Volatility Custom Profiling for Automated Hybrid ELF Malware Detection

Rahul Varshney, Nitesh Kumar, Anand Handa(✉), and Sandeep Kumar Shukla

C3i Center, Department of CSE, Indian Institute of Technology, Kanpur, Kanpur, India
{rvarshney20,niteshkr,ahanda,sandeeps}@cse.iitk.ac.in

Abstract. The increasing prevalence of Linux malware poses a severe threat to private data and expensive computer resources. Hence, there is a dire need to detect Linux malware automatically to comprehend its capabilities and behavior. In our work, we attempt to analyze the ELF binary files before, during, and after execution (or postmortem inspection) using open-source tools. We analyze the ELF binaries in a controlled sandboxed space and monitor the activities of these binaries and their child processes to assess their capabilities and behaviors. We set up `INetSim`, and simulate the fake internet services to increase the chances of malware behaving as intended. We also generate a custom `OS profile` of Ubuntu 16.04. The Volatility tool employs this profile to analyze the memory dump and extract the artifacts. We modify the Limon sandbox to use only specific volatility plugins, which reduces the time for report generation. We extract features from these behavior reports and reports from memory forensics and combine them with features extracted using static analysis to build a hybrid model for ELF malware detection. Our trained hybrid model offers a good accuracy of 99.2% on a recent dataset of benign and malware samples and with a minimal false-positive rate of 0.9%. To the best of our knowledge, no one in the literature has performed the memory analysis of ELF malware using the Volatility profile customization for efficient ELF malware detection.

Keywords: ELF malware · Malware detection · Memory forensics · Machine learning · Volatility · Limon sandbox

1 Introduction

Due to the proliferation of Internet usage in the past few years, the quantity of malware has exploded. Therefore, the computer user community requires automated malware detection strategies that are effective and efficient. Linux, a UNIX resembling OS has garnered global adoption. This is because of its open-source nature and widespread popularity on desktop and server systems. Due

S. Goel et al. (Eds.): ICDF2C 2022, LNICST 508, pp. 274–291, 2023.
https://doi.org/10.1007/978-3-031-36574-4_16

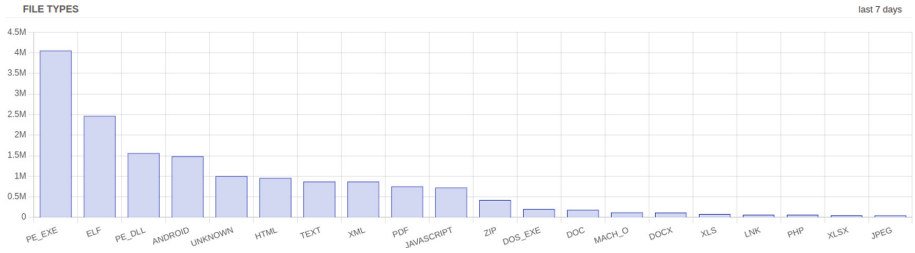

Fig. 1. Samples submitted to VirusTotal [3] in last 7 days

to Linux's extensive use in mobile and server machines, makes it a viable target for malware developers. The quantity of ELF samples reported to VirusTotal [3] in the last week of May 2022 is comparable to that of Windows PE binaries as shown in Fig. 1.

In 2008, a sudden surge in Linux-targeting malware was detected. "Shane Coursen, a senior technical consultant at Kaspersky Lab, stated, The surge of Linux malware is simply attributable to Linux's rising popularity, particularly as a desktop operating system" [7]. Malware (short for "malicious software") is a code chunk or an executable file, typically transmitted over a network, that infects, steals, investigates, or does practically any activity an attacker desires. Malware can be divided into numerous categories based on its behaviours. Some of them are – backdoor, trojan, virus, cryptojackers, etc.

There exist a few approaches to detect malware which are signature based and anomaly based. Signature based technique uses the pre-programmed list of known threats developed by the antivirus companies. This list generally contains signatures of the known threats that uniquely identifies that specific malware and the antivirus keeps on updating the list. In the past, this technique provided adequate protection until the malware authors became more advanced and developed methods like polymorphism to evade such signature based detection. An anomaly-based detection system employs machine learning models to teach the system to estimate a normalised baseline, as opposed to looking through a pre-programmed list to detect known threats. One of the advantages of using anomaly-based systems is that they have the potential to discover unknown threats. However, these systems are easily susceptible to a high number of false detection. This technique was particularly designed to detect suspicious characteristics that can be present in unknown and modified versions of existing known malware samples. "Malware authors are constantly developing new threats, and the anomaly-based approach is the only way to deal with this volume of malware emerging daily [9]." This strategy is one of the few that can combat polymorphic malware which undergoes continual changes and adapts to the environment. This method employs several different techniques such as static, dynamic, and memory analysis.

In static analysis, only static characteristics of a binary file are used. All the analysis should be complete without executing the binary, only using the

contents of the ELF header, embedded strings, and other statically extracted information. This analysis is rapid as we are not executing the binary, but along with its benefits, it has some limitations. Static analysis does not perform well for polymorphic and packed malware. In dynamic analysis, a binary executable is allowed to execute inside a secure and isolated environment. This environment is preferably a controlled virtual machine which is recoverable to a known safe state. This analysis requires the examination of the behaviour of binary under execution to classify it as malware or benign. It has the advantage of remaining unaffected by run-time packing and code obfuscation. Therefore this analysis overcomes the problem of polymorphic and packed malware. Besides these benefits, it has some limitations as well; some of them are – code coverage, anti-VM techniques, etc. Memory analysis can uncover unorthodox malware, such as memory-resident and fileless malware. Memory Analysis is the process of obtaining information about the status of a computer, the processes running on it, network connectivity, and other digital artifacts by analysing a memory image. Analysing the memory after malware execution provides a postmortem perspective and facilitates the extraction of forensics artifacts. Memory analysis also has some limitations like damaged/corrupted memory dump, unsupported memory structure, etc.

A memory image is simply the snapshot of the system's component and the current state of the main memory at a particular instant of time. Memory Image is basically like a photocopy of the main memory, which is helpful in the later examination. The generated image is saved in a format suitable for forensic examination,"Forensic image format namely .vmem, .mem, .dmp, .dump, .crash, .dat and many others". Some of these formats can differentiate between the main memory image and a secondary disk image. For instance, the image of physical memory will have some inaccessible sections as these were used in memory-mapped I/O, while the disk image does not. Various tools can be used to image and analyse the main memory of the machine. The procedure for accessing the main memory is different for different operating systems. After the memory is imaged, it is submitted for memory analysis to determine the system's current state and extract network information and other useful artifacts.

In the current scenario, there is a vast range of malware used to steal personal information, commit financial fraud, and attack vital infrastructures. Top multinational corporations and government organisations are currently investing large sums of money to be protected against these malicious activities. Either these enterprises attempt to rely on antivirus vendors, or they develop their own malware detection systems. Typically, these systems employ signature-based or anomaly-based detection approaches. We seek to construct a model that extracts information before, during and after execution of Linux binaries which is free from shortcomings of signature and anomaly based systems. We attempt to blend all three approaches, as one assists the others in overcoming their limitations. However, avoiding one analysis makes the executable susceptible to others. For instance, packaging and obfuscation prevent the executable from being analysed statically. As the executable file must do additional activities for execution, it is

easily traceable utilising system calls and additional process creation in dynamic and memory analysis. Consequently, it is considerably more difficult for malware programmers to circumvent all techniques simultaneously. Hence, we develop a model that combines characteristics from all of them with a minimum number of false positives and high detection accuracy. In our work, we have faced the following challenges –

- We have faced the challenge of exact memory image creation for the extraction of memory artifacts from the memory dump. One cannot be 100% sure that the formed memory image represents the correct state of `running` system.
- Even after memory image creation, the volatility framework [6] does not able to extract the artifacts from that memory dump because it does not support the kernel version of Ubuntu 16.04 or higher.

The major contributions of our work are as follows –

- To avoid the problem of damaged or corrupted memory image, we have used the `VMWare Workstation` which provides us the exact memory image of the guest OS by first saving and suspending the guest virtual machine.
- To avoid the problem of unsupported memory structure, we have built a specific OS profile [22] for Ubuntu 16.04 using `dwarfdump` tool. This profile is used by the volatility framework to parse the memory dump and provide the relevant information using a variety of volatility plugins.
- We have also customised the Limon sandbox to use specific volatility plugins for Linux OS such as `linux_pslist, linux_pstree, linux_psxview, linux_psaux, linux_malfind, linux_netscan`, etc. Only these plugins are providing some output for memory image of used Linux OS.
- We have extracted the distinctive features before, during and after the execution of ELF. We have also built a machine learning model that uses an aggregate of extracted features from static, dynamic, and memory analysis and performs automatic malware detection on a significant count of binary executable files without the need for manual intervention.

The rest of the paper is organized as – Sect. 1 discusses the various approaches to detect malware, challenges faced in our work and the contributions. We addressed some prior work in the realm of Linux malware detection employing static, dynamic, and memory analysis in Sect. 2. Section 3 explains the design methodology. The experimental results and the dataset description is described in Sect. 4. Section 5 concludes the work with subjective future directions.

2 Related Work

This section examines some of the researches in the field of Linux malware detection. They are discussed as follows –

Static Detection Approaches. Shalaginov Andrii [21] proposed a methodology for the classification of Linux malware into various families using Deep

Neural Network (DNN). Their approach overcomes the limitation of the shallow neural network used for Windows PE files classification. Their dataset includes 10574 ELF files labelled by Microsoft following the naming convention of malware standardised by CARO (Computer Anti-Virus Research Organization). The authors have extracted 30 features from ELF format and VirusTotal for classification and achieved an accuracy of 71% for classification among 19 different malware categories with a concise model of 10 layers. However, adding features from the behavioural analysis may result in an improvement in accuracy.

Jinrong Bai et al. [11] introduced a new technique for detecting malware in which system calls were extracted from the executable's symbol table. They chose 100 of these extracted system calls as features out of the numerous available options. Their dataset collection includes 756 benign ELF executables extracted from Linux systems binaries and 763 malicious ELF executables downloaded from the VX heavens. Their approach achieved a detection rate of 98% for malware.

The writers of ELF-Miner [19], Shahzad F. analysed executable and linkable format (ELF). They have retrieved 383 features from the ELF header. Information gain has been employed as the algorithm for feature selection. For categorisation, they used the well-known algorithms of supervised learning, namely decision tree J48, PART (Partial Decision Tree), RIPPER (Repeated Incremental Pruning to Produce Error Reduction), and C4.5 Rules. Their dataset collection included 709 benign ELF executables scraped from the Linux system binaries and 709 malicious ELF executables scraped from VX heavens and offensive computing. Their approach recorded a detection rate of approximately 99% with less than 0.1% false alarms.

Static Analysis techniques can have less detection time as the binary is not allowed to execute, but it can be easily thwarted by malware authors using packing and obfuscation techniques. Due to the packing of executables, one can not extract much helpful information without allowing it to execute in a system.

Dynamic Detection Approaches. Zhang Zhaoqi [23] proposed a novel low-cost feature extraction approach from dynamic analysis of Windows malware and an effective deep neural network (DNN) architecture for quick malware detection. They have represented API call arguments in the hashed form to keep distinct features. DNN architecture initially transforms those extracted features using Gated-CNNs (convolutional neural network), and these transformed features are further passed through bidirectional LSTM (long short term memory network) to understand the correlation among API calls. They have used a dataset of 27287 malicious and 33400 benign PE samples obtained from SecureAge Technology of Singapore. After various experiments on the count of gated CNNs and bi-LSTM layers, their final configuration achieved an accuracy of 98.80%. They have not mentioned using anything to evade the sandbox detection before the generation of execution logs.

K. A. Ashmita [10] proposed a method depending on the use of system call characteristics. They employ 'strace' to monitor all the system calls made by executables operating in a contained environment. They have classified the system

calls into four classes: union, intersection, and distinctive features for benign and malware files. They have used correlation-based feature reduction in two steps. In order to rank the features, they evaluated feature-class correlation using entropy change and information gain before calculating feature-feature correlation to eliminate redundant features. For the classification of Linux malware, they employed three popular algorithms of supervised learning namely decision tree J48, Random Forest, and AdaBoost, and their feature set had 27 features. The dataset employed by the authors of this work contains 668 files, 442 of which are benign and 226 of which are malware. From this strategy, they achieved a 99.40% accuracy.

Shahzad, F., and Shahzad M. [20] have presented the idea of a genetic footprint that mines the information from the kernel's Process Control Block (PCB) of ELFs and uses that information to determine the behaviour of a process at runtime. In their method, the authors have selected 16 from a total of 118 available "task_struct" parameters for each operational process. The writers claim to have conducted forensics research to determine which factors to use. According to the authors, the selected parameters will describe the semantics and behaviour of the executing process. They have compiled a system call dump containing these parameters collected over 15 s with a 100-ms resolution. In the WEKA environment, all benign and malicious sample processes are classified by employing multiple algorithms, namely the J48 decision tree, SVM, a propositional rule learner (J-Rip), and RBF-Network. They have analysed their results and identified J-48 J-Rip classifiers with the least amount of class noise. The dataset employed by the authors consists of 219 samples, 105 of which are benign processes and 114 newly gathered malicious processes. From this strategy, they achieved a 96% accuracy and 0% false positive rate within less than 100 ms of the onset of malicious operation.

Dynamic analysis can provide a better understanding of malware behaviour than static analysis, but some precautions should be taken to avoid potential security risks. Moreover, dynamic analysis is more expensive both in terms of time and resources used for malware detection. As the malware analyst does not manually interact with the binary during execution, multiple paths for execution remain unexplored.

Memory Based Detection. "Memory analysis has been demonstrated to be a potent analysis tool that can effectively examine the activities of malware" [18]. Memory analysis draws malware experts because it provides a full examination of malware by examining malicious hooks and code outside the regular scope of a function. It analyses information about executing processes and the general overview of the system using stored memory image.

Sihwail Rami [17] presented a novel approach for classification and detection of Windows malware, which retrieves memory-based characteristics from images utilising memory forensic procedures. Those characteristics may reveal the malware's true behaviour, such as demanding elevated rights to carry out particular tasks, interaction with the operating system, connecting with the command and control server, and DLL and process injection. Their dataset collection consists

of 966 benign and 2502 malicious executables retrieved from VirusTotal. Their approach represents malicious behaviour by six feature types, namely API calls, the process handles, network, DLLs, code injection, and privileges resulting in a total feature count of 8898 features. The authors claim to do feature selection using Information gain and correlation as well, but it results in degradation of accuracy. Using the SVM classifier, their method achieved a detection accuracy of 98.5% for malware with a false positive rate of as low as 1.24%. Due to a vast number of features, their approach suffers in time complexity, thereby taking much longer to train and test the model.

Mosli Rayan [15] employs difference in 'the use of handles' by benign and malware executables. They have extracted and exploited this usage difference to classify the executables into two categories (malware and benign). The authors have used the cuckoo sandbox for automating malware execution and generation of memory dumps. They have also used the volatility framework for extracting the handles information from the memory dump. Their dataset consists of 3130 malware samples and 1157 benign samples, which are divided in the ratio of 80:20 for training and testing, respectively. They have trained three classifiers, namely KNN, Random Forest, and SVM. Random Forest surpassed the other two methods and achieved an accuracy of 91.4% with precision and recall of 89.8% and 91.1%, respectively. Their approach primarily focuses only on handles information and neglects the other artifacts present in the memory.

Using only the memory analysis technique for malware detection suffers from a drawback that it can only be employed after the system gets infected, thereby making it the second layer of defence against malware detection.

3 Design and Implementation

The complete overview of our proposed framework for malware detection is shown in Fig. 2. We first filter the samples using `ssdeep` to eliminate polymorphic samples and acquire the actual labels for ELFs using VirusTotal API [5]. We use tools like `readelf` [4], `strings` to extract features from various headers of ELF. Our framework submits the ELF executable to the Limon sandbox for their safe execution inside a guest virtual machine. The framework starts `INetSim` [1] to simulate all the fake internet services and their simulation to decrease the chances of sandbox detection by the malware. We execute the samples for 30 s. During this time frame, our framework monitors all the activities such as – system calls, files or directories accessed, IPs contacted, etc. After execution, volatility processes the guest machine's memory dump for extraction of memory forensics artifacts. We then perform feature reduction and classification for malware detection.

For dynamic analysis, we setup Limon Sandbox. Currently, Limon supports Python 2 only. We install some tools on the host OS, some on the guest OS and some tools on both. Some of these tools are preinstalled on recent Linux distributions. The following tools are required to be installed for the proper functioning of the Limon sandbox which is explained as follows:

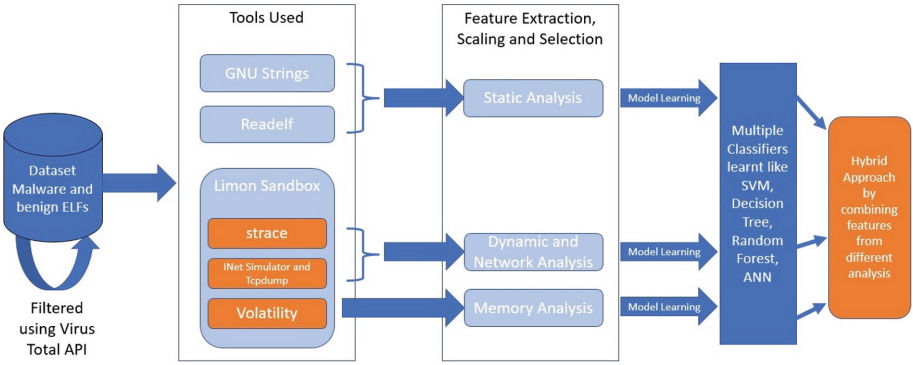

Fig. 2. General Architecture for Linux malware detection

Host System Configurations. We use the host system with specific configurations. The OS on the host machine is Ubuntu 18.04.6 LTS with Intel Core i7 CPU, 1 TB hard drive space, and 16 GB RAM. We install `VMWare Workstation 16.1.2` on the host machine. It is used to create a guest virtual machine and to generate a consistent memory image for analysis. Next, we set up the `ssdeep` on the system because Limon uses it to filter the samples using the fuzzy hashing technique, `GNU strings` utility to extract meaningful strings from executables, and `readelf` to extract header data from ELFs. To detect the packed executables, we install YARA-Python on the host. `INetSim` is installed on the host machine to simulate the fake internet services. Lastly, we configure the `Volatility Framework` to analyze the memory dump provided by the VMWare workstation. We build a custom profile for Ubuntu 16.04 (guest machine) to ensure the proper functioning of the volatility framework. The profile for Linux is a zip file with information about the kernel's data structures and debug information. We built the profile using the tool known as `dwarfdump`, which requires the exact kernel version.

Guest System Configurations. On the VMWare workstation, we install a guest OS with the specifications – Ubuntu 16.04 LTS OS with Intel i7 CPU, 4 GB RAM and 60 GB hard drive space. This guest machine is used by the Limon for the execution of samples and generation of reports. We set the 'root' password of the guest OS and enable the root login using the graphical user interface to execute the malware as root. We install `strace` to collect the system call traces and for execution of 32-bit samples on the 64-bit OS, we add `i386 architecture`. We also install a few library packages namely `libc6:i386, libstdc++6:i386, and libncurses5:i386`. Lastly, we allocate a static IP to the guest machine, clear up the bash history, and capture a snapshot of the guest machine. We name this snapshot as `cleansnapshot`, and it is used as a checkpoint to revert the guest machine to a non-infected state before the execution of the next ELF binary file.

Once the host and guest machine setup is completed, we configure the Limon sandbox's configuration file – `conf.py`. The configuration file includes various settings like the guest machine static IP, directory for sample transfer to the guest machine, root login details of the guest machine, directory for saving final report on the host machine, path for memory dump of the guest machine, path for `tcpdump` to sniff the network activity of the guest machine, path for `strace`, `VirusTotal Public API` to get detection results from antivirus engines, etc. The proposed methodology includes – data generation, feature extraction, feature selection, and classification which are explained further.

3.1 Data Generation

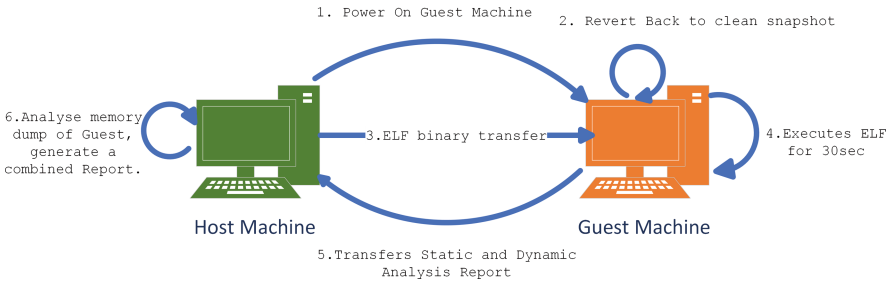

Fig. 3. Process of Report generation

Figure 3 shows the overall setup which is used for data generation for an ELF file. We modify the Limon sandbox source code to extract the output for a specific volatility plugin running on a dump of a particular Linux kernel of Ubuntu 16.04. We submit the sample to the guest machine after the machine gets restored to the uninfected state to ensure that one ELF's effects do not affect others' reports. We start the execution of the ELF binary for 30 s. After this, we stopped the monitoring processes and suspended the guest virtual machine. Next, we acquire the main memory image of the guest machine formed by the VMWare workstation. We analyze the acquired memory image using the Volatility framework 2.6.1 for extracting the list of processes running, list of hidden processes, list of open and closed IP ports, etc. Lastly, we store the complete report, network capture file, and malicious artifacts for further analysis. We repeat this process for all the available ELFs and store their respective reports for feature extraction.

3.2 Feature Engineering

In this section, we discuss the various features extracted from static, dynamic, and memory analysis. We use different tools for the feature extraction in these categories of analysis. The various categories of features are as follows:

Static Features. We extract features from various parts of the ELF structure. These structures are available with the help of two tools; GNU strings and

Table 1. ELF header comparison Benign vs Malware

Features	Mean for Benign	Mean for Malware
Number of section headers	30.609	248.171
Size of ELF header	69.178	52.425
Number of program headers	9.211	4.024
Start of section headers	287014.754	6884126.476
Start of program headers	70.629	5037.257
Section header string table index	28.591	246.143
Size of program headers	58.391	32.977
Size of section headers	67.163	40.804

readelf. Using various arguments to these tools, one can obtain a specific part of the ELF structure. Some of these parts are listed are as follows –

ELF Header: One can extract the ELF header using the '`readelf -h [elf_file]`' command. This header gives us information about the organisation of the ELF File. Mean comparison of the features extracted from the ELF header for benign and malware samples is listed in Table 1.

Program Header Table: One can extract the program header table using the '`readelf -l [elf_file]`' command. We use segment type (LOAD, PHDR, INTERP, NOTE, DYNAMIC) in the feature set as a binary feature.

Section Header Table: One can extract the section header table using the '`readelf -S [elf_file]`' command. We use section name (like .bss, .comment, .data, .dynamic, .rodata, .strtab etc.) and section type (like NOBITS, DYNAMIC, SYMTAB, HASH, RELA, PROGBITS, NULL etc.) as features. We use both section names and section types as binary feature.

Symbol Table: One can extract the symbol table using the '`readelf -s [elf_file]`' command. The symbol table contains considerable data required for linking and debugging files. We use the count of `.dynsym` entries, count of `.symtab` entries, type of `.dynsym` entries, and type of `.symtab` entries as features.

Dynamic Section: One can extract the dynamic section of the ELF file using the '`readelf -d [elf_file]`' command. The runtime linker uses this segment to find all the necessary information needed for dynamic linking and relocation. The number of entries in the dynamic section is not fixed. We use the entries (like FINI, NULL, INIT_ARRAY, FINI_ARRAY, HASH etc.) in the feature list as binary feature.

Strings: We use the '`strings`' tool from GNU Binutils to extract the printable character sequences that are at least four characters long and follow an unprintable character. The output of strings tools lists HTTP GET request to a website as shown in Fig. 4.

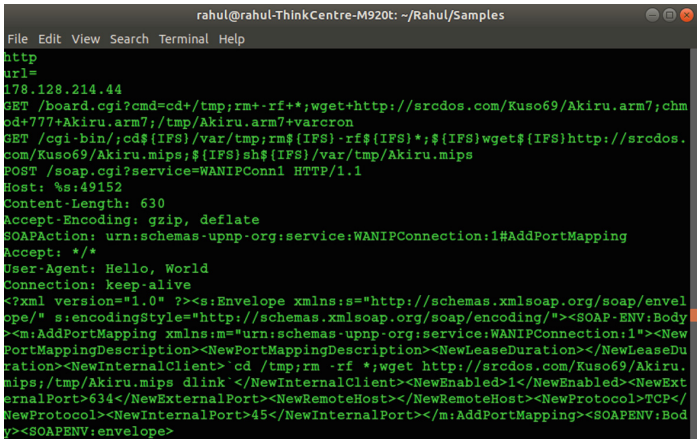

Fig. 4. Portion of `strings` output for ELF

Dynamic Features. The runtime behavior-based characteristics are extracted from the reports of the files provided by Limon. The complete Limon sandbox report is a text file containing system call traces and network activity (captured using Wireshark). The system call describes the operations a process performs, which is referred to as its runtime behavior. Limon sandbox uses "strace" to obtain a comprehensive system call trace of a process and its child processes. In this work, we use the system calls, system call arguments, and TCP packet information as a feature set. Also, we extract the directories and files supplied as arguments to these calls as a feature. Our feature set includes the top 20 directories or files accessed and the generated system calls. Some important features are – `open, close, read, write, connect, clone, /dev, /usr, /proc, etc.`

Memory Artifacts. The Volatility tool uses the snapshot of the main memory and the Volatility profile for the underlying OS. The variety of Volatility plugins provides the information from the memory dump as an output which is further formulated as a feature. Some extracted features are the count of running processes, multiple processes started by an ELF, child processes count, hidden processes count, the number of TCP packets shared with a specific IP, number of IPs contacted, count of distinct IPs contacted, and whether an IP is contacted via multiple ports. Features retrieved using the Volatility tool are listed as follows:

- Using the linux_pslist command, we can determine whether multiple processes are created, or multiple instances of the same process are created. All of these processes are *marked* using `PID` for further feature extraction.
- There are some processes that are not listed using linux_pslist but visible using linux_psxview, all these process are *marked* as hidden processes.
- Count of IPs contacted with single or multiple ports. Count of ports, count of distinct IPs contacted, etc. are used in the feature set.

- We extract the count of malicious processes which contains injected or statically hidden code using the 'linux_malfind' plugin.
- We extract all the network connections made by processes using the 'linux_netstat' plugin. Extraction of connection information (whether the connection is open or closed) for any *marked* process is used in the feature set.
- We also extract the output of other volatility plugins like 'linux_ifconfig', 'linux_check_tty' but they does not provide any distinct data and some plugins like 'linux_bash' currently does not return any output for the version of Ubuntu 16.04 and higher versions.

After feature generation from the report, we attempt to reduce the features using Principal Component Analysis (PCA) [14]. Initially, we have extracted 87 static features, 328 dynamic features, and 26 memory features. However, all the extracted features are not necessary while training the classifiers. Therefore, we apply PCA to reduce the dimensionality of the feature vector. The final feature vector contains 60 static features, 40 dynamic features, and 21 memory features.

3.3 Classification

We use machine learning classifiers such as – Decision Tree [13], Random Forest [12], and SVM [16] for ELF malware detection. To train and test the models, we use Python's `sklearn` library, and ten-fold-cross validation is used for model evaluation on unseen data. In our work, we use stratified k-fold cross-validation. The reason to use cross-validation is that it results in less biased and less promising outcomes. In our work, we utilize the default value of the number of trees as 100 in the case of the Random Forest classifier. Similarly, we use the default parametric values for the rest of the classifiers – SVM and Decision Tree.

4 Experimentation and Results

4.1 Dataset

We collect the ELF malware samples from a publicly available repository VirusShare [8]. We filter the samples using the `ssdeep` utility to remove more than 80% of identical polymorphic samples. To double-check the labels for malware files, we use VirusTotal API to obtain the detection results from various antivirus engines. We label a sample as malware if any sample is marked as malware by more than four antivirus engines. The final dataset (referred as `Dataset-1`) contains 5,772 ELF files, of which 2,268 are benign executables collected from the Linux directories such as `/usr/bin, /usr/sbin`, etc., and 3,504 are malicious. Apart from these samples, we gather a few recent malware samples from Virussamples [2]. We mix the collected recent malware samples with a few benign executables, which are not part of `Dataset-1`. This mixed dataset is referred to as `Dataset-2`, containing 888 samples with 413 benign and 475 malicious samples. We use `Dataset-2` to test the robustness of our model. Table 2 presents the exact details of both the datasets.

Table 2. Dataset Description

Sample Type	Dataset Name	
	Dataset-2	Dataset-1
Benign	413	2268
Malware	475	3504

4.2 Evaluation Metrics

To evaluate our work, we use the following evaluation metrics:

- **True Positive(TP):** Benign Samples predicted correctly.
- **False Positive(FP):** Malicious Samples predicted as benign.
- **False Negative(FN):** Benign Samples predicted as malicious.
- **True Negative(TN):** Malicious Samples predicted correctly.

- **Accuracy:** It is defined as the proportion of times the classifier makes accurate predictions.

$$Accuracy(Acc) = \frac{TP + TN}{TP + FP + TN + FN}$$

- **Precision:** It is defined as the proportion of times the classifier predicts true which are actually true.

$$Precision(Pr) = \frac{TP}{TP + FP}$$

- **True Positive Rate:** It is defined as the proportion of samples classifier predicts true to actual true samples. It is also called as Recall (Re).

$$TPR = \frac{TP}{TP + FN}$$

- **False Positive Rate:** It is defined as the proportion of samples classifier predicts true to actual false samples.

$$FPR = \frac{FP}{FP + TN}$$

4.3 Results

This section presents the results of various classifiers on a different combination of features extracted from static, dynamic, and memory analysis. We perform feature scaling before we train the classifiers and split the `Dataset-1` in the ratio of 80:20 for training and testing the classifiers. We test the robustness of our models, which are trained on `Dataset-1` using recent samples of `Dataset-2`. Table 3 presents the values of different evaluation matrices obtained using the

Table 3. Results for Static Analysis using multiple classifiers

Classifier	Dataset-1				Dataset-2			
	Acc(%)	Pr(%)	TPR(%)	FPR(%)	Acc(%)	Pr(%)	TPR(%)	FPR(%)
SVM	97.9	97.2	99.6	4.9	92	97.4	87.4	2.7
Decision Tree	97.8	97.9	98.6	3.5	97.7	98.7	96.2	1.5
Random Forest	98.6	98.4	99.4	2.8	97.4	95.5	99.1	4.4

static feature set. It is evident from Table 3 that all three classifiers provide almost similar test accuracy on `Dataset-1`. When we test our trained models on samples from `Dataset-2`, the accuracy and TPR of the SVM classifier reduced significantly. In comparison, there is only a slight decrease for the other two classifiers.

Similarly Table 4 and Table 5 presents the results using dynamic and memory-based feature sets. Here, the accuracy and TPR on `Dataset-2` for the SVM classifier is less than the values for other classifiers. Due to fewer samples in the `Dataset-2`, the precision and FPR have not changed significantly. One can infer from the tables that out of all three classifiers, Random Forest outperforms the other two classifiers for both `Dataset-1` and `Dataset-2`. It is also clear from the tables that classifiers on static features set perform slightly better than dynamic and memory-based features set.

Table 4. Results for Dynamic Analysis using multiple classifiers

Classifier	Dataset-1				Dataset-2			
	Acc(%)	Pr(%)	TPR(%)	FPR(%)	Acc(%)	Pr(%)	TPR(%)	FPR(%)
SVM	91.6	91.0	95.9	15.0	85.0	96.7	74.5	2.9
Decision Tree	93.4	95.8	93.3	6.5	90.4	98.0	82.5	1.9
Random Forest	94.5	96.5	94.4	5.4	91.2	98.5	84.0	1.4

Table 5. Results for Memory Analysis using multiple classifiers

Classifier	Dataset-1				Dataset-2			
	Acc(%)	Pr(%)	TPR(%)	FPR(%)	Acc(%)	Pr(%)	TPR(%)	FPR(%)
SVM	83.3	92.9	77.9	8.7	73.3	86.9	58.9	10.1
Decision Tree	89.1	93.5	87.8	9.0	82.4	96.9	68.0	2.4
Random Forest	90.7	95.3	88.8	6.4	85.3	97.7	73.5	1.9

Next, we reduce the features using PCA based on the correlation among features for better generalization. Feature reduction results in a slight decrease in accuracy for the Random Forest classifier on static, dynamic, and memory feature sets, whereas we see an improvement in the results for the Decision

Table 6. Results for Static Analysis after Dimensionality Reduction

Classifier	Dataset-1				Dataset-2			
	Acc(%)	Pr(%)	TPR(%)	FPR(%)	Acc(%)	Pr(%)	TPR(%)	FPR(%)
SVM	97.9	97.2	99.5	4.9	94.1	97.5	91.3	2.7
Decision Tree	98.4	98.5	98.8	2.6	96.9	98.6	94.3	1.5
Random Forest	98.6	97.9	99.8	3.5	98.9	99.1	97.9	1.0

Tree and SVM classifiers. Table 6 shows the results for all three classifiers after dimensionality reduction for the static features set.

Table 7 and Table 8 show the result for the reduced dynamic and memory-based feature set with dimensions 40 and 21, respectively. In the case of the dynamic-based feature set, all three classifiers have similar accuracy on `Dataset-1`, but the Random Forest has the least FPR (which also results in less TPR). Random Forest shows the minimum reduction in accuracy when testing `Dataset-2`, while all three classifiers show a significant decrease in TPR.

Table 7. Results for Dynamic Analysis after Dimensionality Reduction

Classifier	Dataset-1				Dataset-2			
	Acc(%)	Pr(%)	TPR(%)	FPR(%)	Acc(%)	Pr(%)	TPR(%)	FPR(%)
SVM	93.1	95.3	93.2	6.8	83.6	95.5	73.0	3.8
Decision Tree	93.2	95.8	92.8	6.0	86.8	96.3	77.0	3.4
Random Forest	93.2	95.9	92.6	5.8	88.9	98.4	79.9	1.5

Table 8. Results for Memory Analysis after Dimensionality Reduction

Classifier	Dataset-1				Dataset-2			
	Acc(%)	Pr(%)	TPR(%)	FPR(%)	Acc(%)	Pr(%)	TPR(%)	FPR(%)
SVM	83.1	92.3	78.2	9.7	74.1	87.4	60.2	9.9
Decision Tree	90.0	95.7	87.2	5.7	83.4	97.0	69.8	2.4
Random Forest	90.0	94.9	87.9	6.8	85.0	97.7	72.8	1.9

We combine all three categories of features and form a hybrid feature set for detection result improvement on `Dataset-2`. We observe from static, dynamic, and memory analysis that the Random Forest classifier performs the best among all the classifiers in terms of all the evaluation metrics. Therefore, we choose to utilize Random Forest with 100 trees to train various combinations of hybrid classification models. Till now, we have observed that the detection results for ELF malware are not promising. Even though the static analysis performs well compared to dynamic and memory analysis methods, the single feature category results are not enough to detect ELF malware. Hence, we evaluate our model using different combinations of static, dynamic, and memory features as shown

Table 9. Results of Random Forest on Hybrid Features

Feature Set	Dataset-1				Dataset-2			
	Acc(%)	Pr(%)	TPR(%)	FPR(%)	Acc(%)	Pr(%)	TPR(%)	FPR(%)
Static + Dynamic	98.6	98.8	98.8	1.7	97.1	95.9	98.9	4.8
Dynamic + Memory	96.6	98.0	96.2	2.7	93.4	99.7	87.8	2.0
Static + Memory	98.5	98.3	99.2	2.5	97.7	98.3	97.4	1.9
Static + Dynamic + Memory	99.4	99.5	99.5	0.7	99.2	99.1	99.4	0.9

in Table 9. We train the Random Forest classifier using these combinations of features on 80% of the samples from `Dataset-1`, and the remaining 20% are used to test the model. To check the efficacy of our model, we test our trained model using recent samples from `Dataset-2`. Table 9 shows that the combination of dynamic and memory-based features gives the least accuracy among other combinations. The combination of static, dynamic, and memory-based features achieves the best accuracy and least FPR on both the datasets – `Dataset-1` and `Dataset-2`. Also, this combined feature set detects malware with high TPR without predicting much benign as malware (low FPR) which explains the high precision value of the model.

5 Conclusion and Future Work

In this work, we perform analysis on ELF executables using static, dynamic, and memory analysis approaches. We utilize different tools to extract features from all the analysis techniques. We use `readelf` and `strings` tools to retrieve the static features. The Limon sandbox is set up in VMWare to extract the behavioral logs for dynamic analysis. We also customize the Volatility profile for successfully creating and analyzing memory dump for memory forensics using various Volatility plugins, which is one of the key contributions of our work. The experiments are performed using two different datasets: one dataset is used to train and test the model, and the second dataset contains recent samples to test the robustness of the model. We also utilize feature reduction using PCA for faster prediction. After performing the feature engineering, we build three classification models; two of them are tree-based classifiers – Decision Tree and Random Forest, and one is a support vector machine. Our experimental results include the analysis using features obtained from individual approaches and the various combination of all the features from different approaches. We achieve the best result using hybrid features having static, dynamic, and memory analysis features with the Random Forest classifier, which is 99.47% for `Dataset-1` and 99.21% for `Dataset-2`. The results prove that our work can detect unseen malware which is not part of the training and testing dataset with low false positives. The data and codes are available on request.

Currently, our work primarily focuses on the ELF file format. However, there are threats to the Linux OS using other file formats, such as Python scripts, shell scripts, PERL scripts, PDF files, etc. One can add the support for these file formats after adding respective features and training the model on a significant dataset.

References

1. Inetsim: Internet services simulation suite. https://www.inetsim.org/downloads.html
2. Malware and virus samples. https://www.virussamples.com/
3. Malware statistics by virustotal. https://www.virustotal.com/gui/stats
4. readelf: A tool for accessing elf headers. https://sourceware.org/binutils/docs/binutils/readelf.html
5. Virustotal api responses. https://developers.virustotal.com/v2.0/reference/api-responses
6. The volatility foundation - open source memory forensics. https://www.volatilityfoundation.org/#%21releases/component_7140
7. Linux malware (2022). https://en.wikipedia.org/wiki/Linux_malware#cite_note-Yeargin-2
8. Virusshare (2022). https://virusshare.com/
9. Andrade, C.A.B.D., Mello, C.G.D., Duarte, J.C.: Malware automatic analysis. In: 2013 BRICS Congress on Computational Intelligence and 11th Brazilian Congress on Computational Intelligence, pp. 681–686 (2013). https://doi.org/10.1109/BRICS-CCI-CBIC.2013.119
10. Asmitha, K.A., Vinod, P.: Linux malware detection using non-parametric statistical methods. In: 2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI), pp. 356–361 (2014). https://doi.org/10.1109/ICACCI.2014.6968611
11. Bai, J., Yang, Y., Mu, S.G., Ma, Y.: Malware detection through mining symbol table of Linux executables. Inf. Technol. J. **12**, 380–384 (2013)
12. Dogru, N., Subasi, A.: Traffic accident detection using random forest classifier. In: 2018 15th Learning and Technology Conference (L&T), pp. 40–45. IEEE (2018)
13. Gunnarsdottir, K.M., Gamaldo, C.E., Salas, R.M., Ewen, J.B., Allen, R.P., Sarma, S.V.: A novel sleep stage scoring system: Combining expert-based rules with a decision tree classifier. In: 2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), pp. 3240–3243. IEEE (2018)
14. Maćkiewicz, A., Ratajczak, W.: Principal components analysis (PCA). Comput. Geosci. **19**(3), 303–342 (1993)
15. Mosli, R., Li, R., Yuan, B., Pan, Y.: A behavior-based approach for malware detection. In: Peterson, G., Shenoi, S. (eds.) Advances in Digital Forensics XIII, pp. 187–201. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-67208-3_11
16. Noble, W.S.: What is a support vector machine? Nat. Biotechnol. **24**(12), 1565–1567 (2006)
17. Sihwail, R., Omar, K., Arifin, K.A.Z.: An effective memory analysis for malware detection and classification. Comput. Materi. Continua **67**(2), 2301–2320 (2021). https://doi.org/10.32604/cmc.2021.014510, http://www.techscience.com/cmc/v67n2/41330
18. Rathnayaka, C., Jamdagni, A.: An efficient approach for advanced malware analysis using memory forensic technique. In: 2017 IEEE Trustcom/BigDataSE/ICESS, pp. 1145–1150 (2017)
19. Shahzad, F., Farooq, M.: Elf-miner: using structural knowledge and data mining for detecting Linux malicious executables. Knowl. Inf. Syst. **30**, 589–612 (2012)
20. Shahzad, F., Shahzad, M., Farooq, M.: In-execution dynamic malware analysis and detection by mining information in process control blocks of Linux OS. Inf.

Sci. **231**, 45–63 (2013). https://doi.org/10.1016/j.ins.2011.09.016, https://www.sciencedirect.com/science/article/pii/S0020025511004737
21. Shalaginov, A., Øverlier, L.: A novel study on multinomial classification of x86/x64 Linux elf malware types and families through deep neural networks. In: Malware Analysis using Artificial Intelligence and Deep Learning (2020)
22. Volatilityfoundation: Creation of linux volatility profile. https://github.com/volatilityfoundation/volatility/wiki/Linux#creating-a-new-profile
23. Zhang, Z., Qi, P., Wang, W.: Dynamic malware analysis with feature engineering and feature learning (2019). https://doi.org/10.48550/ARXIV.1907.07352, https://arxiv.org/abs/1907.07352