Computers & Security 92 (2020) 101760

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

A dynamic Windows malware detection and prediction method based on contextual understanding of API call sequence

Eslam Amer[a,b,*], Ivan Zelinka[a,c]

[a] Faculty of Electrical Engineering and Computer Science, VSB-Technical University of Ostrava, 17. listopadu 2172/15, Ostrava-Poruba, 708 00 Ostrava, Czech Republic
[b] Misr International University, Cairo, Egypt
[c] Modeling Evolutionary Algorithms Simulation and Artificial Intelligence, Faculty of Electrical & Electronics Engineering, Ton Duc Thang University, Ho Chi Minh, Vietnam

ARTICLE INFO

Article history:
Received 9 October 2019
Revised 14 January 2020
Accepted 11 February 2020
Available online 11 February 2020

Keywords:
API call sequence
Malware detection
Malware prediction
Word embedding
Chain sequence

ABSTRACT

Malware API call graph derived from API call sequences is considered as a representative technique to understand the malware behavioral characteristics. However, it is troublesome in practice to build a behavioral graph for each malware. To resolve this issue, we examine how to generate a simple behavioral graph that characterizes malware. In this paper, we introduce the use of word embedding to understand the contextual relationship that exists between API functions in malware call sequences. We also propose a method that segregating individual functions that have similar contextual traits into clusters. Our experimental results prove that there is a significant distinction between malware and goodware call sequences. Based on this distinction, we introduce a new method to detect and predict malware based on the Markov chain. Through modeling the behavior of malware and goodware API call sequences, we generate a semantic transition matrix which depicts the actual relation between API functions. Our models return an average detection precision of 0.990, with a false positive rate of 0.010. We also propose a prediction methodology that predicts whether an API call sequence is malicious or not from the initial API calling functions. Our model returns an average accuracy for the prediction of 0.997. Therefore, we propose an approach that can block malicious payloads instead of detecting them after their post-execution and avoid repairing the damage.

1. Introduction

The rapid development in computers and Internet technology is also coupled with rapid growth in malicious software (Malware). Malware such as viruses, Trojans, and worms also changed expeditiously and became the most severe threat to the cyberspace. The malware performs malicious activities on the computer system. Usually, malware gains control over computer systems through changing or malfunctioning normal process execution flow.

The number of new malware instances is dramatically increasing. According to AV-TEST,[1] they daily register over 350,000 new malware and potentially unwanted applications (PUA). The massive amount of malware gives a signal to the problem of how to effectively analyze and process such amount of samples. Therefore, automatic malware detection becomes a necessity to handle the increasing amount of new generated malware.

Many researchers have focused on producing different malware detection techniques to alleviate the rapidly increasing rate of malware. Malware detection methods can be divided into static and dynamic malware detection (Cesare et al., 2013; Galal et al., 2016). Static malware detection inspects the content of the portable executable (PE) files without the actual execution of malware samples. During static analysis, analyzers extract features, including string patterns, operation codes, and byte sequences. The extracted features are used to decide whether a file is a malware or not (Gandotra et al., 2014). However, static-based malware detection methods are not sufficient on their own. The major drawback of static analysis methods is that it can be easily bypassed by obfuscation techniques (Ucci et al., 2018; Burnap et al., 2018). Furthermore, static approaches that rely on pattern matching are useful in detecting known malware patterns. However, pattern matching approaches are ineffective in detecting zero-day or polymorphic malware. Hence, static analysis approaches tend to be unreliable.

In response to the drawbacks of static-based malware detection, dynamic malware detection methods analyze the malware behavior during execution. Dynamic analysis is destined to monitor the

* Corresponding author.
E-mail addresses: eslam.amer@vsb.cz (E. Amer), ivan.zelinka@vsb.cz (I. Zelinka).
[1] https://www.av-test.org/.

https://doi.org/10.1016/j.cose.2020.101760
0167-4048/

malware during its real-time performance. In this context, malware is executed in a secured virtual environment that is used to monitor the malicious code behavior during real-time execution.

The dynamic analysis of malware behavior involves extracting features such as network behavior, registry change, system calls, and memory usage (Qiao et al., 2014). Windows Application Programming Interface (API) call sequences are considered one of the most representative characteristics in behavior-based malware detection (Elhadi et al., 2013). The API call analysis reveals how malware behaves.

Machine learning algorithms have been used in malware detection. Algorithms such as K-Nearest Neighbor (KNN), Decision Tree (DT), Naıv̈e Bayes (NB), and Support Vector Machine (SVM) are considered the most commonly used algorithms in malware detection (Fan et al., 2018; Lin et al., 2018; Rieck et al., 2011).

Unfortunately, the performance of most machine learning algorithms is counting on the accuracy of the extracted features. Additionally, it is also complicated to obtain meaningful behavior features that improve the performance of malware detection (Xiao et al., 2019). Furthermore, the handling of features requires professional expertise. Therefore, relying on traditional machine learning algorithms is still unconvincing for malware detection.

In this paper, we propose a method to understand the semantic chain sequence of API calls for malware as well as goodware samples. Through modeling the sequence of API calls for malware and goodware programs, we become able to predict whether any emerging new sequence is malicious or not. Inspired from the contextual understanding in natural language processing (NLP), the sequence of API calls of any program is a kind of context that represents the calling program. Therefore, a sequence of correlated API calls can be viewed as a significant representation of either goodware or malware behaviors. To justify our idea, we used unseen test sets that contain new samples of malware and goodware. The results showed that the proposed work outperforms the previous works in using API call sequence to identify the behaviors, whether it is malicious or normal.

Our contribution in this paper is:

- Introducing a method to analyze, detect, and predict windows malware based on the dynamic API call sequence. Also, we propose a method that reduces the number of API calls that represents a sample (malware or goodware)
- Capturing the relationship between API functions that represent malware and goodware in API call sequences. The relationship is modeled into Markov chain sequences. The resulted Markov chain sequences are representing our dynamic signature for malware.

The rest of the paper is as follows: The related work and background of other researches are discussed in Section 2. Section 3 presents the proposed malware detection model. Datasets and the empirical evaluation of the method are given in Section 4. Finally, Section 5 concludes this paper.

2. Related works

2.1. Overview

Many studies have been used to analyze malware characteristics. Malware analysis methods can be mainly categorized into static and dynamic analysis (Cesare et al., 2013; Galal et al., 2016; Salehi et al., 2017; Lee and Kwak, 2016; Zelinka and Amer, 2019). In static analysis, the malware file is checked through analyzing its executable binaries or codes without executing the malware. In contrast with static analysis, dynamic analysis methods monitor the malware execution process. Malware behavioral features

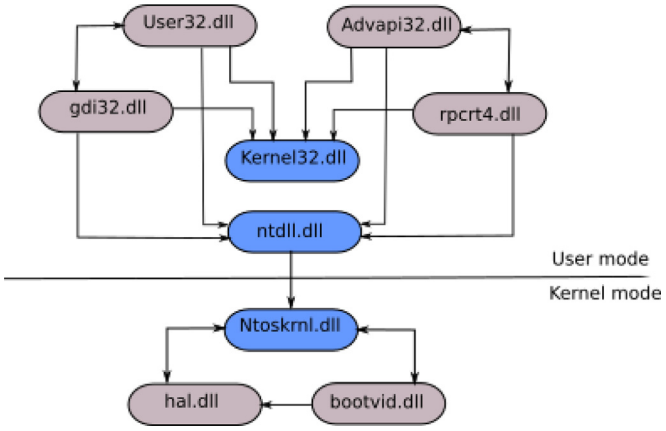


Fig. 1. Windows API call mechanism.

are collected, observed, and recorded during execution time. Dynamic analysis methods are usually performed in a secured virtual environment called sandbox. Common dynamic analysis sandboxes include Cuckoo Sandbox (Cuckoo Sandbox, 2019) and CWSandbox (CWSandbox, 2019). The main objectives of sandboxes are to inspect the malicious behavior of malware while preventing malware from attacking the host system.

Static and dynamic analysis methods have strengths and weaknesses. The major strength of static analysis over dynamic analysis is that it is free from the overhead cost caused by program execution. However, static analysis methods have limitations in terms of the lack of support for packed and complex obfuscated code (Zelinka and Amer, 2019; Han et al., 2019).

In comparison to static analysis, dynamic analysis is able to efficiently analyze packed and obfuscated malware. The reason is that malware must unpack itself while executing. Therefore, its original and malicious code will be loaded into the main memory. However, the main drawback of dynamic analysis is the time and resources' consumption. Malware samples should be analyzed individually which causes the limitation of dynamic analysis in commercial applications.

In this paper, we will focus on dynamic analysis methods. Specifically, we will focus on the analysis of API call sequences that are generated when malware samples are executed. We believe that dynamic analysis methods are considered the most effective and accurate in terms of malware analysis. The reason is that dynamic malware analysis capture feasible features that reflect real malware behaviors. API calls can be extracted using static and dynamic approaches. In static approaches (Ki et al., 2015; Ndibanje et al., 2019), APIs are extracted from the Portable Executable (PE) header of the executable files. However, in dynamic approaches (Qiao et al., 2014; Ki et al., 2015; Qiao et al., 2013), APIs are collected through observing the running executable files. In our approach, we used API call sequences collected using dynamic approaches.

2.2. Windows API

In the Windows operating system, user applications cannot access hardware or system resources directly. Nevertheless, they can rely on interfaces provided by dynamic-link libraries (Choudhary and Vidyarthi, 2015). These libraries such as kernel32.dll, user32.dll, gdi32.dll, and advapi32.dll provide functionalities to access hardware and system resources. The API calling mechanism in Windows is shown in Fig. 1. A brief description of the mentioned DLL files along with their description is shown in Table 1.

Table 1
DLL descriptions.

DLL name	Description
Kernel32.dll	Indexes the core and most common functions such as accessing and manipulating memory, files, hardware.
Ntdll.dll	Considered to be the interface to the Windows kernel. When a process imports this file, it means that it will use functions to create tasks or new functions that are not available to Windows programs. Tasks, such as process manipulation or hiding functions, will rely on this interface.
Advapi32.dll	Provides functionality to core Windows components like the service management and registry.
User32.dll	Provides components for the user interface, like buttons, scroll bars. It also provides functionalities for controlling and responding to user actions.
Gdi32.dll	Contains graphical displaying and manipulating functions.
Rpcrt4.dll	Provides essential functions to remote procedure call (RPC). This file is used by different Windows applications for networking and internet communication.
Ntoskrnl.dll	Known also as the kernel image. This file is considered the fundamental part of the operating system. It provides various system services such as hardware abstraction and memory and process management.
Hal.dll	Contains the functions that act as middleware between the kernel and the hardware. Windows rely on functions provided by this file as interfacing to unique chipsets that are associated with specific motherboards.
Bootvid.dll	Provides functions to video graphics adapter (VGA). Therefore, it is heavily used in the Windows operating system.

The interface shown in Fig. 1 is called the Win32 API. Assume that a user program wants to read a file, it calls the Win32 API function of reading a file. The process initially invokes the NtReadFile function that is indexed in "ntdll.dll". Then the NtReadFile function invokes the associated service routine from the kernel mode, which is also named NtReadFile. Therefore, any program that needs a particular service will call the native API in the kernel mode.

In the case of program monitoring, the leading way to observe the program is through monitoring its API calls (Zhao et al., 2019; Gupta et al., 2016; Ding et al., 2018; Alaeiyan et al., 2019). The API functions by itself are standard; there are no categories called malicious or normal functions. Therefore, malware programs use the standard API functions also to perform its malignant purpose. The API call mechanism doesn't differentiate between malicious and normal programs. Malicious or normal processes can utilize the same API. However, the behavioral sequence of API calls can lead to the contextual property of the calling process (Ki et al., 2015). In other words, whether the API calling sequence is malicious or normal. However, the number of API functions is huge by itself. Therefore, it becomes tough to describe the behavioral attributes of running processes by tracking all APIs at the same time.

The sequence of API function calls between processes, and the operating system is considered the fundamental behavioral difference between malicious and normal processes (Ding et al., 2018). Most of the research work in behavior malware analysis has focused on API calls (Zhao et al., 2019; Choi et al., 2019; Tajoddin and Jalili, 2018). The API call sequences can provide meaningful expressions that support and assist in better malware understanding. API calls encode adequate information about the implicit malware functionalities which take place during the run time of malware.

Lu et al. (2014) and Liu et al. (2011) transform API calls into regular expressions (RE) rules to characterize malware call patterns. They detect malware when a match occurs between the examined call sequence and stored RE rules. Tran and Sato (2017) analyzed the API call sequence using natural language processing. Sequence calls are divided using methods such as n-gram and given weights using term frequency-inverse document frequency (TF-IDF). The objective of TF-IDF is to convert n-grams into numerical input features where machine learning (ML) algorithms can work. However, methods such as TF-IDF don't preserve any contextual relationship that exists between words. Therefore, in our proposed work we applied word embedding on API call sequences to find the contextual relationship between API calls.

Pektaş and Acarman (2017) analyze the n-gram over malware API call sequences to discover some malicious patterns. Frequently occurred patterns are viewed as behavioral representation features of malware. Lee et al. (2018) presented a methodology to group different types of malware mutants through capturing and clustering n-grams of malware mutants. Ravi and Manoharan (2012) used the n-grams to create a 3rd order Markov chain to model the API call sequence. They rely on association mining rules to classify the sequence. Kim et al. (2016) uses multiple sequence alignment (MSA) to generate malware behavioral "feature-chain" patterns. The generated patterns are used to classify malware that has similarities with training patterns. Tungjitviboonkun and Suttichaya (2017) proposed a similar approach to the one used by Kim et al. (2016). In Tungjitviboonkun and Suttichaya (2017), the approach used the longest common API sequence to split malware API call sequence into simple sequences. Then, the extracted longest common sub-sequences were used as a signature to match similar malware.

In this paper, we introduce a methodology that cluster and group related APIs based on their contextual similarity into a single cluster name. Therefore, instead of dealing with a large number of different API calls in the whole sample sequences, the API will be replaced by the name of the cluster that contains it. The customization in grouping the number of APIs into a limited number of cluster names made it possible to describe the behavioral attributes of running processes. The analysis of the newly customized APIs makes it also possible to describe the behavioral attributes of running processes efficiently.

3. Proposed model

3.1. Overview

As mentioned in the related works, previous studies are mainly focused on extracting some patterns such as n-gram from API call sequences. These patterns were used as features for malware detection. However, there have been no attempts to examine the correlation that may exist between individual API calls in the whole sequence. Inspired by this notation, we need to answer the following questions:

(1) Is there any type of correlations that exists between API calls in a malware sequence?
(2) Are there any discoverable differences between normal and malicious API call sequence?

These questions have not been addressed in previous studies. Motivated by this inspiration, we try to introduce a new direction in malware detection through dynamic analysis.

3.2. Challenges

In the proposed work, we need to find relation(s) that may exist between individual API functions that are found in nor-

mal/malware call sequences. However, the main challenges that our approach faced are:

(1) The number of individual API functions is considerably huge, which make it difficult to analyze.
(2) Malware usually tries to escape from being tracked or detected. Therefore, malware authors usually add a large number of unneeded API calls. The main objectives are to add some events that are looking normal to hide its malicious behavior, and made the analysis process too difficult (Firdaus et al., 2018).

3.3. System overview

The execution process of a program can be described in several ways (Sheneamer et al., 2018). API calls and system call usually describes the semantic execution of the program. In this way, it is possible to characterize the behavior of the program. In this paper, we define the semantics of the executed program from API call level, and system call level. Our proposed system which is outlined in Fig. 2 has three phases namely, initialization, learning, and testing phase. In the following subsections, we will describe each phase in detail.

3.3.1. Initialization phase

The main objective of the initialization phase is to group API functions into groups or clusters. The initialization phase contains mainly three steps namely, word embedding, calculate the similarity between APIs, and clustering similarity matrix.

3.3.1.1. Word embedding. Word embedding (Ketkar, 2017), are a form of word representation in an n-dimensional space. The main objective of word embedding is to transfer the human understanding of a language into machines. Word2Vec is considered one of the most used word embedding forms (Alami et al., 2019; Rezaeinia et al., 2019; Martinčić-Ipšić and Miličić, 2019). Word2Vec takes a large corpus of text as input, producing a vector space of several hundred dimensions. Every distinct word in the input corpus is assigned a corresponding vector in the space. The distribution of word vectors in the space is depending solely on the contextual similarity in the input corpus. If two words are similar in context, they will be located close to each other in the neighborhood space, and if two words are different in context, they will be located far from each other.

In the context of malware analysis, we believe that the order of API functions in malware sequence does not randomly exist. Indeed, it may encode some contextual patterns that perform malicious activities. Such patterns are similar at somehow among different malware sequences. Through extracting these patterns from a massive number of malware sequences, we could be able to identify the contextual relations that occur between API functions in sequences.

In our proposed model, the input corpus for the word embedding model is sequences of API calls for goodware and malware. In our experiments, we set the dimensional feature vectors size to 250, a window size of 6, workers of 6.

The output of word embedding as shown in Fig. 2 is two embedding models namely, goodware, and malware embedding model respectively. Each embedding model contains its distinct API call functions as well as the resulting embedding model.

3.3.1.2. Calculate similarity between APIs. The embedding models for goodware and malware can be used to compute the similarity between their API functions. To compute the similarity between two words using Word2Vec, we used the *model.similarity(argument_1, argument_2)* method and pass two API functions as arguments.

Consider the following two API functions *getfileversioninfosize, and getfileversioninfo*. The first function determines if the operating system can retrieve information about the version of specific file call,[2] while the second function retrieves the version information for the specified file.[3] The call to *getfileversioninfosize* is necessary before calling *getfileversioninfo.* According to our embedding model, *model.similarity('getfileversioninfosize', 'getfileversioninfo') = 0.904.* The result represents the Euclidean similarity between the aforementioned two API functions. From the above example, we can say that *getfileversioninfosize* is highly similar to *getfileversioninfo*.

As shown in the initialization phase in Fig. 2, the similarity calculation between APIs takes the API functions as input, then uses the embedding model to compute the similarity between all API functions. We used the Word2Vec similarity method to compute the similarity between every API function and other API functions. The output of this step is two similarity matrices namely, goodware similarity matrix, and malware similarity matrix. Each matrix represents the similarities between distinct API functions in its category sequences. The dimension of each matrix is depending on the number of API functions that are resulted by applying word embedding on the training data.

3.3.1.3. Clustering similarity matrix. The objective of the clustering step is, segregating individual API functions that have similar traits into clusters. The input to the clustering step is a similarity matrix, which represents the similarity relations between individual API function with other remaining functions. We used the k-means algorithm (Muller and Guido, 2017), to cluster the similarity matrix.

In order to determine the optimal number of clusters for the k-means algorithm, we relied on one of the most common approaches called the elbow method (Syakur et al., 2018). The approach involves running the k-means algorithm multiple times over a range of an increasing values of k. It then calculates the Within-Cluster Sum of Squared (WSS) errors for different values of k as described by Eq. (1).

$$WSS = \sum_{k=1}^{k} \sum_{x_i \in C_k} (x_i - \mu_k)^2 \tag{1}$$

where x_i is an observation belonging to the cluster C_k, and μ_k is the mean value of the points assigned to the cluster C_k . The optimal number of clusters is the value of k at the "elbow" where the distortion/inertia start decreasing linearly. In our experiments, we evaluated the value of k in range 2:21 for all datasets. The output of the clustering step, as well as the initialization phase, is a limited number of clusters for goodware and malware. Each category has its clusters that contain the related API functions.

3.3.2. Learning phase.

The objective of the learning phase is to capture the relations that exist between API calls in malware or goodware sequences — then creating behavioral models for malware and goodware. In the learning phase, we have two levels of outputs, namely:

1. Goodware/Malware cluster transition matrix and,
2. Goodware/Malware transition model.

In the learning phase, we have three main processes in this phase, namely, find API function in clusters, create sequence chain transition matrix, and calculate maximum transition sequence probability.

[2] https://docs.microsoft.com/en-us/windows/win32/api/winver/nf-winver-getfileversioninfosizew.
[3] https://docs.microsoft.com/en-us/windows/win32/api/winver/nf-winver-getfileversioninfoa.

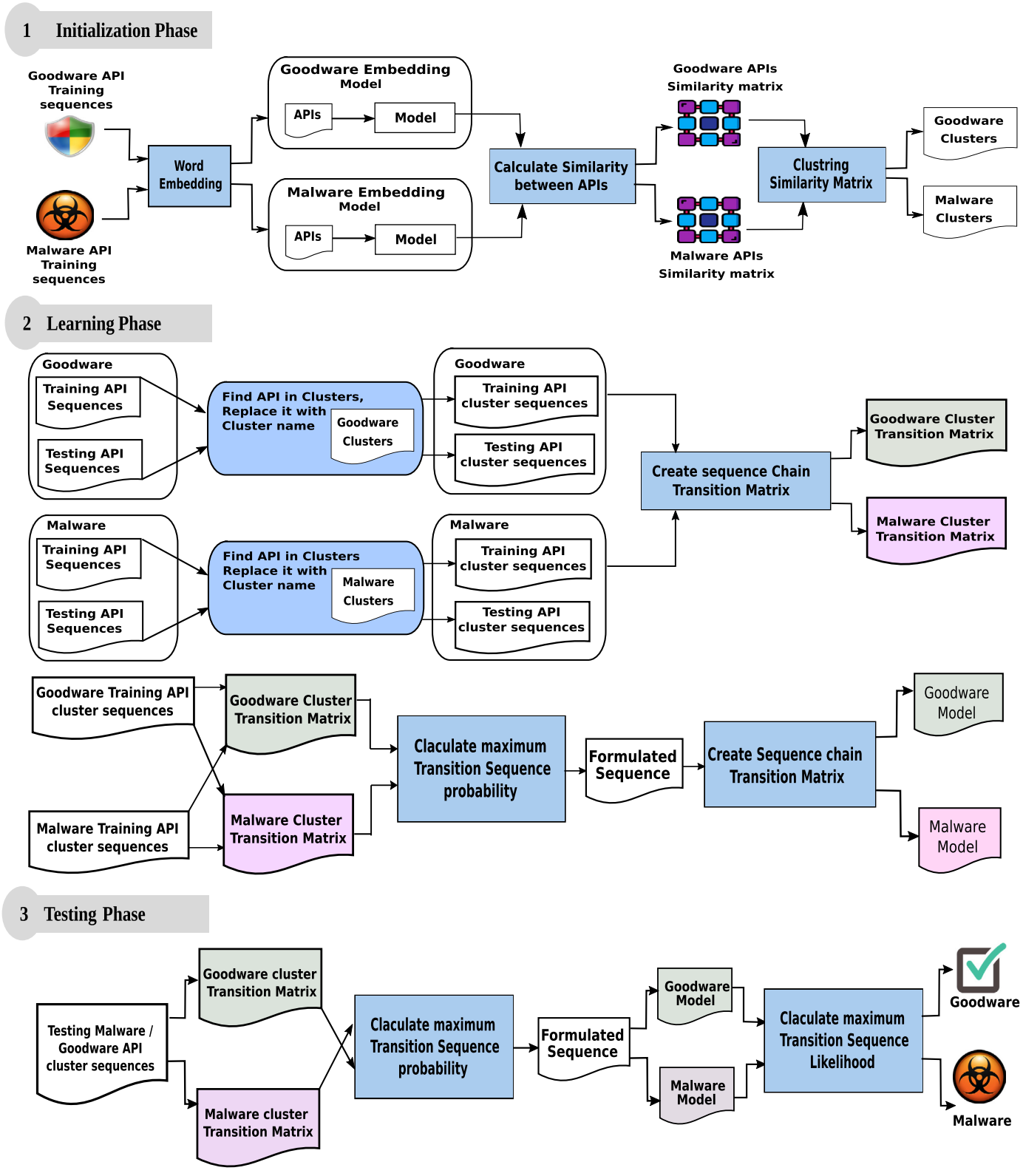


Fig. 2. The proposed detection model overview.

3.3.2.1. Find API function in cluster. In this step, we aim to transfer the original API call sequence into clusters sequence. This step takes the API call sequences as an input. Each API function inside the sequence is replaced with the cluster name that contains the API function. For example, the following sequence is the API call sequence for malware called Worm.Win32.Zwr.c

['GetSystemDirectoryA', 'IsDBCSLeadByte', 'LocalAlloc', 'CreateSemaphoreW', 'CreateSemaphoreA', 'GlobalAddAtomW', 'lstrcpynW', 'LoadLibraryExW', 'SearchPathW', 'CreateFileW', 'CreateFileMappingW', 'MapViewOfFileEx', 'GetSystemMetrics', 'RegisterClipboardFormatW', 'SystemParametersInfoW', 'GetDC', 'GetDeviceCaps', 'ReleaseDC', 'LocalAlloc',

'GetSysColor', 'GetSysColorBrush', 'GetStockObject', 'GetSystemMetrics', 'LoadCursorW', 'RegisterClassW', 'RegisterClassExW', 'LoadLibraryExW', 'LoadLibraryW', 'GetCommandLineA', 'GetStartupInfoA', 'LockResource', 'GetModuleFileNameA', 'IsBadWritePtr', 'RegisterClipboardFormatW', 'SystemParametersInfoW', 'GetSystemMetrics', 'LocalAlloc', 'GetSysColor', 'GetSysColorBrush', 'GetStockObject', 'LoadLibraryW', 'LoadLibraryExW', 'LoadCursorW', 'RegisterClassW', 'GetKeyboardType', 'GetCommandLineA', 'GetStartupInfoA', 'GetVersion', 'GetModuleFileNameA', 'lstrcpynA', 'GetThreadLocale', 'GetLocaleInfoW', 'GetLocaleInfoA', 'lstrlenA', 'LoadLibraryExW', 'SearchPathW', 'FindResourceExW', 'LoadResource', 'LoadStringA', 'LocalAlloc', 'VirtualAllocEx', 'GetThreadLocale', 'GetLocaleInfoA', 'GetLocaleInfoW', 'EnumCalendarInfoA', 'CreateEventA', 'LoadLibraryExW', 'lstrcpyA', 'CompareStringA', 'lstrcmpA', 'WaitForSingleObjectEx', 'WaitForSingleObject', 'GetProcessVersion', 'GlobalAlloc', 'DuplicateHandle', 'WSAStartup', 'LoadLibraryExW', 'CreateSemaphoreA', 'CreateSemaphoreW', 'ReleaseSemaphore', 'WaitForSingleObject', 'WaitForSingleObjectEx', 'GetWindowsDirectoryW', 'LocalAlloc', 'FindFirstFileExW', 'FindFirstFileA', 'GetModuleFileNameA', 'CreateFileW', 'WriteFile', 'CopyFileExW', 'CopyFileA', 'OpenEventW', 'WaitForSingleObject', 'WaitForSingleObjectEx', 'LoadLibraryW', 'LoadLibraryExW', 'DuplicateHandle', 'DeviceIoControl', 'SwitchToThread', 'WSACleanup', 'FreeLibrary', 'VirtualQueryEx', 'ResetEvent', 'VirtualFreeEx', 'UnregisterClassW']

Each API function that appears in the API call sequence will be searched against the clusters. When found, it will be replaced by the cluster number that contains it. The following representation is the cluster sequence which represents the above original API call sequence: 9,7,0,5,5,0,0,0,4,4,4,4,8,8,8,3,3,3,0,3,3,3,8,3,3,3, 0,3,1,1,1,1,1,8,8,8,0,3,3,3,3,0,3,3,1,1,1,1,1,1,1,1,1,0,4,1,1,2,0,1,1,1,1,1,1,0,1,1,1, 2,2,1,1,1,2,0,5,5,1,2,2,1,0,1,2,1,4,1,2,2,1,2,2,3,0,1,1,2,2,1,1,1,1,2

The replacement of the API function sequence into a clusters' sequence is considered the most crucial step. The reason for its importance comes from the fact that the number of API functions is enormous; therefore, it becomes impossible to trace all the API functions that are existing in malware calling sequences. Through the word embedding step, we represented the API functions according to their contextual similarity. When grouping API functions that are contextually similar into a single cluster, we can replace the API function with the cluster name that holds it. With a limited number of clusters, we have an excellent opportunity to customize the possibilities of combinations for sequences that malware can have. For example, the malware dataset in Ki et al. (2015) is grouped into ten clusters that index 1165 API functions extracted from malware sequences. The big difference between the number of the API functions and the number of clusters dramatically limits the combinations that malware sequence can take, which ease the possibility of analysis. The final output of the step of finding API functions in clusters is a set of clusters' sequences.

Grouping contextually similar API functions into clusters can raise a remarkable question about the effectiveness of this way. For example, what if the API functions are grouped in a non-contextual manner (i.e., based on the DLL files that index them)? In Section 4.3, we showed an experimental comparison between both clustering approaches.

3.3.2.2. Create a sequence of transition matrix. The clusters that are produced in the initialization phase can be viewed as a finite set of states S where $S = \{S_1, S_2, S_3,\ldots,S_n\}$. The proposed model assumes that the process, whether it is malicious or not, always in a finite number of states called Markov states. The process initially starts in one of these states and proceed to move successively into another state. Each move or shift is called a step. When the process starts at state S_i, then it can move to another state S_j as its next step with a probability P_{ij}. The probability of movement is not depending on the previous states where the chain was before the current state. The process can also move or loop in the same state based on the current sequence. The initial probability distribution, defined on S, determines the starting state S_0. Usually, specifying a particular state as a starting state. In the proposed work, since any state can be a start state, we set the probability of S_0 to be equally divided among the number of states that already exist in our model.

The process usually moves from one state to another generating a sequence of states $S_{i,1}, S_{i,2}, S_{i,3},\ldots,S_{i,k}$. The sequence of movement over the different states are viewed as transitions between different states. The probability of moving from one state to another state in a sequence is called a *transition probability*. The transition sequences is described using first-order Markov chain, where, the current state depends only on the previous state. A Markov model of n states will have n^2 transition probabilities. The transition probability can be represented as $n\ x\ n$ matrix.

The proposed model relied on maximum likelihood estimation (MLE) (Andrews et al., 2006) to create the transition probability that characterizes the sequence of state transitions. MLE is a method that determines the transition probability values between states in the Markov model. Given some observations produced from a model, the transition probability values are found such that they maximize the likelihood that characterizes the process described by the model. The first output in the learning phase is two transition matrices, namely, *malware and goodware cluster transition matrix*.

Figs. 3a and 4a describe real cluster transition matrices that resulted from one of our experiments on Ki et al. (2015) dataset. Fig. 3.a shows malware transition probabilities that occur between the number of malware states, while Fig 4a shows goodware transition probabilities that occur between the number of goodware states. The graph representation which characterize malware and goodware transition matrix is shown in Figs. 3b, and 4b respectively.

3.3.2.3. Calculate the maximum transition sequence probability. Malware and goodware transition matrices are considered the heart of our model. Given any cluster sequence, the transition values between clusters in both goodware and malware transition matrices are the distinctive features between malware and goodware. However, it could be more understandable if we reformulate the cluster sequences for malware and goodware into a more straightforward form. The reason behind such reformulation is that we want to reveal the traversing behavior of a given sequence. In other words, we want to observe the malicious likelihood behavior of malware, and the benevolent likelihood behavior of a goodware sequence.

To perform the reformulation, we used the malware/goodware cluster sequences training set along with the malware/goodware cluster transition matrices. Each sequence in both malware/goodware training clusters is traversed against both malware and goodware cluster transition matrices. We used Eq. (2) to compare and reformulate each transition sequence according to its transition probabilities in both transition matrices.

$$Transition\ sequence(i,j) = \begin{cases} 1 & p(Malware|(i,j)) > p(Goodware|(i,j)) \\ 0 & p(Malware|(i,j)) < p(Goodware|(i,j)) \end{cases} \quad (2)$$

where *(i,j)* is a transition sequence from state i to state j, *p(Malware|(i,j))* is the transition probability for transition sequence *(i,j)* in malware transition matrix, and *p(Goodware|(i,j))* is the transition probability for transition sequence *(i,j)* in goodware transition matrix. A transition move is considered malicious or not according to its maximal transition probability in both

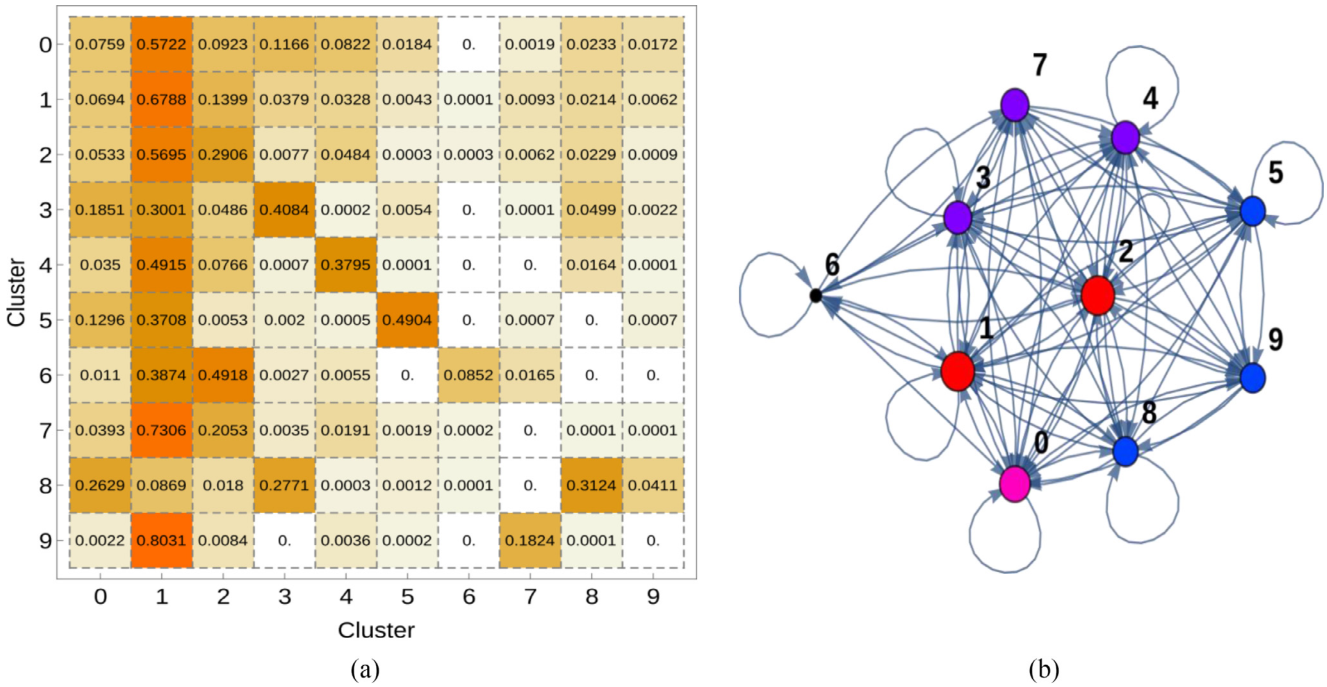


Fig. 3. (a) Malware transition matrix; (b) Malware transition graph.

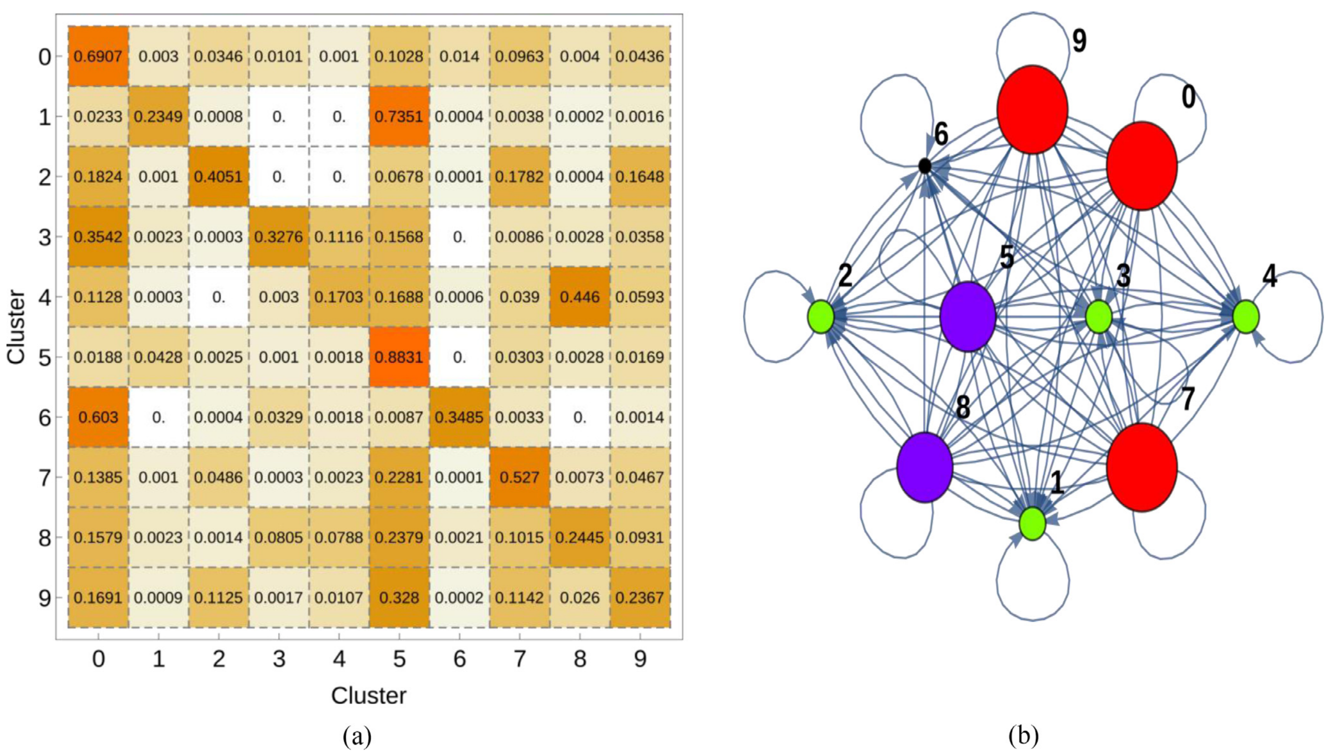


Fig. 4. (a) Goodware transition matrix; (b) Goodware transition graph.

malware and goodware transition matrix. According to Eq. (2), a transition sequence is replaced with *one* if it is a malicious transition and *zero* if it is a goodware one. For example, the cluster transition sequence in Section 3.3.2.2 is formulated according to malware/goodware cluster transition matrices in Figs. 3a and 4a into: 1,0,0,0,1,0,0,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,0,1,1,1,1,1, 1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,0,0,0, 1,1,0,1,1,1,1,1,1,1,1,0,1,1,0,1,0,1,1,1,0,1,1,1,1,1

We used the formulated sequences for malware and goodware to create general models that describe malware and goodware. Again, we relied on the maximum likelihood estimation to create the malware and goodware transition model. The malware transition model is characterizing the reformulated malware clustering training sequences. Similarly, the goodware transition model is characterizing the reformulated goodware clustering training sequences.

Table 2
Tracing of transition probabilities for sequence ‘1,1,1,1,1,0,4,1,1,2,0,1’.

Transition											
Sequence	p(1,1)	p(1,1)	p(1,1)	p(1,1)	p(1,0)	p(0,4)	p(4,1)	p(1,1)	p(1,2)	p(2,0)	p(0,1)
p(sequence, Malware)	0.6788	0.6788	0.6788	0.6788	0.0694	0.0822	0.4915	0.6788	0.1399	0.0533	0.5722
p(sequence, Goodware)	0.2349	0.2349	0.2349	0.2349	0.0233	0.0010	0.0003	0.2349	0.0008	0.1824	0.0030

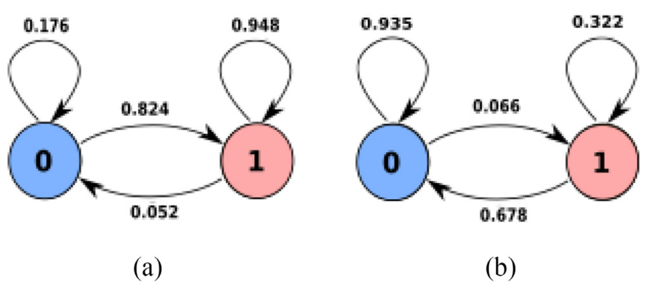


Fig. 5. (a) Malware model; (b) Goodware model.

The final output of the learning model is two Markov models, namely malware and goodware. Fig. 5a and b are showing an example of the resulted malware and goodware models.

3.3.3. Testing phase

The objective of the testing phase is to measure the accuracy of the proposed model in identifying and classifying unseen sequences into their categories. In this phase, the proposed system is tested using unseen goodware and malware cluster sequences. The cluster test sequences are fed as an input to both malware and goodware cluster transition matrices. Each sequence traverses both transition matrices, and while transiting from a state to another state, the probability of state transition is stored. When the sequence traversing finished, a comparison is made between traversing probabilities for both malware and goodware, according to Eq. (2). The output is a formulated new sequence of *ones* and *zeros*.

The formulated sequences are fed as input to both malware and goodware models. The final decision is determined based on the maximum cumulative likelihood transition probabilities that are gained from tracing both malware/goodware transition models.

Let us test our model against real sub-sequences of malware and goodware samples. In example 1, our proposed work is tested against real malware API sub-sequence, while in example 2, our work is tested against real goodware API sub-sequence. We used the cluster transition matrices in Figs. 3a and 4a. and malware/goodware models in Fig. 5a and b.

Example (1) :

Given a sub-sequence of API calls resulted for Worm.Win32.Zwr.c:

“‘lstrcpynA’, ‘GetThreadLocale’, ‘GetLocaleInfoW’, ‘GetLocaleInfoA’, ‘lstrlenA’, ‘LoadLibraryExW’, ‘SearchPathW’, ‘FindResourceExW’, ‘LoadResource’, ‘LoadStringA’, ‘LocalAlloc’, ‘VirtualAllocEx’”

Initially, each API function that is exists in the call will be searched within the clusters. When found, we write down the cluster that contains it. After searching clusters to find each of the API call function in the above sub-sequence, we got the following clusters’ sequences: 1,1,1,1,1,0,4,1,1,2,0,1

We want to know whether the given sequence is malicious or not. Therefore, the system will have to calculate the following transition probabilities which describe the input sequence: p(1,1), p(1,1), p(1,1), p(1,1). p(1,0), p(0,4), p(4,1), p(1,1), p(1,2), p(2,0), p(0,1)

Step 1: check transition probability for each transition in both malware and goodware transition matrices. Table 2 outlines the tracing of transition probabilities for the above cluster sequences.

Step 2: Reformulate the cluster sequence using Eq. (2), which results in a new sequence: 1,1,1,1,1,1,1,1,1,0,1

Step 3: calculate the likelihood of the new sequence against malware and goodware models:

Input sequence: p(1,1)+*p*(1,1)+*p*(1,1)+*p*(1,1)+*p*(1,1)+*p*(1,1) +*p*(1,1)+*p*(1,1)+*p*(1,0)+*p*(0,1)

likelihood (sequence, malware) = 0.948 + 0.948 +0.948 +0.948 + 0.948 + 0.948 + 0.948 + 0.948 + 0.052 + 0.824 = 8.46

likelihood (sequence, goodware) = 0.322 + 0.322 +0.322 + 0.322 + 0.322 + 0.322 + 0.322 + 0.322 + 0.678 + 0.066= 3.32

Step 4: Calculating the maximum likelihood accumulated transitions value:

Max((sequence, Malware), (sequence, Goodware)) = Max(8.46, 3.32) = 8.46.

So, the sequence is considered malicious sequence.

Example (2) :

Given a sub-sequence of API call sequence resulted by running *AriaMaestosaSetup-1.4.13.exe*[4]

“PeekMessageW’, ‘MsgWaitForMultipleObjects’, ‘MessageBoxW’, ‘LoadStringW’, ‘GetSystemMetrics’, ‘ExitWindowsEx’, ‘DispatchMessageW’, ‘DestroyWindow’, ‘CharUpperBuffW’, ‘CallWindowProcW’, ‘WriteFile’, ‘WideCharToMultiByte’”.

The following representation is the cluster sequence which represents the above original API call sequence:

5,5,5,5,5,5,5,1,5,5,5,5,5,5,1,1,5,5,1,5,5,5,5,5,1,1,5,1,5,5,5,1,1,5,5,5,5,5, 5,5,1,5,5,5,5,5,5,5,1,1,1,1,5,5,5,5,5,5,7,5,5,5,5,5,5,7,5,5,0,5,5,5,5,5,5,5,5,5, 5,5,5,5,5,5,5,5,5,5,9,9,5,9,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,4,8, 8,3,4,4,8,8,0,0,0,0,0,0,0,0,6,6,0,6,6,0,0,3,3,3,0,0,0,5,5,5,7

After searching clusters to find each of the API call function in the above sub-sequence, we got the following clusters’ sequences: 5,5,5,5,5,5,5,9,9,5,9,5

Now, we want to know whether the given sub-sequence is malicious or not?. The system will have to calculate the following transition probabilities which describe the input sequence: p(5,5),p(5,5), p(5,5), p(5,5), p(5,5), p(5,5), p(5,9), p(9,9),p(9,5), p(5,9), p(9,5)

Step 1: check transition probability for each transition in both malware and goodware transition matrices. Table 3 outlines the tracing of transition probabilities for the above cluster sequences.

Step 2: Reformulate the cluster sequence using Eq. (2), which results in a new sequence: 0,0,0,0,0,0,0,0,0,0,0

Step 3: calculate the likelihood of the new sequence against malware and goodware models:

Input sequence: p(0,0)+*p*(0,0)+*p*(0,0)+*p*(0,0)+*p*(0,0)+*p*(0,0)+ *p*(0,0)+*p*(0,0)+*p*(0,0)+*p*(0,0)

likelihood (sequence, malware) = 0.176 + 0.176 + 0.176 + 0.176 + 0.176 + 0.176 + 0.176 + 0.176 + 0.176 + 0.176 = 1.76

likelihood (sequence, goodware) = 0.935 + 0.935 + 0.935 + 0.935 + 0.935 + 0.935 + 0.935 + 0.935 + 0.935 + 0.935 = 9.35

[4] https://osdn.net/projects/sfnet_ariamaestosa/downloads/ariamaestosa/1.4.13/AriaMaestosaSetup-1.4.13.exe/ [accessed on 27/8/2019].

Table 3
Tracing of transition probabilities for sequence '5,5,5,5,5,5,5,9,9,5,9,5'.

Transition											
Sequence	p(5,5)	p(5,5)	p(5,5)	p(5,5)	p(5,5)	p(5,5)	p(5,9)	p(9,9)	p(9,5)	p(5,9)	p(9,5)
p(sequence, Malware)	0.4904	0.4904	0.4904	0.4904	0.4904	0.4904	0.0007	0.0000	0.0002	0.0007	0.0002
p(sequence, Goodware)	0.8831	0.8831	0.8831	0.8831	0.8831	0.8831	0.0169	0.2367	0.3280	0.0169	0.3280

Table 4
Dataset used for evaluation.

Reference name	Size	Description
Ki et al. (2015)	23,080 malware (Ki et al., 2015) 21,116 Benign[a]	API call sequences
Kim (2018)	151 malware 69 Benign	API call sequences
CSDMC2010 Dataset[b]	320 malware 68 Benign	API call sequences
Catak and Yazı (2019)	7107 malware 169[c] Benign	System call sequences

[a] https://github.com/fabriciojoc/brazilian-malware-dataset/blob/master/goodware.csv [Accessed 5 July-2019].
[b] **Intelligence and Security Informatics Data Sets [**http://www.azsecure-data.org/].
[c] https://www.researchgate.net/publication/336024802_Windows_PE_API_calls_for_malicious_and_benigin_programs [accessed:20 November-2019].

Step 4: Calculating the maximum accumulated transitions value:
Max(*(sequence, Malware), (sequence, Goodware)*) = Max(1.76, 9.35) = 9.35. So, the sequence is considered *a goodware sequence*

4. Results and discussion

In this section, we present our evaluation using different datasets. We evaluated how our model can correctly detect and predict whether a given API call sequence is malicious or not.

4.1. Dataset

To validate our approach, we collected several API call sequences. Our experiments are carried out with different datasets of different size. Most of the authors didn't provide access to their used datasets, or they provide URL links that are not working anymore. Therefore, it was not an easy task to gather datasets for malware and goodware. We noticed that authors provide only malware API call sequence datasets while ignoring to provide goodware API call sequences like in Ki et al. (2015), Catak and Yazı (2019). It was a necessity to have an API call sequence dataset for goodware also to compare the differences in the execution behavior of malware and goodware. Therefore, we provided some available goodware API call sequence datasets from open sources like Github[5] to make the comparison between malware and goodware behavioral execution. Table 4 lists details of the datasets along with its size, and description.

4.2. Evaluation metrics

We relied on evaluation metrics such as accuracy, precision, recall, and F-measure to evaluate the performance of our proposed model. The calculations of evaluation metrics are shown in Eqs. (3)–(6).

$$Accuracy = \frac{True\ Positives\ (TP) + True\ Negatives\ (TN)}{True\ Positives\ (TP) + True\ Negatives\ (TN) + False\ Positives\ (FP) + False\ Negatives(FN)} \tag{3}$$

$$Precision = \frac{True\ Positives\ (TP)}{True\ Positives\ (TP) + False\ Positives(FP)} \tag{4}$$

$$Recall = \frac{True\ Positives\ (TP)}{True\ Positives\ (TP) + False\ Negatives(FN)} \tag{5}$$

$$F - measure = \frac{2 \times precision \times recall}{precision + recall} \tag{6}$$

4.3. Malware detection evaluation

In our experimentation, we selected a range of 50% of data as a training set, while preserving the remaining 50% for testing. In our training, we rely on random subsamples (with replacement), which is similar to the k-fold strategy; however, in each iteration, we randomly select some samples for training and some others for testing. Such a method has an advantage over k-fold is that we can freely decide the number of iterations and the length of each train-test. The training set was randomly populated while preserving a condition that there is no duplicated occurrence of items in training and test sets. To avoid training bias, the experiments were run *ten* times for each dataset. We got the average of the returned results for a total of ten experiments to be the final evaluation measures for each dataset.

To verify the performance of our proposed work, we will use two additional evaluation metrics. The other metrics are inspired by the confusion matrix which is a table that describes the performance of the classification methods. The evaluation metrics are *false positive rate (FPR)*, and *false negative rate (FNR)*. The FNR is an indicating factor for the falsely predicted classes Eq. (7), and FNR is indicating the incorrect negatively classified classes Eq. (8).

$$FPR = \frac{False\ Positives\ (FP)}{False\ Positives\ (FP) + True\ Negatives\ (TN)} \tag{7}$$

$$FNR = \frac{False\ Negatives\ (FN)}{False\ Negatives\ (FN) + True\ Positives\ (TP)} \tag{8}$$

Experimental results showed that our proposed work returns a highly accurate malware detection rate with very little false positives. As shown in Table 5, our method returns an average precision accuracy of 0.990 with an average false positive rate of 0.010, and average false negative rate of 0.010. Results showed high proficiency in the detection and classifying unseen malware with great accuracy.

[5] https://github.com.

Table 5
Average accuracy of Detection.

Dataset	Accuracy measures							
	Precision	Recall	F-measure	Accuracy	TP	FP	FPR	FNR
Ki et al. (2015)	0.999	0.998	0.999	0.999	0.999	0.001	0.001	0.001
Kim (2018)	0.994	0.986	0.990	0.990	0.994	0.007	0.006	0.014
CSDMC	0.987	0.982	0.985	0.985	0.987	0.013	0.012	0.018
Catak and Yazı (2019)	0.980	0.994	0.987	0.987	0.980	0.020	0.020	0.007
Average	**0.990**	**0.990**	**0.990**	**0.990**	**0.990**	**0.010**	**0.010**	**0.010**

Table 6
Detection accuracy of our model to new test samples.

Dataset	Accuracy measures							
	Precision	Recall	F-measure	Accuracy	TP	FP	FPR	FNR
Validation Dataset	0.965	1.000	0.983	0.983	0.965	0.035	0.034	0.000

Table 7
Average Detection accuracy based on non-contextual clustering.

Dataset	Accuracy measures							
	Precision	Recall	F-measure	Accuracy	TP	FP	FPR	FNR
Ki et al. (2015)	0.001	0.418	0.001	0.500	0.001	0.999	0.500	0.001
Kim (2018)	0.093	0.638	0.162	0.520	0.093	0.907	0.489	0.053
CSDMC	0.007	1.000	0.013	0.503	0.007	0.993	0.498	0.000
Catak and Yazı (2019)	0.170	0.828	0.282	0.567	0.170	0.830	0.462	0.035
Average	**0.068**	**0.721**	**0.115**	**0.523**	**0.068**	**0.932**	**0.483**	**0.022**

To prove the validity and efficiency of our proposed model, we tested our model against new malware and goodware datasets. The malware dataset[6] consists of 701 samples, while the goodware dataset[7] consists of 300 samples. As shown in Table 6, our model returns a detection accuracy of 0.983 with a false positive rate of 0.034.

In Section 3.3.2.1, we raised a question about the effectiveness of clustering API functions according to their contextual similarity. What if we consider the DLL files that index the API functions as clusters by themselves? This way, the API calling sequence for malware or goodware can be represented with the highest level of abstraction. We experimented with our model this type of non-contextual clustering approach to compare it with our contextual one. According to the Windows API calling mechanism shown in Fig. 1, we got nine clusters that represent the DLL files.

Results outlined in Table 7 showed that relying on non-contextual clustering yields an average accuracy of 0.523, with an average false positive rate of 0.483. In comparison to our contextual clustering approach, the non-contextual grouping proved to be ineffective in identifying malware API call sequences.

We compared our work to previous works that rely on the API call sequence. As shown in Table 8, we found that our work outperforms other works in terms of malware detection accuracy. With an average accuracy of 0.999, our approach considered the most accurate approach compared to other methods. For example, our work on the dataset provided by Ki et al. (2015) outperforms works of Tran and Sato (2017) with accuracy value of 0.999 compared to 0.998 in Ki et al. (2015), and 0.961 in Tran and Sato (2017).

4.3.1. Mimicry Malware "fake goodware" detection

Although our model provides a distinguishable detection accuracy, some API sequences couldn't be identified correctly. Concerning malware false positives, we found that they have a considerable amount of goodware transitions compared to malware ones. In other words, such malware samples are almost behave as goodware ones. Our model can detect such type of *mimicry malware* or *fake goodware* sequences by tracking the likelihood behavior of a given sequence. Experimentations showed that most malware sequences tend to have a majority of malicious transitions. Even if the malware sequence includes partial *non-malicious* sub-sequences, it doesn't affect at all its *collective likelihood behavior.* On the contrary, in the case of mimicry malware sequences, we noticed that the sequences have a majority of *non-malicious* sub-sequences compared to *malicious* ones. We also noticed that the behavior of fake goodware is continually changing during progressive transitions. Such a contradiction in the fake goodware sequence behavior can be viewed as a sign that the sequence is behaving maliciously.

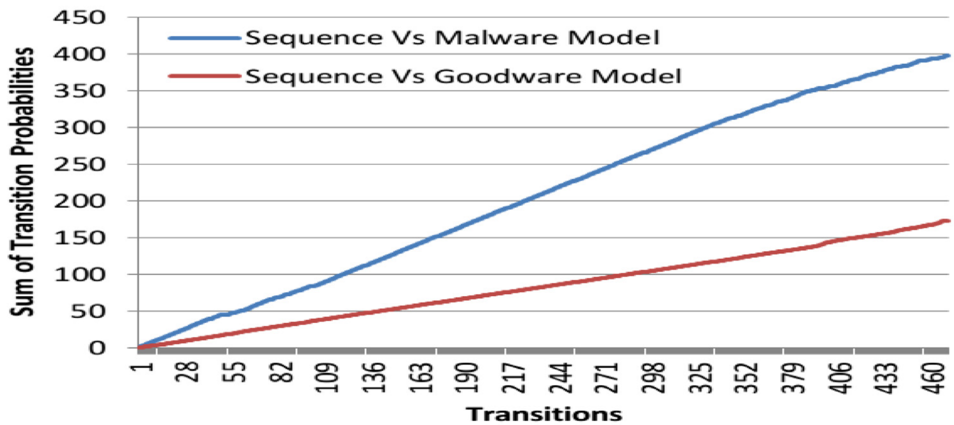
To clarify our idea, consider the following two sequences. The first sequence in Fig. 6a is a transition sequence for malware that our model correctly identified it as malware, while the second transition sequence in Fig. 7a is for a malware sequence that is falsely identified as a goodware. Figs. 6b and 7b are showing the relation between the incremental transitions sequence (x-axis), and the cumulative likelihood summation (y-axis). We omit the API functions for both sequences for space concerns. The first sequence, as shown in Fig. 6a, has a majority of malicious sub-sequences (sub-sequences of consecutive 1 s) compared to non-malicious sub-sequences (sub-sequences of consecutive 0 s). However, the second sequence in Fig. 7.a has a majority of non-malicious sub-sequences (sub-sequences of consecutive 0 s) compared to the malicious sub-sequences (sub-sequences of consecutive 1 s).

We checked the behavior of malware sequence against the malware model and the goodware model, as shown in Fig. 6b. we noticed that, although both behaviors are increasing, nevertheless, they are not growing at the same rate. Yet, they have limited convergence to each other at initial transitions. In other words, there is always a gap that separates both behaviors in progressive transitions. Conversely, in Fig. 7b, we noticed that the sequence behavior against malware and goodware models intercede and converge at some transitions.

[6] https://github.com/duj12/cnn-lstm-based-malware-document-classification.
[7] https://github.com/leocsato/detector_mw.

1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0,1,0,1,1,1,0,0,1,1,1,0,1,1,1,1,1,1,1
,1,1,1,1,1,1,1,1,1,0,1,1,1,1,0,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,0,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
,1,1,1,1,1,1,0,1,1,1,0,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,0,1,1,0,0,0,0,0,0,1,0,0,1,
1,1,1,1,1,1,0,1,1,1,1,0,1,1,1,1,1,1,1,0,1,0,1,1,1,0,1,1,1,1,1,0,1,1,1,0,0,0,1,0,1,0,1,1,1,1,1,1,1,0,1,0,0,1,1,0,0,1,0,0,0,0,1,1,0

(a)

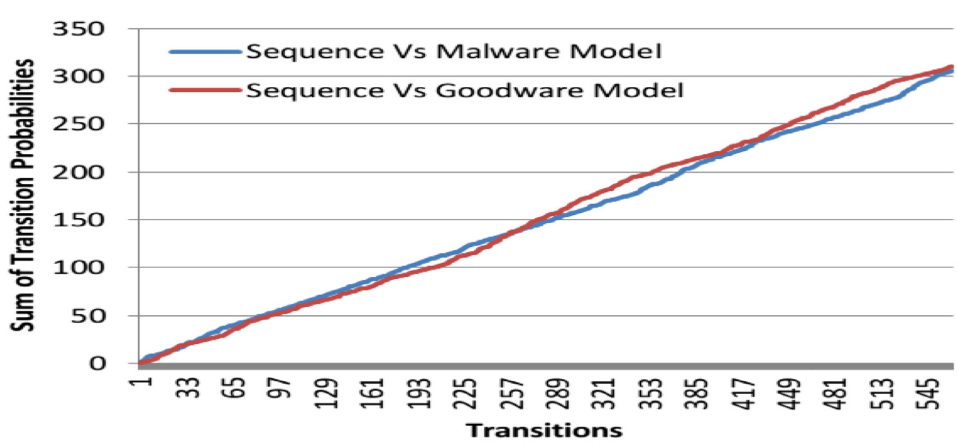


(b)

Fig. 6. (a) Malware sequence that our model correctly identified as malware; (b) Evolutionary summation of likelihood transition sequence in Fig. 6a.

1,1,1,1,1,1,1,1,1,0,1,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,1,1,1,1,1,1,0,1,0,1,1,1,1,1,0,1,1,1,1,1,1,0,1,0,1,1,1,1,1,0,0,0,0,0,0,0,0
,1,1,0,0,0,0,0,0,0,0,1,1,0,0,0,1,1,0,1,1,0,1,1,0,0,1,1,0,1,0,1,1,0,1,1,0,1,1,0,0,1,1,0,0,0,1,1,0,1,1,0,1,1,0,0,1,1,0,1,0,1,1,0,1,1,0,
1,1,0,0,1,1,0,0,0,1,1,0,1,1,0,1,1,0,0,1,1,0,1,0,1,1,0,1,1,0,0,0,0,1,1,0,0,0,0,0,1,1,1,0,1,1,0,1,1,1,1,0,1,1,0,0,1,1,0,1,0,1,1,1,0,1,1
,1,0,1,0,1,0,1,1,0,1,0,0,0,1,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,0,1,1,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,1,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1,1,0,0,0,0,0,0,1,1,1,1,0,0,1,0,0,0,0,0,0
,0,0,1,0,0,0,0,0,1,0,0,1,1,1,1,1,1,1,1,1,1,0,1,0,0,0,0,0,0,1,1,1,1,0,0,1,1,1,1,1,1,1,1,1,1,1,1,0,0,1,1,0,1,1,1,1,1,0,1,1,0,1,1,1,0,1,1,
0,1,0,1,1,0,0,0,0,0,0,0,0,0,1,1,0,1,1,0,0,0,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,1,0,0,0,0,0,0,0,1,0,1,0,0,1,1,0,0,0
,0,0,1,1,0,0,0,0,1,1,1,0,1,1,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,1,1,0,0,0,1,1,1,1,0,1,0,1,1,0,0,0,0,0,1,1,0,0,0,0,0,1,0,0,1,1,1,1,1,1,1,1,
1.1.1.1.1.1.1.1.1.0.1.1.0.1.1.1.1.0.1.1.1.1.1.0.1.0.0.0.0.1

(a)



(b)

Fig. 7. (a) Malware sequence that our model falsely identified as goodware. (b) Evolutionary summation of likelihood transition sequence in Fig. 7a.

Table 8
Comparison with other works.

Study	Method	# of malware	F-measure	Accuracy	Used feature
Alazab et al. (2011)	Static analysis	66,703	0.984	0.985	Frequency of API usage
Sathyanarayan et al. (2008)	Static analysis	800	0.909	0.841	API call sequence
Tian et al. (2010)	Dynamic analysis	1368	0.969	0.973	Frequency of API usage
Sami et al. (2010)	Static analysis	32,000	0.878	0.983	Frequency of API usage
Ye et al. (2007)	Static analysis	17,366	0.941	0.930	API call sequence
Ahmed et al. (2009)	Dynamic analysis	416	–	0.980	API call sequence
Rieck et al., 2011[10]	Dynamic analysis	3133	0.950	–	API call sequence
Qiao et al. (2014)	Dynamic analysis	3131	0.909	–	API call sequence
Qiao et al. (2013)	Dynamic analysis	3131	0.947	–	API call sequence
Ki et al. (2015)	Dynamic analysis	23,080	0.999	0.998	API call sequence
Tran and Sato, 2017)	Dynamic analysis	23,080	–	0.961	API call sequence
Our method	Dynamic analysis	23,080	0.999	0.999	API call sequence
		151	0.990	0.990	API call sequence
		320	0.985	0.985	API call sequence
		7107	0.987	0.987	System call sequences

Table 9
Average false positives detection accuracy.

Dataset	Identification accuracy
Ki et al. (2015)	0.999
Kim (2018)	1.000
CSDMC	1.000
Catak and Yazı (2019)	0.973
Average	**0.993**

The *behavior intersection ratio* (BIR) between behaviors, as indicated in Eq. (9), is utilized as a heuristic that shows how the sequence keeps or changes its behavior when analyzed by our model.

$$BIR\ (sequence) = \frac{\sum_{i=1}^{n}(Malware|\sum p(transition(1:i)) < \ (Goodare|\sum p(transition(1:i))}{Length\ (Sequence)} \quad (9)$$

where n denotes the total number of transitions in a given sequence, the inner sums denotes the accumulation of transition probabilities up to the ith transition in both malware and goodware models. The outer sum denotes the cases where the comparison condition between the two inner sums is evaluated as *true*. The behavior intersection ratio for any sequence is the result of dividing the numerator outer sum by the length of the sequence.

In our analyses, we asserted that a sequence is identified as malware if it has a behaviors intersection ratio of over 10%. We examined our assertion against malware false positives that appeared during our experiments in Table 5. As illustrated in Table 9, our heuristic can detect mimicry malware as a probably malicious sequence with an average identification accuracy of 0.993.

4.4. Evaluating early stage malware prediction

The detection of a malicious sequence requires having the whole API calling sequence in advance. However, it is more effective if we can predict whether a given sequence is malicious or not based on its initial API calling functions. The accuracy of prediction is more significant. Through prediction of whether a call is malicious or not, it becomes possible to block malicious payloads instead of detecting them after their post-execution. Therefore, we can be avoiding repairing the damage caused by malware.

We investigated the possibility to predict whether a given sequence is malicious or not based on its initial API functions calls. The prediction model, as shown in Fig. 8, uses the same models that are used in the detection model. However, the prediction process relies on a part of the whole sequence to perform malware prediction. That is, given a short snapshot of the API call sequence, we proved that our model could effectively predict whether a subsequence is malicious or not. We found that given a snapshot for the API call sequence is considered enough to predict whether the whole sequence is malicious or not. In our work, we experimented and evaluated the performance of prediction when we have a snapshot of size 10,12,30,40, and 50 of consequent API functions.

Results showed that the performance of the system is refined when the length of snapshot increased. In Table 10, the system relied only on the initial 10-API functions in a sequence to predict whether a sample is a malware or not. The system yields an average accuracy of 0.839, with average false positive rate of 0.128, and average false negative rate of 0.080.

We noticed from Table 10 that our model showed a poor prediction performance regarding the dataset in Muller and Guido (2017). When we inspected the dataset, we found that both malware and goodware datasets almost have typical initial API functions. That similarity between API calls causes ambiguity to our model, and hence resulted in the inability to differentiate between goodware and malware sequences.

The prediction accuracy increases when we relied on the initial 20-API functions sequences, as shown in Table 11. The average prediction accuracy increased to 0.888, with an average false positive rate and a false negative rate of 0.105 and 0.060, respectively

The performance of our model prediction accuracy increased when the system relied on the initial 30-API functions in the sequence, as shown in Table 12. The system returned an average accuracy of 0.982, with an average false positive rate of 0.003, and an average false negative rate of 0.033. Table 13 showed more refinement in system performance when the system used the initial 40 API functions in samples. The system showed an average prediction accuracy of 0.993, with an average false positive rate and a false negative rate of 0.000, and 0.014, respectively.

Table 14 also showed a slight refinement to the prediction accuracy performance compared to that in Table 13, when we relied on the initial 50 API functions as prediction features in samples. Our model showed an average accuracy of 0.997, with an average false positive rate of 0.000, and an average false negative rate of 0.007.

We noticed that the performance of our prediction model increased or became stable at some experiments when we rely on more API functions in the samples. However, we noticed also a decrease in the performance of prediction at some experiments. The reason for such decreasing is regarded to the prediction perfor-

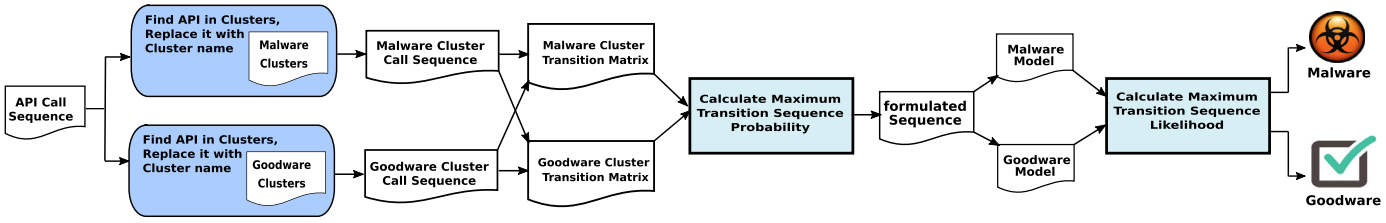


Fig. 8. The proposed Prediction Model.

Table 10
Malware Prediction (first-10 API functions in the sequence).

Dataset	Accuracy Measures							
	Precision	Recall	F-measure	Accuracy	TP	FP	FPR	FNR
Ki et al. (2015)	0.997	0.998	0.998	0.998	0.997	0.003	0.003	0.003
Kim (2018)	0.039	1.000	0.076	0.520	0.039	0.961	0.490	0.000
CSDMC	1.000	0.974	0.987	0.987	1.000	0.000	0.000	0.026
Catak and Yazı (2019)	0.987	0.773	0.867	0.849	0.987	0.013	0.018	0.289
Average	**0.756**	**0.936**	**0.732**	**0.839**	**0.756**	**0.244**	**0.128**	**0.080**

Table 11
Malware Prediction (first-20 API functions in the sequence).

Dataset	Accuracy measures							
	Precision	Recall	F-measure	Accuracy	TP	FP	FPR	FNR
Ki et al. (2015)	1.000	0.998	0.999	0.999	1.000	0.000	0.000	0.002
Kim (2018)	0.276	1.000	0.433	0.638	0.276	0.724	0.420	0.000
CSDMC	1.000	0.950	0.974	0.974	1.000	0.000	0.000	0.053
Catak and Yazı (2019)	1.000	0.844	0.916	0.908	1.000	0.000	0.000	0.184
Average	**0.819**	**0.948**	**0.831**	**0.880**	**0.819**	**0.181**	**0.105**	**0.060**

Table 12
Malware Prediction (first-30 API functions in the sequence).

Dataset	Accuracy measures							
	Precision	Recall	F-measure	Accuracy	TP	FP	FPR	FNR
Ki et al. (2015)	1.000	0.999	0.999	0.999	1.000	0.000	0.000	0.001
Kim (2018)	0.987	1.000	0.993	0.993	0.987	0.013	0.013	0.000
CSDMC	1.000	0.974	0.987	0.987	1.000	0.000	0.000	0.026
Catak and Yazı (2019)	1.000	0.905	0.950	0.947	1.000	0.000	0.000	0.105
Average	**0.997**	**0.970**	**0.982**	**0.982**	**0.997**	**0.003**	**0.003**	**0.033**

Table 13
Malware Prediction (first-40 API functions in the sequence).

Dataset	Accuracy measures							
	Precision	Recall	F-measure	Accuracy	TP	FP	FPR	FNR
Ki et al. (2015)	1.000	0.999	0.999	0.999	1.000	0.000	0.000	0.001
Kim (2018)	1.000	1.000	1.000	1.000	1.000	0.000	0.000	0.000
CSDMC	1.000	1.000	1.000	1.000	1.000	0.000	0.000	0.000
Catak and Yazı (2019)	1.000	0.950	0.974	0.974	1.000	0.000	0.000	0.053
Average	**1.000**	**0.987**	**0.993**	**0.993**	**1.000**	**0.000**	**0.000**	**0.014**

Table 14
Malware Prediction (first-50 API functions in the sequence).

Dataset	Accuracy measures							
	Precision	Recall	F-measure	Accuracy	TP	FP	FPR	FNR
Ki et al. (2015)	1.000	0.999	1.000	1.000	1.000	0.000	0.000	0.001
Kim (2018)	1.000	1.000	1.000	1.000	1.000	0.000	0.000	0.000
CSDMC	1.000	1.000	1.000	1.000	1.000	0.000	0.000	0.000
Catak and Yazı (2019)	1.000	0.974	0.987	0.987	1.000	0.000	0.000	0.026
Average	**1.000**	**0.993**	**0.997**	**0.997**	**1.000**	**0.000**	**0.000**	**0.007**

mance on Catak et al. (Muller and Guido 2017) dataset. Due to the high sub-sequence overlap between goodware and malware system call sequences, our predictor performance could have some degrading performance in terms of accuracy, false positive/negative rate.

5. Conclusion

In this paper, we introduced a malware detection and prediction approach based on the contextual understanding of the API call sequence. We utilized word-embedding to get the contextual relationships between API functions in malware and goodware sequences. Through clustering API functions that are contextually similar, a new method is proposed that group API calls in a limited number of clusters. We generated a semantic chain transition matrix which depicts the actual relation between API functions. Markov chain representation is used to model the behavioral execution models for malware and goodware. The resulted models were considered as our representative features, which describe malware and goodware. We proved that there were significant distinctions between the contextual execution behavior of malware and goodware. Results showed a very promising accuracy in detecting and predicting malware. Our model returned an average detection precision and accuracy of 0.990, with a false positive rate of 0.010. The model is also provided an early prediction of malware according to the given sequence. We showed that we could rely on a limited API sub-sequence to predict whether the whole sequence is malicious or not. Our proposed model can be used to block malicious payloads instead of detecting them after their post-execution and avoiding repairing the damage. Results showed that our models returned an average prediction accuracy of 0.997, with an average false positive rate of 0.000 and a false negative rate of 0.007.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

The following grants are acknowledged for the financial support for this research. Grant of SGS no. SP2020/78, VSB Technical University of Ostrava.

References

Ahmed, F., Hameed, H., Shafiq, M.Z, Farooq, M., 2009. Using spatio-temporal information in API calls with machine learning algorithms for malware detection. In: Proceedings of the 2nd ACM Workshop on Security and Artificial Intelligence. ACM, pp. 55–62.

Alaeiyan, M., Parsa, S., Conti, M., 2019. Analysis and classification of context-based malware behavior. Comput. Commun. 136, 76–90.

Alami, N., Meknassi, M., En-nahnahi, N., 2019. Enhancing unsupervised neural networks based text summarization with word embedding and ensemble learning. Expert Syst. Appl. 123, 195–211.

Alazab, M., Venkatraman, S., Watters, P., Alazab, M., 2011. Zero-day malware detection based on supervised learning algorithms of API call signatures. In: Proceedings of the Ninth Australasian Data Mining Conference-Volume 121. Australian Computer Society, Inc., pp. 171–182.

Andrews, B., Davis, RA., Jay Breidt, F., 2006. Maximum likelihood estimation for all-pass time series models. J. Multivar. Anal. 97 (7), 1638–1659.

Burnap, P., French, R., Turner, F., Jones, K., 2018. Malware classification using self organising feature maps and machine activity data. Comput. Secur. 73, 399–410.

Catak, F.O., and Yazı, A.F. "A benchmark API call dataset for windows PE malware classification." (2019) arXiv:1905.01999.

Cesare, S., Xiang, Y., Zhou, W., 2013. Control flow-based malware variant detection. IEEE Trans. Depend. Secure Comput. 11 (4), 307–317.

Choi, C., Esposito, C., Lee, M., Choi, J., 2019. Metamorphic malicious code behavior detection using probabilistic inference methods. Cognit. Syst. Res. 56, 142–150.

Choudhary, S.P., Vidyarthi, M.D., 2015. A simple method for detection of metamorphic malware using dynamic analysis and text mining. Procedia Comput. Sci. 54, 265–270.

Cuckoo Sandbox Automated malware analysis. https://cuckoosandbox.org (Accessed 8 December 2019).

CWSandbox. https://cwsandbox.org/ (Accessed 8 December 2019).

Ding, Y., Xia, X., Chen, S., Li, Ye, 2018. A malware detection method based on family behavior graph. Comput. Secur. 73, 73–86.

Elhadi, AAE., Maarof, M.A., Barry, B.I., 2013. Improving the detection of malware behaviour using simplified data dependent API call graph. Int. J. Secur. Appl. 7 (5), 29–42.

Fan, M., Liu, J., Luo, X., Chen, K., Tian, Z., Zheng, Q., Liu, T., 2018. Android malware familial classification and representative sample selection via frequent subgraph analysis. IEEE Trans. Inf. Forensics Secur. 13 (8), 1890–1905.

Firdaus, A., Anuar, N.B., Razak, M.F.A.B., Sangaiah, A.K., 2018. Bio-inspired computational paradigm for feature investigation and malware detection: interactive analytics. Multimed. Tools Appl. 77 (14), 17519–17555.

Galal, H.S., Mahdy, Y.B., Atiea, M.A., 2016. Behavior-based features model for malware detection. J. Comput. Virol. Hacking Tech. 12 (2), 59–67.

Gandotra, E., Bansal, D., Sofat, S., 2014. Malware analysis and classification: a survey. J. Inf. Secur. 5 (02), 56.

Gupta, S., Sharma, H., Kaur, S., 2016. Malware characterization using windows API call sequences. In: International Conference on Security, Privacy, and Applied Cryptography Engineering. Springer, Cham, pp. 271–280.

Han, W., Xue, J., Wang, Y., Huang, Lu, Kong, Z., Mao, L., 2019. MalDAE: detecting and explaining malware based on correlation and fusion of static and dynamic characteristics. Comput. Secur. 83, 208–233.

Ketkar, N., 2017. Deep Learning with Python. Apress.

Ki, Y., Kim, E., Kim, H.K., 2015. A novel approach to detect malware based on API call sequence analysis. Int. J. Distrib. Sens. Netw. 11 (6), 659101.

Kim, C.W. "NtMalDetect: a machine learning approach to malware detection using native API system calls." (2018) arXiv:1802.05412.

Kim, H.-J., Kim, J.-H., Kim, J.-T., Kim, I.-K., Chung, T.-M., 2016. Feature-Chain based malware detection using multiple sequence alignment of API call. IEICE Trans. Inf. Syst. 99 (4), 1071–1080.

Lee, T., Choi, B., Shin, Y., Kwak, J., 2018. Automatic malware mutant detection and group classification based on the n-gram and clustering coefficient. J. Supercomput. 74 (8), 3489–3503.

Lee, T., Kwak, J., 2016. Effective and reliable malware group classification for a massive malware environment. Int. J. Distrib. Sens. Netw. 12 (5), 4601847.

Lin, Z., Xiao, F., Sun, Yi, Ma, Y., Xing, C.-C., Huang, J., 2018. A secure encryption-based Malware detection system. TIIS 12 (4), 1799–1818.

Liu, W., Ren, P., Liu, K., Duan, H., 2011. Behavior-based malware analysis and detection. In: 2011 First International Workshop on Complexity and Data Mining. IEEE, pp. 39–42.

Lu, H., Zhao, B., Su, J., Xie, P., 2014. Generating lightweight behavioral signature for malware detection in people-centric sensing. Wirel. Pers. Commun. 75 (3), 1591–1609.

Martinčić-Ipšić, S., Miličić, T., 2019. The influence of feature representation of text on the performance of document classification. Appl. Sci. 9 (4), 743.

Muller, AC., Guido, S., 2017. Introduction to Machine Learning With Python: a Guide for Data Scientists. O'Reilly Media.

Ndibanje, B., Kim, KiH, Kang, Y.J., Kim, HHo, Kim, T.Y., Lee, H.J., 2019. Cross-Method--Based analysis and classification of malicious behavior by api calls extraction. Appl. Sci. 9 (2), 239.

Pektaş, A., Acarman, T., 2017. Malware classification based on API calls and behaviour analysis. IET Inf. Secur. 12 (2), 107–117.

Qiao, Y., Yang, Y., He, J., Tang, C., Liu, Z., 2014. CBM: free, automatic malware analysis framework using API call sequences. In: Knowledge Engineering and Management. Springer, Berlin, Heidelberg, pp. 225–236.

Qiao, Y., Yang, Y., Ji, L., He, J., 2013. Analyzing malware by abstracting the frequent itemsets in API call sequences. In: 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications. IEEE, pp. 265–270.

Ravi, C., Manoharan, R., 2012. Malware detection using windows api sequence and machine learning. Int. J. Comput. Appl. 43 (17), 12–16.

Rezaeinia, S.M., Rahmani, R., Ghodsi, A., Veisi, H., 2019. Sentiment analysis based on improved pre-trained word embeddings. Expert Syst. Appl. 117, 139–147.

Rieck, K., Trinius, P., Willems, C., Holz, T., 2011. Automatic analysis of malware behavior using machine learning. J. Comput. Secur. 19 (4), 639–668.

Salehi, Z., Sami, A., Ghiasi, M., 2017. MAAR: robust features to detect malicious activity based on API calls, their arguments and return values. Eng. Appl. Artif. Intell. 59, 93–102.

Sami, A., Yadegari, B., Rahimi, H., Peiravian, N., Hashemi, S., Hamze, A., 2010. Malware detection based on mining API calls. In: Proceedings of the 2010 ACM Symposium on Applied Computing. ACM, pp. 1020–1025.

Sathyanarayan, V.S, Kohli, P., Bruhadeshwar, B., 2008. Signature generation and detection of malware families. In: Australasian Conference on Information Security and Privacy. Springer, Berlin, Heidelberg, pp. 336–349.

Sheneamer, A., Roy, S., Kalita, J., 2018. A detection framework for semantic code clones and obfuscated code. Expert Syst. Appl. 97, 405–420.

Syakur, M.A., Khotimah, B.K., Rochman, E.M.S., Satoto, B.D., 2018. Integration k-means clustering method and elbow method for identification of the best customer profile cluster. IOP Conference Series: Materials Science and Engineering, 336. IOP Publishing.

Tajoddin, A., Jalili, S., 2018. HM3alD: polymorphic Malware detection using program behavior-aware hidden Markov model. Appl. Sci. 8 (7), 1044.

Tian, R., Islam, R., Batten, L., Versteeg, S., 2010. Differentiating malware from cleanware using behavioural analysis. In: 2010 5th International Conference on Malicious and Unwanted Software. IEEE, pp. 23–30.
Tran, T.K., Sato, H., 2017. NLP-based approaches for malware classification from API sequences. In: 2017 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES). IEEE, pp. 101–105.
Tungjitviboonkun, T., Suttichaya, V., 2017. Complexity reduction on API call sequence alignment using unique API word sequence. In: 2017 21st International Computer Science and Engineering Conference (ICSEC). IEEE, pp. 1–5.
Ucci, D., Aniello, L., Baldoni, R., 2018. Survey of machine learning techniques for malware analysis. Comput. Secur..
Xiao, F., Lin, Z., Sun, Yi, Ma, Y., 2019. Malware detection based on deep learning of behavior graphs. Math. Probl. Eng. 2019.
Ye, Y., Wang, D., Li, T., Ye, D., 2007. IMDS: intelligent malware detection system. In: Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, pp. 1043–1047.
Zelinka, I., Amer, E., 2019. An ensemble-based malware detection model using minimum feature set. Mendel 25 (2), 1–10.
Zhao, Y., Bo, B., Feng, Y., Xu, C.Y.U., Yu, B., 2019. A feature extraction method of hybrid gram for malicious behavior based on machine learning. Secur. Commun. Netw. 2019.

Eslam Amer is an associate professor of computer science. Currently, he is working as a postdoctoral research fellow at faculty of electrical engineering and computer science – technical university of Ostrava – Czech Republic. Eslam is working on malware analysis using natural language processing. His main research interests are natural language processing, information retrieval.

Ivan Zelinka is currently working as a professor at the Technical University of Ostrava (VŠB-TU), Faculty of Electrical Engineering and Computer Science and national supercomputing center IT4 Innovations. Dr. Zelinka has also participated in numerous grants and two EU projects as a member of the team (FP5 - RESTORM) and as supervisor of (FP7 - PROMOEVO) of the Czech team. Currently, he is the head of the Department of Applied Informatics and throughout his career he has supervised numerous MSc. and Bc. diploma theses in addition to his role of supervising doctoral students, including students from abroad. He was awarded the Siemens Award for his Ph.D. thesis and received an award from the journal Software news for his book about artificial intelligence. Ivan Zelinka is a member of the British Computer Society, IEEE (a committee of Czech section of Computational Intelligence), and serves on international program committees of various conferences and three international journals (Soft Computing, SWEVO, Editorial Council of Security Revue.) He is the author of numerous journal articles as well as books in Czech and the English language.