framework within the kernel of Linux. Last but not least, our system detects a malware while it is executing and hence can kill it.

The rest of the paper is organized as follows. Section II provides a brief overview of the previous work related to behavior-based malware detection. Section III presents the in-execution malware detection framework. In Section IV, we present the details of experiments and discussion. In Section V, we discuss the effectiveness of our framework against evasion attempts. Finally, we conclude the paper in Section VI with an outlook to future work.

II. Related Work

The run-time analysis of the behavior of an executing program is an active area of research. The idea of using the state of a process to do intrusion detection is proposed in

[4]. The authors train a neural network for monitoring the information related to a user's activities – user activity times, user login hosts, user foreign hosts, command set, CPU usage and memory usage patterns. In another interesting study [5], task structure parameters are used for performance monitoring and system audit on x86-architecture based systems. In [6], an object monitoring framework for Linux kernel is proposed that detects the hidden malicious processes using task structure for embedded Linux systems. In [7], a secure auditing system in Linux kernel is proposed that uses loadable kernel modules that intercepts system calls' to collect information from the kernel and suggests strategies to audit and protect the kernel. In [8], proposed system protects OS kernel from kernel level rootkits that hides its running processes and threads. It generates signatures for the kernel data structures and builds profiles using their common fields. It then fuzzes these fields to determine the most important parameters required for the correct operation of an operating system.

The other type of systems utilize the system calls sequence of a program to analyze its behavior. The seminal work reported in [9] leveraged the temporal pattern of system calls to discriminate a malicious process form a benign one. Later on a number of enhanced variants of the above-mentioned technique are proposed in [10] and [11]. The authors of [12] used code-based static analysis of system calls. The improvement of existing solutions continued and researchers added more precision and accuracy. In another work, the authors leveraged spatial information in the arguments of parameters [2]. In [17], the authors presented a solution – based upon spatio-temporal features of APIs. Moreover, some authors used information related to the environment – local or global – of a call [13]. Others used system call stack and program counter information [14] [2]. All system calls based malware detection techniques have built in shortcomings of larger processing overheads and detection delays. These techniques are also vulnerable to mimicry attacks (adding fictitious system calls in the code to evade system calls based security solutions).

In [3], the authors proposed a system, korset, that automatically constructs the graph about the behavior of a program behavior. This information is used to prevent unknown code injection/buffer over flow attacks that guarantees a zero false alarm rate. They have also shown that their system is more robust to *mimicry attacks* as well; but it can also be evaded. To the best of our knowledge, the idea of investigating the information in task structures has not received any attention by security researchers. This provides us the grist for the mill to investigate this unexplored dimension and use it to build an in-execution malware detection framework.

III. Architecture of In-Execution Malware Detection Framework

The architecture of our framework is depicted in Figure 1 and it consists of four modules: (1) features logger, (2) features analyzer, (3) features preprocessor, and (4) classifier. We now provide details of each sub-module.

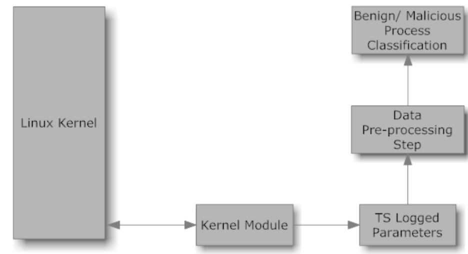


Fig. 1. System Flow Diagram of Framework

Features Logger. The job of features logger is to periodically dump an image of the fields in Linux kernel's process structure i.e. `task_struct` (we dump 118 fields). The parameters include the process state, virtual memory used, CPU time in the system, mode of a process (user or kernel), number of page tables etc. These parameters are logged using a kernel module that extracts relevant information from task structure of a process every millisecond for 15 seconds; as a result, we have a maximum of 15000 samples for each process. (Remember the processes that finish earlier will have less number of samples). We take the dump of these parameters – extracted from the task structure – for 30 benign processes e.g. `ls, ps, vi, firefox, text editor etc.` taken from Linux system directories */bin and /usr/bin* and 30 malicious processes e.g. `kaiowas variants, silvio and dataseg etc.` taken from VX Heavens [1] web-site.

Features Analyzer & Preprocessor. The features analyzer uses the tools of statistical and information theory – to rank the classification potential of all features. In order to short-list features that have high potential to discriminate between a benign and malicious process, the logged parameters are preprocessed in two phases. In the first stage, the parameters which are flags, constants, static identifiers, empty or zero valued fields – are eliminated from the dataset. In the second stage, the time-series analysis of the short-listed parameters is performed. We maintain a time-series mean of the parameters – extracted from both benign and malicious processes – over a variable size time-windows: 1ms, 2ms, 3ms, ... and so on. We then plot the time-series mean and discard those parameters that show same mean values at different time-windows for both benign and malicious processes. Finally, the preprocessor selects a minimal subset of strong features – the features with a very high classification potential – that can discriminate between a benign and malicious process with a high accuracy.

Classification. Our classification system works in two phases – training and testing. In the training phase, we provide features' set to a number of classifiers – Naive Bayes, Bayes Net, J48 and JRip. Finally, the testing is performed after every millisecond on the basis of extracted values of short-listed task structure parameters. The objective of using more classifiers is to select a classifier that can do accurate classification in realtime.

A. A Case Study of Features Selection Process for DataSet

We now show the features selection process with concrete examples from our dataset. Our features selection process consists of following two analysis steps.

Useless Features Analysis: In this analysis, we eliminate the features that offer no value to the classification process. After a thorough study, observation and analysis, we remove the fields from time series dataset that are of type i.e flags, constants, static identifiers, empty or zero valued parameters etc. We have been able to short-list 39 parameters out of 118 task structure in this initial selection process. Moreover, we also have excluded 20 more parameters that do not show a consistent and distinct behavior for benign and malicious processes. An example of such a parameter is process identifier. Although each process created in the operating system has unique process identifier, however it can't be used to discriminate between benign and malware processes.

Time-Series Mean Analysis. Now we do the time-series mean analysis of remaining 19 parameters. In computing the time-series mean, we cumulate the corresponding parameter values of different processes and divide the sum with the total number of processes. The process of computing time-series mean is shown in Equation (1):

$$a_{i,j} = \frac{1}{m} \sum_{k=1}^{m} v_{i,j,k} \tag{1}$$

where i represents the parameter, j represents the time instance, and k represents the processes' identification. $a_{i,j} \in A$, where an element of A is the time-series mean of parameter i at the time instant j. Note that i varies from 1 to n, j varies from 1 to the time instant a process runs, and k varies from 1 to m (the number of processes). In case of both benign and malicious processes $m = 30$. In the next sub-section, we show that 11 parameters are finally selected, so $n = 11$. $v_{i,j,k}$ is the value of ith parameter for k^{th} process at the time instance j. Figure 2 shows the graphs of 11 features out of the remaining 19 that have significant differences in terms of time-series mean values in both benign and malicious processes.

B. Features Preprocessing using Time Series Mean Analysis

We compute and plot the time series mean of 11 parameters of task structure. In the first phase, we have short-listed 19 parameters. Now we have concluded after analyzing their time-series mean that the time series mean of 8 parameters are the same for both benign and malicious processes; as a result, they are simply discarded. Now we discuss these parameters briefly.

TABLE I
DESCRIPTION OF THE SHORT-LISTED PARAMETERS

Parameter Name	Minor Description
task-st⟶mem-manager⟶as_users	Number of processes using current address space
task-st⟶mem-manager⟶page_table_lock	Used to manage the page table entries
task-st⟶mem-manager⟶hiwater_rss	Number of page frames – a process owns
task-st⟶mem-manager⟶shared_vm	Number of pages in shared file memory mapping of a process
task-st⟶mem-manager⟶exec_vm	Number of pages in exec. memory mapping of process
task-st⟶mem-manager⟶nr_ptes	Number of page tables owned by a process
task-st⟶utime	Execution time of a process in user mode (tick count)
task-st⟶stime	Execution time of a process in kernel mode (tick count)
task-st⟶nvcsw	Volunteer context switches of a process
task-st⟶min_flt	Minor page faults of a process
task-st⟶alloc_lock.raw_lock.slock	It locks memory manager, file system and files etc

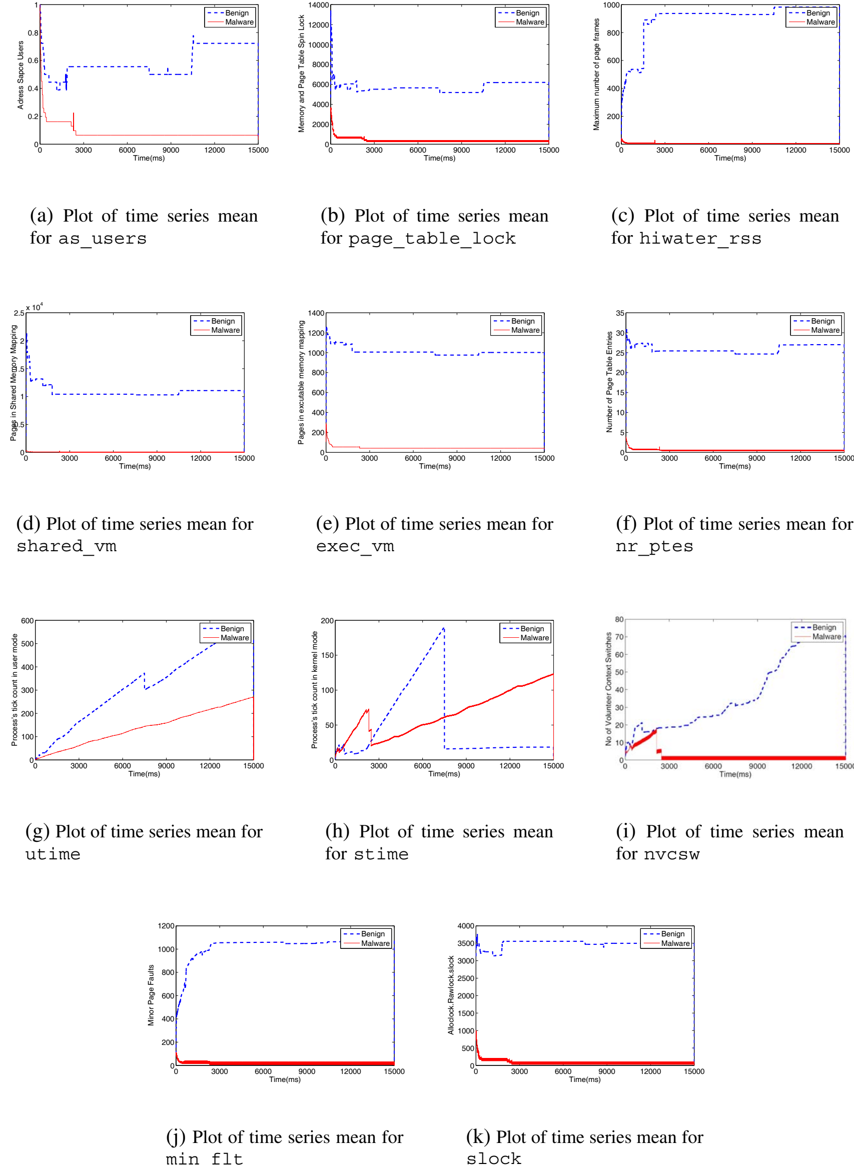
In Figures 2(a) and 2(b), the time-series mean of `as_users` and `page_table_lock` (spin lock) is plotted. The `as_users` field shows the number of processes using current address space; while manipulation and traversal of the page table entries requires the spin lock. For `as_users` parameter, both malicious and benign processes start with a similar decreasing pattern but the frequency of processes using current workspace gradually increases in the benign processes while it decreases in the malicious processes. It can be seen that initially both malicious and benign processes create threads; however, the malicious processes keep killing their threads while benign processes continue to create and kill their threads in a periodic manner. In case of the spin locks, both types of processes start with a high value but its rate decreases with time and then attains a steady state value.

Figures 2(c) and 2(d) respectively show the behavior `hiwater_rss` parameter – the maximum number of page frames ever owned by a process – and `shared_vm` (the number of pages in shared file memory mappings) both for benign and malicious processes. In both cases, malicious processes have almost zero page frames and they keep zero pages in the shared memory mappings. On the other hand, the values of these fields are significantly high in the case of benign processes – `hiwater_rss` forms a gradually increasing pattern while `shared_vm` shows a decreasing pattern.

In Figures 2(e) and 2(f), the time-series mean of `exec_vm` – the number of pages in executable memory mappings – and `nr_ptes` (the number of page tables of a process) are plotted. We can see that the usage of pages in executable memory mapping have mean values around 100 and 1050 for malicious and benign processes respectively. Similarly, the number of page table field has an average between 0 and 5 in case of malicious processes and between 25 and 30 in case of benign processes. As a result, these parameters are really suitable for a malicious process's detection.

In Figure 2(g), `utime` parameter is plotted. `utime` represents the tick count of a process that is executing in the user mode. The patterns of `utime` for benign and malicious process are almost similar but the benign processes spend more time in the user mode as compared to the malicious ones.

Figures 2(h) and 2(i) show the time-series mean of `stime` and `nvcsw` fields. `stime` stores the tick counts of a process in the kernel mode and `nvcsw` represents the number of volunteer context switches. We can see that the switching pattern of malicious processes in the kernel mode is relatively more complex as compared to the benign processes. Initially it increases exponentially, drops to a lower level after 2.5 seconds and then exhibits a linear increase till the end of the time limit of 15 seconds. During the first 2 seconds benign processes have relatively smooth switching behavior, but in next 5 seconds their switching to kernel mode increases exponentially; however, after 7.5 seconds they maintain a constant rate. Similarly, the volunteer context switches pa-



(a) Plot of time series mean for `as_users`

(b) Plot of time series mean for `page_table_lock`

(c) Plot of time series mean for `hiwater_rss`

(d) Plot of time series mean for `shared_vm`

(e) Plot of time series mean for `exec_vm`

(f) Plot of time series mean for `nr_ptes`

(g) Plot of time series mean for `utime`

(h) Plot of time series mean for `stime`

(i) Plot of time series mean for `nvcsw`

(j) Plot of time series mean for `min_flt`

(k) Plot of time series mean for `slock`

Fig. 2. Plots of the parameters in `task_struct` (Table I) used for classification

rameter exponentially increases for both benign and malicious processes for the first 2 seconds. Later the benign processes follow a linear increase but malicious process maintain a constant context switching rate.

In Figures 2(j) and 2(k), the `min_flt` and `slock` fields are plotted. `min_flt` field represents the minor page faults of a process and `slock` fields are used to lock the memory manager, files, file system, and `tty` etc. In benign processes, value for the minor page fault parameter increases exponentially for the first 2 seconds and then it stabilizes at a value between 1000 to 1200 for the rest of the execution time. However, in malicious processes its value varies between 50 and 150. Similarly, value for `slock` parameter in malicious process initially fluctuates between 3000 and 4000 and then stabilizes to 3500 in the steady state. In malicious processes `slock` parameter decreases from 1000 to 200 in the first 2 seconds and then converges to a steady state value of around 100 for the rest of the time. The List of short-listed parameters with a brief description is provided in Table I.

IV. Experiments and Results

In this section, we provide the details of experimental setup and discuss the classification results obtained using the 11 parameters of task structure identified in Section III-B.

A. Experimental Setup

We have used Wakaito Environment for Knowledge Acquisition (WEKA) [15] for evaluating the classification accuracy of different classifiers on our features' set. In WEKA, we

have used four different classifiers – Bayes Net, JRip, J48, and Naive Bayes. Bayes Net is a probabilistic classifier that uses graph theory and Naive Bayes is a simple probabilistic classifier. In comparison, JRip is an iterative rule learning classifier and J48 is a decision tree classifier. We have used *Knowledge Flow Tool of WEKA* to build a model for classification and class prediction for each classifier. We have built the training profiles of classifiers using 10% random samples of 50% benign and malicious processes. The remaining 50% benign and malicious processes are used for testing and the results of tested processes are reported in Table II.

B. Discussion of Experiments & Results

Remember that for two-class problems, such as malware detection, the classification decision of an algorithm fall into one of the four categories: (1) True Positive (TP) – a malicious executable is marked as malicious, (2) True Negative (TN) – a benign process is classified as benign, (3) False Positive (FP) – a benign process is misclassified as malicious, and (4) False Negative (FN) – a malicious process is misclassified as benign. For malware processes, we report TP and FN (number of individual time series instances of a process) and for benign processes we report TN and FP (in terms number of instances). Our another challenge is to classify a malicious process while it is executing. We have observed that most of the malicious process finish in relatively small time, while benign programs run for comparatively longer duration. Moreover, we observe that the process state becomes stable in a few millisecond after the start of its execution. Therefore, we set an upper limit of 100 milliseconds (100 instances of a process) to take a final decision. A classifier gives a decision once it has analyzed 100 samples in a time-window of 100 milliseconds – it declares a process as malicious if its more than 50% samples are classified as malicious and vice versa. It is important to note that if processes finish in less than 100 ms (e.g. 7 ms), we classify such a process with the help of these 7 instances using the overall training profile. The results of our experiments show that our framework can detect malicious processes successfully – right at the start of their execution; however, in few exceptional cases it misclassifies the processes. Remember, we do not kill a process until it is declared malicious on 100 consecutive samples (at maximum). It is ensured that the testing instances of processes are never used in the training. It is important to note that Table II reports the TP and FN results (all sample of a process) of 15 malicious processes and the TN and FP (all samples) results of 15 benign processes only. The other benign and malicious processes are used in the training profile; therefore, they are excluded from the table. The overall detection accuracy (DA) and false alarm rate (FAR) (Table II) are calculated on the basis of majority vote within a window for a process. On the other hand, the detailed results in Table II are computed by taking a decision on each individual instance of a process after 1 ms.

We have tabulated the classification results of four classifiers of malicious processes (M1 to M15) in Table II. We can see that all classifiers have misclassified the malware processes M13 and M15 (on majority decision). These processes executed only for 98 and 7 millisecond respectively. In comparison, only a few instances of other malicious processes are misclassified (4 instances of M14) – during their execution. From M1 to M12, not even a single instance of these malicious processes is misclassified. Finally, we can conclude that an overall 87% of malicious processes are correctly classified by all classifiers.

Similarly, the results for benign processes (B1 to B15) are provided in Table II. We can see that J48 misclassifies two benign processes – B1 and B2 – during the first two milliseconds and B15 for initial 12 millisecond only. In comparison, JRip misclassifies first instance of both the processes (B1 and B2) and initial 17 and 25 instances of B12 and B15 respectively. So the classification accuracy of both the classifiers using available (less than 100 or 100 milliseconds) time is 100%. The accuracy of other two classifiers is significantly small compared to J48 and JRip. The classification results show that benign files are never misclassified.

To conclude, the overall detection accuracy (DA) of all benign (30) and malicious processes (30) is 93% with 0% false alarm rate (FAR) (only two malicious processes are misclassified in the 100 ms dataset). Since our framework will become part of a kernel, it is very important that we should select a classifier with a zero FP; otherwise, we will accidently kill a benign process. If we closely analyze the results in Table II, we can safely conclude that J48 and JRip provide best classification accuracy for both benign and malicious processes. To conclude, our short-listed features – when used in conjunction with Jrip and J48 – provide high classification accuracy.

C. Timing Analysis

In a run-time system, it is very important to analyze training and the testing times of the proposed scheme to make the theme of "in-execution" relevant. We need just 34.43 microseconds to extract features from the task structure. Afterwards, we add the training time of two best classifiers which is approximately 18 and 30 milliseconds for J48 and Jrip respectively. Remember that the training is done only once; therefore, these timings are acceptable. The testing times for J48 and Jrip are 45 and 100 microseconds respectively. It means J48 will just take 79 microseconds after every 1 millisecond. This is an acceptable overhead (less than 7%). Therefore, we feel justified that our framework can be embedded in the kernel of an operating system.

V. Evasion

Finally, we study the robustness of our features' set from another perspective: how easily a crafty attacker can evade these features. In order to systematically undertake this study, we took features' set of 15 malicious processes and manually replaced them with those of benign processes. We have observed that once we forge 4 features simultaneously, only one additional misclassification of malware is observed. Once we forge 6 features, 4 malicious processes are misclassified as

TABLE II

DETECTION RESULTS AND ACCURACY FOR BENIGN AND MALICIOUS PROCESSES

Malicious Processes								
	J48		**JRip**		**Naive Bayes**		**Bayes Net**	
File	TP	FN	TP	FN	TP	FN	TP	FN
M1	70	0	70	0	70	0	70	0
M2	40	0	40	0	40	0	40	0
M3	2000	0	2000	0	2000	0	2000	0
M4	267	0	267	0	276	0	267	0
M5	148	0	148	0	148	0	148	0
M6	146	0	146	0	146	0	146	0
M7	380	0	380	0	380	0	380	0
M8	2000	0	2000	0	2000	0	2000	0
M9	67	0	67	0	67	0	67	0
M10	84	0	84	0	84	0	84	0
M11	199	0	199	0	199	0	199	0
M12	24	0	24	0	24	0	24	0
M13	4	94	3	95	8	90	22	76
M14	200	0	200	0	200	0	196	4
M15	1	6	0	7	2	5	3	4

Benign Processes								
	J48		**JRip**		**Naïve Bayes**		**Bayes Net**	
File	TN	FP	TN	FP	TN	FP	TN	FP
B1	311	1	311	1	311	1	311	1
B2	1999	1	1999	1	1999	1	1999	1
B3	47	0	47	0	46	1	46	1
B4	282	0	282	0	276	6	280	2
B5	2000	0	2000	0	2000	0	2000	0
B6	1150	0	1150	0	1149	1	1149	1
B7	611	0	611	0	522	89	611	0
B8	66	0	66	0	66	0	66	0
B9	217	0	217	0	217	0	217	0
B10	184	0	184	0	184	82	184	53
B11	15000	0	15000	0	15000	0	15000	0
B12	519	0	519	17	519	56	519	43
B13	5	0	5	0	5	0	5	0
B14	343	0	343	0	343	0	343	0
B15	7872	12	7872	25	7872	121	7872	151

Overall Accuracy of Framework									
	J48		**JRip**		**Naive Bayes**		**Bayes Net**		
	DA	**FAR**	**DA**	**FAR**	**DA**	**FAR**	**DA**	**FAR**	
	93%	**0%**	**93%**	**0%**	80%	26%	86%	13%	

benign. Finally, the accuracy become unacceptable once we forge 8 features. This shows that it is possible to evade our system by forging features.

The above-mentioned evasion is made possible because we knew the exact values of features for benign processes. Remember that most of our parameters depend on a particular configuration of a system – cache, RAM, secondary storage, processor, paging mechanism, stack and heap managers – and therefore they must be estimated for each host. The values can change from one host to another. As a result, a crafty attacker has to first estimate these values for a particular system on which it wants to execute malware; as a result, a malware needs hooks to sample fields of task structures for benign processes. In Linux, this feature is only allowed to super user processes; as a result, a malicious process cannot easily evade our system.

VI. CONCLUSION

In this paper, we have proposed a novel *in-execution* detection scheme that detects a malicious process during its execution on Linux platform. After a thorough analysis of 118 parameters in Linux task structure, we have short-listed 11 parameters that have high classification potential. Our results show that our framework achieves 93% DA with 0% FAR. Moreover, our framework, once it is trained – takes few microseconds to take a decision after every millisecond. As a result, our framework can be embedded in the kernel of Linux. Moreover, our features are system specific and we must estimate them for benign and malware processes in each system. The features can only be extracted by a super user; as a result, it becomes difficult to evade our detection system.

ACKNOWLEDGMENTS

The work presented in this paper is supported by the National ICT R&D Fund, Ministry of Information Technology, Government of Pakistan. The information, data, comments, and views detailed herein may not necessarily reflect the endorsements of views of the National ICT R&D Fund.

REFERENCES

[1] VX Heavens, A Malware Collection Web-site, http://hvx.netlux.org

[2] J. Giffin, S. Jha and B. Miller, Efficient Context-sensitive Intrusion Detection, Network and Distributed Systems Security Symposium, 2004.

[3] O. Ben-Cohen, A. Wool, Korset: Automated, Zero False-Alarm Intrusion Detection for Linux, Linux Symposium, pp. 31, 2008.

[4] L. Vokorokos, A. Balaz, M.Chovanec, Intrusion Detection System using Self Organizing Maps, Acta Electrotechnica et Informatica No. 1, pp. 1–5, 2006.

[5] S. Bandyopadhyay, A Study on Performance Monitoring Counters in x86-Architecture, Indian Statistical Institute.

[6] L. Sun, T. Katori, A Lightweight Kernel Object Monitoring Infrastructure for Embedded Linux Systems, IEEE Conference on Embedded and Real-Time Computing Systems and Applications, pp. 55–60, 2008.

[7] K. Zhao, Q. Li, J. Kang, D. Jiang, L. Hu, Design and Implementation of Secure Auditing System in Linux Kernel, IEEE Workshop on Anti-counterfeiting, Security, Identification, pp. 232–236, 2007.

[8] B. Dolan-Gavitt , A. Srivastava , P. Traynor , J. Giffin, Robust Signatures for Kernel Data Structures, ACM Conference on Computer and Communications Security, pp. 566–577, 2009.

[9] S. Forrest, S. A. Hofmeyr, A. Somayaji, T. A. Longstaff, A Sense of Self for Unix Processes, IEEE Symposium on Security and Privacy, pp. 120–128, 1996.

[10] W. Xun, Y. Wei, A. Champion, F. Xinwen, X. Dong, Detecting Worms via Mining Ddynamic Program Execution, International Conference on Security and Privacy in Communication Networks and the Workshops, pp. 412–421, 2007.

[11] J. B. D. Cabrera, L. Lewis, R. K. Mehra, Detection and Classification of Intrusions and Faults using Sequences of System Calls, ACM SIGMOD Record, vol.30, no.4, pp. 25–34, 2001.

[12] D. Wagner, D. Dean, Intrusion Detection via Static analysis, IEEE Symposium on Security and Privacy, pp. 156–168, 2001.

[13] Jonathon T. Giffin, D. Dagon, S. Jha, W. Lee, B. P. Miller, Environment-sensitive Intrusion Detection, International Conference on Recent Advances in Intrusion Detection, pp.185–206, 2006.

[14] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, B. P. Miller, Formalizing Sensitivity in Static Analysis for Intrusion Detection, IEEE Symposium on Security and Privacy, pp. 194–208, 2004.

[15] I. H. Witten, E. Frank, Data mining: Practical Machine Learning Tools and Techniques, Morgan Kaufmann, Second Edition, 2005.

[16] P. Beaucamps, J. Y. Marion, Optimized Control Flow Graph Construction for Malware Detection, International Workshop on the Theory of Computer Viruses, 2008.

[17] F. Ahmed, H. Hameed, M. Z. Shafiq, M. Farooq, Using Spatio-Temporal Information in API Calls with Machine Learning Algorithms for Malware Detection, ACM workshop on Security and Artificial Intelligence, pp. 55–62, 2009.