

In-Execution Dynamic Malware Analysis and Detection by Mining Information in Process Control Blocks of Linux OS

Farrukh Shahzad, M. Shahzad, Muddassar Farooq

Next Generation Intelligent Networks Research Center (nexGIN RC)

National University of Computer and Emerging Sciences (NUCES)

Islamabad, Pakistan

{farrukh.shahzad, muhammad.shahzad, muddassar.farooq}@nexginrc.org

Abstract

Run-time behavior of processes – running on an end-host – is being actively used to dynamically detect malware. Most of these detection schemes build model of run-time behavior of a process on the basis of its data flow and/or sequence of system calls. These novel techniques have shown promising results but an efficient and effective technique must meet the following performance metrics: (1) high detection accuracy, (2) low false alarm rate, (3) small detection time, and (4) the technique should be resilient to run-time evasion attempts.

To meet these challenges, a novel concept of *genetic footprint* is proposed, by mining the information in the kernel Process Control Blocks (PCB) of a process, that can be used to detect malicious processes at run time. The *genetic footprint* consists of selected parameters – maintained inside the PCB of a kernel for each running process – that define the semantics and behavior of an executing process. A systematic forensic study of the execution traces of benign and malware processes is performed to identify discriminatory parameters of a PCB (`task_struct` is PCB in case of Linux OS). As a result, 16 out of 118 task structure parameters are short listed using the time series analysis. A statistical analysis is done to corroborate the features of the *genetic footprint* and to select suitable machine learning classifiers to detect malware.

The scheme has been evaluated on a dataset that consists of 105 benign

processes and 114 recently collected malware processes for Linux. The results of experiments show that the presented scheme achieves a detection accuracy of 96% with 0% false alarm rate in less than 100 milliseconds of the start of a malicious activity. Last but not least, the presented technique utilizes partial knowledge that is available at a given time while the process is still executing; as a result, the kernel of OS can devise mitigation strategies. It is also shown that the presented technique is robust to well known run-time evasion attempts.

1. Introduction

Computer malware is becoming an increasingly significant threat to the computer systems and networks world-wide. In recent years, security experts have observed a massive increase in the number and sophistication of new malware. According to a recent threat report by Symantec, in 2008 alone, 5491 new software vulnerabilities were reported, 1.6 million new malware signatures were created, 245 million new attacks were reported, and the financial losses caused by malware soared to more than 1 trillion dollars [14]. It is, however, interesting to note that though 50% of new malware are simply repacked versions of known malware [50], even then they successfully evade existing commercial-off-the-shelf antivirus software (COTS AV) because they utilize static signature-based techniques [49] that are not robust to code obfuscation and polymorphism.

To overcome shortcomings of existing COTS AV, malware researchers are focusing their attention to non-signature behavior based intelligent detection techniques that can detect new malware on zero day (at the time of its launch). Such techniques can be broadly classified into two categories: (1) static, and (2) dynamic. The main objective of static behavior based techniques is to analyze the information and contents of a given file – contained in the header and payload – to build intelligent malware models and then use different classification engines to detect malware [44][45][47]. An inherent shortcoming of static techniques is to identify “robust” features that are difficult to evade by novel packing schemes. (Remember a packer only changes the contents of an

executable without modifying its real semantics [12].)

In comparison, dynamic detection techniques try to model run-time behavior of a process; as a result, they cannot be evaded by mere code obfuscation or polymorphism. Two types of well known dynamic schemes have been proposed: (1) system call sequencing, and (2) flow graphs. The system call sequences based techniques (presented in [6] [7][58] [61][13] and [29]) build the behavioral model on the basis of the sequence of invoked system calls and the information passed in their arguments. These techniques have four well known shortcomings: (1) processing overhead of logging system calls, (2) high false alarm rate, (3) ability to make a decision only after analyzing the complete trace of the execution sequence of a process (by that time a malicious process has already done the damage), and (4) they can be easily evaded by simply reordering the system calls or adding irrelevant system calls to invalidate the sequence that represents malware.

The basic concept behind flow graph techniques is to construct taint graph of a running process by analyzing its data flow model [61][23]. The graph provides valuable insight about the activities of a running process. Most of the proposed schemes require a virtual machine with a shadow memory to keep track of taint information. A related system is NICKLE [40], but it has been demonstrated that return oriented rootkits [20] can evade it. These schemes – like their system call counterparts – have high false alarm rate, high processing overheads and high detection time (sometimes of the order of minutes). It is rightly concluded in a recent paper [24] that most of the above-mentioned dynamic techniques are novel and promising but still a leap jump is required in their detection accuracy, detection time, and processing overheads to establish their effectiveness to replace or complement traditional anti-virus software at an end host.

To meet these challenges, a novel technique based on dynamic analysis is proposed that uses *genetic footprint* of a running process. The thesis of using *genetic footprint* is: *the state diagram of malicious processes should be different from that of benign processes because of difference in their activities.* The *genetic*

footprint is defined as a set of 16 out of 118 parameters, which are maintained by the kernel of an operating system, in the PCB of a process¹, to keep track of the state of an executable process². To be more specific, malicious processes that try to steal or corrupt data and information – like passwords, keystrokes, files etc. – try to covertly capture the information that is not intended for them [24] without a user’s consent. In comparison, the benign processes follow access control policies to acquire the legitimate information. Similarly, the state of a malicious process that tries to replicate itself on storage media or network interfaces will be different from the benign process that runs only once, performs its given tasks and exits.

As mentioned before, the *genetic footprint* is maintained in the “task structure” by the kernel of every operating system. Linux operating system is selected because of its open source advantage that provides fine grain control over kernel structures; as a result, systematic studies/evaluations are conducted on them. In case of Linux, the information about the current state of a process is maintained in the `task_struct` structure. The aim is to analyze the temporal behavior of the *genetic footprint*; therefore, a system call is developed that dumps 118 fields of `task_struct` structure for 15 seconds with a resolution of one millisecond (ms). This time interval is reconfigurable and at the moment it is based on the observation that most of the malicious processes in the selected dataset either finish their intended activity within this time duration or at least start showing a distinct behavior to characterize them as benign or malicious.

The proposed scheme is evaluated on a dataset that consists of 105 benign processes and 114 recently collected malware processes of Linux. The results of the experiments prove that proposed scheme achieves a detection accuracy of 96% with 0% false alarm rate. Moreover, it is able to detect a malicious process

¹Throughout this paper, we will use the terms PCB, task structure, and `task_struct` interchangeably.

²The feasibility of using parameters of “task structure”, maintained in the kernel of Linux, is investigated by the authors in a preliminary pilot study reported in [46].

in less than 100 ms from “the start of a malicious activity”. It is emphasized the use of “the start of a malicious activity” instead of “the start of a malicious process” because the use of *genetic footprint* of a process enables the detection of even those malicious processes which mostly behave like benign processes and perform malicious activity for a very short duration somewhere in the middle (such as backdoors). Most of existing run-time behavior analysis techniques fail to detect such malicious processes. Also to the best of our knowledge, no existing runtime analysis technique is able to detect a malicious activity in less than 100 ms. A robustness analysis of the proposed technique is also presented in circumstances when a crafty attacker splices the *genetic footprint* of a benign process with that of a malicious process at different positions. The results of experiments show that the proposed scheme is robust to such evasion attempts. It is also demonstrated that it is difficult to evade proposed technique if it is used in conjunction with recently proposed techniques – discussed in Section 6.

To conclude, the major contributions of the work presented in this paper are: (1) architecture of the proposed dynamic malware detection framework based upon the concept of *genetic footprint* – a set of selected parameters (maintained inside the task structure of a kernel of an operating system for each running process) – that defines the semantics and behavior of an executing process, and (2) a testing and validation framework to prove that our framework has the capability to not only detect a running malware within 100 ms of the start of a malicious activity but it also provides 96% detection accuracy with a 0% false alarm rate.

The rest of the paper is organized as follows. Section 2 builds the motivation for using information in the task structure by doing a time series forensic study of selected malware and benign processes. In Section 3, the characteristic of dataset are discussed that is used for experiments. Section 4 explains in detail the functionality of three modules of the proposed scheme: (1) features logging, (2) features selection, and (3) classification. The design of the scheme is based on systematic investigation and evaluations. Section 5 describes the results of experiments and in doing so intriguing insights are provided about the behavior

of the proposed scheme. Section 5.7 provides the performance comparison of the proposed scheme with multiple system calls sequence based solutions. Section 6 provides a detailed description of the robustness of proposed scheme to evasion. Section 7 provides a brief overview of the previous work related to behavior-based malware detection and finally the paper concludes with an outlook to our future work.

2. Forensic Analysis of Benign and Malicious Processes

The intuition is systematically built in order to answer a tricky question: *why genetic footprint of a malware process should be different from that of a benign one?* The forensic investigation is conducted to analyze the execution behavior of benign and malware processes. Both the benign and malware processes are selected from different categories. The benign processes belong to text editors, image utilities and system utilities categories; while malware processes belong to trojan horses, net-worms, and virus categories. A brief description of each type of process is provided that will help in correlating their behavior with their *genetic footprint*.

2.1. Benign Processes.

hwclock (CLK). This utility accesses the hardware clock of the system. It displays the current time and is responsible for synchronizing the system and hardware clocks periodically.

ED. It is a command line text editor that can create, display, and modify text files. When it is invoked with a filename, it copies the text of original file into its buffer in a temporary file and all subsequent changes are made to this file and are written into the original file once explicitly saved. This utility is invoked with a text file but the user does not edit the file for 15 seconds.

ppmtoterm (PPMT). This is a command-line image utility that converts a .ppm image to ANSI ISO 6429 ASCII image and displays it on the terminal of Linux OS. It performs color approximations, measures the minimum cartesian

distance between RGB vectors and finally generates palettes. Unlike the text editor *ED*, it does all above-mentioned steps in an automated fashion.

2.2. Malware Processes.

Linux.Backdoor.Kaiten.a (TKTN). It is a trojan horse [3] that opens a backdoor on the victim computer and uses an IRC client to connect to IRC servers on the port TCP 6667. It connects to a predetermined IRC channel and listens for commands, which are then used to perform malicious activities on the victim host. It performs the following malicious activities: (1) launches DDOS attacks using SYN and UDP packets, (2) downloads and executes remote scripts and files, (3) changes a client's nickname, (4) changes servers, (5) spoofs an IP address, (6) kills running processes, (7) generates floods, and (8) changes system files: `/etc/rc.d/rc.local` and `/etc/rc.conf`.

Net-Worm.Linux.Sorso-b4264 (WSRO). It is a Linux worm that infects the samba server [5]. It establishes a connection and sends exploit code to the shell on the server. Once the server runs the exploit code, it in turn downloads malicious executables that try to infect more samba servers, replace the `http` daemon with the hijacked one. Moreover, it hides its created process, steals IP addresses and sends them to remote hosts to perform various actions enlisted in [5].

Virus.Linux.Satyr.a (VSTR). It is a non-memory resident parasitic Linux virus [4] that has multiple aliases. Its main task is to look for other executable files in the system and infect them. The virus infects files in the directories and it shifts down the contents of a victims' file and writes itself in the file header. To release control to the host file, the virus "disinfects" it to a temporary file and then executes it. The virus does not manifest itself in anyway but its body contains a copyright text string.

Virus.Linux.ELF_X23 (VX23). Once `ELF_X23` [1] executes, it infects other ELF files in the current directory. It checks if the file has a `.X23` extension and whether it is an executable in the user group. If it does not have this extension and if it is an executable in the user group, it concatenates the `.X23` extension

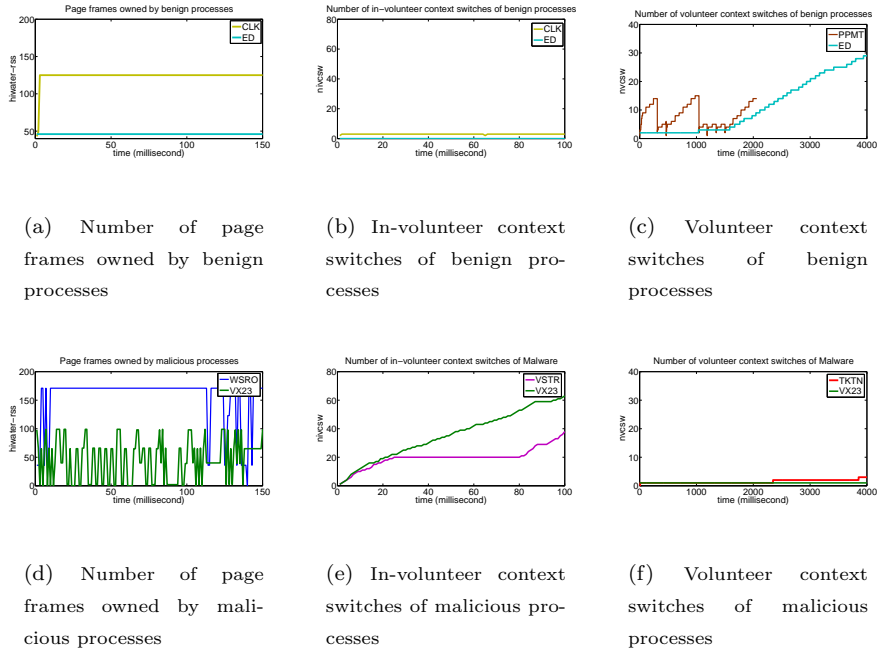


Figure 1: Forensic analysis of benign and malicious processes

to the host file name. This serves as a copy of the original host file. After this, it copies its virus code to the original host file. Finally, it changes the attribute of the file to readable, writable, and executable for all user groups. Once the virus is finished, it transfers control back to the copy of the original file with the .X23 extension.

2.3. Forensic Analysis.

The forensic analysis is presented on three parameters of Linux `task_struct` structure to build an insight into the execution behavior of benign and malicious processes. The aim is to get an understanding that how the execution behavior of a process is correlated with the `task_struct` parameters. The sample parameters are: (1) the number of page frames owned by a process, (2) the number of in-volunteer context switches of a process, and (3) the number of volunteer context switches of a process. A time series analysis of these processes is done.

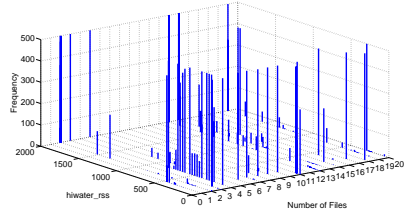
In order to show distinct patterns, only two processes are shown in a figure. (It is experienced that once the parameters of four processes are plotted in a single figure, the legibility of the figure is severely impaired.)

Page frames owned by a process. It is interesting to note in Figures 1(a) and 1(d) that benign programs allocate memory in the beginning of their execution and then their memory usage does not change in small intervals of milliseconds. A malware, on the other hand, wants to cause the damage as quickly as possible; therefore, its memory usage shows oscillations. (Note that in the selected dataset 3 malware samples finish in less than 10 ms; while 11 finish in less than 30 ms.) Moreover, benign processes – CLK and ED – show a uniform memory usage pattern in small intervals while VX23 malware shows an oscillatory behavior. The moment WSRO starts the malicious activity, its memory usage also changes in small intervals of time.

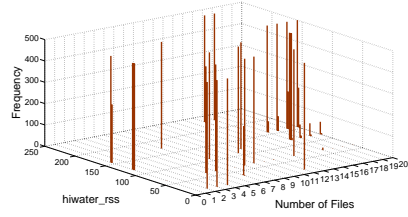
Volunteer and In-volunteer context switching. A volunteer context switch means that a processes relinquishes control of the processor before expiry of its allocated time quantum. On the other hand, if a scheduler intervenes to stop a running process at the expiry of its allocated time quantum, it is called an in-volunteer context switch.

It is again interesting to see in Figures 1(b) and 1(c) that benign processes mostly execute in a non-greedy fashion; consequently, they have an increasing number of volunteer context switches and near zero in-volunteer context switches. On the other hand, it can be seen in Figures 1(e) and 1(f) that the malicious processes mostly operate in a greedy mode; as a result, they are preempted by the scheduler resulting in an increasing number of in-volunteer context switches and near zero volunteer context switches. The above analysis provides a preliminary insight that Linux `task_struct` can be used to distinguish between benign and malicious processes.

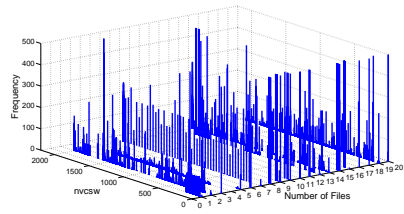
To generalize the above mentioned assertion, the histograms of three task structure parameters is plotted of 20 benign and malware processes each in Figure 2. (Note that the processes are randomly selected from the chosen dataset.) From Figures 2(a) and 2(b), it is concluded that `hiwater_rss` plots of benign



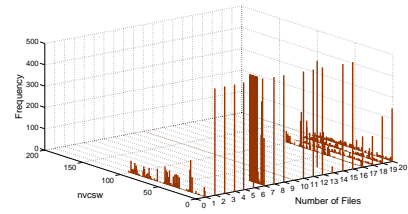
(a) Magnified histogram of benign processes hiwater_rss parameter



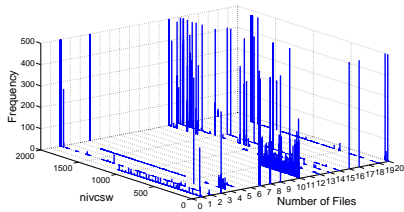
(b) Magnified histogram of malware processes hiwater_rss parameter



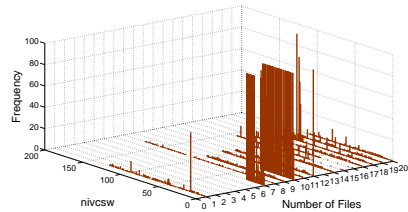
(c) Magnified histogram of benign processes nvcsw parameter



(d) Magnified histogram of malware processes nvcsw parameter



(e) Magnified histogram of benign processes nivcsw parameter



(f) Magnified histogram of malware processes nivcsw parameter

Figure 2: Histogram comparison of benign and malicious processes

processes have large spreads, implying continuous memory allocation. In contrast, malicious processes show sparsely located spikes for `hiwater_rss` depicting that the allocated memory remains constant during most of the execution

trace. Furthermore, `hiwater_rss` spans over a shorter domain (i.e. 0-250) for malicious processes as compared to the benign ones (i.e. 0-2000). (Frequency and `hiwater_rss` axis are magnified for didactic purpose.)

Similarly, Figures 2(c) to 2(f) show voluntary and involuntary context switching behavior of benign and malicious processes. It is visible in both figures that benign processes are more likely to switch their context voluntarily as compared to malicious processes. The same behavior can be inferred from Figures 2(e) and 2(f), where it is visible that benign processes are rarely preempted by the OS scheduler. Since malicious processes have higher predilection to hold CPU; therefore, they are more frequently preempted. Also, it can be observed that malicious processes mostly finish quickly (in less than 200 ms) and the majority of them in less than 50 context switches. In comparison, the majority of benign processes execute till 2000 context switches (or even more).

It can be concluded that the forensic and histogram analysis both ascertain the ground truth: *some parameters of `task_struct` of Linux have the potential to characterize a process as a benign or malicious.*

3. Dataset

In section 2, the fact is established that Linux `task_struct` parameters can be used to discriminate between benign and malicious processes. Now a larger dataset of benign and malicious processes is taken for a detailed analysis of more `task_struct` parameters and their classification potential.

In this section, the dataset³ and the methodology is presented to log parameters of task structure after every 1 ms. 114 malware and 105 benign samples are used for this purpose. The statistics of all these samples are tabulated in Table 1.

In Table 1, it can be seen that benign samples are divided into four major categories: games (Gms), image utilities (ImgUtl), text editing utilities & con-

³Linux `task_struct` mined datasets of both benign and malware processes collected for this study are available on the website <http://www.nexginrc.org>

verters (TEd), and miscellaneous Linux shell commands (SCmd). These benign files are collected from Linux operating system’s directories `/bin`, `/sbin`, and `/usr/bin`. The diversity in the benign dataset is ensured by selecting files of different sizes – ranging between 10 KiloBytes (KB) to 2 MegaByte (MB) – and categories from above-mentioned directories.

Malware samples have been collected from “Offensive Computing” [35] and “VX Heavens” [53] malware collections. These 114 malware samples can be divided into 8 different malware categories – Exploits (Exp), Flooders (Fldrs), Net-Worms (NWrm), Rootkits + Hacktools (Rkt), Backdoors (BkDr), Trojans (Trjn), and Virus (Vrs). Malware processes execute in an automated manner and usually don’t take input from the user while benign processes consist of a variety of execution patterns; automated (system processes), semi-automated (take users’ input only once e.g. image converters) and manually operative (interactive and non-interactive text editors, firefox etc). Both the malware and benign processes have been dumped in almost all execution states. In order to have a balanced dataset, it is ensured that the percentage of benign and malicious files in a specific category (based upon size) is approximately the same. For an interested reader, the complete list of 114 malware and 105 benign processes is provided in Table 6.

Table 1: Benign and malware file size distribution

Sizes	Benign Files						Malware Files								
	SCmd	ImgUtl	TEd	Gms	All	%	Exp	Fldrs	NWrm	Rkt	BkDr	Trjn	Vrs	All	%
<10K	6	19	1	0	26	25	1	0	0	1	19	0	8	29	26
10-50k	35	9	3	4	51	48.5	5	7	5	3	9	14	22	65	57
50-100K	7	0	3	0	10	9.5	0	0	2	0	0	2	1	5	4
100-500K	10	0	0	5	15	14.2	0	0	1	2	0	1	6	10	9
500K-2048K	2	0	1	0	3	2.8	0	0	0	1	0	0	4	5	4
Total	60	28	8	9	105	100	6	7	8	7	28	17	41	114	100

4. Architecture of Dynamic Malware Detection Framework

In this section, the architecture of proposed scheme is presented that consists of three modules: (1) features logger, (2) features analyzer, and (3) classifier. In the following subsections a methodology is presented to select the features

that define the *genetic footprint*. Moreover, a systematic analysis is performed on features' set to select a suitable machine learning classifier. The architecture of the proposed framework is shown in Figure 3. Each module is discussed individually.

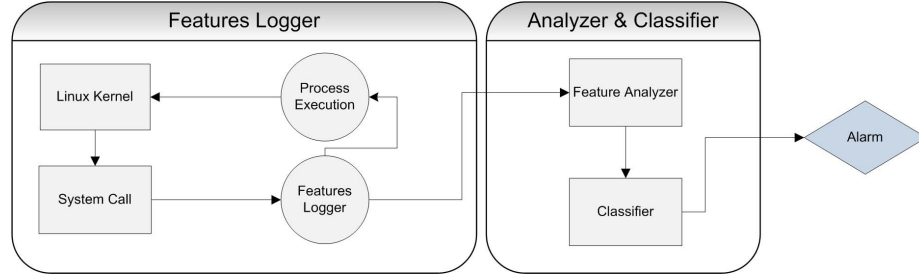


Figure 3: Block diagram of the dynamic malware detection framework

4.1. Features Logger.

The job of features logger is to periodically dump 118 fields of `task_struct` structure of Linux. Some of the most notable parameters are number of page frames, volunteer and in-volunteer context switches, number of page table locks, number of page faults, virtual memory used, CPU time in system, mode of a process, number of page tables etc. The parameters are logged using customized kernel system call framework that invokes customized system calls to extract relevant information from `task_struct` structure of a desired process after every millisecond for 15 seconds. As system calls are endemic part of kernel, so these can access kernel structures directly. Linux kernel stores the state of processes in a doubly circular linked list of `task_struct` structure and maintains a global variable of this structure named – `current` – that provides access to the task structure of the current process. Next or previous processes can be accessed using next or previous members of `current`. In this way, the state of any process can be accessed by moving forward or backward in this circular linked list. A customized system call tracks the process under consideration by its name in the circular list of Linux kernel. As soon as it finds the process, it logs the fields of `task_struct` in a separate data file with the same name as of

the running process. Through this system call, a maximum of 15000 samples for each process have been dumped. The processes that finish earlier, of course have the number of samples corresponding to their total execution time. This methodology is used to dump the task structure parameters for 105 benign and 114 malicious processes.

4.2. Features Analyzer.

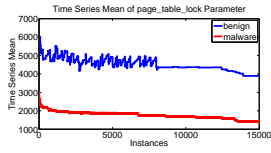
In order to short-list features with high classification potential, feature pre-processing has been done in two steps: (1) identifying and filtering the fields that are not relevant to the behavior of a process, and (2) the time series analysis of remaining fields is performed to identify features with a high classification potential. The fields that do not depend on the behavior of a process are constant fields, process identifiers, bit combinations (flags) etc; therefore, it is important to remove them from the features' set so that they do not misguide a classifier during the training process. In case of Linux, 23 parameters are various kinds of offsets, 9 are flags, and 50 are either constants or zeros that show no discriminating behavior. Once the first step is finished, only 36 parameters are left on which the second step is performed.

In the second step, the time series mean of each parameter for all benign and malicious files is calculated using the following equation:

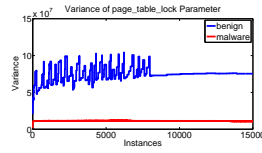
$$\mu_{i,k} = \frac{1}{N} \sum_{j=1}^N x_{i,j,k} \quad i = 1 \rightarrow t_x, k = 1 \rightarrow 36, \quad (1)$$

where i represents the time instance, t_x is the termination time of a process (in milliseconds) or 15000 whichever is smaller, j represents the processes' identification number, $N = 105$ for benign and $N = 114$ for malicious processes. $\mu_{i,k} \in T_k$, where T_k are the time series means of parameters $k = 1$ to 36 (remember in this step only 36 parameters are processed).

The time series means of these 36 fields of benign and malicious processes help in short listing the fields that are different in benign and malicious processes. Moreover, it is also equally important that the selected parameter should have low variance in all samples of benign and malware processes; otherwise, it



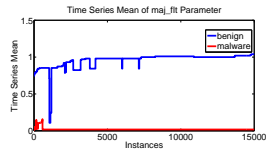
(a) Accepted Parameter `page_table_lock` showing difference in time series mean for benign and malicious processes



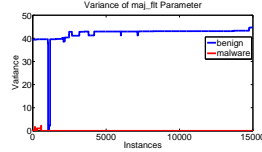
(b) Accepted Parameter `page_table_lock` showing low variance



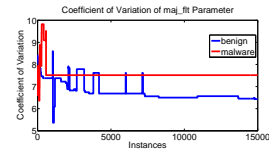
(c) Accepted Parameter `page_table_lock` showing low coefficient of variation



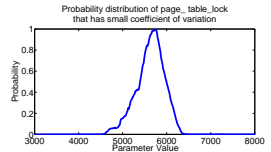
(d) Rejected Parameter `maj_fit` showing difference in time series mean for benign and malicious processes



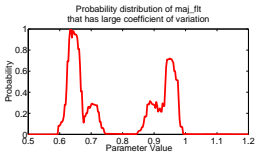
(e) Rejected Parameter `maj_fit` showing high variance



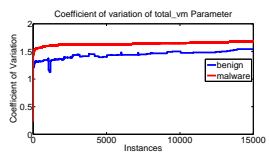
(f) Rejected Parameter `maj_fit` showing high coefficient of variation



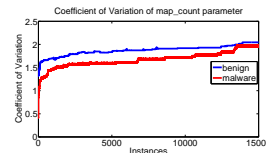
(g) Probability distribution of `page_table_lock` that has small coefficient of variation



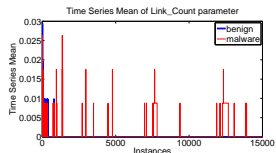
(h) Probability distribution of `maj_fit` that has large coefficient of variation



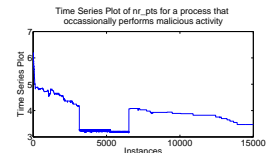
(i) Accepted Parameter `total_vm` showing low coefficient of variation for benign and malicious processes



(j) Accepted Parameter `map_count` showing low coefficient of variation for benign and malicious processes



(k) Rejected Parameter `Link_Count` showing absence of difference in time series mean for benign and malicious processes



(l) Time Series Mean of parameter `exec_vm` for a file that behaves like benign at start and like malicious further down the execution time

Figure 4: Plots of various statistical aspects of short listed parameters

would lead to high false positives and false negatives respectively. In order to get an understanding about the spread of a feature, its variation is computed using the following equation:

$$\sigma_{i,k}^2 = \frac{1}{N} \sum_{j=1}^N (x_{i,j,k} - \mu_{i,k})^2 \quad i = 1 \rightarrow t_x, k = 1 \rightarrow 36 \quad (2)$$

where i, j, N, t_x and k represent the same quantities as in Eq. (1). $\sigma_{i,k}^2 \in V_k$, where V_k are the time series variances of parameter k . In order to factor out the impact of the value of a feature, a coefficient of variance is defined. The coefficient of variance of a feature can be computed using the following equation:

$$cv_{i,k} = \frac{\sigma_{i,k}}{\mu_{i,k}} \quad i = 1 \rightarrow t_x, k = 1 \rightarrow 36 \quad (3)$$

where i, j, μ, σ, t_x and k represent the same quantities as in Eqs. (1) and (2). $cv_{i,k}$ represents the coefficient of variance of parameter k at time instant i . If the coefficient of variation approaches 1, it means that the standard deviation of a parameter is approximately the same as its mean. If a feature has a large coefficient of variance, intuitively speaking it is not a good idea to use it because it will confuse the learning process that ultimately would result in reducing the detection accuracy.

Before going into the details of the parameter selection procedure, a formal definition of the *genetic footprint* is provided. Let Y_k be a time series of parameter k that terminates at time instant n , then Y_k is given by:

$$Y_k = \{y | y = (t_{i,k}, x_{i,k}) \wedge i = [0, n]\} \quad (4)$$

where $(t_{i,k}, x_{i,k})$ is the ordered pair which represents the value $x_{i,k}$ of the time series Y_k at time instant t_i . The termination time of the process is defined by the symbol t_x . In this scenario, $n = t_x$ if $t_x < 15000$ and $n = 15000$ if $t_x \geq 15000$. In terms of its parameters, the task structure \mathbb{T} is defined as:

$$\mathbb{T} = \{Y_k | y \in Y_k \text{ is a parameter of Linux } \mathbf{task_struct}\} \quad (5)$$

This equation means that the \mathbb{T} is basically a set of time series of the parameters of the `task_struct` structure. The *genetic footprint* \mathbb{G} of a process, is

essentially a subset of the set \mathbb{T} i.e. $G \subset \mathbb{T}$ whose elements are time series of those parameters that have certain required properties. Formally, \mathbb{G} is defined in the following equation:

$$\mathbb{G} = \{Y_k | \forall Y_k \in \mathbb{T} \Rightarrow |\mu_b(Y_k) - \mu_m(Y_k)| > \epsilon \wedge 0 < cv_{b,m}(Y_k) < 3 \wedge |Z| \geq |Y_k| \times 0.9545\} \quad (6)$$

where $\mu_b(Y_k)$ is the time series mean of the time series Y_k of benign processes and $\mu_m(Y_k)$ shows the time series mean of the time series Y_k for malicious process. ϵ models the smallest difference that the time series mean of benign processes should have from the time series mean of malicious processes. ϵ is given by the following equation:

$$\epsilon = \frac{1}{|Y_k|} \sum_{i=1}^{|Y_k|} \left(\mu_b(Y_k) - \frac{\sum_{i=1}^{|Y_k|} \mu_b(Y_k)}{|Y_k|} \right)^{1/2} \quad (7)$$

In equation 6, $cv_{b,m}(Y_k)$ represents the coefficient of variation of the elements of time series Y_k for both benign and malicious processes. $|Y_k|$ is the value of total number of elements in the time series Y_k . Z is given by:

$$Z = \left\{ z | \left(\mu_b(Y_k) - 3\sigma_b(Y_k) \right) < z < \left(\mu_b(Y_k) + 3\sigma_b(Y_k) \right) \right\} \quad (8)$$

Having developed a formal definition of *genetic footprint*, let us now demonstrate the usefulness of the above-mentioned two step feature selection methodology. A first look at Figures 4(a) and 4(d) might provide the temptation to use both of them in the features' set because their time series mean is significantly different in benign and malware samples. However, once their variance is plotted in Figures 4(b) and 4(e), and their corresponding coefficient of covariance in Figures 4(c) and 4(f) respectively, two important conclusions can be drawn: (1) the parameter `page_table_locks` should be made part of the *genetic footprint* because of its low *cv* both in benign and malicious processes, and (2) the parameter `maj_ft` should not be considered for inclusion in *genetic footprint* because of its relatively high *cv* value in benign samples. To further confirm these findings about the effectiveness of selected and rejected parameters, their distributions are plotted in Figures 4(g) and 4(h). It is clear that `page_table_locks` with a low

cv has a gaussian distribution; while maj_flt with a high cv does not have a well known distribution. It is known that in case of a gaussian distribution, 95.45% of the values lie between the interval $\mu \pm 3\sigma$. On the basis of this analysis, the following rule set is evolved to make a feature part of the *genetic footprint*:

1. Its time series mean is significantly different for both benign and malicious processes,
2. coefficient of variation is less than 3,
3. 95.45% of its values are within the interval $\mu \pm 3\sigma$.

Once these rules are iteratively applied to the above-mentioned 36 features, 20 more features are discarded; as a result, *genetic footprint* now consists of 16 features. Just to efficiently utilize the space, the plots of three other parameters are shown: (1) $total_vm$ is accepted because of low cv in Figure 4(i), (2) $link_count$ in Figure 4(k) is rejected because of its similar pattern in benign and malicious processes, and (3) map_count is accepted in Figure 4(j) because of its low cv . Finally, the *genetic footprint* is created and the final features are included in Table 2. Now we focus our attention towards identifying a classifier that meets three requirements: (1) low training and testing times, (2) high detection rate, and (3) low false alarm rate.

4.3. Classification.

Recently, the role of a given dataset in selecting an appropriate classifier is being investigated [51] [52]. A well known measure – class noise [8] – is defined to quantify the challenging nature of a dataset. In [62], the authors have shown that the classification accuracy is more dependent on class noise instead of attribute noise. The authors classify a given dataset by using a number of well known trained classifiers and the intersection of correctly classified instances constitutes non-noisy dataset. Consequently, the intersection of all misclassified instances by all classifiers is defined as the class noise. We follow the same approach: use all instances of benign and malware processes and classify them with neural networks (RBF-Network), Support Vector Machines (SVM) with

Table 2: Description of the fields constituting *genetic footprint*

Parameter Name	Description
task→fpu_counter	Usage counter of floating point units
task→active_mm→map_count	No. of memory regions of a process
task→active_mm→page_table_lock	Needed to traverse & manipulate the page table entries
task→active_mm→hiwater_rss	Max no. of page frames ever owned by a process
task→active_mm→hiwater_vm	Max no. of pages appeared in memory region of process
task→active_mm→total_vm	Size of process's address space in terms of no. of pages
task→active_mm→shared_vm	No. of pages in shared file memory mappings of process
task→active_mm→exec_vm	No. of pages in executable memory mappings of process
task→active_mm→nr_ptes	No. of page tables of a process
task→utime	Tick counts of a process that is executing in user mode
task→stime	Tick counts of a process in the kernel mode
task→nvcs	Number of volunteer context switches
task→nivcs	Stores the no. of in-volunteer context switches
task→minflt	Contains the minor page faults
task→alloc_lock.raw_lock.slock	Used to lock memory manager, files and file system etc
task→fs→count	fs_struct's usage count to indicate the restrictions

Table 3: Class noise of dataset used in experiments

	RBF Network	SVM POLY-K	SVM PUK-K	SVM RBF-K	J48	J-Rip	Class Noise
Benign Instances (%)	31.87	18.89	34.65	44.70	5.71	3.29	3.29
Malware Instances (%)	14.87	10.28	25.65	27.08	6.91	15.48	6.91
Overall Class Noise	10.2(%)						

multiple kernels (Polynomial kernel, universal Pearson VII function based kernel (Puk) and Radial Basis Function based kernel (RBF)), decision tree (J48) and a propositional rule learner (J-Rip) in Wakaito Environment for Knowledge Acquisition (WEKA) [59]. The results of misclassified instances are reported in Table 3 for all classifiers. It is obvious from this empirical study that the intersection of misclassified instances of benign and malicious processes is 3.29% and 6.91% respectively. (This leads to an overall class noise of 10.2 %). In comparison, SVM and neural networks are unable to cope with the complexity of our time series dataset. In [60], it is shown that J48 is resilient to class noise because it avoids over fitting during learning and also prunes a decision tree for an optimum performance. Similarly, J-Rip also applies pruning during the formulation of rules to achieve better accuracy. Therefore, we have short listed

J-48 and J-Rip.

Now we cross validate our preliminary decision of selecting J-48 and J-Rip by analyzing the classification potential of *genetic footprint*. (Higher detection accuracy means that *genetic footprint* has higher classification potential). In order to do this, renowned statistical measures (commonly used in data mining and knowledge extraction), Information Gain and Information Gain Ratio, are applied. Information gain IG for a parameter p is given by the following equation:

$$IG(Ts, g) = H(Ts) - \sum_{v \in \text{values}(g)} \frac{|\{s \in Ts | V(s, g) = v\}|}{|Ts|} \times H(\{s \in Ts | V(s, g) = v\}) \quad (9)$$

Ts is the training samples that is used. $V(s, g)$ defines the value of sample $s \in Ts$ for parameter $g \in G$ where G represents the *genetic footprint* that consists of 16 selected features. $|Ts|$ is the number of elements in the set Ts . H defines the entropy of Ts and is calculated using the following equation:

$$H(X) = - \sum_{i=1}^n p(x_i) \log_b p(x_i), \quad (10)$$

where b is the base of the logarithm and Euler's number e is used as base of the logarithm. X is a discrete random variable, p denotes the probability mass function of X , and n is the total number of values that X can have.

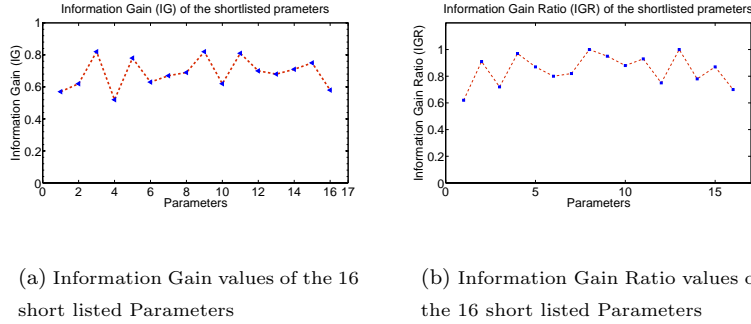


Figure 5: Information Gain and Information Gain Ratio

The information gain of *genetic footprint* is plotted in Figure 5(a). It is seen

that information gain of most of the features is relatively high. It is a well known fact in data mining research: *decision tree based algorithms and propositional rule learners achieve high classification accuracy for classification of instances where parameters have reasonably high information gain* [60]. However, their accuracy can significantly degrade, even if the selected features have high information gain because they might have large number of distinct values. (In this case, the features of *genetic footprint* can also possibly have large number of distinct values). In this scenario, an effective decision tree is not created because of relatively high bias towards features that can have large number of distinct values. In order to remove this bias, another measure – information gain ratio (*IGR*) (defined in the following) – is used.

$$IGR(Ts, g) = \frac{IG(Ts, g)}{-\sum_s \frac{|\{s \in Ts\}|}{|Ts|} * \log_b\left(\frac{|\{s \in Ts\}|}{|Ts|}\right)} \quad (11)$$

If information gain ratio is approximately 1, decision trees and propositional rule learners give good classification accuracy. *IGR* of features in the *genetic footprint* is plotted in Figure 5(b). It can be seen that *IGR* of most of the features is near to 1, and hence it concurs with our preliminary decision to use decision tree algorithms and propositional rule learning algorithms.

5. Experiments and Results

In this section, the accuracy of two short listed classifiers is evaluated using the features in *genetic footprint*. Specifically, following issues are discussed chronologically: (1) the overall accuracy of a classifier using *genetic footprint*, (2) the impact of increasing detection time on the classification accuracy, (3) detection of a malicious process if it mostly behaves like a benign process and performs malicious activity for a short duration later during its execution, (4) the false alarm rate of the proposed system, (5) detection of the malicious processes before their exit even if they finish within 30 ms of their launch, (6) the processing overheads of logging, training and testing of the proposed system, and (7) a comparison of the accuracy of the proposed scheme with other

sequence calls based solutions. Later in Section 6, the “robustness” of the proposed system is discussed to evasion attempts if a malicious process tries to imitate the behavior of a benign process.

In order to evaluate the effectiveness of *genetic footprint* in detecting malware using J48 and J-Rip, a stratified 10–fold cross validation strategy is used. The dataset is divided into 10 folds – a fold on the average consists of 11 malware and 10 – 11 benign samples – and classifiers are trained on 9 folds and are tested on the remaining 1 fold. WEKA [59] is used for this 10 fold analysis to remove any bias in evaluation due to implementation of a classifier.

It is important to note that most of the processes finish before 15 seconds, hence it is important to use the concept of step size to ensure that learning is not biased towards the processes running longer. Therefore, the step size is defined as:

$$step_size = \frac{x}{n} \tag{12}$$

where *step_size* is the number of samples after which a sample of a process is selected for training, *x* is total number of samples of a process and *n* is number of selected samples. For our experiments, we have taken *n*=30. Step size helps us to select training samples at almost equally spaced intervals.

In a typical two-class problem, such as malicious process detection, the classification decision of an algorithm may fall into one of the following four categories: (1) true positive (TP), classification of a malicious process as malicious, (2) true negative (TN), classification of a benign process as benign, (3) false positive (FP), classification of a benign process as malicious, and (4) false negative (FN), classification of a malicious process as benign.

The detection accuracy of proposed system is reported using three separate metrics: (1) detection rate (DR), (2) false alarm rate (FAR), and (3) detection accuracy (DA). These metrics are defined in the following:

$$DR = \frac{TP}{TP + FN} \tag{13}$$

$$FAR = \frac{FP}{FP + TN} \tag{14}$$

$$DA = \frac{TP + TN}{TP + TN + FP + FN} \quad (15)$$

We now discuss the issues raised in the beginning of this section.

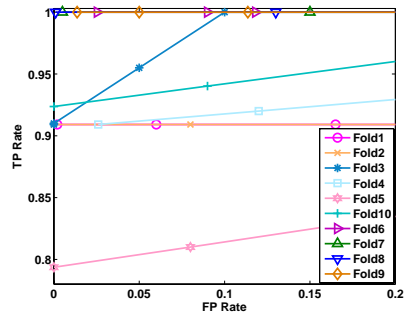
5.1. Overall classification accuracy using genetic footprint.

The accuracy of the two classifiers is reported – using *genetic footprint* – in Table 4⁴. The average results and the results for testing each fold are reported. In Table 4, the results are tabulated for window sizes of 10, 30, and 100 respectively. The “window size” defines the number of instances considered from each testing sample before making a decision. In case of 10, a classifier just takes first 10 instances of each feature in the *genetic footprint* and makes a decision. Remember, a process is declared as malicious or benign on the basis of the majority vote within a given window. In a window of 10, if six classifications are malware, the process is declared as malware. In case of equal votes for malware and benign, the system defers the decision to the next window. It can be seen that J-Rip – in case of a window size of 100 – DR, FAR and DA are 93.7%, 0% and 96.65% respectively. (For J48 these parameters are 1-4% inferior). To the best of our knowledge, no existing dynamic analysis system has achieved such detection and false alarm rates. In order to get a better insight, we plot ROC curves of different classifiers in Figure 6 for a window size of 100. (Note that a fold number in Figure 6 corresponds to the same fold number in Table 4.) The detection rate of J-Rip of 10 folds varies from 0.79 to 1 for a false alarm rate of 0. In comparison, on some folds, J48 has a false alarm rate of 0.1 to 0.2.

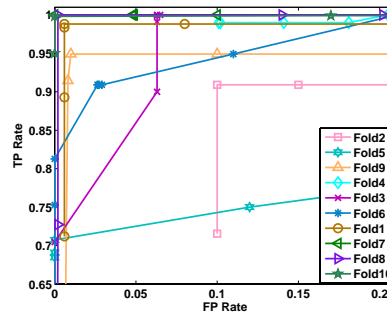
In order to do a comparative study about the effectiveness of other classifiers, we plot ROC of commonly used classifiers – RBF neural networks [16] and Naive Bayes [32] – for malware detection in Figures 6(c) and 6(d) respectively. It

⁴Note that DA and FAR reported in Table 4 are calculated on the basis of classification assigned to a process by a majority vote within a window. In comparison, the results of Table 3 are computed by taking an independent decision on each instance of a process after every millisecond.

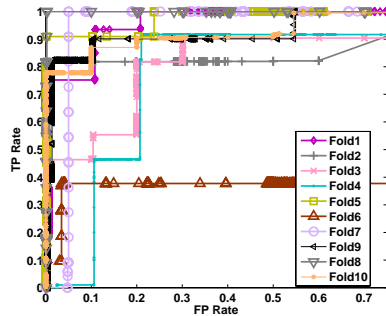
is obvious from the figure that both of them are unable to provide accurate detection with low false alarm on all folds on our *genetic footprint* dataset.



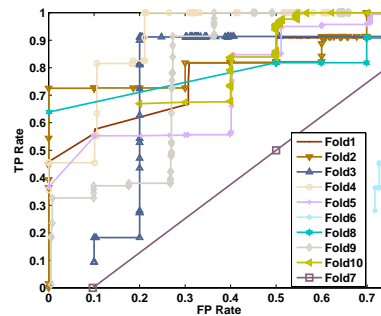
(a) ROC plot for J-Rip classifier



(b) ROC plot for J48 decision tree



(c) ROC plot for RBF neural network



(d) ROC plot for Naive Bayes

Figure 6: Magnified ROC Plots for different classifiers

5.2. The impact of increasing detection time on the classification accuracy.

It is evident in Table 4 that as the detection time increases from 10 to 30, FAR is significantly reduced and DR also improves – resulting in an overall improvement of 1-2% in detection accuracy. Moreover, if the detection time is allowed to increase beyond 30 to 100, a significant improvement in the FAR is observed.

Table 4: Accuracy results of J48 and J-Rip on each of the 10 folds

Classifier	Time (ms)	Fold1	Fold2	Fold3	Fold4	Fold5	Fold6	Fold7	Fold8	Fold9	Fold10	Avg		
J48	10	TP	10	10	11	11	08	10	11	11	11	11	10.40	
		FP	00	01	01	02	00	01	01	00	01	00	0.70	
		TN	10	09	09	08	10	10	10	09	10	10	10	09.50
		FN	01	01	00	00	03	01	01	00	00	00	00	0.60
		DR (%)	91.0	91.0	100	100	72.7	91.0	100.0	100.0	100.0	100	100	94.6
		FAR (%)	0.00	10.0	10.0	20.0	00.0	9.09	10.0	10.0	0.00	9.09	0.00	06.80
	30	DA (%)	95.2	90.5	95.2	90.5	85.7	91.0	95.2	100	95.5	100	93.88	
		TP	11	10	11	11	08	10	11	11	11	11	11	10.50
		FP	00	01	00	02	00	01	01	01	00	00	00	0.50
		TN	10	09	10	08	10	10	10	09	10	11	10	09.70
		FN	00	01	00	00	03	01	01	00	00	11	00	01.60
		DR (%)	100	91.0	100	100.0	72.7	91.0	100	100	100	100	100	95.47
	100	FAR (%)	0.00	10.0	0.00	20.0	0.00	9.09	10.0	10.0	0.00	0.00	0.00	04.90
		DA (%)	100	90.5	100	90.5	85.7	91.0	95.2	100	100	100	95.29	
		TP	11	10	11	11	08	10	11	11	11	11	11	10.50
FP		00	01	01	02	00	00	00	00	00	00	00	0.40	
TN		10	09	10	08	10	11	10	10	10	11	10	09.90	
FN		00	01	00	00	03	01	01	00	00	00	00	0.50	
J-Rip	10	DR (%)	100	91.0	100.0	100.0	72.7	91.0	100	100	100	100	95.47	
		FAR (%)	0.00	10.0	10.00	20.0	0.00	9.09	10.0	10.0	0.00	0.00	04.0	
		DA (%)	100	90.5	100	90.5	85.7	95.5	100	100	100	100	95.74	
		TP	10	10	10	10	09	11	11	11	11	11	11	10.30
		FP	00	00	00	01	00	01	01	00	00	01	00	0.30
		TN	10	10	10	09	10	10	10	10	10	10	10	9.90
	30	FN	01	01	01	01	02	00	00	00	00	01	0.70	
		DR (%)	91.0	91.0	91.0	100	82.0	100	100	100	100	91.0	94.6	
		FAR (%)	0.00	0.00	0.00	10.00	0.00	9.09	10.0	0.00	9.09	0.00	2.81	
		DA (%)	95.2	95.2	95.2	90.5	90.5	95.5	100	100	95.5	95.2	95.28	
		TP	10	10	10	10	09	11	11	11	11	11	11	10.30
		FP	00	00	00	01	00	01	01	00	00	01	00	0.30
	100	TN	10	10	10	09	10	10	10	10	10	10	9.90	
		FN	01	01	01	01	02	00	00	00	00	01	0.70	
		DR (%)	91.0	91.0	91.0	91.0	82.0	100	100	100	100	91.0	93.70	
FAR (%)		0.00	0.00	0.00	10.0	0.00	9.09	10.0	0.00	9.09	0.00	2.81		
DA (%)		95.2	95.2	95.2	90.5	90.5	95.5	100	100	95.5	95.2	95.28		
TP		10	10	10	10	09	11	11	11	11	11	11	10.30	
100	FP	00	00	00	00	00	00	00	00	00	00	0.00		
	TN	10	10	10	10	10	11	10	10	11	10	10.20		
	FN	01	01	01	01	02	00	00	00	00	01	0.70		
	DR (%)	91.0	91.0	91.0	91.0	82.0	100	100	100	100	91.0	93.70		
	FAR (%)	0.00	0.00	0.00	10.0	0.00	9.09	10.0	0.00	9.09	0.00	0.00		
	DA (%)	95.2	95.2	95.2	90.5	90.5	95.5	100	100	95.5	95.2	96.65		

This behavior of proposed system can be easily explained if the basic motivation of a malware writer is understood: do the intended damage as quickly as possible. In the selected dataset about 15% malware finish in less than 100ms; as a result, most of them start malicious activity just after their launch; as a result, they can be detected within 30 ms.

5.3. Detection of backdoors-type processes.

Let’s now consider the typical behavior of backdoors – they might start their first malicious activity after 1000 ms. Recall that the proposed technique do not just perform a one time check at the start of a process; rather, it keeps on sliding the “window” to continuously invoke the classifier. As a result, system will detect an anomaly within 30 ms of its launch (assuming a window size of 30 ms). This phenomenon can be seen in Figure 4(1) that plots parameter `exec_vm` of a Linux backdoor Excedoor. It is clear that the process behaves like a benign one till 3148 ms and then it starts its malicious activity. The proposed system has detected it as a malware. To conclude, proposed system can detect a malicious activity because of its ability to do continuous real-time monitoring.

5.4. False alarm rate of proposed system.

It is depicted in Table 4 that for a window size of 100 ms, the FAR of J48 and J-Rip is 4% and 0% respectively – 2-6% reduction if a window size of 10 ms is chosen. This shows that 10 ms is relatively a small window size and within this time interval the true behavior of a process can’t be learnt. A low false alarm is very important from the usability perspective; otherwise, the users will be annoyed if their legitimate applications are frequently stopped.

5.5. Detection of tiny malicious processes.

It is already mentioned that 17 (15%) of malware finish in less than 100 ms, 11 malware finish in less than 30 ms and 3 even in less than 10 ms. Now it becomes a real challenge to detect them while they are still executing. A separate set of experiments are conducted to investigate the facts: how many of

17 malware that finish in less than 100 ms are correctly classified. The proposed technique is able to successfully classify all 14 malware processes that execute for more than 10 ms. In comparison, 1 out of 3 malware that finish before 10 ms are misclassified. (Remember it is a tight constrain on the system to give a decision before termination of a malware). As a result, DR for this challenging dataset is 93.7% and this substantiates the claim that it can accurately classify even those malware samples that terminate very quickly.

5.6. Processing overheads of proposed scheme.

In an online run-time system, it is very important to measure the processing overhead of each module of the proposed system. The processing overhead of the system is a sum of feature logging and testing times. Training time is not critical because it happens only once at the beginning and then after every one hour. The feature logging time is 40 microseconds for each instance (it includes the context switching time of 6 microseconds). J48 and J-Rip take 18 ms and 30 ms during the training phase respectively. Moreover, J48 and J-Rip have 45 and 100 microseconds testing time respectively. If the features' logging time is added to the testing time, then J48 and J-Rip take 85 and 140 microseconds per instance. The results are intriguing enough to make serious efforts to embed the detection module inside the Linux kernel for online analysis and detection.

5.7. A comparison of the accuracy of the proposed scheme with other sequence calls based solutions.

Table 5: Comparison with different system call sequences based techniques

Markov n-grams	J48		JRip		Bayesnet		IMAD	
	DA	FAR	DA	FAR	DA	FAR	DA	FAR
5-grams	75.85	3.90	75.35	2.28	74.41	9.89	73.41	5.89
6-grams	78.30	4.79	77.40	2.95	75.40	14.55	76.01	3.2
7-grams	79.87	5.99	78.78	3.29	75.33	17.45	76.0	1.09
8-grams	81.36	6.30	80.45	4.48	76.44	17.62	79.1	0.0
9-grams	81.65	6.75	80.66	4.63	76.70	17.65	80.41	1.65
10-grams	83.74	6.05	82.44	4.35	76.71	18.86	81.57	0
Average	80.13	5.63	79.18	3.66	75.83	16.00	77.75	1.97
Hyper-grams								
DA				FAR				
86.34				0				

The focus of this study is to compare the detection accuracy of our framework with recently proposed sequence calls based malware detection techniques. Most of these techniques use n-gram (typically a sequence of 5 to 10 system calls) based representation for dynamic malware detection [39], [18] [17], *IMAD* [31], and *Hyper-grams* [30]. The techniques selected for the comparison are: n-grams (ranging from 5 to 10), *IMAD* and *Hyper-grams*. J48, J-Rip and Baysiannet are used for classification of n-gram based sequence representation. These techniques classify a process by matching its system call sequences to that of benign or malware call sequences.

IMAD is a realtime and efficient in-execution malware detection scheme. It uses n-gram representation and optimizes the learning process (using a genetic algorithm) for system call sequences that are present in both benign and malicious processes. It tunes different parameters to improve the accuracy of classification.

Hyper-grams technique uses variable length n-grams (called Hyper-grams) instead of using fixed length n-grams. It is a generalized scheme in which n-gram of the trace of a process is visualized in a k-dimensional hyperspace by following the sequence of its system calls. Here, k represents the number of unique system call sequences followed by a process. The path taken by a process in hyperspace is used to model its behavior by matching it with the paths of benign and malicious processes.

In order to evaluate these techniques, a log of system calls of all benign and malicious processes is collected by executing them on Linux OS. The malware detection accuracy (DA) and false alarm rate (FAR) of these schemes are reported in Table 5. One can see that n-gram with J48 achieves on the average DA and FAR of 80.13% and 5.63% respectively. (For J-Rip DA is 1% inferior but FAR is 2% superior). The maximum accuracy (86%) with a 0% FAR is achieved by *Hyper-grams* technique. In comparison, our technique achieves more than 96% DA with a 0% FAR.

6. Evasion

The evasion of this system is also studied from two different perspectives: (1) the robustness of proposed scheme to evasion attempts if a malicious process tries to imitate the benign behavior for various parameters, and (2) access restrictions to `task_struct` parameters so that a process is unable to access or modify their values directly. The answer to the second question is critical because a malware has to first estimate the benign *genetic footprint* before imitating it. (Note all features of *genetic footprint* are system and machine dependent and hence must be learned for each system).

6.1. Robustness of proposed scheme to evasion attempts.

In this section, it is analyzed that how robust the proposed scheme is to evasion attempts if a malicious process tries to imitate the benign behavior for various parameters. In order to systematically undertake the study of robustness of the set of 16 features of *genetic footprint*, different features in the *genetic footprint* are replaced of a malware with that of corresponding features in the *genetic footprint* of a benign file. This is the logical way of imitating a benign process as malicious. The effect of this “forgery” on the accuracy of proposed system is analyzed. (All results are reported for a window size of 100). It is observed that once 4 features are forged, only 3% more malicious processes are misclassified. If 6 features are forged, accuracy further deteriorates by 2% and a total of 11% malware processes are misclassified as benign. Finally, the accuracy becomes totally unacceptable once 8 features are forged. The accuracy deteriorates by 13% (becomes 83%), which is still very competitive compared with existing state-of-the-art run-time systems.

Here, It should be emphasized that the above-mentioned forgery is done because the values of benign and malware genetic footprints are known. Remember that most of these features depend on a particular configuration of a system like cache, RAM, secondary storage, processor, paging mechanism, stack and heap managers etc and therefore, they must be estimated for each host. These 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 imitate benign process. This demands that malware be equipped with hooks to log different fields of *genetic footprint* (`task_struct`) for benign processes. This brings us to the second issue related to access restrictions and it is discussed in the following.

6.2. Access restrictions to `task_struct` parameters.

The most dangerous threat to the security of a kernel and its structures is posed by kernel rootkits. These rootkits can hide themselves by subverting “reference monitor” of an operating system [11]. As a result, they tamper with OS functionalities to launch various types of well known attacks: OS backdoors, stealing private information, escalating privileges of malicious processes, and disabling defense mechanisms. Such rootkits can surely get an access to the fields of `task_struct`, and then enable a running malware processes to learn *genetic footprint* of benign processes and then it can directly modify its *genetic footprint* to evade the system.

But the above-mentioned problem is well known in the OS security community for years now. Recently, researchers have proposed a number of novel and effective methodologies that can stop rootkits and other malicious process that try to access or modify the kernel structures. Some of these schemes are: kernel module signing [21], $W\oplus X$ [2], NICKLE [40], and SecVisor [42]. Their authors have shown that these schemes effectively and efficiently protect kernel structure from illegitimate access and modifications in all possible scenarios. Hund et al. in a recent paper [20] have introduced the concept of return oriented rootkit, which circumvents all above-mentioned schemes of protection of kernel structures. Luckily, Wang et al. have proposed a solution in [57] that again blocks the access of these return oriented rootkits to the kernel structures.

To conclude, a number of schemes exist that can stop the malicious codes to access kernel structures. As a consequence, these schemes make it not-so-easy for malware writers to learn *genetic footprint* of a benign process on a given system. As a result, it becomes a challenge for a “crafty attacker” to forge the

features and evade the proposed scheme. Nevertheless, this fact is acknowledged that rootkits are (by any means) not a solved problem and in future they can pose a potential threat to this proposed scheme.

7. Related Work

Today, most of the commercially available antivirus programs identify a malware on the basis of strings or instruction sequences that are typical to that particular malware [49]. These strings or instruction sequences define the signature of the malware file. As this technique is based upon syntactic appearance of a malware, so it is easily evaded by code obfuscation and polymorphism [12][49]. To overcome this short coming of signature based detection schemes, researchers proposed some higher order properties that can capture the intrinsic characteristics of a program and are thus difficult to disguise. One of the most commonly used such property is the n-grams character distribution of a file [27][28][43]. This property is very useful in identifying embedded malware in benign files. Another technique to detect polymorphic variants of a malware is control flow graph [9][25]. More sophisticated static analysis approaches utilize code templates that capture the malicious functionality of certain malware families. For this purpose symbolic execution [26], model checking [22], or compiler verification [13] and semantic aware [38] techniques are applied to recognize arbitrary code fragments that implement a specific function. These techniques learn a functionality independent of the specific machine instructions that express it.

Although a number of sophisticated static analysis techniques have been developed that show promising results but they also face some significant shortcomings. For example, as the malware programs rely heavily on run-time packing and self-modification of code, the instruction present in the binary on a disk are different than those executed at runtime. Although generic unpackers [41] can be helpful in obtaining the actual instructions, binary analysis of obfuscated code is still very difficult [33]. Furthermore, many advanced static analysis approaches are extremely slow (at times of the order of minutes [13]), and are thus

unsuitable for detection in real-world deployment scenarios.

Dynamic analysis techniques intend to detect malicious processes by analyzing the execution of a program or the effects that this program has on the operating system. The former generally utilizes the system calls sequence of a program to analyze its behavior. The seminal work reported in [17] leveraged the temporal pattern of system calls to discriminate a malicious process from a benign one. Later, a number of enhanced variants of the above-mentioned technique have been proposed in [55], [10] and [34]. Another technique is the code-based static analysis of system calls proposed in [54]. In [19], the authors have used spatial information in the arguments of parameters to identify malicious processes. Yet another technique uses system calls stack and program counter information [15] [19]. An example of the technique that uses the effects of a running program on the operating system is Strider GhostBuster [56]. It has the ability to detect certain kinds of rootkits that hide themselves from the user by filtering the results of system calls. It does so by comparing the view of the system provided by a possibly compromised OS to the view that is gathered when accessing the file system directly. A novel worm detection scheme by utilizing neural networks is proposed in [48]. The authors have used performance counters of an operating system to train the neural network to detect the unknown worms in realtime. Another dynamic malware detection technique is based on the analysis of disk access patterns [37] as the malicious processes usually access disk in a manner that can be distinguished from a benign program. The biggest advantage of this approach is that it can be incorporated into the disk controller, and is thus difficult to bypass; however, it can detect a malware only if it modifies a large numbers of files stored on the disk. In a recent work [36], the authors build clusters of transaction audit data streams on hosts to detect any malicious activity.

Such approaches show promising results but their large processing overheads and detection time (sometimes in the order of even minutes) make them infeasible for real world deployment. Moreover, these techniques are also evadable. A malware can easily change its sequence of system calls or add irrelevant calls

and these schemes fail in such situations. Similarly, return oriented rootkits can evade the systems like Strider GhostBuster. In comparison, our technique uses a novel concept of *genetic footprint* that enables it to detect a malware in less than 100 ms. Moreover, it provides more than 96% accuracy with 0% FAR.

8. Conclusion and Future Work

In this paper, a novel concept of *genetic footprint* of a process is used to detect malicious processes at run time. We have formally defined the *genetic footprint* and provided a comprehensive analysis of the selection process of the parameters that constitute it. As a result, 16 task structure parameters have been selected from 118 parameters after a thorough statistical analysis. A suitable classifier is selected on the basis of the complexity of dataset and information gain analysis of *genetic footprint*. The idea of using *genetic footprint* for detection of malware is validated on a dataset of 105 benign processes and 114 recently collected malware processes for Linux. The results of experiments demonstrate that the proposed system achieves a detection accuracy of 96% with 0% false alarm rate. Moreover, it detects a malicious process in less than 100 milliseconds of the start of a malicious activity. To the best of our knowledge, this is the shortest detection time achieved to date by any dynamic detection scheme. The presented technique is robust to run-time evasion attempts and has small processing overheads. In future, it has been planned to embed the scheme in the kernel of Linux and evaluate its effectiveness for other operating systems as well.

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] Elf_x23 virus description, trendmicro - an online malware descriptions database, in: <http://threatinfo.trendmicro.com> (last-viewed-on November 5, 2010).
- [2] Grsecurity, complete documentation for the pax project, in: <http://pax.grsecurity.net/docs/>.
- [3] Kaitan backdoor description, symantec - an antivirus solution provider, in: <http://www.symantec.com> (last-viewed-on November 5, 2010).
- [4] Satyr virus description, virus-list - a malware description database, in: <http://www.viruslist.com> (last-viewed-on November 5, 2010).
- [5] Sorso-b4264 net-worm description, symantec - an antivirus solution provider, in: <http://www.symantec.com> (last-viewed-on November 5, 2010).
- [6] F. Ahmed, H. Hameed, M. Z. Shafiq, M. Farooq, Using Spatio-Temporal Information in API Calls with Machine Learning Algorithms for Malware Detection, in: Proceedings of the 2nd ACM workshop on Security and Artificial Intelligence, 2009, pp. 55–62.
- [7] U. Bayer, A. Moser, C. Kruegel, E. Kirda, Dynamic analysis of malicious code, *Journal in Computer Virology* 2 (1) (2006) 67–77.
- [8] C. Brodley, M. Friedl, Identifying mislabeled training data, *Journal of Artificial Intelligence Research* 11 (1999) 131–167.
- [9] D. Bruschi, L. Martignoni, M. Monga, U. di Milano, Detecting self-mutating malware using control-flow graph matching, in: *Lecture Notes in Computer Science*, vol. 4064, 2006, pp. 129–143.
- [10] G. Casas-Garriga, P. Diaz, J. Balcazar, ISSA: An integrated system for sequence analysis, Tech. rep., Technical Report DELIS-TR-0103, Universitat Paderborn, 2005.

- [11] M. Castro, M. Costa, T. Harris, Securing software by enforcing data-flow integrity, in: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, 2006, pp. 147–160.
- [12] M. Christodorescu, S. Jha, Testing malware detectors, ACM SIGSOFT Software Engineering Notes 29 (4) (2004) 34–44.
- [13] M. Christodorescu, S. Jha, S. Seshia, D. Song, R. Bryant, Semantics-aware malware detection, in: IEEE symposium on security and privacy, 2005, pp. 32–46.
- [14] M. F. et. al., Symantec global Internet security threat report, volume XV, in: Trends for 2009, 2010.
- [15] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, B. Miller, Formalizing sensitivity in static analysis for intrusion detection, in: IEEE Symposium on Security and Privacy, 2004, pp. 194–210.
- [16] D. Fisch, A. Hofmann, B. Sick, On the versatility of radial basis function neural networks: A case study in the field of intrusion detection, Information Sciences 180 (12) (2010) 2421–2439.
- [17] S. Forrest, S. Hofmeyr, A. Somayaji, T. Longstaff, et al., A sense of self for Unix processes, in: IEEE Symposium on Security and Privacy, 1996, pp. 120–128.
- [18] D. Gao, M. Reiter, D. Song, Behavioral distance measurement using hidden markov models, in: Recent Advances in Intrusion Detection, 2006, pp. 19–40.
- [19] J. Giffin, S. Jha, B. Miller, Efficient context-sensitive intrusion detection, in: Proceedings of the 11th Network and Distributed System Security Symposium, 2004.
- [20] R. Hund, T. Holz, F. Freiling, Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms, in: Proceedings of the USENIX Security Symposium, 2009, pp. 383–398.

- [21] M. Inc., Microsoft. Digital Signatures for Kernel Modules on Systems Running Windows Vista. .
- [22] J. Kinder, S. Katzenbeisser, C. Schallhart, H. Veith, et al., Detecting malicious code by model checking, in: Conference on Detection of Intrusions and Malware & Vulnerability Assessment, 2005, pp. 174–187.
- [23] E. Kirda, C. Kruegel, G. Banks, G. Vigna, R. Kemmerer, Behavior-based spyware detection, in: Usenix Security Symposium, vol. 15, 2006.
- [24] C. Kolbitsch, P. Comparetti, C. Kruegel, E. Kirda, X. Zhou, X. Wang, U. Santa Barbara, Effective and efficient malware detection at the end host, in: Proceedings of the 18th USENIX Security Symposium, 2009, pp. 351–366.
- [25] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, G. Vigna, Polymorphic worm detection using structural information of executables, in: Lecture Notes in Computer Science, vol. 3858, 2006, pp. 207–226.
- [26] C. Kruegel, W. Robertson, G. Vigna, Detecting kernel-level rootkits through binary analysis, in: Proceedings of the Annual Computer Security Applications Conference, 2005, pp. 91–100.
- [27] W. Li, S. Stolfo, A. Stavrou, E. Androulaki, A. Keromytis, A Study of Malcode-Bearing Documents, Detection of Intrusions and Malware, and Vulnerability Assessment 4579 (2007) 231–250.
- [28] W. Li, K. Wang, S. Stolfo, B. Herzog, Fileprints: Identifying file types by n-gram analysis, in: Proceedings of the IEEE Workshop on Information Assurance and Security, 2005, pp. 64–71.
- [29] Z. Li, X. Wang, Z. Liang, M. Reiter, AGIS: Towards automatic generation of infection signatures, in: IEEE International Conference on Dependable Systems and Networks With FTCS and DCC, 2008, pp. 237–246.

- [30] B. Mehdi, F. Ahmed, S. A. Khayyam, M. Farooq, Towards a Theory of Generalizing System Call Representation For In-Execution Malware Detection, in: Proceedings of the IEEE International Conference on Communication, 2010, pp. 1–5.
- [31] S. B. Mehdi, A. K. Tanwani, M. Farooq, Imad: In-execution malware analysis and detection, in: Proceedings of the Genetic and Evolutionary Conference, 2009, pp. 1553–1560.
- [32] E. Menahem, A. Shabtai, L. Rokach, Y. Elovici, Improving malware detection by applying multi-inducer ensemble, *Computational Statistics & Data Analysis* 53 (4) (2009) 1483–1494.
- [33] A. Moser, C. Kruegel, E. Kirda, Limits of static analysis for malware detection, in: Annual Computer Security Applications Conference (ACSAC), 2007, pp. 421–430.
- [34] D. Mutz, F. Valeur, C. Kruegel, G. Vigna, Anomalous System Call Detection, *ACM Transactions on Information and System Security* 9 (1) (2006) 61–93.
- [35] Offensive-Computing, an online malware collection database, <http://offensivecomputing.net>.
- [36] N. Park, S. Oh, W. Lee, Anomaly intrusion detection by clustering transactional audit streams in a host computer, *Information Sciences* 180 (12) (2010) 2375–2389.
- [37] N. Paul, S. Gurumurthi, D. Evans, Towards Disk-Level Malware Detection, *Code Based Software Security Assessments* (2005) 13.
- [38] M. Preda, M. Christodorescu, S. Jha, S. Debray, A semantics-based approach to malware detection, *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30 (5) (2008) 25.

- [39] Q. Qian, M. Xin, Research on hidden Markov model for system call anomaly detection, *Intelligence and Security Informatics (2007)* 152–159.
- [40] R. Riley, X. Jiang, D. Xu, Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing, in: *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, 2008, pp. 1–20.
- [41] P. Royal, M. Halpin, D. Dagon, R. Edmonds, W. Lee, Polyunpack: Automating the hidden-code extraction of unpack-executing malware, in: *Proceedings of the Computer Security Applications Conference*, 2006, pp. 289–300.
- [42] A. Seshadri, M. Luk, N. Qu, A. Perrig, SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes, *ACM SIGOPS Operating Systems Review* 41 (6) (2007) 350.
- [43] M. Z. Shafiq, S. A. Khayam, M. Farooq, Embedded Malware Detection using Markov n-grams, *Lecture Notes in Computer Science* 5137 (2008) 88–107.
- [44] M. Z. Shafiq, S. M. Tabish, M. Farooq, PE-probe: leveraging packer detection and structural information to detect malicious portable executables, in: *Proceedings of the 18th Virus Bulletin Conference*, 2009.
- [45] M. Z. Shafiq, S. M. Tabish, F. Mirza, M. Farooq, PE-Miner: Mining Structural Information to Detect Malicious Executables in Realtime, in: *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, 2009, pp. 121–141.
- [46] F. Shahzad, S. Bhatti, M. Shahzad, M. Farooq, In-Execution Malware Detection using Task Structures of Linux Processes, in: *IEEE International Conference on Communication (InPress)*, 2011.
- [47] F. Shahzad, M. Farooq, Elf-miner: Using structural knowledge and data mining methods to detect new (linux) malicious executables, *Knowledge and Information Systems* (2011) 1–24.

- [48] D. Stopel, R. Moskovitch, Z. Boger, Y. Shahar, Y. Elovici, Using artificial neural networks to detect unknown computer worms, *Neural Computing and Applications* 18 (7) (2009) 663–674.
- [49] P. Szor, *The art of computer virus research and defense*, Addison-Wesley Professional, 2005.
- [50] G. Taha, Counterattacking the packers, in: McAfee Avert Labs, 2007.
- [51] A. K. Tanwani, J. Afridi, M. Z. Shafiq, M. Farooq, Guidelines to select machine learning scheme for classification of biomedical datasets, *EVOBIO, Lecture Notes in Computer Science* 5483 (2009) 128–139.
- [52] A. K. Tanwani, M. Farooq, The role of biomedical dataset in classification, *AIME, Lecture Notes in Artificial Intelligence* 5651 (2009) 370–374.
- [53] VX-Heavens, A free malware collection web site, <http://vx.netlux.org/>.
- [54] D. Wagner, D. Dean, Intrusion detection via static analysis, in: *IEEE symposium on security and privacy*, 2001, pp. 156–169.
- [55] X. Wang, W. Yu, A. Champion, X. Fu, D. Xuan, Detecting worms via mining dynamic program execution, in: *Proceedings of the 3rd International Conference on Security and Privacy in Communication Networks and the Workshops*, 2007, pp. 412–421.
- [56] Y. Wang, D. Beck, B. Vo, R. Roussev, C. Verbowski, A. Johnson, Detecting stealth software with strider ghostbuster, in: *Proceedings of the International Conference on Dependable Systems and Networks table of contents*, 2005, pp. 368–377.
- [57] Z. Wang, X. Jiang, W. Cui, P. Ning, Countering Kernel Rootkits with Lightweight Hook Protection, in: *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009, pp. 545–554.
- [58] C. Willems, T. Holz, F. Freiling, Toward automated dynamic malware analysis using cwsandbox, *IEEE Security & Privacy* (2007) 32–39.

- [59] I. Witten, E. Frank, Data mining: Practical machine learning tools and techniques with Java implementations, *ACM SIGMOD Record* 31 (1) (2002) 76–77.
- [60] I. H. Witten, E. Frank, Data mining: Practical machine learning tools and techniques, second edition, Morgan Kaufmann, 2005.
- [61] H. Yin, D. Song, M. Egele, C. Kruegel, E. Kirda, Panorama: Capturing system-wide information flow for malware detection and analysis, in: *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 116–127.
- [62] X. Zhu, X. Wu, Class noise vs. attribute noise: A quantitative study, *Artificial Intelligence Review* 22 (3) (2004) 177–210.

Table 6: Malware dataset with categories, names and file sizes[53] [35]

Malware Dataset				Benign Dataset							
No.	Name	Size (KB)	No.	Name	Size (KB)	No.	Name	Size (KB)	No.	Name	Size (KB)
1	Exploit.Linux.	12.3	55	Small.ao	5.74	1	pbmtobbbng	6.7	62	pnmtile	6.5
2	Local.p	14	56	Rpctime21527	6	2	arecord	49	63	pnmtorast	12.8
3	Espacec	14	57	Neo.a	7	3	blackjack	165	64	pnmtosir	6.7
4	Local.g	16.2	58	Rptime	7	4	blkid	10	65	ppncolors	6
5	Freezer	16.5	59	Small.a	7	5	cat	22	66	ppmdm	6.7
6	Soutown	5.13	60	Small.aa	7	6	cfdisk	48	67	ppmflash	6.7
7	Flooder.Linux.	18	61	Small.e	7	7	clock	32	68	ppmmake	6.7
8	pepsy.b	16.4	62	Exceedoor	8	8	gnuchess	173	69	ppmrelief	6.7
9	slice.c	15.6	63	Small.ap	8	9	colrm	8	70	ppmfpccx	20
10	small.n	15.2	64	Neo.b	9	10	cp	68	71	ppmtoppm	6.2
11	small.r	12.6	65	Batamacer.a	11	11	notification-ar	46.4	72	ppmatoterm	7
12	syn.a	23.2	66	Blachole.b	13	12	dbus-monitor	13	73	ppmtocxpm	11.1
13	Syn.b	15.3	67	Blachole.b	13.3	13	mingetty	13	74	ppmtv	6.7
14	Hactool.Linux.	820	68	Andrada.ca	14	14	dmesg	7	75	ps	78
15	bf.d	18.8	69	Andrada.a	14	15	writer	36	76	qritoppm	6.5
16	Net-Worm.Linux.	12	70	Bluez.a	17	16	dumpkeys	49	77	rm	44
17	Ramen14202	14	71	Agent.a	29	17	ed	45	78	same.gnome	93
18	Lion21442	17	72	BO.d	38.4	18	fc-cache	12.8	79	script	12
19	mworm.b	39.4	73	BO.c	48	19	tomboy	340	80	serviceTag	15.3
20	Mworm.a	43	74	quais	0.91	20	file	13.2	81	setsid	6
21	scalper.a	59.6	75	rick1627	16.9	21	find	94	82	python	5.7
22	scalper.b	60.2	76	satyr.a	10.9	22	gcaltool	193	83	shred	47
23	lupper.bm	432	77	snoopy.b	15.9	23	Gedit	717	84	slabtop	12.4
24	Rootkit.Linux.	1.57	78	svat.b	16.8	24	getkey	8	85	sendmail	880
25	agent.x	13.4	79	thebe.b	20	25	getkeycodes	7.9	86	Sort	79
26	agent.y	29.9	80	zipworm	3.83	26	giftotiff	15	87	splay	49
27	agent.30.chfn	138	81	Thebe.a	2.31	27	gkcytool	6.6	88	su	26
28	agent.30.chsh	138	82	Winter.340	2.85	28	glines	107	89	tie	12.2
29	Trojan.Linux.	11.4	83	Winter.343	3.2	29	glxgears	16	90	trashapplet	43
30	WrapLogin.c	11.9	84	Silvio.a	5.31	30	gnomevfs	486	91	toe	34
31	(Spy)Lsd.a	14	85	Silvio.59166313	6.41	31	gnomevfs.df	16.7	92	traceroute	49
32	(DDoS)Ris	15	86	Silvio.5916	9.89	32	gnome	103	93	ul	12.4
33	(DDoS)XChatSouls	15	87	RST.a6313	10.8	33	gst-feedback	11	94	unix2dos	15
34	(DDoS)PaulCyber.20	17	88	Satyr.b	10.9	34	gstinspect	11	95	metafy	5.15
35	(Spy)XKeyLogger.a	18	89	Ovets.b	12.1	35	gst-typefind	11	96	usleep	8
36	(DDoS)BlowFish	20	90	Vit.40966313	12.4	36	gtali	106	97	xload	38.8
37	(PSW)Small.b	20.1	91	Silvio.b	14.4	37	gtk-demo	163	98	yum-updatesd	13.3
38	(psw)small.cb	20.1	92	Snoopy.c	16.8	38	gucharmap	16.6	99	automount	410
39	(DDoS)Fork	20.4	93	Nel.b	17.1	39	gzip	68	100	xxd	16.5
40	Reboot.b	34	94	Snoopy.a	17.5	40	hunspell	43	101	aplay	11
41	Hactop	38	95	RST.b14933	19.2	41	hwclock	32	102	gnobots2	20
42	(Proxy)Hopbot.18	47	96	RST.b17168	19.8	42	kudzu	127	103	firefox	150.6
43	(Spy)Logftp	81	97	alaeda	20.3	43	fast-user-switc	96.9	104	evolution-alarm	132
44	Mircforce.b	96	98	RST.b12178	21.6	44	bluetooth-applet	62.2	105	gnome-screensav	173
45	(dropper)skitt.c	113	99	clifax	21.9	45	planner	36			
46	Baadoer.Linux.	2	100	RST.b18016	24.9	46	more	31			
47	Fpath.s	2	101	Osrf.875921380	27	47	netstat	128			
48	lhc.a	2	102	Osrf.875915904	40.9	48	oclock	32			
49	Promptt.e.a	2	103	Radix6313	57.7	49	pamflp	86			
50	Rootin.a	2	104	Telf.800019244	106	50	parted	86			
51	Rootin.c	2	105	Telf.800015631	154	51	gam-server	86			
52	Adore.b	4	106	RST.b12188	168	52	pbmtext	11.1			
53	Bofishy.a21527	4	107	RST.b18659	447	53	pbmtext	6.7			
54	Small.al	4	108	RST.b10114	462	54	pbmtext	6.7			
			109	RST.b17867	598	55	pbmtext	6.7			
			110	RST.b	826	56	pbmtext	6.7			
			111	Quasi6313	932	57	pbmtext	6.7			
			112	RST.b6313	1.31MB	58	pbmtext	6.7			
			113			59	pbmtext	6.7			
			114			60	pbmtext	6.7			
			114			61	ping	30			