Learning Fast and Slow: PROPEDEUTICA for Real-time Malware Detection

Ruimin Sun[1*], Xiaoyong Yuan[1*], Pan He[1], Qile Zhu[1], Aokun Chen[1], Andre Gregio[2], Daniela Oliveira[1], and Xiaolin Li[1]

[1]University of Florida, Florida, USA
[2]Federal University of Parana, Paraná, Brazil
{gracesrm, chbrian, pan.he, valder, chenaokun1990}@ufl.edu, gregio@inf.ufpr.br, {daniela, andyli}@ece.ufl.edu
[*]Equal Contribution

***Abstract*—**
In this paper, we introduce and evaluate PROPEDEUTICA[1], a novel methodology and framework for efficient and effective real-time malware detection, leveraging the best of conventional machine learning (ML) and deep learning (DL) algorithms. In PROPEDEUTICA, all software processes in the system start execution subjected to a conventional ML detector for fast classification. If a piece of software receives a borderline classification, it is subjected to further analysis via more performance expensive and more accurate DL methods, via our newly proposed DL algorithm DEEPMALWARE. Further, in an exploratory fashion, we introduce delays to the execution of software subjected to DEEPMALWARE as a way to "buy time" for DL analysis and to rate-limit the impact of possible malware in the system.

We evaluated PROPEDEUTICA with a set of 9,115 malware samples and 877 commonly used benign software samples from various categories for the Windows OS. Our results show that the false positive rate for conventional ML methods can reach 20%, and for modern DL methods it is usually below 6%. However, the classification time for DL can be 100X longer than conventional ML methods. PROPEDEUTICA improved the detection F1-score from 77.54% (conventional ML method) to 90.25% (16.39% increase), and reduced the detection time by 54.86%. Further, the percentage of software subjected to DL analysis was approximately 40% on average. Further, the application of delays in software subjected to ML reduced the detection time by approximately 10%. Finally, we found and discussed a discrepancy between the detection accuracy offline (analysis after all traces are collected) and on-the-fly (analysis in tandem with trace collection). Conventional ML experienced a decrease of 13% in accuracy when executed offline (89.05%) compared to online (77.54%) with the same traces.

Our insights show that conventional ML and modern DL-based malware detectors in isolation cannot meet the needs of efficient and effective malware detection: high accuracy, low false positive rate, and short classification time.

Keywords: Malware Detection, Deep Learning, Machine Learning

I. INTRODUCTION

Malware has been continuously evolving [1]. Existing protection mechanisms do not cope well with the increased sophistication and complexity of modern malware attacks, especially those performed by advanced persistent threats (APTs) [2]. Furthermore, malware campaigns are not homogeneous—malware sophistication varies depending on the target, the type of service exploited as part of the attack (*e.g.*, Internet banking, relationship sites), the attack spreading source (*e.g.*, phishing, drive-by downloads), and the geographic location of the target.

The industry still relies heavily on anti-virus technology for threat detection [3], [4]. While it is effective for malware with known signatures, it is not sustainable given the massive amount of samples released daily, as well as its inefficacy in dealing with zero-day and polymorphic/metamorphic malware (practical detection rates ranging from 25% to 50%) [5], [6]. Confinement-based solutions for running suspicious software, such as Bromium [6] are also suboptimal because they cannot guarantee complete isolation—some types of malware will accomplish their tasks even while confined (*e.g.*, a keylogger can still record an employees credentials).

Behavior-based approaches attempt to identify malware behaviors using instruction sequences [7], [8], computation trace logic [9] and system (or API) call sequences [10]–[12]. These solutions have been mostly based on conventional ML models, such as K-nearest neighbor, SVM, neural networks, and decision tree algorithms [13]–[16]. However, current solutions based on ML still suffer from high false-positive rates, mainly because of (i) the complexity and diversity of current software and malware [1], [11], [17]–[19], which are hard to capture during the learning phase of the algorithms, (ii) sub-optimal feature extraction, (iii) limited training/testing datasets, and the challenge of concept drift [20].

The accuracy of malware classification depends on gaining sufficient context information and extracting meaningful abstraction of behaviors. For system-call/API call malware detection, longer sequences of calls likely contain more information. However, conventional ML-based detectors (*i.e.*, Random Forest [21], Naïve Bayes [22]) often use short windows of system calls during the training process to avoid the curse of dimensionality (when the dimension increases, the classification needs more data to support and becomes harder to solve [23]), and may not be able to extract useful features for accurate detection. Thus, the main drawback of such

[1]In Medicine, propedeutics refers to diagnose a patient condition by first performing initial non-specialized, low-cost exams, and then proceeding to specialized, possibly expensive, diagnostic procedures if preliminary exams are inconclusive.

approaches is that they might lead to many false-positives, since it is hard to analyze complex and longer sequences of malicious behaviors with limited window sizes, especially when malicious and benign behaviors are interposed.

In contrast, DL models [24] are capable of analyzing longer sequences of system calls and making more accurate classification through higher level information extraction. However, DL requires more time to gain enough information for classification and to predict the probability of detection. The trade-off is challenging: fast and not-so-accurate (conventional ML methods) versus time-consuming and accurate classification (emerging DL methods).

In this paper, we introduce and evaluate PROPEDEUTICA, a novel methodology and a proof-of-concept prototype for the Windows OS for efficient and effective on-the-fly malware detection, which combines the best of conventional machine learning (ML) and deep learning (DL) algorithms. In PROPEDEUTICA, all software in the system is subjected to conventional ML for fast classification. If a piece of software receives a borderline malware classification probability, it is then subjected to further analysis via our newly proposed DEEPMALWARE. Further, in an exploratory fashion, PROPEDEUTICA adds delays to the execution of software subjected to DL analysis as a way to "buy time" for DEEPMALWARE to finish analysis and to rate-limit the impact of possible malware in the system while analysis is underway.

The inspiration for our methodology is the practice of propedeutics Medicine. In Medicine, propedeutics refers to diagnose a patient condition by first performing initial non-specialized, low-cost exams or by patient data collection based on observation, palpation, temperature measurement, and proceeding to specialized, possibly expensive and diagnostic procedures only if preliminary exams are inconclusive. In this paper, our proposal is to first attempt to classify ("diagnose") a piece of software ("patient") as malicious ("experiencing a medical condition") using fast conventional ML (simple and non-expensive "diagnostic procedures"). If classification results are borderline ("inconclusive"), the software is subjected to accurate, but more performance expensive DL methods (complex, expensive "diagnostic procedures").

We evaluated PROPEDEUTICA with a set of 9,115 malware samples and 877 common benign software from various categories for the Windows OS. Our results show that for a (configurable) borderline interval of classification probability [30%-70%], approximately 10% of the system software needed to be subjected to DL analysis. In this case, software that were classified as malware by conventional ML with probability less than 30% were considered benign and software classified as malware with probability over 70% were considered malicious. For these 10% of borderline cases, our novel DL algorithm malware achieved an accuracy of 95.54% and false positive rate of 4.10%. Further, we found a discrepancy between detection accuracy offline (analysis after all traces are collected) and on-the-fly (analysis in tandem with trace collection). For example, Random Forest (conventional ML) experienced a decrease of 13% in accuracy when executed offline (89.05%) compared to online (78%) with same traces. This show that real-time detection differs from offline detection as interactions between the system and the detectors are involved, making the malware detection an even more challenging problem. We also found that adding delays to software subjected to DEEPMALWARE decreases the malware detection time in approximately 10% on average. These results corroborate our main hypothesis that conventional ML and modern DL-based malware detectors in isolation cannot meet the needs of high accuracy, low false positive rate, and short detection time of the challenging task of real-time malware detection.

This paper presents the following contributions: (i) We introduce a new methodology for efficient and effective real-time malware detection, (ii) we implement and evaluate this methodology in a proof-of-concept prototype, PROPEDEUTICA, for the Windows OS with a comprehensive collection of malware and benign software and (iii) we introduce a novel DL algorithm, DEEPMALWARE, which specializes in malware classification using enriched features from system calls (not API calls).

The remainder of this paper is organized as follows. In Section II, we describe the threat model that motivates our work. In Section III, we introduce an overview of the architecture design of PROPEDEUTICA. Section IV shows the implementation details of the system call monitoring driver and the HYBRID DETECTOR in our prototype. In Section V, we discuss the comprehensive experiments with offline and on-the-fly analysis on a real-time system. Related work is discussed in Section VI. Finally, Section VII concludes this paper.

II. THREAT MODEL

PROPEDEUTICA's protection is designed for corporate security, in which it is not common to find deep learning specific GPUs available on regular employees' devices. We assume that organizations require on-the-fly malware detection in a timely manner with good accuracy and few false positives, and not to interfere with employee's primary tasks. We also assume that if an organization is a target of a motivated attacker, malware will eventually get in. Further, our trusted computing base includes the Windows OS kernel, the learning models running in user land and the hardware stack.

III. THE ARCHITECTURAL DESIGN OF PROPEDEUTICA

This section provides an overview of the architectural design of PROPEDEUTICA. PROPEDEUTICA comprises three main components (see Figure 1) (i) a system call monitoring driver, (ii) a hybrid malware detector module, and (iii) an interaction module. The system call monitoring driver works in kernel mode and is responsible for intercepting system calls and probabilistically adding delays to the execution of software subjected to DEEPMALWARE. The hybrid malware detector module, called HYBRID DETECTOR, operates in user land and is composed of a system call *reconstruction module*, a conventional ML-detector and our newly proposed DEEPMALWARE

DL-based detector. The interaction module mediates the interactions between HYBRID DETECTOR and the system call monitoring driver.

As Figure 1 shows, when the system starts, system calls invoked by all the processes will be logged into the system call logging queues in user space (Step 1). Each system call will be associated with the `PID` of the invoking process. The HYBRID DETECTOR will follow the tail of the system call logging queue (Step 2), and generate sliding windows of system calls for each `PID` with the help of the *reconstruction module* (Step 3). The conventional ML detector will take in these sliding windows and start classifying the system call traces for each process.

If the ML detector predicts a piece of software S as malware with a classification probability p within the borderline interval range, S will be subjected to further analysis by DEEPMALWARE for a definite classification, and a signal for applying delays to S will be sent to the interaction module (Step 4a). The interaction module will request the system call monitoring driver to apply delays (Step 5a). If the ML detector classifies S as malware with a probability higher than the borderline interval, a kill signal will be sent to the interaction module, and S will be killed (Step 4c and 5b). If the ML detector classifies S as malware with a probability smaller than the borderline interval, the monitoring of Ss execution will continue via the ML detector or a remove delay signal will be sent to the interaction module (Step 4b), which will request the system call monitoring driver to stop applying delays to S's execution (Step 5a).

A. The System Call Monitoring Driver

The goal of the system call monitoring driver is to (i) continuously intercept Windows system calls and record them into logging queues to serve as input to the HYBRID DETECTOR and (ii) apply delays in selected system calls for all software subjected to DEEPMALWARE.

The driver was implemented for Windows 7 SP1 32-bit. PROPEDEUTICA's operation relies on obtaining comprehensive information about processes behavior in the form of system calls. In Windows 7 32-bit system, there are 400 entries of system calls [25]. However, only a subset of them is officially documented by Microsoft. We found that unofficial documentation about system call parameters could be misleading and in some cases lead to Blue Screen Of Death (BSOD). Thus, we decided to collect only information about system calls that would not cause BSOD when intercepted. In total, our driver was able to successfully intercept 155 system calls (listed in the Appendix), including network-related system calls (*e.g.*, `NtListenPort` and `NtAlpcConnectPort`), file-related system calls (*e.g.*, `NtCreateFile` and `NtReadFile`), memory related system calls (*e.g.*, `NtReadVirtualMemory` and `NtWriteVirtualMemory`), process-related system calls (*e.g.*, `NtTerminateProcess` and `NtResumeProcess`), and other types (*e.g.*, `NtOpenSemaphore` and `NtGetPlugPlayEvent`).

To the best of our knowledge, this represents the largest system call set that has been hooked by a driver in the literature [26]–[31]. Our driver is publicly available at [32].

Delay Mechanism In an exploratory fashion we introduced a delay mechanism to PROPEDEUTICA, whose goal is to "buy time" for DEEPMALWARE analysis and rate-limit the actions of potential malware while analysis is underway. Based on the analysis of our malware dataset we identified a set of common malware behaviors and the associated system calls (Table I). It is worth to notice that these system calls alone do not imply malicious behavior, but they can be part of a chain of actions that may lead to a security violation.

TABLE I: Common malware behaviors and examples of system calls invoked to accomplish such behaviors.

Common Malware Behavior	Associated System Calls
Hiding specific processes	NtOpenThread NtOpenProcess NtQuerySystemInformation
Modifying virtual memory	NtReadVirtualMemory NtWriteVirtualMemory
Code Injection	NtDebugActiveProcess NtQueueApcThread NtMapViewOfSection NtSetContextThread
Modifying system files	NtReadFile NtWriteFile
Privilege escalation	NtPrivilegeCheck

We chose 18 system calls (from the 155 intercepted system calls) we found particularly relevant to malware behavior to be subjected to delays when their invoking process was being analyzed by DEEPMALWARE (see Table VIII in the Appendix).

We introduce the following delay strategies:

1) Slowing down the access to critical files: This strategy slows down malware from trojanizing system binaries and infecting other software. It is implemented through the addition of sleep time in system calls returning a file handle related to system file set, *e.g.*, `NtCreateFile`. The system file set is configurable (in this paper we set it to $C:\backslash Windows\backslash$).

2) Slowing down memory accesses: This strategy reduces the probability of malware accessing unshared memory. It is implemented through adding a sleep time in system calls such as `NtCreateSection`.

3) Slowing down the creation of processes/threads: This strategy slows down malware children processes creation. It is implemented through adding sleep to system calls such as `NtCreateProcess` and `NtCreateThread`.

4) Slowing down network connections: This strategy goal mitigates the effect of flooders and is applied to system calls using a network handles, such as `NtListenPort`.

The strength of the delays can be adjusted by a system administrator, through manipulating a `THRESHOLD`. The higher the `THRESHOLD`, the higher probability that a delay will be applied to a system call. In our work, we set the `THRESHOLD` at 10%. This means that whenever a system call subjected to the delay mechanism (see examples in Table I) is invoked, there is a 10% probability that a delay strategy will be applied to it.

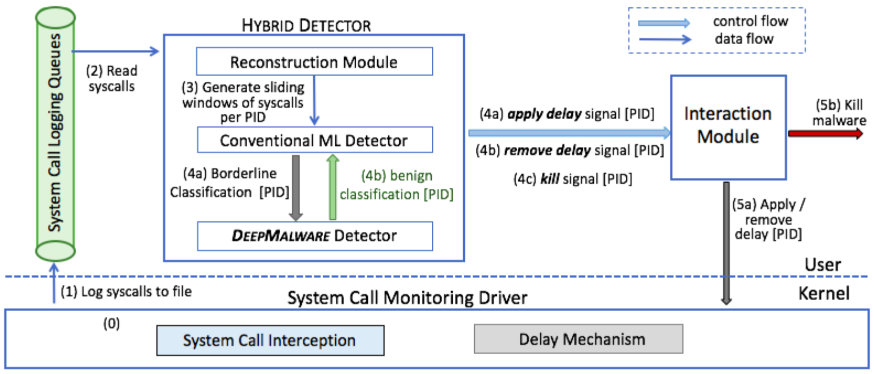

HYBRID DETECTOR
System Call Logging Queues
(2) Read syscalls
Reconstruction Module
(3) Generate sliding windows of syscalls per PID
Conventional ML Detector
(4a) Borderline Classification [PID]
(4b) benign classification [PID]
DEEPMALWARE Detector
control flow
data flow
(4a) apply delay signal [PID]
(4b) remove delay signal [PID]
(4c) kill signal [PID]
Interaction Module
(5b) Kill malware
(5a) Apply / remove delay [PID]
User
Kernel
(1) Log syscalls to file
System Call Monitoring Driver
(0)
System Call Interception
Delay Mechanism

Fig. 1: The workflow of PROPEDEUTICA. The system call monitoring driver will record system call into the system call logging queues. The HYBRID DETECTOR will read system calls from the queues, analyze on the system calls, and signal the interaction module with classification results. The interaction module will take in the signal and carry out the action for the corresponding software.

B. HYBRID DETECTOR

The HYBRID DETECTOR operates in user land and is composed of: (i) a *reconstruction module* to generate sliding windows of system calls to be used as input to the ML and DL classifiers; (ii) a *conventional ML classifier* with a configurable borderline interval, and (iii) our newly proposed *DL classifier*, DEEPMALWARE.

In PROPEDEUTICA, system calls invoked by software in the system are collected and parsed by the *reconstruction module*. The *conventional ML classifier* and DEEPMALWARE share the preprocessed input stream from *reconstruction module*. The *reconstruction module* reconstructs the observed sequence of system calls in a compressed format that is appropriate to be consumed by the learning models (see Section IV-B1 for details).

The *conventional ML classifier* receives as input sliding windows of system calls traces labeled by the PID of the process which invoked them. The ML classifier introduces a configurable borderline probability interval [lower bound, upper bound], which determines the range of classification probability that is considered borderline, *i.e.*, inconclusive. For example, consider a borderline interval in the range [20%-80%]. In this case, if a piece of software receives a "malware" classification probability by the ML classifier, which falls into the range [20%-80%], we consider that the classification for this software is inconclusive. If the software receives a "malware" classification probability of less than 20%, we consider that the software is benign. In contrast, if the software receives a malware classification probability greater than 80%, we consider the software malicious. For the inconclusive case, this piece of software continues to execute in the system, but now it is subject to analysis by DEEPMALWARE for a definite classification. If DEEPMALWARE classification for the software is "malware", the software is killed. Otherwise, the software is considered benign and continues execution being monitored by the conventional ML classifier in the system. Lower and upper bounds are configurable depending on how conservative or lax the administrator wants the detector to be.

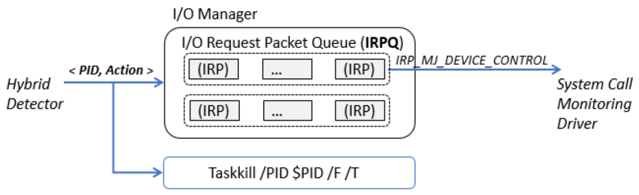


Fig. 2: The architecture of the interaction module. The I/O manager enables the communication between the user land the HYBRID DETECTOR and the kernel land system call monitoring driver. `Taskkill` helps forcefully kill malware (labeled with PID) and all its child processes.

DEEPMALWARE is a multi-stream DL-based malware classification algorithm leveraging CNN, RNN, and their variants to analyze traces of system calls. DEEPMALWARE learns from the multi-stream input, extracts semantic sequences, classifies, and detects the suspicious sequences (see Section IV-B for details).

C. The Interaction Module

The goal of the interaction module is to allow the HYBRID DETECTOR (user land) to communicate with the kernel land system call monitoring driver, and with other user land processes (*e.g.*, to send a kill signal to a piece of malware).

As Figure 2 shows, the HYBRID DETECTOR will send a signal with <*PID, Action*> to the interaction module. The *Action* can apply/remove delay on a process, or simply kill the process. A signal of applying/removing delay will be forwarded to the delay mechanism in the system call monitoring driver. This is implemented by sending I/O request packets (IRP) to the queue of IRP managed by the I/O manager. The I/O manager will associate the packet to a corresponding device, which in our case is created by the system call monitoring driver. The driver uses `IRP_MJ_DEVICE_CONTROL` to read the I/O packets.

A signal of kill will be forwarded to `Taskkill` with the PID information. `Taskkill` will forcefully kill the process and all its child processes.

IV. The Implementation of Propedeutica

In this section, we will show the implementation details of the system call monitoring driver and the Hybrid Detector.

A. *The System Call Monitoring Driver*

The system call monitoring driver is responsible for (i) intercepting Windows system calls and (ii) introducing delays to processes subject to DeepMalware.

There are multiple tools to monitor API or system calls, such as Process Monitor [33], drstrace library [34], Core OS Tool [35], and WinDbg's Logger [36]. Process Monitor and drstrace library collect API calls. However, obfuscated malware will detect the tracing operation and change their behavior accordingly to prevent analysis. WinDbg Logger is mainly designed for debugging and is cumbersome for automated analysis because one has to click a button to save current traces to a file. Core OS Tool leverages ETW (Event Tracing for Windows) and only collect traces for a target software. Thus, while very useful for other contexts, these tools are not sufficient for real-time monitoring and analysis of all processes running on a Windows system. Therefore, we designed and implemented our own system call monitoring driver to monitor system calls and address some issues from the mentioned tools: our driver operates at the kernel level, making it hard for user-level malware to tamper with it or evade interception—most malware today can detect user level tracing and change behavior [37]; our driver is able to perform whole-system system call interception, since that DeepMalware does not analyze processes in isolation—the context of processes' interactions with other processes in the system is considered.

The system call monitoring driver is implemented through hooking the System Service Dispatch Table (SSDT), which contains an array of function pointers to important system service routines, including system call routines. To get access to SSDT, we leverage `ntoskrnl.exe` in Windows 7, which provides the kernel and executive layers of the Windows NT kernel space and at the meantime exports `KeServiceDescriptorTable`. The `KeServiceDescriptorTable` points to the System Service Table (SST), and SST contains a field naming `ServiceTable` pointing to the virtual addresses of SSDT. The SSDT entries are located in read-only memory, so we toggled the WP flag (default 0 for read-only) in the CR0 register to gain write access to the entries.

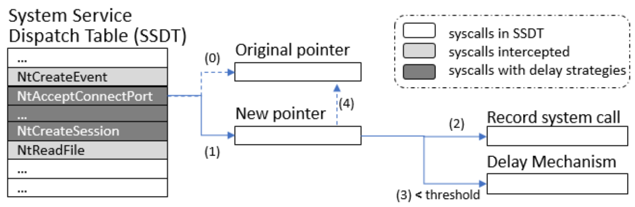


Fig. 3: The process of hooking SSDT structure. When system call `NtAcceptConnectPort` is invoked, we save the original pointer through the dashed arrow (Step 0). Then the system call will be redirected to a new pointer (Step 1) and the invocation will be recorded to file (Step 2). If the current process is identified as suspicious (borderline classification), we apply the delaying strategy with a threshold (Step 3a). Finally, the system call will be returned to the original pointer (Step 4).

As Figure 3 shows, when a system call is invoked, the original function pointer will be saved as the dashed arrow shows (Step 0). Then the system call will be redirected to a new function pointer (Step 1). At this place, the system call will be recorded (Step 2). If the current process in not in the list of processes receiving borderline classification from the conventional ML detector, the invoked system call will be returned to the original pointer directly (Step 4). Otherwise, the system call will go through the delay mechanism with a probability (defined by a `threshold`), sleep for a while (Step 3), and then return to the original pointer (Step 4). In this last case, the driver is not only recording a newly invoked system call to the log file, but also subjecting this system call to the delay mechanism.

We maintain a `Borderline_PIDs` list in the driver to keep a record of processes that get a borderline classification from the conventional ML detector and use kernel function `GetCurrentProcessId()` to get the PID of the current process that invokes the system call. A random float between 0 to 1 will be generated. If the float is greater than the `threshold` (by default 10%), the system call will be returned; otherwise, a sleep with `sleep_time` (by default 50ms) will be added to the call, and then the system call will be returned. PIDs in `Borderline_PIDs` can be added or removed depending on the mediation of the Interaction Module.

System call logs are collected using DbgPrint Logger [38]. DbgPrint Logger enables real-time kernel message logging to a specified IP address (localhost or remote IP)—therefore the logging pool and the hybrid detector can either reside on a PC or a cloud server. For devices having limited computational resources, cloud-based logging and analysis make the detection more efficient and scalable.

The format of system call logging record is <timestamp, PID, syscall>. It contains the time the system call is invoked, the PID of the process, and the system call number (identification).

B. Hybrid Detector *Implementation*

In this section, we describe the implementation of Hybrid Detector, including the *reconstruction module*, the conventional ML detector, and DeepMalware, our newly proposed DL based multi-stream malware detector.

Both the conventional ML and the DeepMalware use the *reconstruction module* to preprocess the input system call sequences and obtain the same preprocessed data. Since considerable amount of research on conventional ML-based malware detection has been done (see Section VI), we only describe DeepMalware in this section.

Figure 4 provides a workflow of our DL-based malware detection approach. System call sequences are taken as input for all processes subjected to DL analysis (those that received a borderline classification from the conventional ML detector). The next subsection explains this workflow in details.

*1) **Reconstruction module**:*

The *reconstruction module* first splits system calls sequences according to the PIDs of processes invoking them. Next, it parses these sequences into three types of sequential data: process-wide n-gram sequence, process-wide density sequence, and system-wide frequency feature vector, explained below. Then, it converts the sequential data into windows using the sliding window method. The sliding window method is usually used to translate a sequential classification problem into a classical classification problem, and also works well with a large amount of data [39].

Process-wide n-gram sequence and density sequence

N-gram model is widely used in problems of natural language processing (NLP). Because of the similarity between sentences and system call sequences, many pieces of previous work have been proposed leveraging n-gram model for malware detection [11], [40]–[42]. We define n-gram as a combination of n contiguous system calls. N-gram model encodes low-level features with simplicity and scalability.

We use the n-gram model to compress sequences. N-gram compresses the data by reducing the length of each sequence with encoded information. The workload of processes is intensive in our model—more than 1,000 system calls can be generated in one second, resulting in very large sliding windows. Such long sequences of intensive system calls not only make it hard to train ML/DL but also consume time for detection. Therefore, we further compress the system call sequences and translate them into two-stream sequences: ***n-gram sequences*** and ***density sequences***. Given the encoded n-gram model sequences, we group the repeated n-gram units and convert them into two sequences. For instance, using 5-gram, we reduce the average sequence length from 52,806 to 4,935 with a compression ratio of 10.7. Thus, *n-gram sequence* is a list of n-gram units, while *density sequence* is the corresponding frequency statistics of repeated n-gram units.

There are many variants such as n-gram, n-tuple, n-bag, and other non-trivial and hierarchical combinations (*i.e.*, "tuples of bags," "bags of bags," and "bags of grams") [11]. We only use 2-gram in the experiments, because (i) n-gram model is considered the most appropriate for such classification problems [11] and (ii) the embedding layer and first few convolutional neural layers (Section IV-B2) can automatically extract high-level features from neighbor grams, which can be seen as hierarchical combinations of multiple n-grams.

Once n-gram sequences fill up a sliding window, the *reconstruction module* delivers the window of sequences to either the ML or DeepMalware and redirects incoming system calls to new n-gram sequences.

System-wide frequency feature vector As we mentioned before, our learning models make use of system calls from all processes running in the system. Our hypothesis is that such holistic (opposite to process-specific) approach will prove more effective for malware detection than current approaches, since modern malware perform interactions among multiple processes in the system to accomplish malicious behaviors [43], [44]. System-wide information helps the models learn the interactions among all processes running on the system. To gain whole system information, the *reconstruction module* collects the frequency of different types of n-grams from all processes during the sliding window and extract them as a frequency feature vector. Each element of the vector represents the frequency of an n-gram unit in the sequence.

To match the n-gram sequence with the produced sliding window, deep learning-based classifier uses the frequency feature vector to represent the system calls invoked during the referred sliding window.

*2) **DeepMalware**:*

DeepMalware is a multi-stream malware classifier in which two types of DL networks are applied: (i) recurrent neural networks (RNN) [45], which can gather broad context information with a sequence model and can achieve state-of-the-art performance in processing sequence data and (ii) convolutional neural networks (CNN) [46], which extract low-level features and allowing DeepMalware to gain strong spatially local correlation without handcrafted feature engineering.

DeepMalware leverages n-gram sequences of processes and frequency feature vectors of the system. First, two streams (process-wide n-gram sequence and density sequence) model the sequence and density of n-gram system calls of the process. The third stream represents the global frequency feature vector of the whole system. DeepMalware consists of four main components, namely *N-gram Embedding*, *(Atrous) Convolutional Layers*, *Long Short-Term Memory (LSTM) Layers*, and *Fully Connected Layers* (Figure 5).

N-gram Embedding. We adopt an encoding scheme called *N-gram Embedding*, which converts sparse n-gram unit into dense representation. After Bengio *et al.* introduced *word embedding* [47], it has been formulated as a foundation module in NLP problems. In DeepMalware, n-gram units are treated as discrete atomic inputs (word). *N-gram Embedding* helps in understanding the relationship between functionally correlated system calls and provides meaningful information of each n-gram to the learning system. Unlike vector representations such as *one-hot vectors*, which may cause data sparsity, *N-gram Embedding* mitigates sparsity and reduces the dimension of input n-gram units.

The embedding layer maps system call sequences to a high-dimensional vector. It helps in the extraction of semantic knowledge from low-level features (system call sequences) and largely reduce feature dimension. In the DL model, 256 neurons are used in the embedding layer, which reduces the dimension of n-gram model from 3,526 (number of unique n-grams in the evaluation) to 256. Because the sequence length (number of n-gram units) in each sample varies, longer samples are truncated and shorter samples are padded with zeros.

(Atrous) Convolutional Layers. Conventional sliding window methods leverage small windows of system call sequences and, therefore, are severe difficulties modeling long sequences.

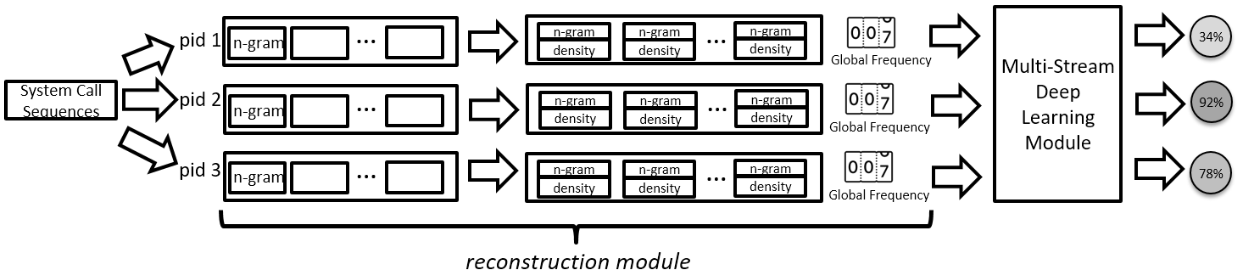


Fig. 4: Workflow of the deep learning based malware detection

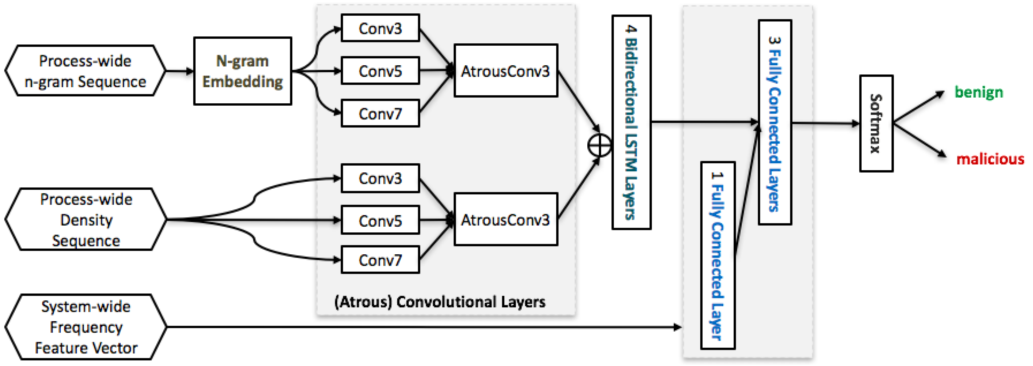

Process-wide n-gram Sequence
N-gram Embedding
Conv3
Conv5
Conv7
AtrousConv3
Process-wide Density Sequence
System-wide Frequency Feature Vector
(Atrous) Convolutional Layers
4 Bidirectional LSTM Layers
1 Fully Connected Layer
3 Fully Connected Layers
Softmax
benign
malicious

Fig. 5: Illustration of DEEPMALWARE. The model consists of four main components—n-gram embedding, (atrous) convolutional layers (Conv3, Conv5, Conv7, and AtrousConv3), bidirectional LSTM layers, and fully connected layers. N-gram embedding is only applied to the process-wide n-gram Sequence. Convolutional layers are inception-like with 1×3, 1×5, 1×7 kernel filters, where all the outputs are concatenated to extract local and global information. Two atrous convolutional layers are followed to enlarge the receptive field. The two streams are combined with element-wise multiplication. Four bidirectional LSTM layers model the global context information. We get the final prediction after three fully connected layers and a softmax activation layer.

A key concern is that these methods represent features in a rather simple way, which may be inadequate for a classification task.

We borrow the inception design from GoogleNet [48], which applies multi-scale convolutional kernel filters on the same inputs in parallel and concatenates the output features. Features are extracted with different receptive fields, and lower level features and higher level features are fused together. This design has been proved to be robust and speed-up large-scale image classification. We use 1×3, 1×5, and 1×7 convolutional layers, and applies padding on each convolutional branch to keep their lengths aligned.

We use the atrous convolutional layer [49] (a variant of convolutional layers) after the inception model. Atrous convolution layer as a dilated convolutional layer allows us to enlarge the field-of-view of filters. It employs spatially small convolution kernels to keep both computation and number of parameters contained without increasing the number of parameters and the amount of computation. Thus, it represents an efficient mechanism to increase the field of view in the convolutional layers. The output of the atrous convolutional layer can be described as:

$$y(i) = \sum_{k=1}^{K} x(i + r \cdot k)w(k) \tag{1}$$

where $y(i)$ denotes the output of Atrous convolutional layer of $x(i)$ with filter $w(k)$ of length K. r is the ratio of dilation.

We deploy batch normalization after convolutional layers to speed up the training process [50] and a non-linear activation function, ReLU to avoid saturation.

Long Short-Term Memory (LSTM) Layers. The internal dependencies between system calls include meaningful context information or unknown patterns for identifying malicious behaviors. To leverage this information, we adopt one of the recurrent architectures, LSTM [45], for its strong capability on learning meaningful temporal/sequential patterns in a sequence and reflecting internal state changes modulated by the recurrent weights. This architecture is proved robust against noises in input sequences.

Our LSTM layers gather information from the first two streams: process-wide n-gram sequence and density sequence.

Fully Connected Layers. A fully connected layer is deployed to encode system-wide frequency. Then, it is concatenated with n-gram three fully-connected layers to gather both sequence-based process information and frequency-based system-wide information. The output of last fully-connected layer is transformed into the probability of a process being malicious, through a softmax activation layer.

V. Evaluation

Our evaluation's goal is to answer the following research questions: (i) is the combination of conventional ML and DL more effective for online malware detection than using such approaches in isolation (*i.e.*, to what extent does our Hybrid Detector outperform ML on accuracy and DL on classification time)? (ii) to what extent the application of delays to software subject to DL helps the detection process?

In this section, we will first describe in details the malware and benign software dataset used for our evaluation. Next, we show and compare the results we obtained regarding malware classification using ML and DL in an offline (post-processing mode, analysis machine), and on-the-fly on a real system.

A. Dataset

For the evaluation, we used Windows malware samples collected since 2013 from a major financial institution incident response team through customer notification of malware in-the-wild, a security mechanism installed on online banking clients, and phishing messages. To ensure the effectiveness of the experiment, other malware in-the-wild are not considered as they may have been outdated or could not function properly. This set of malware samples comprises 9,115 files of PE32 format. Our evaluation also used 7 APTs collected from Rekings [51]. For benign software, we used 877 samples—866 Windows benchmark-based software, 11 GUI-based software, and dozens of system software (*e.g.*, svchost.exe, dllhost.exe, smss.exe), listed in Table II. For the GUI-based software, we used WinAutomation [52] to simulate keyboard and mouse operations.

TABLE II: Categories of benign software used in our evaluation.

Benchmark-based software	**GUI-based software**
Apache Bench [53]	Microsoft Word
NovaBench [54]	Microsoft Office
WinSAT [55]	Windows Media Player
WMIC [56]	Chrome Browser
HCK Test Suite [57]	Calculator
GeekBench [58]	Windows Folder

As we described before, all conventional ML and DL models used sliding windows of Windows OS system calls as features for classification. We did not include system call arguments as the features for the learning models, because of the overhead of collecting system call arguments and the significant increase these arguments would cause in the dimension of the learning models. We collected five datasets of system calls via running five datasets in different combinations:

1) *1M1B*: one malware sample, one benchmark-based software and dozens of system software;
2) *1M1R*: one malware sample, one GUI-based software [2] and dozens of system software;
3) *2M3B*: two malware samples, three benchmark-based software and dozens of system software;
4) *1M1R2*: one malware sample, one GUI-based software and dozens of system software;
5) *APT*: one APT sample, one GUI-based software and dozens of system software.

[2]This includes test cases with daily used benign software, *e.g.*, editing a Microsoft Word file, calculating with Excel, playing music and so on.

In each experiment, we ran malware, benchmark-based/GUI based software and system software based on the combinations from the aforementioned five datasets. During the life cycle of an experiment, only one or two malicious processes will be opened and closed, but more than 200 benign processes will be opened and closed. Thereby we collected system calls from many more benign processes (647,138) than malicious processes (9,115). The datasets are described in detail in Table III.

TABLE III: Description of the five datasets generated for our evaluation. *Size* denotes the size in GB of the formatted system call logs collected in the experiments.

	Number of Experiments	Number of Malware Processes	Number of Benign Processes	Size (GB)
1M1B	780	5,395	195,498	8.38
1M1R	3,322	824	131,360	3.17
2M3R	497	1,149	20,492	0.71
1M1R2	839	886	133,167	3.22
APT	7	8	1,818	0.07
Total	6,249	9,115	647,138	19.66

Each experiment lasted for five minutes to collect system-wide system calls. Some malware samples, however, caused blue screen of death (BSOD) before the five minutes had elapsed. Thus, we set a minimum running time of one minute for a malware sample to be included in our analysis. We use the *1M1B*, *1M1R*, and *2M3B* dataset for training and validation. We use the *1M1R2* and *APT* dataset for testing on different settings and different malware. (In the future, we will test malware based on different malware families.)

Imbalanced datasets (in which the number of malware and benign processes are not equally represented) may adversely affect the training process, because the ML classifier is prone to learn from the majority class [59]. We under-sampled the data by reducing the number of benign sliding window samples the same as the number of malware ones. We applied sliding window to analyze traces and tested feasible combinations of window sizes and strides. Window size and stride are two important hyper-parameters in sliding window methods indicating the total and the shifting number of n-gram system calls in each process. Large window sizes provide more context for DL models. In our experiments, we selected and compared three window size and stride pairs: (100, 50), (200, 100), (500, 250). Then we divided dataset *1M1B*, *1M1R*, and *2M3B* into training and validation dataset with ratio $9:1$.

Table IV shows a comparison among existing datasets based on Windows API or system calls for malware detection. The datasets we generated in this paper are larger and more up-to-date than existing ones. We included not only benchmark software, but also GUI-based software which are challenging to configure and automate because of the need to simulate user activity.

Another dataset not listed in Table IV (because it is not public anymore) is Anubis [31], which collected hybrid traces from benign and malicious behaviors and used to play a major role in many research projects.

TABLE IV: Comparison among existing datasets with PROPEDEUTICA.

Dataset Name	Year	Number of Malware Processes	Number of Benign Software Processes
ADFA-IDS [26]	2014	855	0
CSDMC [27]	2010	320	68
Nitro [28]	2011	1943	285
SPADE [29]	2011	640	570
Xiao and Stibor [30]	2010	2176	161
PROPEDEUTICA	2017	**9115**	**647138**

B. Offline Post-processing of Traces: ML, DL, and the DEEPMALWARE

In this section, we compare the performance of conventional ML (in isolation), DL (in isolation), and the HYBRID DETECTOR when the system call traces from malware and benign software are completely collected before being analyzed by the models—offline post-processing. We compared the three approaches separately and considered the values of accuracy, precision, recall, f1 score, false positive rate, detection time.

For conventional ML, we considered the following algorithms: *Random Forest (RF)* [21], *eXtreme Gradient Boosting (XGBoost)* [60], and *Adaptive Boosting (AdaBoost)* [61]. Random Forest, neural networks, and boosted trees have been shown to be the best three supervised models on high-dimensional data [62]. We used AdaBoost and XGBoost, a new fast gradient based boosting algorithm, as the representatives of boosted trees. The input features of these algorithms are the frequency of n-gram in a sliding window of system calls belonging to a process.

For DL, we considered the following three models: DEEPMALWARE (our newly proposed model), and two DL control models — *CNNLSTM*, *LSTM*. Compared to DEEPMALWARE (see Section IV-B), *CNNLSTM* and *LSTM* have a similar architecture, but do not have dilated and convolutional layers respectively. The architecture of CNNLSTM and LSTM models are depicted in Appendix A.

The HYBRID DETECTOR used Random Forest for conventional ML classification and DEEPMALWARE DL classification. As described later in this section, we chose these algorithms because they produced the best overall performance when executed in isolation (see Table V for details).

We use scikit-learn [63] and PyTorch [64] to implement all conventional ML and DL algorithms. The training and testing processes ran on an Ubuntu 14.04 Server with four Nvidia Tesla K80 GPUs and 16 Intel(R) Xeon(R) E5-2667 CPU cores.

We combined dataset *1M1B*, *1M1R*, and *2M3B* dataset as the training/validation dataset in the offline post-processing experiments. The testing dataset was Dataset *1M1R2*. During the testing process, we randomly choose traces from 856 malware and 856 benign software respectively (about 10% of the whole datasets) and tested the models for 10 times for a more accurate average result.

Table V shows the results we obtained by comparing conventional ML and DL algorithms on Dataset *1M1R2* with offline post-processing analysis. We measured the accuracy, precision, recall, F1 score, and false positive (fp) rate as the metrics of model performance. We also compared the detection time for each algorithm using CPU and GPU. GPU devices not only reduce the training time for DL models, but also help DL models achieve classification time one order of magnitude less than compared to conventional CPUs. We did not apply GPU devices to ML models because their classification time with conventional CPUs is already much smaller (at least one order of magnitude) than those measured for DL algorithms. We denote 'N/A' as not applicable in Table V. In real life, GPU devices, especially specific DL GPUs for accelerating DL training/testing are not widespread in end-user or corporate devices. Thus, anti-malware solutions for such public currently does not rely on GPUs.

As shown in Table V, the performance of DEEPMALWARE model improves with the increase of window size and stride. **The best deep learning model is DEEPMALWARE with window size 500 and stride 250 (97.03% accuracy, 97.02%, and 2.43% false positive rate). The best conventional ML model is Random Forest with window size 500 and stride 250.** The DEEPMALWARE model outperforms Random Forest by 3.09% (accuracy) and 3.29% (F1 score) with window size 500 and stride 250. Large window size and stride provide more context for DL to analyze at a time. However, the detection time of DEEPMALWARE is higher than that of Random Forest. On average, it is approximately 100 times slower in conventional CPUs than conventional ML algorithms. Even in GPU devices, DEEPMALWARE can be 3 to 5 times slower on average than conventional ML algorithms.

Next, we evaluate the performance of the HYBRID DETECTOR for different borderline intervals, but still leveraging offline processing of system call traces.

Based on the results from Table V, we use the overall best DL model (DEEPMALWARE) and the best overall conventional ML model (Random Forest) and evaluate them using various borderline intervals. However, on-the-fly processing brings extra workloads (see details in Section V-D), where PROPEDEUTICA takes more time to fill up a sliding window compared to offline post-processing. Hence, we used a smaller sliding window (window size 100 and stride size 50) in this evaluation. If a piece of software receives a malware classification probability from Random Forest that is smaller than the lower bound or greater than the upper bound, it is considered benign software or malware respectively. If its malware probability falls within the borderline interval, the software is subjected to DEEPMALWARE analysis.

Table VI shows the performance of the HYBRID DETECTOR for various borderline intervals. We chose lower bound as 10%, 20%, 30%, and 40%, upper bound as 60%, 70%, 80%, and 90%. With a borderline interval of [40%-60%], the hybrid approach moved approximately 20% of the samples for DL analysis. In other words, 20% of samples (malware and benign software) received a classification score between 40-60% with Random Forest. For these samples, DEEPMALWARE performed with almost 90% accuracy and less than 6% of false positive rates. This highlights the potential of the PROPEDEUTICA paradigm: 80% of the samples were quickly classified with high accuracy as malicious or benign using an initial triage with fast conventional ML. Only 20% of

TABLE V: Comparison between conventional ML and DL models in isolation and in offline analysis

Model		Window Size	Stride	Accuracy	Precision	Recall	F1 score	FP Rate	Analyzing Time with GPU (s)	Analyzing Time with CPU (s)
ML	AdaBoost	100	50	79.25%	78.11%	81.31%	79.67%	22.80%	N/A	0.0187
	Random Forest	**100**	**50**	**89.05%**	**84.63%**	**95.44%**	**89.71%**	**17.35%**	**N/A**	**0.0089**
	XGBoost	100	50	84.78%	90.99%	77.22%	83.54%	7.66%	N/A	0.0116
DL	CNNLSTM	100	50	94.82%	92.98%	96.96%	94.93%	7.32%	0.0410	1.5533
	LSTM	100	50	94.10%	90.91%	98.01%	94.32%	9.81%	0.0098	2.8076
	DEEPMALWARE	**100**	**50**	**94.84%**	**92.63%**	**97.43%**	**94.97%**	**7.76%**	**0.0383**	**1.4104**
ML	AdaBoost	200	100	78.27%	72.84%	90.19%	80.59%	33.64%	N/A	0.0132
	Random Forest	**200**	**100**	**93.49%**	**89.99%**	**97.90%**	**93.77%**	**10.91%**	**N/A**	**0.0063**
	XGBoost	200	100	70.81%	63.65%	97.08%	76.89%	70.81%	N/A	0.0073
DL	CNNLSTM	200	100	95.78%	93.54%	98.36%	95.89%	6.80%	0.0492	1.4332
	LSTM	200	100	94.86%	92.96%	97.08%	94.97%	7.36%	0.0947	2.6715
	DEEPMALWARE	**200**	**100**	**94.96%**	**93.05%**	**97.20%**	**95.08%**	**7.27%**	**0.0543**	**1.3784**
ML	AdaBoost	500	250	81.81%	79.44%	85.86%	82.52%	22.24%	N/A	0.0108
	Random Forest	**500**	**250**	**93.94%**	**97.16%**	**90.54%**	**93.73%**	**2.65%**	**N/A**	**0.0048**
	XGBoost	500	250	79.19%	94.14%	62.27%	74.95%	3.88%	N/A	0.0062
DL	CNNLSTM	500	250	95.33%	97.20%	93.34%	95.23%	2.69%	0.0902	1.8908
	LSTM	500	250	95.81%	97.38%	94.16%	95.74%	2.54%	0.1529	3.5626
	DEEPMALWARE	**500**	**250**	**97.03%**	**97.54%**	**96.50%**	**97.02%**	**2.43%**	**0.0797**	**1.3344**

TABLE VI: Comparison on different borderline policies for the HYBRID DETECTOR in offline analysis with window size 100 and stride 50. Borderline intervals are described with a lower bound and an upper bound. The move percentage represents the percentage of software in the system that received a borderline classification with Random Forest (according to the borderline interval) and was subjected to further analysis with DEEPMALWARE.

Lower Bound	Upper Bound	Accuracy	Precision	Recall	F1	FP Rate	Analyzing Time with GPU (s)	Analyzing Time with CPU (s)	Move Percentage
10%	90%	94.71%	92.41%	97.43%	94.85%	8.00%	0.0223	0.3810	55.95%
20%	80%	94.61%	92.23%	97.43%	94.76%	8.21%	0.0173	0.1543	48.32%
30%	70%	94.34%	91.77%	97.43%	94.51%	8.75%	0.0146	0.0884	41.45%
40%	60%	93.72%	90.79%	97.37%	93.96%	9.92%	0.0123	0.0497	34.96%
20%	90%	94.00%	91.25%	97.38%	94.21%	9.38%	0.0196	0.2095	37.73%
20%	70%	93.93%	91.11%	97.41%	94.14%	9.56%	0.0168	0.1244	37.43%
20%	60%	93.59%	90.54%	97.43%	93.85%	10.25%	0.0156	0.1021	36.61%
30%	90%	93.81%	90.93%	97.42%	94.05%	9.79%	0.0175	0.1506	37.10%
30%	80%	93.92%	91.12%	97.40%	94.14%	9.57%	0.0157	0.1075	36.58%
30%	60%	93.71%	90.76%	97.41%	93.95%	9.99%	0.0139	0.0735	35.27%
40%	90%	93.86%	91.06%	97.36%	94.09%	9.64%	0.0152	0.1140	35.03%
40%	80%	93.93%	91.22%	97.31%	94.15%	9.45%	0.0137	0.0778	34.15%
40%	70%	93.92%	91.22%	97.29%	94.14%	9.44%	0.0127	0.0616	33.04%

the samples needed to be subjected to a more expensive analysis using DL. Please notice that the borderline interval is a configurable parameter of the system. Depending on the user or organization more or less conservative intervals can be used.

We also evaluated the performance of the HYBRID DETECTOR on 7 APTs with various borderline intervals. The HYBRID DETECTOR successfully detected all the malware with all borderline intervals. There existed at most one false positives (false positive rate about 10%) in all the configurations. However, Random Forest detector (in isolation) performed a much higher false positive rate (52.5%).

C. On-the-Fly Processing of Traces: HYBRID DETECTOR *with and without Delay Strategies*

In this section, we show the experiments we carried out for malware classification using the HYBRID DETECTOR on-the-fly and with delay strategies for software subjected to DL. Our definition of on-the-fly is an analysis that is performed *while the system is executing*, contrasting with the offline analysis, in which the system call traces are first logged and later analyzed.

We compare the performance of the HYBRID DETECTOR with offline post-processing and on-the-fly processing, try to explain the reasons behind the different performance results we found in these two settings, and discuss ways our methodology, PROPEDEUTICA, can be applied in practice.

Table VII compares the performance of conventional ML models and the HYBRID DETECTOR (with and without delay strategies) when running on-the-fly experiments. Each experiment ran with the same configuration, inputs, and parameters as those shown for Table VI. Similarly to the offline results, the HYBRID DETECTOR (for various borderline intervals) outperforms conventional ML models, with lower false positive rate and higher F1-score. However, three aspects diverge between online and offline results: 1) the offline results are slightly better than on-the-fly results in terms of performance (accuracy, precision, f1, fp rate, etc.). 2) the HYBRID DETECTOR's performance was not sensitive to the type of borderlines intervals applied. 3) the percentage of software (malicious and benign) that is subjected to DL analysis (*i.e.*, move percentages) increases in the on-the-fly results (*e.g.*, move percentage changes from 34% to 44% in the borderline interval [40%, 60%]). One reason for these differences could be the real-time interactions, among the ML and DL detectors, and the system processes, such as the change of environments and the application of the delay. We hypothesize that these interactions in the case of on-the-fly analysis increase the overhead of the system, in which more processes are opened and closed during each experiment, causing a slow-down to the execution of the offline test cases. To test this hypothesis, we measured the total number of processes monitored for each experiment and found that there was approximately an 50% increase in the number of processes when the detection was performed online. Another result that corroborates our hypothesis is that in the on-the-fly experiments, the system call monitoring driver collects fewer system calls than in the offline experiments. In 5-minute experiments, many processes cannot fill up one sliding window (the total number of collected n-gram system calls is less than the window size). This brings incomplete data to DEEPMALWARE—an obstacle to the classification task.

We also carried out experiments running longer time intervals (10 minutes) (see Table VII). In the 10-minute experiments, the processes have more time to fill up the sliding windows, providing DEEPMALWARE with sufficient information for classification, thus performing better than the 5-minute experiments. Table VII shows that the overall performance of the HYBRID DETECTOR in 10-minute runs increases compared with that in 5-minute runs, and gets closer to the offline performance. Our hypothesis is corroborated with the increase in accuracy, precision, recall, and F1. Further, with longer running times, broader borderlines perform better than narrow ones.

The Effect of Delaying Strategies As described before, in an exploratory fashion, our DEEPMALWARE prototype adds random and probabilistic delays to selected system calls of process subjected to DEEPMALWARE. Table VII shows the effect of delay strategies with sleep time 50 ms and `Threshold` 10%. Within each borderline interval, the results for accuracy, precision, recall, fp rate, F1 and move percentage are similar to those without delay strategies. Detection time, however, decreased over 10% when we applied delay. By analyzing our results, we discovered that benign software were affected more by the delaying strategies than malware. An explanation could be that the rate at which benign software usually invoke system calls is higher than that of malware. At the same threshold (probability of applying a strategy on a system call), adding delay will slow down benign processes more than malicious processes, and hence fewer system calls are queued for DEEPMALWARE analysis. To detect the malware, DEEPMALWARE needs the same amount of traces from malware in each experiment. Everything happens as if malware traces are consumed faster.

D. Summary and Discussions

Real-world malware detection is challenging — on the one hand, mission critical software should not be mistakenly killed by a malware detector. On the other hand, the detector should not risk allowing possible malware running in a system, especially in the perimeter of an organization. Conventional ML and modern DL-based malware detectors in isolation cannot meet the needs of high accuracy, low false-positive rate, and short detection time. Our results show that the false-positive rate for conventional ML method can reach 20%, and for modern DL methods is usually below 6%. However, the computation time for DL can be 100x longer than ML on a conventional CPU.

The HYBRID DETECTOR leverages the fast speed of conventional ML and the high accuracy of modern DL. In this work, only software receiving borderline classification from ML detector needed further analysis by a DL detector, which saved computational resources and shortened the detection time without loss of accuracy. For the on-the-fly experiments, the HYBRID DETECTOR improved the detection F1-score from 77.54% (conventional ML method) to 90.25% (12.71% increase), and reduced the detection time by 54.86%. Furthermore, the amount of software that were subjected to DL analysis was just a fraction (approximately 32% on average for online detection) of the software running in the system, as conventional ML was able to provide a high probability for part of the clear-cut cases.

Despite the good performance of the DEEPMALWARE, there are still challenges to be faced when applying this type of detection model in a real system. On-the-fly analysis brings extra workloads and more computational pressure, which is introduced by the interactions between the HYBRID DETECTOR, its supporting system, and the processes opened. Also, traces from one process may not be able to fill up one sliding window sometimes. One possible solution could be to apply a smaller window size and stride in ML detector to conduct fast detecting and to use big window size in DEEPMALWARE to gain enough context information. We compare the accuracy between offline and on-the-fly analysis based on Random Forest model and HYBRID DETECTOR without delay strategies with window size 100 and stride 50 (Figure 6). On-the-fly analysis did much less impact on HYBRID DETECTOR than that did on Random Forest model, specially when we use a longer time interval in the experiments.

Delaying system calls from processes subjected to DL analysis have the potential to gain time for DL detectors to classify malware correctly. Since the rate at which benign software usually invokes system calls is usually higher than that of malware, delay strategies affected benign software more than malware in the DL environment. As future work, we plan to analyze the effect of different thresholds and sleeping time for delay strategies.

The worst-case scenario for software in PROPEDEUTICA would be looped between the ML and DEEPMALWARE. For example, it could receive a borderline classification from the ML detector, and then be moved to DEEPMALWARE. Then DEEPMALWARE would classify it as benign, and the software would continue being analyzed by the ML, which would again provide a borderline classification, and so on.

As we discussed in Section II, a resourceful and motivated adversary can bypass any protection mechanism. Even though PROPEDEUTICA demonstrated itself a promising paradigm for

TABLE VII: Comparison between conventional ML and DEEPMALWARE for on-the-fly detection. Notice that the detection time may be longer than the experiment length, as in this case the system is still running without collecting further traces. All these experiments leveraged a conventional non-GPU machine.

Experiment Length	Model	Upper Bound	Lower Bound	Accuracy	Precision	Recall	F1	FP rate	Detection Time (s)
5 minutes	Random Forrest	N/A		78.00%	79.18%	75.79%	77.54%	20%	255.88
5 minutes	HYBRID DETECTOR (no delay strategies)	20%	80%	86.13%	83.64%	80.00%	87.47%	8%	615.87
		30%	70%	86.37%	84.77%	78.67%	87.88%	6%	353.26
		40%	60%	87.83%	84.65%	81.85%	90.25%	6%	270.84
10 minutes	HYBRID DETECTOR (no delay strategies)	20%	80%	91.96%	89.05%	95.83%	91.83%	12%	682.00
		30%	70%	86.94%	84.78%	84.77%	88.10%	11%	536.56
		40%	60%	88.75%	83.42%	87.98%	91.54%	10%	447.73
10 minutes	HYBRID DETECTOR (sleep time=50 ms threshold=10%)	20%	80%	86.61%	84.20%	83.81%	87.49%	10%	555.53
		30%	70%	87.84%	82.32%	86.46%	91.22%	11%	501.02
		40%	60%	88.67%	86.01%	87.30%	92.85%	10%	403.49

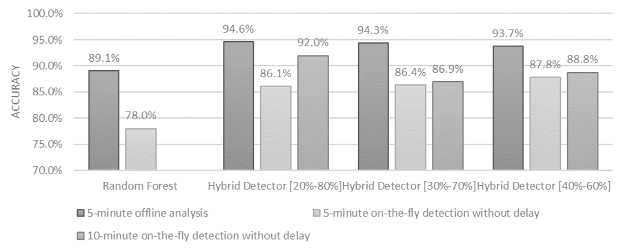


Fig. 6: Accuracy comparison between offline and on-the-fly analysis based on Random Forest, HYBRID DETECTOR without delays.

on-the-fly practical malware detection, a highly sophisticated malware could still evade detection. Further, PROPEDEUTICA relies on the integrity of the OS for correct operation, and attacks compromising the OS could directly compromise any of PROPEDEUTICA's components.

VI. RELATED WORK

Our work intersects the areas of behavior-based malware detection with machine-learning and deep-learning. This section summarizes the state-of-the-art in these areas and highlights topics currently under-studied.

Behavior-based Malware Detection. Most of the work on dynamic behavior-based malware detection [7], [8], [17], [65] evolved from Forrest *et al.*'s seminal work [66] on detecting attacks through system calls.

Christodorescu *et al.* [7], [8] extract high-level and unique behavior from the disassembled binary to detect malware and its variants. The detection is based on predefined templates covering potentially malicious behaviors, such as mass-mailing and unpacking. Willems *et al.* proposed CWSandbox, a dynamic analysis system that runs malware into a virtual environment and monitor its API calls [67]. Rieck *et al.* [68] processed these API calls and used them as features to separate malware into families using Support Vector Machines (SVM). Rieck *et al.* [16] uses a representation for system calls to generate malware behavioral traces (transformed in q-grams), which will be grouped into clusters with an algorithm similar to the k-nearest neighbors (KNN). Mohaisen *et al.* [69] introduce AMAL, a framework for dynamic analysis and classification of malware using SVM, linear regression, classification trees, and KNN. Kolosnjaji *et al.* [70] propose the use of maximum-a-posteriori (MAP) to classify malware samples into families through their behavioral extracted from dynamic analysis with Cuckoo's Sandbox [71]. Although these techniques were successfully applied to classifying malware, as the seminal work from Bailey *et al.* [72], they do not consider benign samples. Thus, they are limited to label an unknown malware as pertaining to one of the existing clusters.

Xiao and Stibor used system calls to distinguish among harmless programs, network-based malware (email, IM, IRC, net) and system-based malware (backdoors, Trojans, others) [30]. They combined several techniques, such as 1- and 2-grams, supervised topic transition (STT), SVM and others in a dataset of 3,048 programs, accomplishing accuracy rates ranging from 32 to 63%. Wressnegger *et al.* [73] propose Gordon, a detection method for Flash-based malware. Gordon uses information from static analysis and the execution behavior of benign and malicious Flash applications to produce *n-grams*, which are used as input for an SVM. Gordon was evaluated with 26,000 samples and detected from 90 to 95% of Flash-based malware.

Bayer *et al.* monitor the software behavior through API call hooking and breakpoints using a modified version of QEMU [12]. This approach was used to build Anubis [31], which leverages sandboxing techniques to analyze malware (*e.g.*, through system calls invoked). These system calls are then used to clusterize malware based on behavior similarity [14]. Kirat *et al.* introduce BareBox, a dynamic analysis system based on hooking system calls directly from SSDT [74]. Barebox is able to run in a bare metal system, i.e., outside a virtual machine or emulator, thus potentially obtaining behavioral traces from malware equipped with anti-analysis techniques. PROPEDEUTICA also uses system call hooking techniques to monitor software behavior, and has a testbed to run malware in a large scale automatically. Anubis works best for advanced malware analyzers who can read the detailed execution report while PROPEDEUTICA delivers on-thy-fly detection result to end users. PROPEDEUTICA improves the experiment of Anubis by running longer time (10 minutes) and using cutting-edge DL models to help on the detection.

PROPEDEUTICA is also inspired by many system-wide monitoring research on malware detection, such as VMScope [75], TTAnalyze [12], and Panorama [43]. Emulated with Qemu, they use tainting techniques to trace the data-flow and intercept malicious behavior from whole-system processes. PROPEDEUTICA differs by monitoring software behavior through system-wide system call hooking instead of data tainting, and maintains the interactions among different processes in a lightweight way. Some researchers have shown that malware have developed strong obfuscation capabilities to evade detection [76]–[78], and suggested detecting tools to avoid high level (user level) monitoring which may inform the malware of the existence of a tracing tool. Agreeably, the monitoring driver and the delay mechanism in PROPEDEUTICA are both in kernel space. Canali *et al.* presented the closest environment with PROPEDEUTICA [11]. The goal of their work is to evaluate different types of models in malware detection. Their findings confirm that *the accuracy of some widely used models is very poor, independently of the values of their parameters*, and *few, high-level atoms with arguments model is the best one*, which corroborates our assumption.

Even though a lot of work has been done on behavior-based malware classification/detection, on-the-fly detection still suffers from high false-positive rates due to the diversity of applications and the diverse nature of system calls invoked [17].

ML-based Malware Detection: Xie *et al.* proposed a one-class SVM model with different kernel functions [79] to classify anomalous system calls in the ADFA-LD dataset [26]. Ye *et al.* proposed a semi-parametric classification model for combining file content and file relation information to improve the performance of file sample classification [80]. Abed *et al.* used bag of system calls to detect malicious applications in Linux containers [81]. Kumar *et al.* used K-means clustering [82] to differentiate legitimate and malicious behaviors based on the NSL-KDD dataset. Fan *et al.* utilized an effective sequence mining algorithm to discover malicious sequential patterns and trained an All-Nearest-Neighbor (ANN) classifier based on these discovered patterns for the malware detection [83].

The Random Forest algorithm has been applied to classification problems as diverse as offensive tweets, malware detection, de-anonymization, suspicious Web pages, and unsolicited email messages [84]–[90]. ML-based malware detectors suffer, however, from high false-positive rates because of the diverse nature of system calls invoked by applications, as well as the diversity of applications [17].

DL-based Malware Detection: There are recent efforts to apply deep learning for malware detection with the advances in deep learning and big data analytics.

Pascanu *et al.* first applied deep neural networks (recurrent neural networks and echo state networks) to modeling the sequential behaviors of malware. They collected API call sequences of the operating system and C run-time library and detected malware as a binary classification problem [91]. David *et al.* [92] used deep belief network with denoising autoencoders to automatically generate malware signature and classified malware based these signatures. Saxe and Berlin [93] proposed a DL-based malware detection technique with two dimensional binary program features. They also provided a Bayesian model to calibrate detection. Hou *et al.* collected the system calls from kernel and then constructed the weighted directed graphs and use DL framework to make dimension reduction [94].

Recently, Kolosnjaji *et al.* [42] proposed a DL method to detect and predict malware family based only on system call sequences. The neural network architecture proposed by them is similar to DEEPMALWARE. However, Their neural networks do not use atrous convolutional layers, inception design, and bidirectional recurrent neural networks, which have been proved to increase the performance of detection in our experiments. To deal with fast evolving of malware, [95] recently proposed a framework called Transcend to detect concept drift in malware. Transcend can detect the aging machine learning models before their degradation.

Although there has been several proposals for DL-based malware detection, most of them focus on modeling the malicious behavior in an offline manner. Our work also pays attention to the performance of DL-based malware detection algorithms running in a real system.

Despite initial successes on malware classification, recent work has demonstrated that DL is vulnerable to misclassification by well-designed data samples, called *adversarial examples* [96], [97]. Papernot *et al.* tried network distillation to defend adversarial examples [98], but it is still vulnerable to some strong attacks (*e.g.*, C&W attack [96]). Wang *et al.* proposed a robust DL technique with randomly nullifying features for malware detection to resist adversarial examples [99]. As future work, we plan to apply such defending method to our DEEPMALWARE.

VII. CONCLUSIONS

In this paper, we introduced PROPEDEUTICA, a novel paradigm and proof-of-concept prototype for malware detection combines the best of conventional machine learning and emerging deep learning techniques. Our paradigm is inspired by the practice of propedeutics in Medicine. Propedeutics refers to diagnose a patient condition by first performing initial non-specialized, low-cost exams then proceeding to specialized, possibly expensive, diagnostic procedures if preliminary exams are inconclusive.

The main idea proposed was that all software processes in the system start execution subjected to a conventional ML detector for fast classification. If a piece of software receives a borderline classification, it is subjected to further analysis via more performance expensive and more accurate DL methods, via our newly proposed DL algorithm DEEPMALWARE. We also evaluated whether adding delays to processes during deep learning analysis would help the classification accuracy. Further, in a exploratory fashion, we introduce delays to the execution of software subjected to DEEPMALWARE as a way to "buy time" for DL analysis and to rate-limit the impact of possible malware in the system. Our results showed that such paradigm is promising as it showed better performance (in all standard machine learning metrics) than conventional machine learning and deep learning in isolation. We also discussed the different performance results (and possible causes) for malware classification performed online (decreased performance) and offline.

In sum, our work provided evidence that conventional machine learning and emerging deep learning methods in isolation are not enough. PROPEDEUTICA, by combining the best capabilities of such methods intelligently has the potential to transform the next generation of practical on-the-fly malware detection.

REFERENCES

[1] A. Calleja, J. Tapiador, and J. Caballero, "A look into 30 years of malware development from a software metrics perspective," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016, pp. 325–345.

[2] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 377–396.

[3] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "AVClass: A Tool for Massive Malware Labeling," in *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses*, Evry, France, September 2016.

[4] G. Vigna and R. A. Kemmerer, "NetSTAT: A Network-Based Intrusion Detection Approach," ser. ACSAC '98, 1998.

[5] L. Bilge and T. Dumitras, "Before we knew it: an empirical study of zero-day attacks in the real world," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 833–844.

[6] "Bromium end point protection," https://www.bromium.com/, 2010.

[7] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, ser. SP '05, 2005.

[8] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07, 2007, pp. 5–14.

[9] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Detecting malicious code by model checking," *Detection of Intrusions and Malware, and Vulnerability Assessment*, vol. 3548, pp. 174–187, 2005.

[10] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and efficient malware detection at the end host," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09, 2009, pp. 351–366.

[11] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "A quantitative study of accuracy in system call-based malware detection," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012, 2012, pp. 122–132.

[12] U. Bayer, C. Kruegel, and E. Kirda, *TTAnalyze: A tool for analyzing malware*. na, 2006.

[13] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of computer security*, vol. 6, no. 3, pp. 151–180, 1998.

[14] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering." in *NDSS*, vol. 9. Citeseer, 2009, pp. 8–11.

[15] S. Revathi and A. Malathi, "A detailed analysis on nsl-kdd dataset using various machine learning techniques for intrusion detection," *International Journal of Engineering Research and Technology. ESRSA Publications*, 2013.

[16] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," *Journal of Computer Security*, vol. 19, no. 4, pp. 639–668, 2011.

[17] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "Accessminer: using system-centric models for malware protection," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 399–412.

[18] "Modern malware exposed," http://www.nle.com/literature/FireEye_modern_malware_exposed.pdf, 2009.

[19] "The modern malware review," https://media.paloaltonetworks.com/documents/The-Modern-Malware-Review-March-2013.pdf, March 2013.

[20] G. Widmer and M. Kubat, "Learning in the presence of concept drift and hidden contexts," *Machine learning*, vol. 23, no. 1, pp. 69–101, 1996.

[21] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[22] H. Zhang, "The optimality of naive bayes," *AA*, vol. 1, no. 2, p. 3, 2004.

[23] Y. Bengio, Y. LeCun *et al.*, "Scaling learning algorithms towards ai," *Large-scale kernel machines*, vol. 34, no. 5, pp. 1–41, 2007.

[24] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[25] "NTAPI," http://j00ru.vexillium.org/syscalls/nt/32/.

[26] G. Creech and J. Hu, "Generation of a new ids test dataset: Time to retire the kdd collection," in *2013 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2013, pp. 4487–4492.

[27] "CSDMC," http://csmining.org/index.php/malicious-software-datasets-.html, 2010.

[28] J. Pfoh, C. Schneider, and C. Eckert, *Nitro: Hardware-Based System Call Tracing for Virtual Machines*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[29] B. Rozenberg, E. Gudes, Y. Elovici, and Y. Fledel, "Newapproach for detecting unknown malicious executables. j forensic res 1: 112. doi: 10.4172/2157-7145.10001 12," 2010.

[30] H. Xiao and T. Stibor, "A supervised topic transition model for detecting malicious system call sequences," in *Proceedings of the 2011 Workshop on Knowledge Discovery, Modeling and Simulation*, ser. KDMS '11. New York, NY, USA: ACM, 2011, pp. 23–30. [Online]. Available: http://doi.acm.org/10.1145/2023568.2023577

[31] "Anubis dataset," http://anubis.iseclab.org/, 2010.

[32] "Propedeutica Driver Publicly available at <blinded for anonymity >."

[33] "Process monitor v3.40," https://technet.microsoft.com/en-us/sysinternals/bb896645.aspx, 2017.

[34] "drstrace," http://drmemory.org/strace_for_windows.html, 2017.

[35] "Core os tools," https://msdn.microsoft.com/en-us/magazine/ee412263.aspx, 2009.

[36] "Windbg logger," https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/logger-and-logviewer, 2017.

[37] J. Desfossez, J. Dieppedale, and G. Girard, "Stealth malware analysis from kernel space with kolumbo," *Journal in computer virology*, vol. 7, no. 1, pp. 83–93, 2011.

[38] "DbgPrint Logger," https://alter.org.ua/soft/win/dbgdump/.

[39] T. Dietterich, "Machine learning for sequential data: A review," *Structural, syntactic, and statistical pattern recognition*, pp. 227–246, 2002.

[40] S. Forrest, A. Somayaji, and D. Ackley, "Building diverse computer systems," in *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, 1997.

[41] Y. Zhang, Q. Huang, X. Ma, Z. Yang, and J. Jiang, "Using multi-features and ensemble learning method for imbalanced malware classification," in *Trustcom/BigDataSE/I SPA, 2016 IEEE*. IEEE, 2016, pp. 965–973.

[42] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences," in *Australasian Joint Conference on Artificial Intelligence*. Springer, 2016, pp. 137–149.

[43] H. Yin, D. Song, M. Egele, C. Kruegel, and 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*. ACM, 2007, pp. 116–127.

[44] B. Caillat, B. Gilbert, R. Kemmerer, C. Kruegel, and G. Vigna, "Prison: Tracking process interactions to contain malware," in *HPCC, 2015 CSS, 2015 ICESS*. IEEE, 2015, pp. 1282–1291.

[45] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[46] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[47] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[48] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9.

[49] F. Yu and V. Koltun, “Multi-scale context aggregation by dilated convolutions,” *arXiv preprint arXiv:1511.07122*, 2015.

[50] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” *arXiv preprint arXiv:1502.03167*, 2015.

[51] “Rekings.com,” http://www.rekings.com/.

[52] “Winautomation - http://www.winautomation.com.” [Online]. Available: http://www.winautomation.com

[53] “Apache bench http://httpd.apache.org/docs/2.2/en/programs/ab.html.”

[54] “Novabench - https://novabench.com/.” [Online]. Available: https://novabench.com/

[55] “WinSAT,” https://technet.microsoft.com/en-us/library/cc770542(v=ws.11).aspx.

[56] “Wmic - https://msdn.microsoft.com/en-us/library/bb742610.aspx.” [Online]. Available: https://msdn.microsoft.com/en-us/library/bb742610.aspx

[57] “Windows hardware certification kit (windows hck),” https://developer.microsoft.com/en-us/windows/hardware/windows-hardware-lab-kit.

[58] “Geekbench 4,” https://www.geekbench.com/.

[59] F. Provost, “Machine learning from imbalanced data sets 101,” in *Proceedings of the AAAI2000 workshop on imbalanced data sets*, 2000, pp. 1–3.

[60] T. Chen and C. Guestrin, “Xgboost: A scalable tree boosting system,” in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM, 2016, pp. 785–794.

[61] Y. Freund and R. E. Schapire, “A desicion-theoretic generalization of on-line learning and an application to boosting,” in *European conference on computational learning theory*. Springer, 1995, pp. 23–37.

[62] R. Caruana, N. Karampatziakis, and A. Yessenalina, “An empirical evaluation of supervised learning in high dimensions,” in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008, pp. 96–103.

[63] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[64] PyTorch, https://github.com/pytorch/pytorch, 2017.

[65] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, “A survey on automated dynamic malware-analysis techniques and tools,” *ACM computing surveys (CSUR)*, vol. 44, no. 2, p. 6, 2012.

[66] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff., “A sense of self for Unix processes,” in *Proceedings of the IEEE Symposium on Security and Privacy*, 1996, pp. 120–128.

[67] C. Willems, T. Holz, and F. Freiling, “Toward automated dynamic malware analysis using cwsandbox,” *IEEE Security and Privacy*, vol. 5, no. 2, pp. 32–39, Mar. 2007. [Online]. Available: http://dx.doi.org/10.1109/MSP.2007.45

[68] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, “Learning and classification of malware behavior,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2008, pp. 108–125.

[69] A. Mohaisen, O. Alrawi, and M. Mohaisen, “Amal: High-fidelity, behavior-based automated malware analysis and classification,” *Computers & Security*, vol. 52, pp. 251 – 266, 2015.

[70] B. Kolosnjaji, A. Zarras, T. Lengyel, G. Webster, and C. Eckert, “Adaptive semantics-aware malware classification,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 419–439.

[71] C. Guarnieri, “Cuckoo sandbox,” https://www.cuckoosandbox.org/, 2017.

[72] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, “Automated classification and analysis of internet malware,” in *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, ser. RAID’07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 178–197. [Online]. Available: http://dl.acm.org/citation.cfm?id=1776434.1776449

[73] C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, “Comprehensive analysis and detection of flash-based malware,” in *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA*, 2016, pp. 101–121.

[74] D. Kirat, G. Vigna, and C. Kruegel, “Barebox: efficient malware analysis on bare-metal,” in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011, pp. 403–412.

[75] N. M. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song, “Differential slicing: Identifying causal execution differences for security applications,” in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 347–362.

[76] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna, “Efficient detection of split personalities in malware.” in *NDSS*, 2010.

[77] A. Kharraz and E. Kirda, “Redemption: Real-time protection against ransomware at end-hosts.”

[78] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel, “Insights into current malware behavior,” in *2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[79] M. Xie, J. Hu, and J. Slay, “Evaluating host-based anomaly detection systems: Application of the one-class svm algorithm to adfa-ld,” in *Fuzzy Systems and Knowledge Discovery (FSKD), 2014 11th International Conference on*. IEEE, 2014, pp. 978–982.

[80] Y. Ye, T. Li, S. Zhu, W. Zhuang, E. Tas, U. Gupta, and M. Abdulhayoglu, “Combining file content and file relations for cloud based malware detection,” in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2011, pp. 222–230.

[81] A. S. Abed, T. C. Clancy, and D. S. Levy, “Applying bag of system calls for anomalous behavior detection of applications in linux containers,” in *2015 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 2015, pp. 1–5.

[82] V. Kumar, H. Chauhan, and D. Panwar, “K-means clustering approach to analyze nsl-kdd intrusion detection dataset,” *International Journal of Soft*, 2013.

[83] Y. Fan, Y. Ye, and L. Chen, “Malicious sequential pattern mining for automatic malware detection,” *Expert Systems with Applications*, vol. 52, pp. 16–25, 2016.

[84] D. Chatzakou, N. Kourtellis, J. Blackburn, E. D. Cristofaro, G. Stringhini, and A. Vakali, “Mean birds: Detecting aggression and bullying on twitter,” 2017, https://arxiv.org/abs/1702.06877v1.

[85] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, “Mamadroid: Detecting android malware by building markov chains of behavioral models,” in *Proceedings of the 24th Network and Distributed System Security Symposium*, ser. NDSS. Internet Society, 2017.

[86] E. Mariconti, J. Onaolapo, G. Ross, and G. Stringhini, “What’s your major threat? on the differences between the network behavior of targeted and commodity malware,” in *11th International Conference on Availability, Reliability and Security*, ser. ARES. Los Alamitos, CA, USA: IEEE Computer Society, 2016, pp. 599–608.

[87] A. Caliskan-Islam, F. Yamaguchi, E. Dauber, R. Harang, K. Rieck, R. Greenstadt, and A. Narayanan, “When coding style survives compilation: De-anonymizing programmers from executable binaries,” 2016, https://arxiv.org/abs/1512.08546v2.

[88] G. Stringhini, G. Wang, M. Egele, C. Kruegel, G. Vigna, H. Zheng, and B. Y. Zhao, “Follow the green: Growth and dynamics in twitter follower markets,” in *Proceedings of the 2013 Conference on Internet Measurement Conference*, ser. IMC ’13. New York, NY, USA: ACM, 2013, pp. 163–176. [Online]. Available: http://doi.acm.org/10.1145/2504730.2504731

[89] D. Canali, M. Cova, G. Vigna, and C. Kruegel, “Prophiler: A fast filter for the large-scale detection of malicious web pages,” in *Proceedings of the 20th International Conference on World Wide Web*, ser. WWW ’11. New York, NY, USA: ACM, 2011, pp. 197–206. [Online]. Available: http://doi.acm.org/10.1145/1963405.1963436

[90] G. Stringhini, C. Kruegel, and G. Vigna, “Detecting spammers on social networks,” in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC ’10. New York, NY, USA: ACM, 2010, pp. 1–9. [Online]. Available: http://doi.acm.org/10.1145/1920261.1920263

[91] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas, “Malware classification with recurrent networks,” in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2015, pp. 1916–1920.

[92] O. E. David and N. S. Netanyahu, “Deepsign: Deep learning for automatic malware signature generation and classification,” in *Neural Networks (IJCNN), 2015 International Joint Conference on*. IEEE, 2015, pp. 1–8.

[93] J. Saxe and K. Berlin, “Deep neural network based malware detection using two dimensional binary program features,” in *2015 10th Interna-*

tional Conference on Malicious and Unwanted Software (MALWARE). IEEE, 2015, pp. 11–20.

[94] S. Hou, A. Saas, L. Chen, and Y. Ye, "Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs," in *Web Intelligence Workshops (WIW), IEEE/WIC/ACM International Conference on*. IEEE, 2016, pp. 104–111.

[95] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro, "Transcend: Detecting concept drift in malware classification models," 2017.

[96] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 39–57.

[97] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial perturbations against deep neural networks for malware classification," *arXiv preprint arXiv:1606.04435*, 2016.

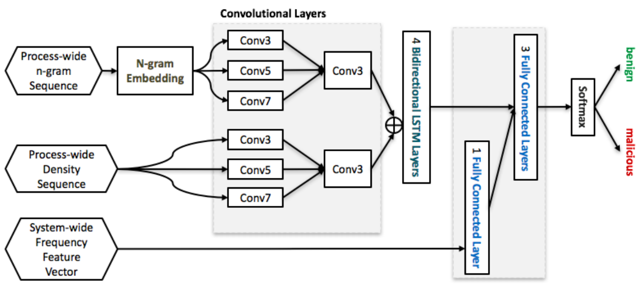
[98] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 582–597.

[99] Q. Wang, W. Guo, K. Zhang, A. G. Ororbia II, X. Xing, X. Liu, and C. L. Giles, "Adversary resistant deep neural networks with an application to malware detection," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2017, pp. 1145–1153.

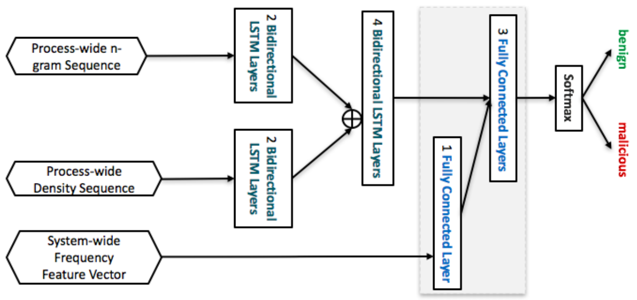
APPENDIX

A. *Architecture of CNNLSTM and LSTM model*

Figure 7b and Figure 7a depict the detailed architecture of *CNNLSTM* and *LSTM* model. We compare DEEPMALWARE with these two models to show the improvement.



(a) Illustration of *CNNLSTM* model


Process-wide n-gram Sequence
2 Bidirectional LSTM Layers
Process-wide Density Sequence
2 Bidirectional LSTM Layers
System-wide Frequency Feature Vector
4 Bidirectional LSTM Layers
1 Fully Connected Layer
3 Fully Connected Layers
Softmax
benign
malicious

(b) Illustration of *LSTM* model

Fig. 7: Architecture of *CNNLSTM* and *LSTM* model

B. *System Call Description*

TABLE VIII: The set contains 155 system calls, and is the **largest** set that have been hooked to our best knowledge.

System Call Name	System Call Name	System Call Name
oldNtCreateThread	oldNtCreateThreadEx	oldNtSetContextThread
oldNtCreateProcess	oldNtCreateProcessEx	oldNtCreateUserProcess
oldNtQueueApcThread	oldNtSystemDebugControl	oldNtMapViewOfSection
oldNtOpenProcess	oldNtCreateProcess	oldNtCreateProcessEx
oldNtOpenThread	**oldNtQuerySystemInformation**	oldNtSetInformationFile
oldNtQueryInformationFile	oldNtCreateMutant	oldNtDeviceIoControlFile
oldNtTerminateProcess	oldNtDelayExecution	oldNtQueryValueKey
oldNtQueryAttributesFile	oldNtResumeThread	**oldNtCreateSection**
oldNtLoadDriver	oldNtClose	**oldNtOpenFile**
oldNtNotifyChangeMultipleKeys	oldNtQueryMultipleValueKey	oldNtQueryObject
oldNtRenameKey	oldNtSetInformationKey	oldNtAllocateLocallyUniqueId
oldNtCreateDirectoryObject	oldNtCreateKey	oldNtCreateKeyTransacted
oldNtSetQuotaInformationFile	oldNtSetSecurityObject	oldNtSetValueKey
oldNtSetVolumeInformationFile	oldNtUnloadDriver	oldNtUnlockFile
oldNtUnmapViewOfSection	oldNtWaitForSingleObject	oldNtFlushInstructionCache
oldNtQueryInformationProcess	oldNtSetInformationProcess	oldNtAlertThread
oldNtCallbackReturn	oldNtGetContextThread	oldNtAlertResumeThread
oldNtContinue	oldNtImpersonateThread	oldNtRegisterThreadTerminatePort
oldNtSuspendThread	oldNtTerminateThread	oldNtOpenMutant
oldNtQueryMutant	oldNtReleaseMutant	oldNtSetTimerResolution
oldNtSetSystemTime	oldNtQueryTimerResolution	oldNtQuerySystemTime
oldNtQueryPerformanceCounter	oldNtLockFile	oldNtOpenEvent
oldNtQueryInformationThread	oldNtQueryDirectoryFile	oldNtQueryEaFile
oldNtSetInformationThread	oldNtAccessCheckByTypeAndAuditAlarm	oldNtCreateEvent
oldNtCreateFile	oldNtDeleteFile	oldNtFlushVirtualMemory
oldNtFreeVirtualMemory	oldNtLockVirtualMemory	oldNtProtectVirtualMemory
oldNtUnlockVirtualMemory	oldNtReadVirtualMemory	oldNtWriteVirtualMemory
oldNtReadFile	oldNtWriteFile	**oldNtWriteRequestData**
oldNtCreatePort	**oldNtImpersonateClientOfPort**	**oldNtListenPort**
oldNtQueryInformationPort	**oldNtRequestPort**	**oldNtAlpcAcceptConnectPort**
oldNtAlpcConnectPort	oldNtAlpcCreatePort	**oldNtAlpcCreatePortSection**
oldNtAlpcDeleteResourceReserve	**oldNtAlpcDisconnectPort**	oldNtReplyWaitReceivePortEx
oldNtPrivilegeCheck	**oldNtAlpcOpenSenderProcess**	oldNtAlpcQueryInformation
oldNtAreMappedFilesTheSame	oldNtAssignProcessToJobObject	oldNtCancelSynchronousIoFile
oldNtCompressKey	oldNtCreateEventPair	oldNtCreateKeyedEvent
oldNtCreateProfile	oldNtCreateSemaphore	oldNtCreateSymbolicLinkObject
oldNtCreateTransactionManager	oldNtDebugContinue	oldNtDeletePrivateNamespace
oldNtDisableLastKnownGood	oldNtDisplayString	oldNtDrawText
oldNtEnumerateDriverEntries	oldNtEnumerateTransactionObject	oldNtGetCurrentProcessorNumber
oldNtGetNlsSectionPtr	oldNtGetPlugPlayEvent	oldNtGetWriteWatch
oldNtImpersonateAnonymousToken	oldNtInitiatePowerAction	oldNtIsProcessInJob
oldNtIsSystemResumeAutomatic	oldNtLoadKey	oldNtLoadKey2
oldNtMakeTemporaryObject	oldNtMapUserPhysicalPagesScatter	oldNtModifyBootEntry
oldNtOpenPrivateNamespace	oldNtOpenResourceManager	oldNtOpenSemaphore
oldNtOpenSession	oldNtPrePrepareEnlistment	oldNtQueryInformationEnlistment
oldNtQueryInformationResourceManager	oldNtQueryInformationTransaction	oldNtQueryInformationWorkerFactory
oldNtReadOnlyEnlistment	oldNtRegisterProtocolAddressInformation	oldNtReplacePartitionUnit
oldNtResetWriteWatch	oldNtResumeProcess	oldNtSaveKeyEx
oldNtSetDefaultLocale	oldNtSetInformationDebugObject	oldNtSetInformationJobObject
oldNtSetInformationResourceManager	oldNtSetInformationTransactionManager	oldNtSetIntervalProfile
oldNtSetSystemPowerState	oldNtSetTimer	oldNtSinglePhaseReject
oldNtVdmControl	oldNtWaitLowEventPair	