Journal of Network and Computer Applications 218 (2023) 103704

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

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

API-MalDetect: Automated malware detection framework for windows based on API calls and deep learning techniques

Pascal Maniriho *, Abdun Naser Mahmood, Mohammad Jabed Morshed Chowdhury

Department of Computer Science and Information Technology, La Trobe University, Melbourne, VIC, Australia

ARTICLE INFO

MSC:
00-01
99-00

Keywords:
Malware analysis
Malware detection
Dynamic analysis
Convolutional neural network
API calls
Machine learning
Deep learning

ABSTRACT

This paper presents API-MalDetect, a new deep learning-based automated framework for detecting malware attacks in Windows systems. The framework uses an NLP-based encoder for API calls and a hybrid automatic feature extractor based on convolutional neural networks (CNNs) and bidirectional gated recurrent units (BiGRU) to extract features from raw and long sequences of API calls. The proposed framework is designed to detect unseen malware attacks and prevent performance degradation over time or across different rates of exposure to malware by reducing temporal bias and spatial bias during training and testing. Experimental results show that API-MalDetect outperforms existing state-of-the-art malware detection techniques in terms of accuracy, precision, recall, F1-score, and AUC-ROC on different benchmark datasets of API call sequences. These results demonstrate that the ability to automatically identify unique and highly relevant patterns from raw and long sequences of API calls is effective in distinguishing malware attacks from benign activities in Windows systems using the proposed API-MalDetect framework. API-MalDetect is also able to show cybersecurity experts which API calls were most important in malware identification. Furthermore, we make our dataset available to the research community.

1. Introduction

As Internet-based applications continue to shape various businesses around the globe, malware threats have become a severe problem for computing devices such as desktop computers, smartphones, local servers, and remote servers. According to statistics, it is expected that in this year (2023) the total number of devices connected to IP networks will be around 29.3 billion (Cisco, 2020), resulting in a massive interconnection of various networked devices globally. As the number of connected devices continues to rise exponentially, this has also become a motivating factor for cyber-attackers to develop new advanced malware programs that disrupt, steal sensitive data, damage, and exploit various vulnerabilities. The widespread use of different malware variants makes the existing security systems less effective whereby, millions of devices are infected by various forms of malware such as worms, ransomware, backdoors, computer viruses, and Trojans (Jovanovic, 2022; Maniriho et al., 2022). Accordingly, there has been a significant increase in new malware targeting Windows devices over the last decade. For instance, the number of reported malware increased by 23% (9.5 million) (Drapkin, 2022) from 2020 to 2021. About 107.27 million of new malware samples were created to compromise Windows devices in 2021, showing an increase of 16.53 million of malware samples over 2020 (with an average of 328,073 malware samples produced daily) (Drapkin, 2022). According to the AtlasVPN report (Ruth, 2023), more than 95% of all malware attacks were against Windows desktop devices in 2022.

As a solution to address malware attacks, the application of signature-based malware detection systems such as anti-virus programs that rely on a database of signatures extracted from the previously identified malware samples has become popular (Anon, 2023e). In static malware analysis, signatures are malware's unique identities which are extracted from malware without executing the suspicious program (Zhang et al., 2019; Naik et al., 2021). Some of the static-based malware detection techniques were implemented using static signatures such as printable strings, opcode sequences, and static API calls (Singh and Singh, 2020; Sun et al., 2019; Huda et al., 2016). As signature-based malware detection systems rely on previously seen signatures to detect malware threats, they have become ineffective due to a huge number of new malware variants coming out every day (Alazab et al., 2010). Moreover, static-based techniques are unable to detect obfuscated malware (malware with evasion behaviors) (Zelinka and Amer, 2019; Anon, 2019). Such obfuscated malware include Agent Tesla, BitPaymer, Zeus Panda, and Ursnif, to name a few (Anon, 2022a).

* Corresponding author.
E-mail address: P.Maniriho@latrobe.edu.au (P. Maniriho).

https://doi.org/10.1016/j.jnca.2023.103704
Received 15 April 2023; Received in revised form 20 June 2023; Accepted 16 July 2023
Available online 22 July 2023

In contrast to static analysis-based techniques, dynamic analysis is developed to counter obfuscation techniques. Dynamic or behavior-based malware detection techniques are implemented based on the dynamic malware analysis approach which allows monitoring the suspicious program's behaviors and vulnerability exploits by executing it in a virtual environment (Udayakumar et al., 2017). Dynamic analysis can reveal malware's behavioral characteristics such as running processes, registry key changes, web browsing history (such as DNS queries), malicious IP addresses, loaded DLLs, API calls, and changes in the file system (Han et al., 2019; Maniriho et al., 2022). Vemparala et al.'s work (Vemparala et al., 2019) demonstrated that the dynamic-based malware detection technique outperforms the static-based technique in many cases, after comparing their performance using extracted dynamic API calls and static sequence of opcode features. The dynamic analysis approach can produce relevant feature representations that reveal what a malware intends to perform in the victim's system (Suaboot et al., 2020; Maniriho et al., 2022). As in many cases benign software programs evolve (e.g., when new vulnerabilities are discovered, new services are getting created), it is important to mention that malicious software programs (malware) also evolve at the same, i.e., the behaviors of malware change over time. Therefore, new features such as API call functions that can be relevant to developing more accurate and robust malware detection techniques for the present tasks could emerge from these new malware behaviors, which creates the need to perform regular analysis of emerging malware in order to capture new features. Nevertheless, extracting dynamic/behavioral features from malware executable files is a critical task as malware can damage organizational resources such as corporate networks, confidential information, and other resources when they escape the analysis environment (Suaboot et al., 2020; Maniriho et al., 2022). This makes the extraction of up-to-date feature representation of malware a challenging task (Mimura and Ito, 2022; Suaboot et al., 2020). In the case of API calls, it also remains a challenge to obtain relevant API call features as the number of API calls made by malware executable files is relatively long which makes their processing difficult (Suaboot et al., 2020).

Given the potential of dynamic malware analysis, this work is focused on analyzing dynamic-based API call sequences extracted from Windows executable files to identify malware attacks. Existing techniques for dynamic-based malware detection have used machine learning (ML) algorithms to detect malware attacks. These algorithms learn from given data and make predictions on new data. Over the last decade, the use of ML-based algorithms has become more prevalent in the field of cybersecurity such as malware detection (Apruzzese et al., 2018; Gibert et al., 2020). However, conventional ML techniques rely on manual feature extraction and selection process, which requires human expert domain knowledge to derive relevant or high-level patterns (features) to be used to represent a set of malware and benign files (Gibert et al., 2022; Le et al., 2018). This process is known as manual feature engineering and is time-consuming and error-prone as it depends on a manual process, considering the current plethora of malware production. On the other hand, deep learning (DL) algorithms have also emerged for malware detection (Bostami and Ahmed, 2020; Li et al., 2022a). Different from conventional ML techniques, DL algorithms can perform automatic feature extraction (Tirumala et al., 2020). Nonetheless, the majority of existing ML and DL-based techniques operate as black boxes (Moraffah et al., 2020; Mehrabi et al., 2019; Maniriho et al., 2022). These models receive input X which is processed through a series of complex operations to produce Y as the predicted outcome/output. Unfortunately, such operations cannot be interpreted by humans, as they fail to provide human-friendly insights and explanations (for example: which features contributed to the final predicted outcome) (Moraffah et al., 2020; Mehrabi et al., 2019). Practically, it is ideal to have an ML or DL-based malware detection technique that can detect the presence of malicious files with high detection accuracy. However, the prediction of such a technique should not be blindly trusted, but instead, it is important to have confidence about the features or attributes that contributed to the prediction. By using explainable modules researchers and security analysts can derive more insights from the detection techniques/models and understand the logic behind the final model's predictions (Ribeiro et al., 2016).

Therefore, in direction to address the problem of inefficiency observed in the existing malware detection techniques, we propose API-MalDetect, a new DL-based automated framework for detecting malware attacks in Windows. The motivation for using deep learning is to automatically identify unique and high relevant patterns from raw and long sequences of API calls which distinguish malware attacks from benign activities. API-MalDetect uses an encoder based on natural language text processing (NLP) techniques to construct numerical representations and embedding vectors of API call sequences based on their semantic relationships. It also uses an automatic hybrid feature extractor based on a convolutional neural network (CNN) and bidirectional gated recurrent unit (BiGRU). Although CNN has the potential to extract high-level text-based features, it extracts local features (API call in our case) which lack contextual semantic information between them due to the limitation of the sliding filter (sliding window). In order to address this problem, we combine CNN and BiGRU to improve the feature extraction process. Specifically, BiGRU receives local features of API calls extracted by CNN and processes them to capture more contextual semantic information between them. Hence, combining CNN and BiGRU techniques allows us to effectively capture relevant features that can be used in detecting malicious executable files. Features generated by the CNN-BiGRU feature extractor are fed to a fully connected neural network (FCNN) module for malware classification.

We have also integrated LIME into our framework to make it explainable. LIME is a framework for interpreting ML and DL black box models and was proposed by Ribeiro et al. (2016). It allows API-MalDetect to produce explainable predictions, which reveal feature importance, i.e., LIME produces features of API calls that contributed to the final prediction of a particular benign or malware executable file, and to the best of our knowledge, none of the previous techniques has attempted to use LIME on API call features. Explainable results produced by LIME can help cybersecurity analysts or security practitioners to better understand API-MalDetect's predictions and to make decisions based on explainable insights. It is worth noting that we have also taken into consideration the practical constraints suggested in Pendlebury et al. (2019) to address the issue of temporal bias and spacial bias encountered in previous works such as (Suaboot et al., 2020; Singh and Singh, 2020; Amer et al., 2021). Temporal bias occurs when the time-based split of samples is not considered during training while spatial bias refers to the unrealistic distribution of malware samples over benign samples in the test set.

Contributions

This paper makes several significant contributions to the field of malware detection in Windows systems.

1. We introduce a new benchmark dataset for evaluating malware detection techniques. The dataset is created by using the dynamic analysis approach to extract sequences of API calls from both benign and malware executable files. We make this dataset publicly available for the research community to use in their experiments.
2. We propose a new deep learning-based automated framework called API-MalDetect for detecting malware attacks in Windows. The framework uses an NLP-based encoder for API calls and a hybrid automatic feature extractor based on deep learning techniques such as CNNs and BiGRUs to extract features from raw and long sequences of API calls. This approach allows us to automatically identify unique and highly relevant patterns from API call sequences that distinguish malware attacks from benign activities.

3. We introduce practical experimental factors for training and testing malware detection techniques to avoid temporal bias and spatial bias in the experiments. These factors include using different time periods for training and testing, and using different proportions of benign and malware samples. We demonstrate that API-MalDetect is effective in detecting unseen malware attacks even when trained on data from a different time period or tested on a different machine with different hardware configurations.
4. We evaluate the performance of API-MalDetect on benchmark datasets of API call sequences. Experimental results show that API-MalDetect outperforms existing state-of-the-art malware detection techniques Karbab and Debbabi (2019), Li et al. (2022a), Qin et al. (2020), Xiaofeng et al. (2019) and Avci et al. (2023) in terms of accuracy, precision, recall, F1-score, and AUC-ROC. These results demonstrate that the ability to automatically identify unique and highly relevant patterns from raw and long API call sequences effectively distinguishes malware attacks from benign activities in Windows systems using the proposed API-MalDetect framework.
5. Finally, by using LIME, API-MalDetect is able to produce local interpretability and explainability for its predictions. This allows security practitioners to understand how the framework makes its predictions and which API call features are more important in distinguishing malware attacks from benign activities.

Structure: The remaining part of this paper is structured as follows. Section 2 presents the background and Section 3 discusses the related works. Section 4 presents the proposed framework while Section 5 discusses the experimental results. Section 6 presents limitations and future work. The conclusion of this work is provided in Section 7.

2. Background

This section presents basic background on Windows application programming interface (Win API) and executables' API calls monitoring. Moreover, it discusses the use of deep learning algorithms for malware detection.

2.1. Windows API

The Windows application programming interface (API), also known as Win32 API, is a collection of all API functions that allow Windows-based applications/programs to interact with the Microsoft Windows OS (Kernel) and hardware (Stenne, 2021; Silberschatz Abraham, 2018; Uppal et al., 2014). Apart from some console programs, all Windows-based applications must employ Windows APIs to request the operating system to perform certain tasks such as opening and closing a file, displaying a message on the screen, creating, writing content to files, and making changes in the registry. This implies that both system resources and hardware cannot be directly accessed by a program, but instead, programs need to accomplish their tasks via Win32 API. All available API functions are defined in the dynamic link libraries (DLLs), i.e., in .dll files included in C:\Windows\System32*. For example, many commonly used libraries include Kernel32.dl, User32.dll, Advapi32.dll, Gdi32.dll, Hal.dll, and Bootvid.dll (Microsoft, 2021).

2.2. API calls monitoring

Generally, any Windows-based program performs its task by calling some API functions. This functionality makes Win32 API one of the important and core components of the Windows OS as well as an entry point for malware programs targeting the Windows platform since the API also allows malware programs to execute their malicious activities. Thus, monitoring and analyzing Windows API call sequences gives the behavioral characteristics that can be used to represent benign and malware programs (Ammar Ahmed E. Elhadi, 2013; Ki et al., 2015). API calls analysis reveals a considerable representation of how a given malware program behaves. Therefore, monitoring the program's API call sequences is by far one of the effective ways to observe if a particular executable program file has malicious or normal behaviors (Suaboot et al., 2020; Amer and Zelinka, 2020).

2.3. Deep learning algorithms

Deep learning algorithms are subsets of machine learning techniques that use artificial neural network architectures to learn and discover interesting patterns/features from data. DL network architectures can handle big datasets with high dimensions and can perform automatic extraction of high-level features (Sharma et al., 2021; Yuan and Wu, 2021). DL algorithms are designed to learn from both labeled and unlabeled datasets and can produce highly accurate results with low false-positive rates (Najafabadi et al., 2015). The multi-layered structure (network with many hidden layers) adopted by DL algorithms gives them the ability to learn relevant data representations through which low-level features are captured by lower layers and high-level abstract features are extracted by higher layers (Rafique et al., 2020; Pinhero et al., 2021). The next subsections introduces CNN and recurrent neural networks, which are some of the popular categories of DL algorithms that we use to design our framework.

2.3.1. Convolutional neural network

Convolutional neural network is a category of DL techniques that gained popularity over the last decades. Inspired by how the animal visual cortex is organized (Hubel and Wiesel, 1968; Fukushima, 1979), CNN was mainly designed for processing data represented in grid patterns such as images and has been successfully used to solve computer vision problems (Khan et al., 2018). Recently, CNN has been also applied to detect malware attacks based on binary images of executable files (Chaganti et al., 2022; Tekerek and Yapici, 2022). Similar to other DL algorithms, it can automatically learn and extract high-level feature representation from data. Two-dimensional CNN and one-dimensional CNN (1-D CNN) are the two main versions of the CNN algorithm with the Two-dimensional CNN algorithm being mainly applied for images (Khan et al., 2018; Tekerek and Yapici, 2022), while 1-D CNN was designed for processing one-dimensional data such as times series and sequential data (Kim, 2014; Fesseha et al., 2021). One-dimensional CNN is less computationally expensive compared to 2-D CNN and in many cases, it does not require graphics processing units (GPUs) as it can be implemented on a standard computer with a CPU (making it much faster than 2-D CNN) (Kiranyaz et al., 2021). One-dimensional CNN architectures have been also successful in modeling various tasks and solving natural language processing (NLP) problems. For example, they were successfully applied to perform text classification (Kim, 2014) and sentiments classification (Liu et al., 2020).

2.3.2. Recurrent neural networks

A Recurrent neural network (RNN) is a type of DL algorithm suitable for modeling sequential data using a memory function that allows it to discover relevant patterns from data (Medsker and Jain, 1999). Despite their performance, traditional/classic RNNs suffer from vanishing gradients (also known as gradient explosion) and are unable to process long sequences (Lynn et al., 2019). To address this problem, Hochreiter and Schmidhuber (1997) proposed Long short-term memory (LSTM), an improved RNN algorithm that performs well on long sequences. The Gated Recurrent Unit (GRU) was later implemented by Cho et al. (2014) based on LSTM. A GRU network architecture uses a reset gate and update gate to decide which information to be passed to the output. Furthermore, it is important to note that GRU uses a simple network architecture and has shown better performance over regular LSTMs (Chung et al., 2014). Bidirectional GRU (BiGRU) is a variant of GRU that models information in two directions (right and left direction) (Vukotić et al., 2016) and studies have shown better performance with reversed sequences, making it also ideal when modeling sequences of data (Vukotić et al., 2016).

3. Related works

There have been significant efforts in recent works on malware detection through dynamic-based malware analysis. The work in Ki et al. (2015) has adopted a DNA sequence alignment approach to design a dynamic analysis-based method for malware detection. Their experimental outcome revealed that some malicious files possess common behaviors/API call functions despite their categories which may be different. In addition, their study has also indicated that new malware can be detected by identifying and matching the presence of certain API calls as many malware programs perform malicious activities using almost similar API calls. However, the limitation of DNA sequence approaches is that they are prone to consuming many resources and require high execution time, making them computationally expensive given the high volume of emerging malware datasets. Singh and Singh (2020) extracted API calls from benign and malware executable files using dynamic analysis in the Cuckoo sandbox. They computed Shannon entropy over API call features to determine their randomness and then processed them using count factorization to obtain features that were used to train a Random Forest-based malware classifier. While their proposed approach shows an improvement in accuracy, it is worth mentioning that the count vectorization model applied while processing API calls does not preserve semantic relationship/similarity between features (Saket, 2021; Mandelbaum and Shalev, 2016).

Pirscoveanu et al. (2015) used Windows API calls to implement a malware classification system that achieved a detection accuracy of 98%. Malicious features were extracted from about 80,000 malware files including four malware categories (Trojans, rootkit, adware, and potentially unwanted programs) downloaded from VirusTotal and VirusShare. Looking at the detection outcome, their approach only performs well when detecting Trojans. Morato et al. (2018) presented a method for detecting ransomware attacks based on network traffic data. In the work presented by Suaboot et al. (2020), a subset of API call features was extracted from API call sequences using a sub-curve Hidden Markov Model (HMM) feature extractor. Only six malware families (Keyloggers, Zeus, Rammit, Lokibot, Ransomware, and Hivecoin) with data exfiltration behaviors were used for the experimental evaluation. Different malware detection techniques based on ML algorithms such as Random Forest (RF), J48, and SVM were built using the extracted features. Nevertheless, their method was only limited to evaluating executable program files, which exfiltrate confidential information from the compromised systems. In addition, with 42 benign and 756 malware executable programs used in their experimental analysis, there is a significant class imbalance in their dataset which could lead to the model failing to predict and identify samples from minority classes despite its good performance. A fuzzy similarity algorithm was employed by Lajevardi et al. (2022) to develop dynamic-based malware detection techniques based on API calls.

The longest common substring and longest common subsequence methods for malware detection were suggested in Mira et al. (2016). Both methods were trained on API call sequences, which were captured from 1500 benign and 4256 malware during dynamic analysis. A Malware detection approach based on a sequence alignment approach and API calls captured by the Cuckoo sandbox was designed in the work proposed by Cho et al. (2016). The analysis was carried out using 150 malware belonging to ten malware variants while malware detection was achieved by computing similarity between files based on extracted sequences of API calls. Their experimental results show that similar behaviors of malware families can be found by identifying a common list of invoked API call sequences generated during the execution of the executable program. Nonetheless, this method cannot be suitable for high-speed malware detection as it fully relies on a pairwise sequence alignment approach, which introduces overheads. Several graph-based techniques for malware detection were proposed in the previous studies Wüchner et al. (2019), Jha et al. (2013), Blokhin et al. (2013), Hellal et al. (2020), Ding et al. (2018) and Pei et al. (2020), to mention a few. Although graph-based techniques can achieve good performance, the complexity of graph matching is one of their major issues, i.e., as the graph's size increases the matching complexity, the detection accuracy of a given detection model decreases (Singh and Singh, 2020). However, it is worth noting that it is harder for a cyber attacker to modify the behaviors of a malware detection model based on graph method (Hellal et al., 2020).

In the work presented in Tran and Sato (2017), NLP techniques were applied to process sequences of API calls which were fed to the detection technique. API calls were processed using the n-gram method and the weights were assigned to each API call feature using the term frequency-inverse document frequency (TF-IDF) model. The work in Karbab and Debbabi (2019) has also relied on TF-IDF to transform input features for machine learning algorithms such as CART, ETrees KNN, RF, SVM, and XGBoost. Another work in Catak et al. (2020) has used LSTM and TF-IDF model to build a behavior-based malware detection using API call sequences extracted using the Cuckoo sandbox in a dynamic analysis environment. TF-IDF and Anti-colony optimization (Swarm algorithm) were used to implement a graph-based malware detection method based on dynamic features of API calls extracted from executable files (Amer et al., 2022). Unfortunately, like the count vectorization approach, the TF-IDF approach does not reveal or preserve the semantic relationship/similarity that exists between words. In the case of malware detection, this would be the similarity between API calls or another text-based feature such as file name, dropped messages, network operations such as contacted hostnames, web browsing history, and error message generated while executing the executable program file.

Liu and Wang (2019) used Bidirectional LSTM (BiLSTM) to Build an API call-based approach that classifies malware attacks. ALL sequences of API calls were extracted from 21,378 executable files and were processed using the word2vec model. In Li et al. (2022b) a graph convolutional network (GCN) model for malware classification was built using sequences of API calls. Features were extracted using principal component analysis (PCA) and Markov Chain. The work in Maniath et al. (2017) proposed an LSTM-based model that identifies ransomware attacks based on behavioral API calls from Windows EXE files generated through dynamic analysis in the Cuckoo sandbox. Chen et al.'s work (Chen et al., 2022) proposed different malware detection techniques based on CNN, LSTM, and bidirectional LSTM models. These models were trained on raw sequences of API calls and parameters that were traced during the execution of malware and benign files. An ensemble of ML algorithms for malware classification based on API call sequences was implemented in Sukul et al. (2022). Convolutional neural networks and BiLSTM were used to develop a malware classification framework based on sequences of API calls extracted from executables files (Li et al., 2022a). The work proposed in Abbasi et al. (2022) has employed a dataset of API invocations, registry keys, files/directory operations, dropped files, and embedded strings features extracted using Cuckoo sandbox to implement a particle swarm-based approach that classifies ransomware attacks. Jing et al. (2022) proposed Ensila, an ensemble of RNN, LSTM, and GRU for malware detection which was trained and evaluated on dynamic features of API calls.

4. Proposed methodology

Details on the proposed framework for detecting malware attacks in Windows systems are presented in this section.

4.1. System overview

The proposed framework is based on the dynamic analysis approach where API call sequences are extracted from benign and malware executable files while running in a virtual isolated environment. The extracted raw sequences of API calls are then processed and encoded before being fed to a hybrid automatic feature extractor based on CNN

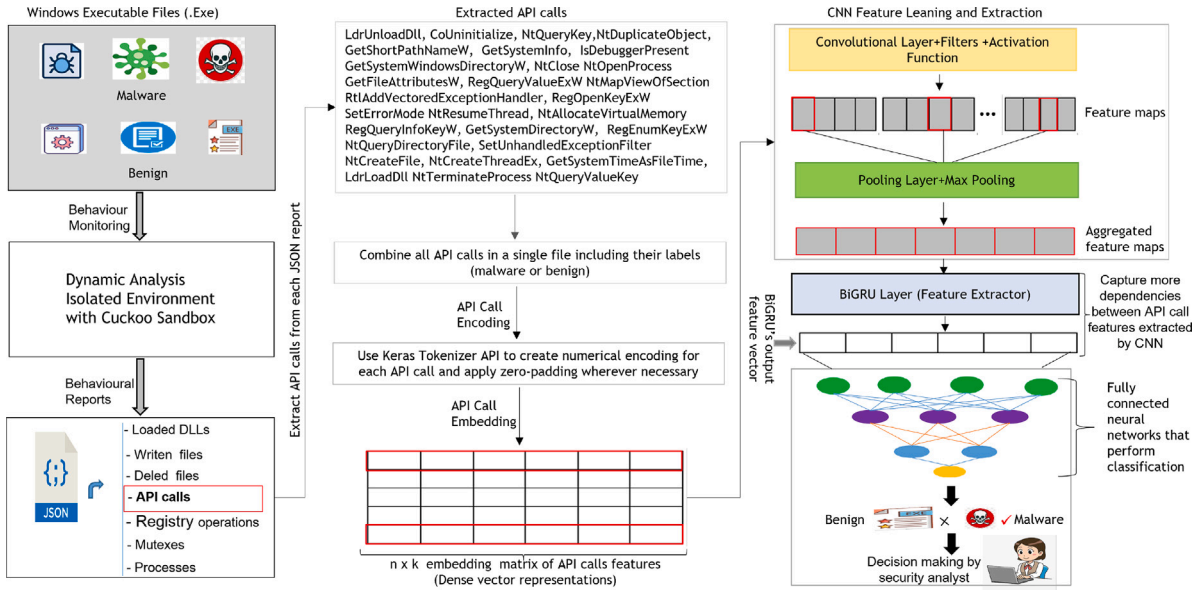

Fig. 1. The proposed API-MalDetect framework for behavior-based malware detection in Windows systems.

and BiGRU deep learning architectures. The final features generated by CNN-BiGRU are then passed to a fully connected layer with neural networks which performs the classification of each sequence of API calls as malicious or benign (normal). The architecture of the proposed framework is depicted in Fig. 1 and below we present more details on each component.

4.2. Generating API calls dataset

It is often challenging to find an up-to-date dataset of API calls of benign and malware executable files. For this reason, we have generated a new dataset of API calls which is used for the experimental evaluations. Different sources of malware executable files such as Malheur (Rieck et al., 2011), Kafan Forum (Pan et al., 2016), Danny Quist (Sethi et al., 2017), Vxheaven (Huda et al., 2016), MEDUSA (Nair et al., 2010), and Malicia (Nappa et al., 2015) were used in the previous studies. However, these repositories are not regularly updated to include new malware samples. Hence, we have collected malware executable samples from VirusTotal (Anon, 2021d), the most updated and the world's largest malware samples repository. Considering millions of malware samples available in the repository, it is worth mentioning that processing all malware samples is beyond the scope of this study due to hardware limitations.

Thus, only a subset of malware samples made available in the second quarter of 2021 was collected. We were given access to a Google Drive folder having malicious EXE files which are shared by VirusTotal. Benign samples were collected from CNET site (Anon, 2021b). The VirusTotal online engine was used to scan each benign EXE file to ensure that all benign files are clean. Therefore, a total number of 2800 EXE files were collected to be analyzed in an isolated analysis environment through dynamic analysis. Nevertheless, we experienced issues while executing some files, resulting in a dataset of 1285 benign and 1285 malware files that were successfully executed and analyzed to generate our benchmark dataset of API calls. Some benign files were excluded as they were classified as malicious by VirusTotal online engine (Anon, 2022b), while some malware files were also excluded because they did not run due to compatibility issues.

Accordingly, our isolated dynamic analysis environment consists of one Ubuntu host machine and Windows virtual machines (VMs). The Cuckoo sandbox (Anon, 2021a) and its dependencies (such as analysis and reporting modules) were installed in the Ubuntu host machine while the Cuckoo agent was deployed in each VM (2 VMs were configured for the dynamic analysis). The Cuckoo agent monitors each file's behaviors while executing and sends the results to the host to be analyzed by Cuckoo processing and reporting modules which create a JSON report containing all API calls captured during execution. As some advanced malware can escape the analysis environment during execution (which could cause serious damage to the production environment), a virtual network was configured to enable communication between the host and the virtual machine.

The analysis reports generated during our dynamic analysis reveal that some sophisticated malware can use different API calls that potentially lead to malicious activities. For instance, Table 1 presents some of the API calls used by ae03e1079ae2a3b3ac45e1e360eaa973.virus, which is a ransomware variant. This ransomware ended up locking one of our Windows VMs and demanded for a ransom to be paid in Bitcoin in order to get access back to the infected VM. Moreover, this variant also encrypted files and made them inaccessible. We have also observed that recent malware variants possess multiple behaviors and can perform multiple malicious activities after compromising the target, making their detection difficult. After the analysis, all JSON reports were processed to extract API calls which resulted in a new benchmark dataset of API call sequences that is publicly available for use by the research community focusing on malware detection.

Our benchmark dataset has 2570 records representing benign executable files and malware files such as ransomware, worms, viruses, spyware, backdoor, adware, keyloggers, and Trojans which appeared in the second quarter of 2021. Each executable file in the dataset has been labeled as benign or malware. The distribution of samples in the dataset is presented in Fig. 2. The dataset is balanced with the same number of malware and benign files and can be accessed from Github (Maniriho, 2022). Furthermore, it has been processed to remove all inconsistencies/noise, making it ready to be used for evaluating the performance of deep learning models. Additionally, a hash value of each file has been included to avoid duplication of files while extending the dataset in the future. Thus, making it easier to include behavioral characteristics of newly discovered malware variants in the dataset or combine the dataset with any of the existing datasets of API calls extracted from Windows PE files through dynamic analysis.

4.2.1. API call sequence encoding

After generating the dataset, the next step is to perform the encoding of API call sequences. Given an input sequence of API call, it is first tokenized to generated API call tokens, which are further

Table 1
Example of potentially malicious API calls observed in ae03e1079ae2a3b3ac45e1e360eaa973.virus while running in a Windows VM during our dynamic analysis.

Malicious API call	Description of API call
WriteConsoleW	Malware uses this API call to establish a command line console.
NtProtectVirtualMemory	This API call was used by the malware to allocate read-write memory usually to unpack itself.
CreateProcessInternalW	A process created a hidden Windows to hide the running process from the task manager. This API call allows the malicious program to spawn a new process from itself which is usually observed in packers that use this method to inject actual malicious code in memory and execute them using CreateProcessInternalW.
Process32FirstW	The malware used this API to search running processes potentially to identify processes for sandbox evasion, code injection, or memory dumping/live image capturing.
FindWindowA	The malware used this API to check for the presence of known Windows forensics tools and debuggers that might be running in the background.

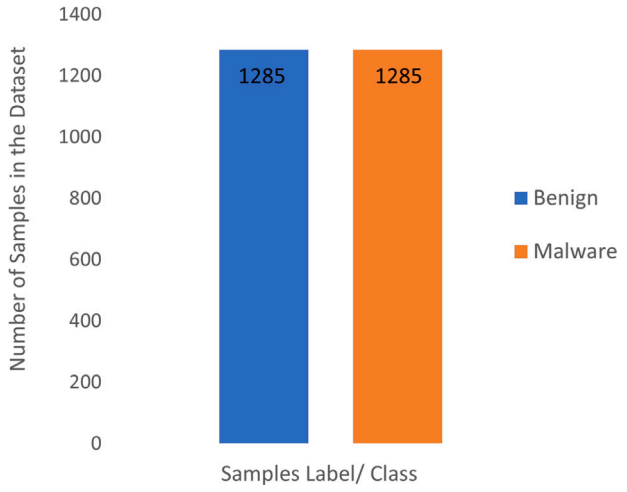

Fig. 2. Distribution of benign and malware samples in our benchmark dataset of API call sequences.

encoded by assigning a unique integer number to each token of API call where similar API calls have similar encoding. After encoding, the output is a sequential numerical representation of the original input sequence (originally represented in text format). All sequences of API calls are tokenized and encoded using an NLP Tokenizer from the Keras framework (Anon, 2023d). Additionally, sequences of API calls were padded using pre-padding (zero padding was applied where necessary) to obtain sequences with the same length. This is very important for our CNN feature extractor as it requires all input sequences to be numeric and have the same length. The encoded sequences are fed to an embedding layer which builds dense vector representations of each API call where all created vectors are combined to generate an embedding matrix of API call features. Fig. 3 illustrates the process of API call encoding where each API call sequence extracted from each benign and malware EXE file is treated as a sentence. Details on creating API call embedding are presented in Section 4.2.2.

4.2.2. Creating API call embedding

In natural language processing, embedding models allow the creation of numerical dense vectors of words while preserving their semantic information (contextual relationship between them). Embedding models have been used in combination with DL techniques to perform NLP-related tasks like text and sentiment classification (Pittaras et al., 2020; Kim, 2014). As we are dealing with API calls represented in text format, our API call embedding approach is linked to these studies. However, in this work we do not use existing pre-trained NLP-based word embedding models (Mikolov et al., 2013; Pennington et al., 2014) because similarities in API call sequences are very dissimilar with ordinary English words/text. Thereby, we use direct embedding with Keras embedding layer (Anon, 2023a) to automatically learn and generate dense embedding vectors of API calls. Direct embedding allows the knowledge to be incorporated inside the detection model (the proposed framework), i.e., the whole knowledge of API-MalDetect is incorporated in one component, which is different from the previous techniques. Additionally, as our proposed framework relies on direct embedding (where embeddings are constructed while training the detection model), it can resist some types of adversarial learning attacks such as the one which can be performed by modifying the pre-trained embedding models (Wang et al., 2022) with the aim to fool the performance of DL-based models. This makes the proposed framework more secure against such attacks. The Keras embedding requires all input of API calls to be integer encoded, the reason why each API call sequence has to be encoded as discussed in Section 4.2.1.

The Keras embedding layer uses deep neural networks to create dense vector representations of API calls from the API calls' corpus. It maps all encoded API calls in the input sequences to dense vectors in a high-dimensional space where similar API calls are located closer together, allowing neural networks to learn and capture contextual relationships between API calls. Therefore, the Keras embedding layer first takes an encoded matrix of API calls as input (matrix of integer indices) where each row of the matrix is a sequence of API calls representing benign or malware executable files. Each integer index is mapped to a dense vector (with a fixed length) which is learned through neural network training. Finally, the output is a matrix of API calls' dense vectors which is used as input to a CNN feature extractor (a subsequent layer to the Keras embedding layer in the proposed framework). Fig. 4 summarizes steps for generating numerical representations of API call sequences through encoding and embedding.

It is worth noting that the Keras embedding layer requires a few parameters to be specified before creating embedding. The embedding layer is first initialized with random weights, and thereafter, other parameters are defined. Such parameters include $Input\_dim$ which specifies the vocabulary size of API calls (for instance, if the API calls data has integer encoded values between 0–1000, then the vocabulary size would be 1001 API calls. The second parameter is the $output\_dim$ which denotes the size of the vector space in which words/tokens of API calls are embedded (i.e., it specifies/defines the size of each API call's output vector from the embedding layer). For example, it can be of any dimension such as 5, 10, 20, or any other positive integer which belongs ($\in$) to $\mathbb{Z}$. In this work, different values of $output\_dim$ are tested to obtain the suitable dimension of the output vector. Hence, this value was empirically chosen after testing different values and has been set to 10 in our experiment. Finally, the third parameter is the $input\_length$, which is the input length of each API call sequence. For instance, if a sequence of API calls extracted from a malware file has 60 API calls, its input length would be 60. As there are sequences with different lengths in the dataset, all sequences are processed to have the same $input\_length$ value during encoding. After this step, the embedding layer constructs and concatenates all constructed embedding vectors of API calls to form an embedding matrix which is used as input to the next layer.

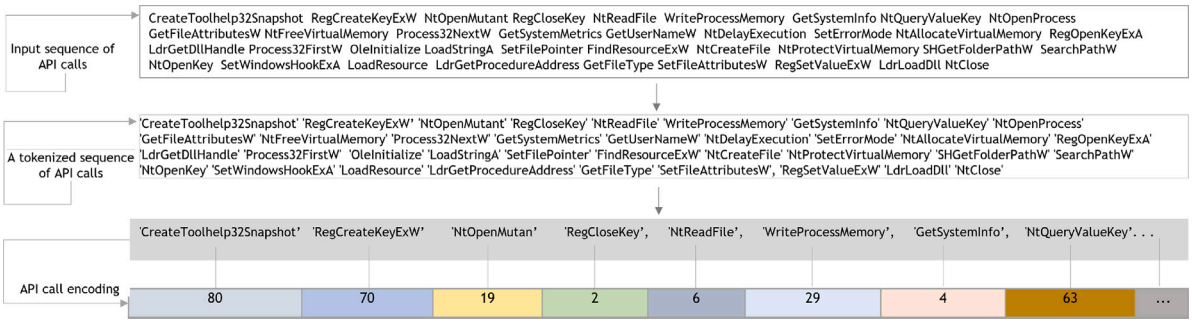

Fig. 3. An example of API call encoding using the Keras API Tokenizer.

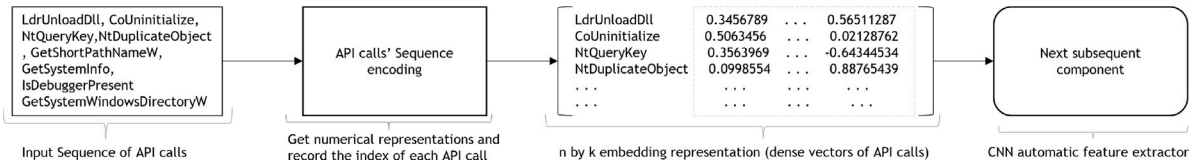

Fig. 4. Steps for generating numerical representations of API call sequences through encoding and embedding.

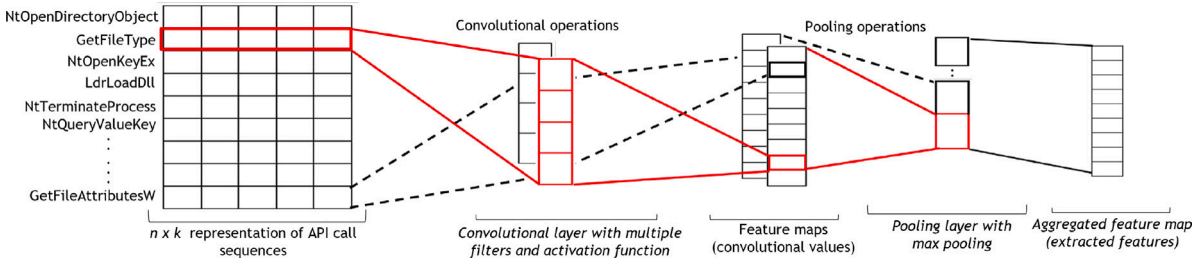

Fig. 5. The architecture of one-dimensional convolution neural networks (1-D CNN) designed for API calls feature learning and extraction.

4.3. Hybrid automatic feature extraction

We have designed a hybrid automatic feature extractor that exploits the potential of CNN and BiGRU deep learning algorithms to effectively learn relevant features of API calls that are used to train a fully connected neural network (FCNN)-based malware classifier/detector.

4.3.1. 1-D CNN feature extractor

The Keras embedding layer described in Section 4.2.2 serves as the key processing layer for our CNN automatic feature extractor which is designed based on the one-dimensional CNN (1-D CNN) technique for text classification proposed by Kim (2014). That is, we use 1-D CNN to extract local features of API calls from the embedding matrix (which is used as input data). As illustrated in Fig. 5, the architecture of the designed 1-D CNN feature extractor is made up of two main components, namely, the convolutional layer and the pooling layer. Given a sequence S of API calls represented as a dense vector from the embedding matrix (created during Keras embedding), let $X_{i\in\mathbb{R}^d}$ denotes a d-dimensional API call vector representing the ith API call in s where d is the dimension of the embedding vector. Therefore, a sequence S consisting of n API calls from a single JSON report can be constructed by concatenating individual API calls using the expression in (1) where the symbol $\oplus$ denotes the concatenation operator and n is the length of the sequence.

$$X_{i:n} = x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus \ldots\ldots \oplus x_n \tag{1}$$

We have padded sequences to generate API calls matrix of k_n dimensions having k number of tokens of API call with embedding vectors of length n. Padding allows sequences to have a fixed number of k tokens (i.e., the same fixed length is kept for all sequences) which is very important as CNN cannot work with input vectors of different lengths. Thus, we have set k to a fixed length. In order to identify and select highly relevant/discriminative features from raw-level features of API calls' embedding vectors, the CNN feature extractor performs a set of transformations to the sequential input vector $X_{i:n}$ through convolution operations, non-linear activation, and pooling operations in different layers. These layers interact as follows.

The convolutional layer relies on defined filters to perform convolutional operations to the input vectors of API calls. This allows the convolutional layer to extract unique features from API call vectors. As convolutional filters extract features from different locations/positions in the embedding vectors (embedding matrix), the extracted features have a lower dimension compared to the original features of API calls. Hence, mapping high-dimensional features to lower-dimensional features while keeping highly relevant features (i.e., it reduces the dimension of features). Positions considered by filters while convolving to the input are independent for each API call and semantic associations between API calls that are far apart in the sequences are captured at higher layers. therefore, we have applied a filter $W \in \mathbb{R}^{m\times n}$ to generate a high-level feature representation, with m moving/shifting horizontally over the embedding matrix based on a stride t to construct a feature map c_i which is computed using the expression in (4). It is important to mention that the multiplication operator (*) which is in Eq. (2) denotes the convolutional operation (achieved by performing element-wise multiplication) which represents API call vectors from X_i to $X_i{+}m{-}1$ (which means m rows at a time) from X which is covered by the defined filter W based on t. To make the operation faster, we have kept stride to a value of 1, however, various strides can be adapted as well. Moreover, in Eq. (2), the bias value is denoted by b_i.

$$C_i = f(W * X_{i:i+m-1+b_i}) \tag{2}$$

CNN supports several activation functions such as hyperbolic tangent, Sigmoid, and rectified linear unit (ReLU). In this work, we have used ReLU, which is represented by f in (2). Once applied to each input x, the ReLu activation function introduces non-linearity by caping/turning all negative values to zero. This operation is achieved using the expression in (3) where activation operation introduces nonlinearity and speeds up the training of the CNN model. This nonlinearity allows the proposed framework (API-MalDetect) to handle complex input data representation, which is impossible when using simple linear ML models (Chng, 2023).

$$f(x) = max(0, 1) \tag{3}$$

After convolving filters to the entire input embedding matrix, the out is a feature map corresponding to each convolutional operation and is obtained using the expression in (4). Note that the convolutional operations were optimized by the Dropout regularizer with a dropout rate of 0.2.

$$C(f) = [C_1, C_2, C_3, \ldots, C_{n-m+1}] \tag{4}$$

The convolutional layer passes its output to the pooling layer which performs further operations to generate a new feature representation by aggregating all values received from the feature maps. This operation is carried out using some well-known statistical techniques such as computing the mean or average, finding the maximum value, or applying the L-norm. One of the advantages of the pooling layer is that it has the potential to prevent the model's overfitting, reducing the dimensionality of features and producing sequences of API call features with the same fixed lengths. In this work, we have used max pooling (Kim, 2014; Collobert et al., 2011) which performs the pooling operation over each generated feature map and then selects the maximum value associated with a particular filter's output feature map. For instance, having a feature map c_i, the max-pooling operation is performed by the expression in (5), and the same operation is applied to each c_i feature map.

$$\hat{c}_i = max(c_i) \tag{5}$$

The goal is to capture high-level features of API calls (the ones with the maximum/highest value for every feature map). Note that the selected value from each feature map corresponds to a particular API call feature captured by the filter while convolving over the input embedding matrix. All values from the pooling operations are aggregated together to produce reduced feature vectors which are passed to the BiGRU feature extractor (also referred to as the BiGRU layer) for further processing.

4.3.2. BiGRU feature extractor

Features of API calls generated by the 1-D CNN feature extractor have low-level semantic information between them compared to the original ones. Fortunately, gated recurrent units can directly receive and process the intermediate feature maps generated by CNN. Thus, we have used a BiGRU network architecture which works as the subsequent layer to the 1-D CNN feature extractor. The BiGRU feature extractor processes sequences in both directions (i.e., it uses both forward and backward recurrence to process sequences), allowing it to capture high dependencies across feature maps (local features) of API calls. Therefore, the BiGRU layer depicted in the proposed malware detection framework in Fig. 1 receives the output feature vector generated after processing each feature map $\hat{c}_i$ of API calls) as its input. It then processes those sequences of API calls using gated units which allows it to control information flow and prevent the learning model from vanishing gradients. Mathematical computation of GRU is performed as shown in Eqs. (6), (7),(8), and (9) where the expression z_t, r_t, $\tilde{h}_t$ and h_t denote the update gate, reset gate, hidden state of current hidden node and output, respectively. The symbol σ denotes the Sigmoid activation, x_t is the current input, $tanh$ is the hyperbolic tangent activation function, w and u are weights matrices to be learned, $\odot$ is the Hadamard product of the matrix, while b_z, b_r, and b_h denotes the bias.

$$z_t = \sigma(w_{zx}x_t + u_{zh}h_{t-1} + b_z) \tag{6}$$

$$r_t = \sigma(w_{rx}x_t + u_{rh}h_{t-1} + b_r) \tag{7}$$

$$\tilde{h}_t = tanh(w_{hx}x_t + r_t \odot u_{hh}h_{t-1} + b_h) \tag{8}$$

$$h_t = (1 - z_t \odot \tilde{h}_t) + z_t \odot h_{t-1} \tag{9}$$

The output produced by the BiGRU layer contains sequences of hidden states that encode the contextual information of API calls features which are passed to the next layer for malware classification.

4.3.3. Fully connected layer

The fully connected layer consists of neural networks with hidden layers, ReLU activation function, and network regularizer. It also has an output layer with a sigmoid activation function. The hidden layer neurons/units receive input feature vectors h_ti from the BiGRU layer, process them, and then compute their activations l_i using the expression in (10) with W being the matrix of weights between the connections of the input neurons and hidden layer neurons while b_i represents the bias. We have used the Dropout regularization technique to prevent the network from overfitting on the training data. A dropout rate of 0.5 was used after each hidden layer, which means that at each training iteration, 50% of the connection weights are randomly selected and set to zero. Dropout works by randomly dropping out/disabling neurons and their associated connections to the next layer, which prevents the network's neurons from highly relying on some neurons and forcing each neuron to learn and to better generalize on the training data (Srivastava et al., 2014).

$$l_i = ReLU(\sum_i W_i * h_ti + b_i) \tag{10}$$

In addition, we have used the binary-cross entropy (Jadon, 2020) to computer the classification error/loss whereas the learning weights are optimized by Adaptive Moment Estimation (Adam) optimizer (Kingma and Ba, 2014; Yaqub et al., 2020). Cross-entropy is a measure of the difference between two probability distributions for a given set of events or random variables and it has been widely used in neural networks for classification tasks. On the other hand, Adam works by searching for the best weights W and bias b parameters which contribute to minimizing the gradient computed by the error function (binary-cross entropy in this case). Note that the network learning weights W are updated through backpropagation operations while for the Sigmoid function, we have used the binary logistic regression (see Eq. (11)).

$$Sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{11}$$

$$Y = Sigmoid(W_i \times l_i + b) \tag{12}$$

After computing the activation, the classification outcome (the predicted output) is computed by the expression in (12). For more details, a summary of parameter settings for the proposed framework is presented in Table 2, where the best parameters such as number filter, kernel size, etc., were determined using the grid search approach.

5. Experiments and results

Details on the experimental evaluations performed while evaluating the proposed framework are presented in this section. The results are based on a binary classification problem of benign and malware executable files in Windows systems.

Table 2
A summary of different parameter settings for the proposed framework.

Layer	Parameter Used	Value
Embedding layer	Input sequence length	Various (20, 30, 40, 60, 80, and 100)
	Embedding dimension	10
	Sequence padding	Pre-padding with zero padding
CNN	Number of convolutional layer	1
	Number of filters	128
	Filter size/kernel size	4
	Activation function	ReLU
	Regularizer	Dropout with a dropout rate of 0.2
	Number of pooling layer	1
	Stride	1
	Pooling method	Max pooling
BiGRU	Number of gated units	3
	Activation function	Tanh
	Recurrent activation function	Sigmoid
Fully connected layer	Number of hidden layers	2
	Number of neurons (hidden layer 1)	25
	Number of neurons (hidden layer 2)	25
	Regularizer (hidden layer 1 and layer 2)	Dropout with a dropout rate of 0.5
Model compilation	Optimizer	Adam with a learning rate of 0.001
	Loss function	Binary Cross Entropy
Other parameters	Number of epochs	16
	Batch size	32

5.1. Experimental setup and tools

The proposed framework was implemented and tested in a computer running Windows 10 Enterprise edition (64-bit) with Intel(R) Core (TM) i7-10700 CPU @ 2.90 GHz, 16.0 GB RAM, NVIDIA Quadro P620, and 500 GB for the hard disk drive. The framework was implemented in Python programming language version 3.9.1 using TensorFlow 2.3.0 and Keras 2.7.0 frameworks. Other libraries such as Scikit-learn, Numpy, Pandas, Matplotlib, Seaborn, LIME, and Natural Language Toolkit (NLTK) have been also used. All these libraries are freely available for public use and can be accessed from PiPy (Pypi, 2021), the Python package management website.

5.2. Eliminating spatial and temporal bias in the experiments

Considering the best practices for building malware detection techniques (systems), we have followed the best practices from Pendlebury et al.'s work Pendlebury et al. (2019) to address the issue of temporal and spatial bias in the experiment. The practical constraints/guidelines in Pendlebury et al. (2019) allow us to adhere to a realistic scenario as closely as possible in our experimental evaluations. Specifically, the following constraints were taken into consideration.

- **Temporal training consistency:** We have imposed a temporal split between the training dataset and testing dataset where malware samples released in 2015 are used for training while testing is performed on malware samples generated in 2018 and 2021. This allows us to evaluate the performance of the proposed framework on unseen malware samples in order to ensure its potential while detecting newly released/unknown malicious executable files.
- **Temporal benign consistency:** As in many cases benign executable files can remain stable over time, we did not collect them based on time. We collected them from various sources and then split them into two portions, with one portion used for training and the other one for testing.
- **Spatial malware and benign consistency in the test set:** Estimating the exact percentage of malware executable samples in the wild is impossible because many of them have a short lifetime and they also get updated more often. On the other hand, it is also clear that the number of daily malware executable samples encountered by individuals or organizations is significantly less than that of benign ones (Mimura, 2023; Pendlebury et al., 2019). Despite being highly produced, malware executable files are rarely used compared to benign executables (Mimura, 2023). As suggested in Moskovitch et al.'s work (Moskovitch et al., 2009), malware represents approximately 10% of the traffic on the Internet (which actually represents the ratio of malware in the test set). Accordingly, in this work, we assume that in a realistic scenario (real-life conditions), there would be 90% benign and 10% malware detected by a malware detection end system. The same ratio has been suggested in the research studies presented in Pendlebury et al. (2019) Nissim et al. (2014), Tien et al. (2020).

5.3. Training and testing datasets

Following the best practices presented in Section 5.2, we have used the API calls dataset in Ki et al. (2015) for training (the dataset can be accessed from Anon (2023c) and Anon (2023b)). Testing was performed using our dataset (Maniriho, 2022) (the descriptions on our dataset can be found in Section 4.2). Because the dataset in Ki et al. (2015) is highly imbalanced where malware files highly outnumber benign files (23,080 malware and 300 benign), we performed downsampling in order to have a good distribution of samples across the dataset. Moreover, we have also taken benign API call samples from Anon (2021c), which were added to the training dataset. Specifically, our training dataset has a total number of 8351 samples (with 4787 benign and 3564 malware) while the testing dataset has three test sets. The first test (testset1) has 1343 samples with 90% and 10% representing benign and malware samples, respectively. The second test set (testset2) has 1511 samples where 80% represent benign while 20% represent malware and it is used to examine if the proposed framework can still perform well when the number of malware files exceeds 10% in the test set. It is worth mentioning that both testset1 and testset2 are from our dataset. The last test set (testset3), is taken from Ceschin et al. (2018). It has 1343 samples with 90% and 10% representing benign and malware, respectively. Overall, the proposed framework is trained and tested on a total number of 12,548 executable samples.

5.4. Evaluation metrics

Different metrics such as precision (P), recall (R), F1-Score (F1), false positive rate (FPR), false negative rate (FNR), and accuracy (Acc)

were measured to evaluate the performance of the proposed framework. The computations of these metrics are presented in Eqs. (13), (14), (15), and (16), with TP, TN, FP, and FN indicating true positive, true negative, false positive, and false negative, respectively.

$$Precision(P) = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{13}$$

$$Recall(R) = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{14}$$

$$Accuracy = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{15}$$

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{16}$$

We have also computed other metrics such as the area under the Roc Curve (AUC), macro average, and the weighted average for precision, recall, and F1-score, respectively. The macro average is calculated using unweighted/arithmetic mean where all classes in the dataset are treated equally irrespective of their support (the total number of times each class appears in the dataset). The weighted average considers each class's support and is computed by the mean of all per-class scores (e.g., precision or recall score). Eqs. (17), (18), and (19) show how to compute the macro average precision (Macro P), macro average recall (Macro R), and macro average F1-score (Macro F1) where i represents the ith class while N is the total number of classes in the dataset. Moreover, Eqs. (20), (21), and (22) show the weighted average precision (weighted P), weighted average recall (weighted R), and weighted average F1-score (Weighted F1) where W_i denotes the weight of class i. Additionally, Eq. (23) shows AUC calculation.

$$MacroP = \frac{1}{N}\sum_{i=1}^{N} P_i \tag{17}$$

$$MacroR = \frac{1}{N}\sum_{i=1}^{N} R_i \tag{18}$$

$$MacroF1 = \frac{2 * MacroP * MacroR}{MacroP + MacroR} \tag{19}$$

$$WeightedP = W_i \sum_{i=1}^{N} P_i \tag{20}$$

$$WeightedR = W_i \sum_{i=1}^{N} R_i \tag{21}$$

$$WeightedF1 = \frac{2 * WeightedP * WeightedR}{WeightedP + WeightedR} \tag{22}$$

$$AUC = \int_0^1 \frac{TP}{TP + FN} d \frac{FP}{TN + FP} \tag{23}$$

5.5. Classification results

The experimental evaluations were conducted to examine how the proposed framework can effectively detect unseen malware attacks based on sequences of API calls representing benign or malware executable files. Thus, various evaluations were carried out, and the results are presented in this subsection. We first present the performance of API-MalDetect against other API call-based malware detection techniques (or frameworks) in Section 5.5.1 and then in Section 5.5.2, we present the performance of API-MalDetect under various experimental conditions. Finally, Section 5.5.4 provides insights about explainable results generated by API-MalDetect based on LIME.

Table 3
Comparisons against other detection techniques based on API call sequences extracted in Windows executable files.

Work	Algorithm used	Accuracy (%)
Maldy (Karbab and Debbabi, 2019)	KNN	97.60
Qin and Wang (Qin et al., 2020)	TextCNN	95.90
Amer et al. (Amer et al., 2022)	Particle swarm-based	95.40
Mathew and Kumara (Mathew and Ajay Kumara, 2020)	LSTM	92.00
Liu and Wang (Liu and Wang, 2019)	BiGRU	93.72
Li et al. (Li et al., 2022a)	CNN-BiLSTM	97.31
Yesir and Soğukpinar (Yesir and Soğukpinar, 2021)	BERT	96.76
Xiaofeng et al. (Xiaofeng et al., 2019)	RF and Bi-residual LSTM	96.70
Catak et al. (Catak et al., 2020)	Two-layer LSTM	95.00
Xue et al. (Xue et al., 2022)	MLP	91.57
Sai et al. (Sai et al., 2019)	DT	90.00
Nawaz et al. (Nawaz et al., 2022)	J48	97.50
Maniath (Maniath et al., 2017)	LSTM	96.67
Pektaş and Acarman (Pektaş and Acarman, 2017)	AROW	93.00
Nunes et al. (Nunes et al., 2019)	RF	96.00
Han et al. (Han et al., 2019)	XGBoost	93.18
Avci et al. (Avci et al., 2023)	BiLSTM	93.16
This work	**CNN-BiGRU**	**99.07**

Table 4
Accuracy, false positive rate, and false negative rate achieved by API-MalDetect on testset1.

Sequence length	Accuracy (%)	False positive rate (%)	False negative rate (%)
20	96.87	3.05	0.07
40	97.32	2.46	0.22
60	97.77	2.08	0.15
80	98.06	1.86	0.07
100	98.81	1.04	0.15

5.5.1. Benchmark comparisons against other techniques

We have examined the performance of the API-MalDetect framework against other techniques for malware detection implemented based on sequences of API calls. Therein, comparative results are presented in Table 3. First, we have compared API-MalDetect against Maldy (Karbab and Debbabi, 2019), an existing framework based on NLP and machine learning techniques. As shown in Table 3, our framework outperformed the Maldy framework (Karbab and Debbabi, 2019) with an improvement of 1.47% (99.07%–97.60%) in detection accuracy. API-MalDetect also performed well over a TextCNN-based malware detection approach presented by Qin et al. (2020), revealing the potential of combining TextCNN (also known as 1-D CNN) with BiGRU in our framework. Moreover, the proposed framework also shows better performance over other malware detection techniques presented by Amer et al. (2022), Mathew and Ajay Kumara (2020), Liu and Wang (2019), Li et al. (2022a), Yesir and Soğukpinar (2021), Xiaofeng et al. (2019), and Avci et al. (2023), to mention a few.

5.5.2. Performance of API-MalDetect under various experimental conditions

As previously mentioned, we have tested the performance of API-MalDetect using testset1, testset2, and testset3. We have examined the effect of API call sequence length (n) on the performance of our framework. Moreover, the results obtained on testset2 also allow us to examine if the performance of API-MalDetect can be affected by

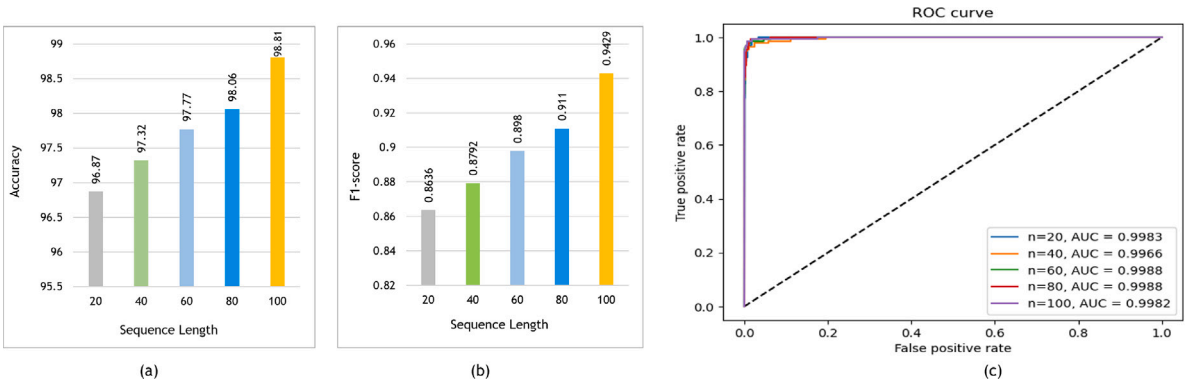

Fig. 6. Effect of sequence length on (a) Accuracy and (b) F1-score (with data taken from Tables 4 and 5) and (c) ROC Curve obtained while testing the proposed framework using testset1.

Table 5
Precision, recall, and F1-score obtained when testing API-MalDetect on testset1.

Sequence length (n)	Predicted class	Precision	Recall	F1-Score
20	Benign	0.9991	0.9661	0.9823
	Malware	0.7644	0.9925	0.8636
40	Benign	0.9975	0.9727	0.9849
	Malware	0.7988	0.9776	0.8792
60	Benign	0.9983	0.9768	0.9875
	Malware	0.8250	0.9851	0.8980
80	Benign	0.9992	0.9793	0.9891
	Malware	0.8418	0.9925	0.9110
100	Benign	0.9983	0.9884	0.9933
	Malware	0.9041	0.9851	0.9429

increasing malware in the test set, i.e., when there are 20% of malware and 80% of benign samples in the test set. The results in Tables 4–6 are based on testset1 while the results in Tables 7, 8, and 9 are generated using testset2. Additionally, Tables 10–12 present the performance of API-MalDetect on testset3 with API calls from one of the existing datasets mentioned in Section 5.3. Throughout our evaluations, five lengths of API call sequences were considered (20, 40, 60, 80, 100), however, the proposed framework can handle other lengths of API calls sequences beyond the mentioned ones. For simplicity, we have kept the embedding size (embedding dimension) to 10, and the evaluations show good performance.

Accordingly, Table 4 shows the detection/classification report achieved by API-MalDetect on unseen sequences of API calls. Looking at the results, we could see that API-MalDetect has successfully classified malicious API call sequences and benign API call sequences with a detection accuracy of 96.56% based on a sequence length of 20 API calls. The accuracy of 97.32%, 97.77%, and 98.06% was also achieved based on sequence lengths of 40, 60, and 80, respectively. The highest accuracy (98.81%) was obtained using the lengths of 100. Interestingly, the testing accuracy varies in accordance with the value of n, revealing the effect of the API call sequence length on the performance, i.e., as n increases, the accuracy also increases. This is shown by the accuracy improvement of 2.25% (98.81% − 96.56%) achieved by increasing the value of n from 20 to 100. Table 4 also shows the false positive rate (FPR) and false negative rate (FNR) obtained on the same test set, which demonstrate that only a few samples of malware were misclassified by our framework, resulting in a lower FNR of 0.15% obtained while testing the framework on API call sequences with a length of 100. Nevertheless, a high FPR of 3.05% was achieved with a sequence length of 20. However, there is a reduction in the false positive rate as the value of n increases. In general, the framework is able to identify malicious API call sequences with a lower FNR and FPR.

Table 5 presents precision, recall, and F1-score generated while testing API-MalDetect, which also demonstrate better performance while detecting both malware and benign on unseen data. For instance, a precision of 0.9991 and 0.7644 was obtained using the length of API call sequence of 20 for benign and malware detection, respectively. Similarly, using the same length (n=20) API-MalDetect achieves a recall of 0.9421 and an F1-Score of 0.8636 for malware detection. In general, the classification report for precision, recall, and F1-score obtained using different values of n reveal a better performance of our framework when distinguishing malicious API calls from benign ones. Our framework has a good measure of separability between both classes (it performs well in identifying malware attacks) and can deal with long sequences. The higher the value of precision (with 1 being the highest), the better the performance of a given detection model. As shown in Table 5, it is also important to highlight that in many cases the precision, recall, and F1-score get increased based on the value of (n). Table 6 provides the macro and weighted average for precision, recall, and F1-score achieved by the proposed framework and were computed based on the classification report presented in Table 5, which also shows better performance of API-MalDetect with a high macro F1 of 0.9681 and weighted F1 of 0.9883.

Fig. 6(a) and (b) illustrate how accuracy and F1-score change based on different values on n while Fig. 6(c) depicts the area under the ROC Curve (AUC or AUC-ROC) values obtained while classifying malware and benign files. Accordingly, all AUC values are close to one with the highest value being 0.9988, revealing how API-MalDetect can effectively distinguish malware from benign files' activities. Tables 7, 8, and 9 present the performance of API-MalDetect obtained on testset2 (with 80% benign and 20% malware). In particular, the classification report in Table 7 shows an improvement in accuracy when detecting previously unseen API call sequences of benign and malware.

For instance, the accuracy was improved from 98.81% (obtained using testset1) to 99.07%. More importantly, the results also demonstrate that there is a reduction in both FPR and FNR which gives assurance that the proposed framework is able to identify malicious activities based on the executable's API call sequences even when the number of malware exceeds the realistic ratio (which is 10% in our case). The classification report in Tables 8 and 9 also shows performance improvement in macro F1, weighted F1, and in other metrics such as recall and precision.

Fig. 7(a) illustrates how the detection accuracy was improved based on the ratio of malware and benign in testset1 (where 10% represent malware) and testset2 (where 20% represent malware) with $n = 100$. Fig. 7(b) and (c) also show how false positive rate and false negative rate increase based on the ratio of malware in the test set (10% and 20%). The results show a trade-off between FPR and FNR, i.e., when FPR increases, FNR decreases, and vice-versa. For example, the FPR decreased from 1.04% to 0.15% (refer to Fig. 7(b) while FNR increased from 0.15% to 0.26% using API call sequences with $n = 100$ from

Table 6
Macro and the weighted average for precision, recall, and F1-score achieved by API-MalDetect on testset1.

Sequences length (n)	Macro P	Macro R	Macro F1	Weighted Pr	Weighted R	Weighted F1
20	0.8818	0.9793	0.9230	0.9757	0.9687	0.9705
40	0.8981	0.9752	0.9321	0.9776	0.9732	0.9744
60	0.9117	0.9810	0.9427	0.9810	0.9777	0.9785
80	0.9205	0.9859	0.9500	0.9835	0.9806	0.9813
100	0.9512	0.9867	0.9681	0.9889	0.9881	0.9883

Table 7
Accuracy, false positive rate, false negative rate, and AUC achieved by API-MalDetect using testset2.

Sequence length (n)	Accuracy (%)	False positive rate (%)	False negative rate (%)	AUC
20	96.56	3.38	0.07	0.9987
40	97.62	2.18	0.20	0.9992
60	98.08	1.72	0.20	0.9993
80	98.68	1.13	0.20	0.9992
100	99.07	0.66	0.26	0.9992

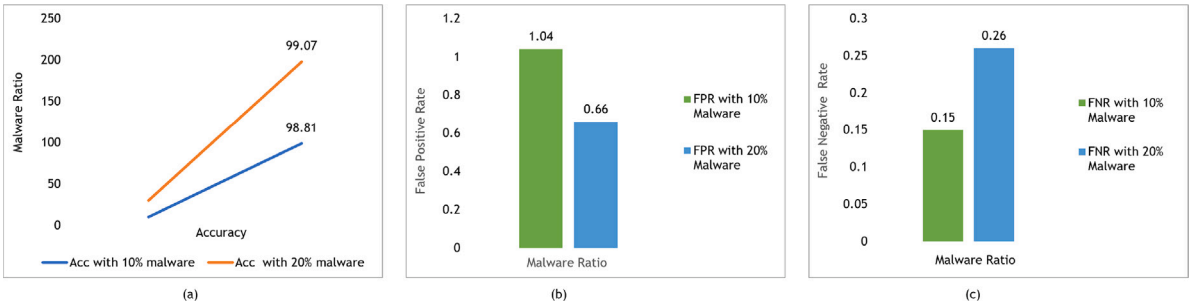

Fig. 7. Variation of (a) accuracy (b) False positive rate (c) False negative rate based on the ratio of malware in the test set.

Table 8
Precision, recall, and F1-Score achieved while examining the performance of API-MalDetect on testset2.

Sequence length (n)	Predicted class	Precision	Recall	F1-Score
20	Benign	0.9991	0.9578	0.9780
	Malware	0.8551	0.9967	0.9205
40	Benign	0.9975	0.9727	0.9849
	Malware	0.9006	0.9901	0.9432
60	Benign	0.9975	0.9785	0.9879
	Malware	0.9200	0.9901	0.9537
80	Benign	0.9975	0.9859	0.9917
	Malware	0.9462	0.9901	0.9676
100	Benign	0.9967	0.9917	0.9942
	Malware	0.9675	0.9868	0.9770

testset1 and testset2, respectively. Note that the data in Fig. 7 are taken from Tables 4 and 7.

As mentioned in Section 5.3, the performance of the proposed framework was also tested on one of the existing datasets of API calls (testset3). Accordingly, Table 10 shows accuracy, false positive rate, false negative rate, and AUC, achieved by API-MalDetect on unseen samples from testset3. The results also show that the performance increases as the length of the API call sequence increases. For example, the accuracy increased from 96.52% (with $n = 20$) to 98.29% (where $n = 100$). In addition, API-MalDetect can detect unseen malware with a low FPR (1.12%) and FNR (0.59%). Tables 11 and 12 present the obtained macro and weighted average for precision, recall, and F1-score, which also shows a good performance of the proposed framework. More importantly, the overall performance achieved in various experimental evaluations proves that the proposed framework can potentially detect known and unseen malware attacks based on sequences of API calls. This gives our framework the ability to deal with malware attacks in Windows systems.

5.5.3. Time complexity analysis of API-MalDetect

In this work, we have also analyzed the detection time complexity based on the input size of the API call sequence (denoted by n) which is fed to the proposed API-MalDetect framework to detect unseen malicious files. That is, the detection time taken by API-MalDetect is measured with respect to the value of n, and our analysis is based on the work in Cormen et al. (2022). We refer to input size (also called input sequence length) as the total number of API calls in each sequence. Accordingly, we have measured the execution time, and the results are visualized in Fig. 8(a), (b), and (c). Looking at the training time, there is an increase in the training time as n increases. For instance, an execution time of 35.721 s was taken while training API-MalDetect on sequences of API calls with $n = 20$ while 220.780 s were taken with $n = 100$. The training and testing time gap is mainly due to the sequence length.

As shown in Fig. 8, the total number of operations performed by API-MalDetect increases if n is increased. In other words, the detection time taken by the proposed framework increase with the growth of n. The best detection case will always occur if the framework detects and classifies malware attacks based on the first twenty API calls (where $n = 20$) invoked by a particular malware file while running the victim's system/device. On the other hand, the complexity in detection time taken by the framework to discover malicious files (malware) also increases as the length of API call sequences becomes longer. Hence, the worst-case detection time of the proposed framework will likely occur when processing longer sequences.

The average detection time/average performance behavior (in terms of detection time) achieved by API-MalDetect when detecting unseen malicious files from our dataset is 0.298 s. It is computed by combining the detection time of all input sizes (n=20, 40, 60, 80, and 100 in this case). In addition, there is an average detection time of 0.107 s, taken when testing API-MalDetect on another dataset of API call sequences (testset3). Therefore, it is important to note that despite a slight increase in detection time when dealing with longer sequences, API-MalDetect does not take a huge amount of time to detect

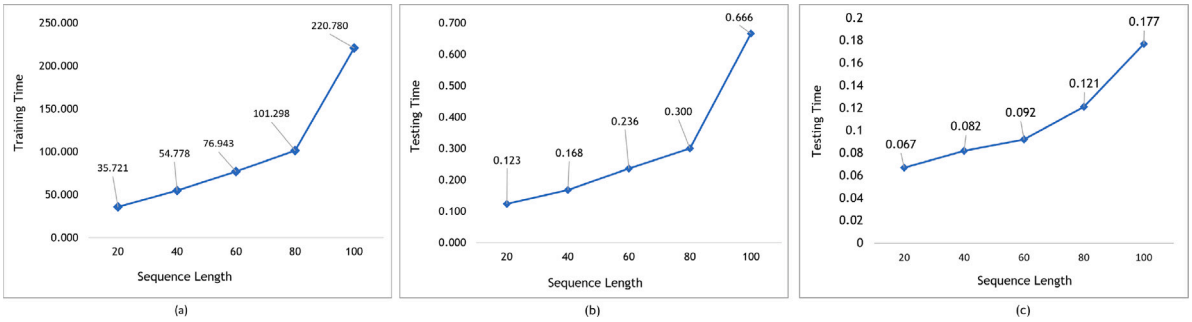

Fig. 8. (a) Training time taken when training API-MalDetect (b) and (c) Testing time taken when testing the framework on various test sets.

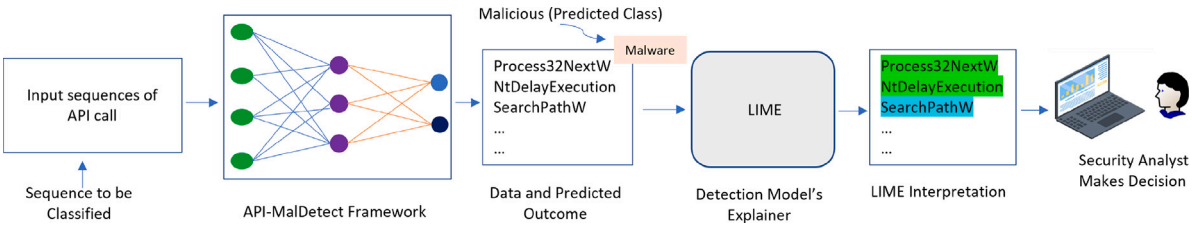

Fig. 9. Explaining the predicted outcome: API-MalDetect predicts that a sequence of API calls is malicious, and LIME highlights the API calls in the sequence that led/contributed to the final prediction. These can help security analysts to make decisions and trust the API-MalDetect's predictions.

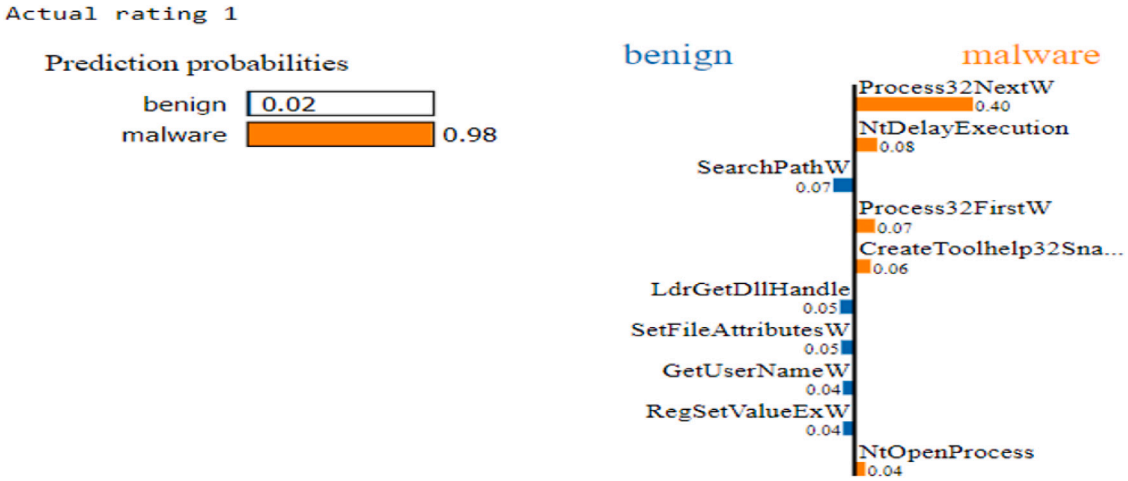

Fig. 10. An example of explanation of the classification outcome generated by LIME when classifying a malware file with API-MalDetect.

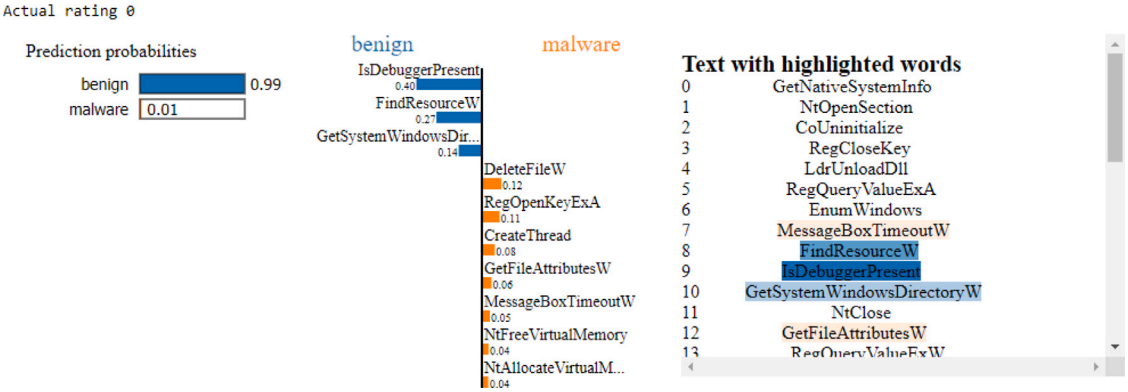

Fig. 11. Explanation of the classification (predicted) outcome generated by LIME when classifying a benign file with the API-MalDetect framework.

Table 9
Macro and the weighted average for precision, recall, and F1-score obtained by API-MalDetect on testset2.

Sequences length (n)	Macro P	Macro R	Macro F1	Weighted P	Weighted R	Weighted F1
20	0.9271	0.9773	0.9493	0.9704	0.9656	0.9665
40	0.9490	0.9814	0.9641	0.9781	0.9762	0.9766
60	0.9587	0.9843	0.9708	0.9820	0.9808	0.9811
80	0.9718	0.9880	0.9797	0.9872	0.9868	0.9869
100	0.9821	0.9892	0.9856	0.9909	0.9907	0.9908

Table 10
Accuracy, false positive rate, false negative rate, and AUC achieved by API-MalDetect using testset3.

Sequence length (n)	Accuracy (%)	False positive rate (%)	False negative rate (%)	AUC
20	96.52	2.61	0.87	0.9594
40	97.03	1.96	1.01	0.9355
60	97.10	2.10	0.80	0.9586
80	97.17	2.17	0.65	0.9620
100	98.29	1.12	0.59	0.9752

Table 11
Precision, recall, and F1-Score obtained while examining the performance of API-MalDetect on testset3.

Sequence length (n)	Predicted class	Precision	Recall	F1-Score
20	Benign	0.9901	0.9710	0.9805
	Malware	0.7778	0.9130	0.8400
40	Benign	0.9886	0.9783	0.9834
	Malware	0.8212	0.8986	0.8581
60	Benign	0.9910	0.9767	0.9838
	Malware	0.8141	0.9203	0.8639
80	Benign	0.9926	0.9758	0.9842
	Malware	0.8113	0.9348	0.8687
100	Benign	0.9933	0.9876	0.9905
	Malware	0.8944	0.9407	0.9170

malware attacks, which is a necessity for any robust and efficient anti-malware detection system. API-MalDetect can still identify malicious files within a reasonable short time, making it ideal for high-speed malware detection on Windows desktop devices.

5.5.4. Understanding API-MalDetect prediction with LIME

It is often complicated to understand the prediction/classification outcome of deep learning models given many parameters they use when making predictions. Therefore, in contrast to the previous malware detection techniques based on API calls, we have integrated LIME (Ribeiro et al., 2016) into our proposed behavior-based malware detection framework which helps to understand the predictions. The local interpretable model-agnostic explanations (LIME) is an automated framework/library with the potential to explain or interpret the prediction of deep learning models. More importantly, in the case of text-based classification, LIME interprets the predicted results and then reveals the importance of the most highly influential words (tokens) which contributed to the predicted results. This works well for our proposed framework as we are dealing with sequences of API calls that represent malware and benign executable files. The LIME framework was chosen because it is open source, has a high number of citations and the framework has been highly rated on GitHub.

Fig. 9 shows how LIME works to provide an explanation/interpretation of a given prediction of API call sequence. LIME explains the framework's predictions at the data sample level, allowing security analysts/end-users to interpret the framework's predictions and make decisions based on them. LIME works by perturbing the input of the data samples to understand how the prediction changes i.e., LIME considers a deep learning model as a black box and discovers the relationships between input and output which are represented by the model (Ribeiro et al., 2016; Hulstaert, 2022). The output produced by LIME is a list of explanations showing the contributions of each feature to the final classification/prediction of a given data sample. This produces local interpretability and allows security practitioners to discover which API call feature changes (in our case) will have the most impact on the predicted output. LIME computes the weight probabilities of each API call in the sequence and highlights individual API calls that led to the final prediction of a particular sequence of API calls representing malware or benign EXE file.

For instance, in Fig. 10, the Process32NextW, NtDelayExecution, Process32FirstW, CreateToolhelp32Snapshot, and NtOpenProcess are portrayed as the most API calls contributing to the final classification of the sequence as "malicious" while SearchPathW, LdrGetDllHandle, SetFileAttributesW, and GetUserNameW API calls are against the final prediction. Another example showing Lime output is presented in Fig. 11 where API calls features that led to the correct classification of a benign file are assigned weight probabilities which are summed up to give the total weight of 0.99. API calls such as IsDebuggerPresent, FindResourceW, and GetSystemWindowsDir are among the most influential API calls that contribute to the classification of the sequence/file into its respective class (benign in this case).

The screenshots of Lime explanations presented in Figs. 10 and 11 are generated in HTML as it generates clear visualizations than other visualization tools such as Matplotlib. In addition, the weights are interpreted by applying them to the prediction probability. For instance, if API calls IsDebuggerPresent and FindResourceW are removed from the sequences, we expect API-MalDetect to classifier the benign sequence with the probability of 0.37 (0.99–0.38-0.26= 0.37). Ideally, the interpretation/explanation produced by Lime is a local approximation of the API-MalDetect framework's behaviors, allowing it to reveal what happened inside the black box. It is crucial to note that in Fig. 11, the tokens under "Text with highlighted words", represent the original sequences of API while the number before each API call (e.g., 0 for RegCreateKeyExW) corresponds to its index in the sequence. The highlighted tokens show those API calls which contributed to the classification of the sequence. Thus, having this information, a security analyst can decide whether the model's prediction should be trusted or not.

6. Limitations and future work

In our future work, we intend to extend our dataset of API calls to include more features from benign files, and newly released malware variants (the current dataset size was limited due to hardware limitations). We also plan to evaluate the proposed framework on API call features extracted in the Android applications where features will be extracted from APK files through dynamic analysis. Although currently, we can explain the results using LIME, in some cases, LIME can be unstable as it depends on the random sampling of new features/perturbed features (Molnar, 2020). LIME ignores correlations between features as data points are sampled from a Gaussian distribution (Molnar, 2020). Therefore, we plan to explore and compare explanation insights

Table 12
Macro and the weighted average for precision, recall, and F1-score obtained by API-MalDetect on testset3.

Sequences length (n)	Macro P	Macro R	Macro F1	Weighted P	Weighted R	Weighted F1
20	0.8840	0.9420	0.9102	0.9689	0.9652	0.9664
40	0.9049	0.9384	0.9208	0.9719	0.9703	0.9709
60	0.9026	0.9485	0.9239	0.9733	0.9710	0.9718
80	0.9020	0.9553	0.9264	0.9745	0.9717	0.9726
100	0.9439	0.9642	0.9537	0.9834	0.9829	0.9831

produced by other frameworks such as Anchor (Ribeiro et al., 2018) and ELI5 (MIT, 2022), which also interpret deep learning models. The proposed framework will also be assessed against adversarial samples (Peng et al., 2021), which can fool malware detection models based on API calls. While this work was focused on dynamic malware analysis, it is also important to note that extracting dynamic features requires more resources and can be much more costly compared to static feature extraction. Another potential drawback of dynamic malware analysis is that some sophisticated malware can detect the virtual analysis environment, and in some cases, malware can present different behaviors which are different from their real behaviors when running in a real/production environment.

7. Conclusion

The paper proposed a deep learning-based framework called API-MalDetect for detecting malware attacks in Windows systems. The framework used an NLP-based encoder for API calls and a hybrid automatic feature extractor based on deep learning techniques such as CNNs and BiGRUs to extract features from raw and long sequences of API calls. The paper also introduced practical experimental factors for training and testing malware detection techniques to avoid temporal bias and spatial bias in the experiments. Additionally, it integrated LIME into the framework to provide local interpretability and explainability for predictions. The proposed framework achieved high detection accuracy, with an F1-score of 0.99 on the training set and 0.98 on the unseen data, demonstrating its effectiveness in detecting both existing and new malware attacks with high performance. The authors also evaluated their approach against several state-of-the-art methods, achieving better or comparable results in terms of accuracy, precision, recall, and F1-score. Overall, the proposed framework showed promising results in detecting malware attacks in Windows systems using deep learning-based techniques.

CRediT authorship contribution statement

Pascal Maniriho: Conceptualization, Methodology, Implementation, Investigation, Validation, Writing – original draft, Writing – review & editing. **Abdun Naser Mahmood:** Conceptualization, Validation, Writing – review & editing, Supervision, Funding acquisition. **Mohammad Jabed Morshed Chowdhury:** Conceptualization, Validation, Writing – review & editing, Supervision.

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.

Data availability

Data will be made available on request.

References

Abbasi, M.S., Al-Sahaf, H., Mansoori, M., Welch, I., 2022. Behavior-based ransomware classification: A particle swarm optimization wrapper-based approach for feature selection. Appl. Soft Comput. 121, 108744.

Alazab, M., Venkataraman, S., Watters, P., 2010. Towards understanding malware behaviour by the extraction of API calls. In: 2010 Second Cybercrime and Trustworthy Computing Workshop. pp. 52–59. http://dx.doi.org/10.1109/CTC.2010.8.

Amer, E., Samir, A., Mostafa, H., Mohamed, A., Amin, M., 2022. Malware detection approach based on the swarm-based behavioural analysis over API calling sequence. In: 2022 2nd International Mobile, Intelligent, and Ubiquitous Computing Conference. MIUCC, IEEE, pp. 27–32.

Amer, E., Zelinka, I., 2020. A dynamic windows malware detection and prediction method based on contextual understanding of API call sequence. Comput. Secur. 92, http://dx.doi.org/10.1016/j.cose.2020.101760.

Amer, E., Zelinka, I., El-Sappagh, S., 2021. A multi-perspective malware detection approach through behavioral fusion of api call sequence. Comput. Secur. 110, 102449.

Ammar Ahmed E. Elhadi, B.I.A.B., 2013. Improving the detection of malware behaviour using simplified data dependent API call graph. Int. J. Secur. Appl. 7, 29–42.

Anon, 2019. MalDAE: Detecting and explaining malware based on correlation and fusion of static and dynamic characteristics. Comput. Secur. 83, 208–233. http://dx.doi.org/10.1016/j.cose.2019.02.007.

Anon, 2021a. Cuckoo sandbox - automated malware analysis. https://cuckoosandbox.org/. (Accessed 14 May 2021).

Anon, 2021b. Free software downloads and reviews for Windows, Android, Mac, and iOS – CNET download. https://download.cnet.com/. (Accessed 15 December 2021).

Anon, 2021c. GitHub: Dataset containing malware and goodware collected in cyberspace over years. https://github.com/fabriciojoc/brazilian-malware-dataset. (Accessed 16 December 2021).

Anon, 2021d. VirusTotal - Home. https://www.virustotal.com/gui/home/upload. (Accessed 29 September 2021).

Anon, 2022a. Obfuscated files or information, technique T1027 - enterprise | MITRE ATT&CK®. https://attack.mitre.org/techniques/T1027/. (Accessed 08 July 2022).

Anon, 2022b. VirusTotal - Home. https://www.virustotal.com/gui/home/upload. (Accessed 04 August 2022).

Anon, 2023a. Embedding layer. https://keras.io/api/layers/core_layers/embedding/. (Accessed 23 February 2023).

Anon, 2023b. GitHub - leocsato/detector_mw: Optimizer for malware detection. Api calls sequence of benign files are provided. https://github.com/leocsato/detector_mw. (Accessed 15 January 2023).

Anon, 2023c. HCRL - [HIDE]APIMDS-dataset. https://ocslab.hksecurity.net/apimds-dataset. (Accessed 20 June 2023).

Anon, 2023d. Tokenizer base class. https://keras.io/api/keras_nlp/tokenizers/tokenizer/. (Accessed 23 February 2023).

Anon, 2023e. What is a signature and how can I detect it? https://home.sophos.com/en-us/security-news/2020/what-is-a-signature. (Accessed 11 February 2023).

Apruzzese, G., Colajanni, M., Ferretti, L., Guido, A., Marchetti, M., 2018. On the effectiveness of machine and deep learning for cyber security. In: 2018 10th International Conference on Cyber Conflict. CyCon, pp. 371–390. http://dx.doi.org/10.23919/CYCON.2018.8405026.

Avci, C., Tekinerdogan, B., Catal, C., 2023. Analyzing the performance of long short-term memory architectures for malware detection models. Concurr. Comput.: Pract. Exper. e7581.

Blokhin, K., Saxe, J., Mentis, D., 2013. Malware similarity identification using call graph based system call subsequence features. In: 2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops. pp. 6–10. http://dx.doi.org/10.1109/ICDCSW.2013.55.

Bostami, B., Ahmed, M., 2020. Deep learning meets malware detection: An investigation. In: Fadlullah, Z.M., Khan Pathan, A.-S. (Eds.), Combating Security Challenges in the Age of Big Data: Powered By State-of-the-Art Artificial Intelligence Techniques. pp. 137–155. http://dx.doi.org/10.1007/978-3-030-35642-2_7.

Catak, F.O., Yazı, A.F., Elezaj, O., Ahmed, J., 2020. Deep learning based sequential model for malware analysis using Windows exe API Calls. PeerJ Comput. Sci. 6, e285. http://dx.doi.org/10.7717/peerj-cs.285.

Ceschin, F., Pinage, F., Castilho, M., Menotti, D., Oliveira, L.S., Gregio, A., 2018. The need for speed: An analysis of brazilian malware classifiers. IEEE Secur. Priv. 16 (6), 31–41.

Chaganti, R., Ravi, V., Pham, T.D., 2022. Image-based malware representation approach with EfficientNet convolutional neural networks for effective malware classification. J. Inf. Secur. Appl. 69, 103306.

Chen, X., Hao, Z., Li, L., Cui, L., Zhu, Y., Ding, Z., Liu, Y., 2022. CruParamer: Learning on parameter-augmented API sequences for malware detection. IEEE Trans. Inf. Forensics Secur. 17, 788–803.

Chng, M.Z., 2023. Using activation functions in neural networks - MachineLearningMastery.com. https://machinelearningmastery.com/using-activation-functions-in-neural-networks. (Accessed 07 June 2023).

Cho, I.K., Kim, T.G., Shim, Y.J., Ryu, M., Im, E.G., 2016. Malware analysis and classification using sequence alignments. Intell. Autom. Soft Comput. 22 (3), 371–377. http://dx.doi.org/10.1080/10798587.2015.1118916.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y., 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078.

Chung, J., Gulcehre, C., Cho, K., Bengio, Y., 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555.

Cisco, 2020. Cisco Annual Internet Report (2018–2023) White Paper.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., Kuksa, P., 2011. Natural language processing (almost) from scratch. J. Mach. Learn. Res. 12 (ARTICLE), 2493–2537.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., 2022. Introduction to Algorithms. MIT Press.

Ding, Y., Xia, X., Chen, S., Li, Y., 2018. A malware detection method based on family behavior graph. Comput. Secur. 73, 73–86. http://dx.doi.org/10.1016/j.cose.2017.10.007.

Drapkin, A., 2022. Over 100 million pieces of malware were made for Windows users in 2021. https://tech.co/news/windows-users-malware. (Accessed 14 June 2022).

Fesseha, A., Xiong, S., Emiru, E.D., Diallo, M., Dahou, A., 2021. Text classification based on convolutional neural networks and word embedding for low-resource languages: Tigrinya. Information 12 (2), 52.

Fukushima, K., 1979. Neural network model for a mechanism of pattern recognition unaffected by shift in position-neocognitron. IEICE Tech. Rep. A 62 (10), 658–665.

Gibert, D., Mateu, C., Planes, J., 2020. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. J. Netw. Comput. Appl. 153, http://dx.doi.org/10.1016/j.jnca.2019.102526.

Gibert, D., Planes, J., Mateu, C., Le, Q., 2022. Fusing feature engineering and deep learning: A case study for malware classification. Expert Syst. Appl. 207, 117957.

Han, W., Xue, J., Wang, Y., Liu, Z., Kong, Z., 2019. MalInsight: A systematic profiling based malware detection framework. J. Netw. Comput. Appl. 125, 236–250. http://dx.doi.org/10.1016/j.jnca.2018.10.022.

Hellal, A., Mallouli, F., Hidri, A., Aljamaeen, R.K., 2020. A survey on graph-based methods for malware detection. In: 2020 4th International Conference on Advanced Systems and Emergent Technologies. pp. 130–134. http://dx.doi.org/10.1109/IC_ASET49463.2020.9318301.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780.

Hubel, D.H., Wiesel, T.N., 1968. Receptive fields and functional architecture of monkey striate cortex. J. Physiol. 195 (1), 215–243.

Huda, S., Abawajy, J., Alazab, M., Abdollalihian, M., Islam, R., Yearwood, J., 2016. Hybrids of support vector machine wrapper and filter based framework for malware detection. Future Gener. Comput. Syst. 55, 376–390. http://dx.doi.org/10.1016/j.future.2014.06.001.

Hulstaert, L., 2022. Understanding model predictions with LIME | by Lars Hulstaert | Towards Data Science. https://towardsdatascience.com/understanding-model-predictions-with-lime-a582fdff3a3b. (Accessed 03 July 2022).

Jadon, S., 2020. A survey of loss functions for semantic segmentation. In: 2020 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology. CIBCB, IEEE, pp. 1–7.

Jha, S., Fredrikson, M., Christodoresu, M., Sailer, R., Yan, X., 2013. Synthesizing near-optimal malware specifications from suspicious behaviors. In: 2013 8th International Conference on Malicious and Unwanted Software: "The Americas". MALWARE, pp. 41–50. http://dx.doi.org/10.1109/MALWARE.2013.6703684.

Jing, C., Wu, Y., Cui, C., 2022. Ensemble dynamic behavior detection method for adversarial malware. Future Gener. Comput. Syst. 130, 193–206.

Jovanovic, B., 2022. A not-so-common cold: Malware statistics in 2022. https://dataprot.net/statistics/malware-statistics/. (Accessed 14 June 2022).

Karbab, E.B., Debbabi, M., 2019. Maldy: Portable, data-driven malware detection using natural language processing and machine learning techniques on behavioral analysis reports. Digit. Investig. 28, S77–S87. http://dx.doi.org/10.1016/j.diin.2019.01.017.

Khan, S., Rahmani, H., Shah, S.A.A., Bennamoun, M., 2018. A guide to convolutional neural networks for computer vision. Synth. Lect. Comput. Vis. 8 (1), 1–207.

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), http://dx.doi.org/10.1155/2015/659101.

Kim, Y., 2014. Convolutional neural networks for sentence classification. arXiv:1408.5882.

Kingma, D.P., Ba, J., 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Kiranyaz, S., Avci, O., Abdeljaber, O., Ince, T., Gabbouj, M., Inman, D.J., 2021. 1D convolutional neural networks and applications: A survey. Mech. Syst. Signal Process. 151, 107398. http://dx.doi.org/10.1016/j.ymssp.2020.107398.

Lajevardi, A.M., Parsa, S., Amiri, M.J., 2022. Markhor: malware detection using fuzzy similarity of system call dependency sequences. J. Comput. Virol. Hacking Techn. 18 (2), 81–90.

Le, Q., Boydell, O., Mac Namee, B., Scanlon, M., 2018. Deep learning at the shallow end: Malware classification for non-domain experts. Digit. Investig. 26, S118–S126.

Li, C., Lv, Q., Li, N., Wang, Y., Sun, D., Qiao, Y., 2022a. A novel deep framework for dynamic malware detection based on api sequence intrinsic features. Comput. Secur. 116, 102686.

Li, S., Zhou, Q., Zhou, R., Lv, Q., 2022b. Intelligent malware detection based on graph convolutional network. J. Supercomput. 78 (3), 4182–4198.

Liu, Y., Wang, Y., 2019. A robust malware detection system using deep learning on API calls. In: 2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference. ITNEC, IEEE, pp. 1456–1460.

Liu, F., Zheng, L., Zheng, J., 2020. HieNN-DWE: A hierarchical neural network with dynamic word embeddings for document level sentiment classification. Neurocomputing 403, 21–32. http://dx.doi.org/10.1016/j.neucom.2020.04.084.

Lynn, H.M., Pan, S.B., Kim, P., 2019. A deep bidirectional GRU network model for biometric electrocardiogram classification based on recurrent neural networks. IEEE Access 7, 145395–145405.

Mandelbaum, A., Shalev, A., 2016. Word embeddings and their use in sentence classification tasks. arXiv:1610.08229.

Maniath, S., Ashok, A., Poornachandran, P., Sujadevi, V., Sankar A.U., P., Jan, S., 2017. Deep learning LSTM based ransomware detection. In: 2017 Recent Developments in Control, Automation & Power Engineering. RDCAPE, pp. 442–446.

Maniriho, P., 2022. MalbehavD-V1: A new Dataset of API calls extracted from Windows PE files of benign and malware. https://github.com/mpasco/MalbehavD-V1. (Accessed 07 April 2022).

Maniriho, P., Mahmood, A.N., Chowdhury, M.J.M., 2022. A study on malicious software behaviour analysis and detection techniques: Taxonomy, current trends and challenges. Future Gener. Comput. Syst. 130, 1–18. http://dx.doi.org/10.1016/j.future.2021.11.030.

Mathew, J., Ajay Kumara, M., 2020. API call based malware detection approach using recurrent neural network—LSTM. In: Intelligent Systems Design and Applications: 18th International Conference on Intelligent Systems Design and Applications, Vol. 1. ISDA 2018, Held in Vellore, India, December 6-8, 2018, Springer, pp. 87–99.

Medsker, L., Jain, L.C., 1999. Recurrent Neural Networks: Design and Applications. CRC Press.

Mehrabi, N., Morstatter, F., Saxena, N., Lerman, K., Galstyan, A., 2019. A survey on bias and fairness in machine learning. arXiv preprint arXiv:1908.09635.

Microsoft, 2021. Programming reference for the Win32 API - Win32 apps | Microsoft Docs. https://docs.microsoft.com/en-us/windows/win32/api/. (Accessed 01 January 2021).

Mikolov, T., Chen, K., Corrado, G., Dean, J., 2013. Efficient estimation of word representations in vector space. arXiv:1301.3781.

Mimura, M., 2023. Impact of benign sample size on binary classification accuracy. Expert Syst. Appl. 211, 118630.

Mimura, M., Ito, R., 2022. Applying NLP techniques to malware detection in a practical environment. Int. J. Inf. Secur. 21 (2), 279–291.

Mira, F., Brown, A., Huang, W., 2016. Novel malware detection methods by using LCS and LCSS. In: 2016 22nd International Conference on Automation and Computing. ICAC, pp. 554–559. http://dx.doi.org/10.1109/IConAC.2016.7604978.

MIT, 2022. GitHub - TeamHG-Memex/eli5: A library for debugging/inspecting machine learning classifiers and explaining their predictions. https://github.com/TeamHG-Memex/eli5. (Accessed 16 July 2022).

Molnar, C., 2020. Interpretable Machine Learning. Lulu. com.

Moraffah, R., Karami, M., Guo, R., Raglin, A., Liu, H., 2020. Causal interpretability for machine learning-problems, methods and evaluation. ACM SIGKDD Explor. Newsl. 22 (1), http://dx.doi.org/10.1145/3400051.3400058.

Morato, D., Berrueta, E., Magaña, E., Izal, M., 2018. Ransomware early detection by the analysis of file sharing traffic. J. Netw. Comput. Appl. 124, 14–32.

Moskovitch, R., Stopel, D., Feher, C., Nissim, N., Japkowicz, N., Elovici, Y., 2009. Unknown malcode detection and the imbalance problem. J. Comput. Virol. 5, 295–308.

Naik, N., Jenkins, P., Savage, N., Yang, L., Boongoen, T., Iam-On, N., 2021. Fuzzy-import hashing: A static analysis technique for malware detection. Forensic Sci. Int. Digit. Investig. 37, 301139. http://dx.doi.org/10.1016/j.fsidi.2021.301139.

Nair, V.P., Jain, H., Golecha, Y.K., Gaur, M.S., Laxmi, V., 2010. Medusa: Metamorphic malware dynamic analysis usingsignature from api. In: Proceedings of the 3rd International Conference on Security of Information and Networks. pp. 263–269. http://dx.doi.org/10.1145/1854099.1854152.

Najafabadi, M.M., Villanustre, F., Khoshgoftaar, T.M., Seliya, N., Wald, R., Muharemagic, E., 2015. Deep learning applications and challenges in big data analytics. J. Big Data 2 (1), 1–21. http://dx.doi.org/10.1186/s40537-014-0007-7.

Nappa, A., Rafique, M.Z., Caballero, J., 2015. The MALICIA dataset: identification and analysis of drive-by download operations. Int. J. Inf. Secur. 14 (1), 15–33. http://dx.doi.org/10.1007/s10207-014-0248-7.

Nawaz, M.S., Fournier-Viger, P., Nawaz, M.Z., Chen, G., Wu, Y., 2022. MalSPM: Metamorphic malware behavior analysis and classification using sequential pattern mining. Comput. Secur. 118, 102741.

Nissim, N., Moskovitch, R., Rokach, L., Elovici, Y., 2014. Novel active learning methods for enhanced PC malware detection in windows OS. Expert Syst. Appl. 41 (13), 5843–5857.

Nunes, M., Burnap, P., Rana, O., Reinecke, P., Lloyd, K., 2019. Getting to the root of the problem: A detailed comparison of kernel and user level data for dynamic malware analysis. J. Inf. Secur. Appl. 48, 102365.

Pan, Z.-P., Feng, C., Tang, C.-J., 2016. Malware Classification Based on the Behavior Analysis and Back Propagation Neural Network. In: 3rd Annual International Conference on Information Technology and Applications, Vol. 7. ITA 2016, pp. 1–5. http://dx.doi.org/10.1051/itmconf/20160702001.

Pei, X., Yu, L., Tian, S., 2020. AMalNet: A deep learning framework based on graph convolutional networks for malware detection. Comput. Secur. 93, 101792. http://dx.doi.org/10.1016/j.cose.2020.101792.

Pektaş, A., Acarman, T., 2017. Classification of malware families based on runtime behaviors. J. Inf. Secur. Appl. 37, 91–100. http://dx.doi.org/10.1016/j.jisa.2017.10.005.

Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., Cavallaro, L., 2019. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In: 28th USENIX Security Symposium. USENIX Security 19, pp. 729–746.

Peng, X., Xian, H., Lu, Q., Lu, X., 2021. Semantics aware adversarial malware examples generation for black-box attacks. Appl. Soft Comput. 109, 107506.

Pennington, J., Socher, R., Manning, C.D., 2014. Glove: Global vectors for word representation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing. EMNLP, pp. 1532–1543.

Pinhero, A., M L, A., P, V., Visaggio, C., N, A., S, A., S, A., 2021. Malware detection employed by visualization and deep neural network. Comput. Secur. 105, 102247. http://dx.doi.org/10.1016/j.cose.2021.102247.

Pirscoveanu, R.S., Hansen, S.S., Larsen, T.M.T., Stevanovic, M., Pedersen, J.M., Czech, A., 2015. Analysis of malware behavior: Type classification using machine learning. In: 2015 International Conference on Cyber Situational Awareness, Data Analytics and Assessment. CyberSA, http://dx.doi.org/10.1109/CyberSA.2015.7166115.

Pittaras, N., Giannakopoulos, G., Papadakis, G., Karkaletsis, V., 2020. Text classification with semantically enriched word embeddings. Nat. Lang. Eng. 1–35. http://dx.doi.org/10.1017/S1351324920000170.

Pypi, 2021. PyPI · The Python package index. https://pypi.org/. (Accessed 22 August 2021).

Qin, B., Wang, Y., Ma, C., 2020. API call based ransomware dynamic detection approach using textCNN. In: 2020 International Conference on Big Data, Artificial Intelligence and Internet of Things Engineering. ICBAIE, IEEE, pp. 162–166.

Rafique, M.F., Ali, M., Qureshi, A.S., Khan, A., Mirza, A.M., 2020. Malware classification using deep learning based feature extraction and wrapper based feature selection technique. arXiv:1910.10958.

Ribeiro, M.T., Singh, S., Guestrin, C., 2016. "Why should i trust you?" Explaining the predictions of any classifier. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 1135–1144.

Ribeiro, M.T., Singh, S., Guestrin, C., 2018. Anchors: High-precision model-agnostic explanations. In: AAAI Conference on Artificial Intelligence. AAAI.

Rieck, K., Trinius, P., Willems, C., Holz, T., 2011. Automatic analysis of malware behavior using machine learning. J. Comput. Secur. 19 (4), 639–668. http://dx.doi.org/10.3233/JCS-2010-0410.

Ruth, C., 2023. Over 95% of all new malware threats discovered in 2022 are aimed at Windows - Atlas VPN. https://atlasvpn.com/blog/over-95-of-all-new-malware-threats-discovered-in-2022-are-aimed-at-windows. (Accessed 11 February 2023).

Sai, K.N., Thanudas, B., Sreelal, S., Chakraborty, A., Manoj, B., 2019. MACA-I: a malware detection technique using memory management API call mining. In: TENCON 2019-2019 IEEE Region 10 Conference. TENCON, IEEE, pp. 527–532.

Saket, S., 2021. (9) count vectorizer vs TFIDF vectorizer | Natural language processing | LinkedIn. https://www.linkedin.com/pulse/count-vectorizers-vs-tfidf-natural-language-processing-sheel-saket/. (Accessed 03 October 2021).

Sethi, K., Tripathy, B.K., Chaudhary, S.K., Bera, P., 2017. A novel malware analysis for malware detection and classification using machine learning algorithms. In: SIN '17: Proceedings of the 10th International Conference on Security of Information and Networks. pp. 107–116. http://dx.doi.org/10.1145/3136825.3136883.

Sharma, N., Sharma, R., Jindal, N., 2021. Machine learning and deep learning applications-a vision. Glob. Transitions Proc. 2 (1), 24–28. http://dx.doi.org/10.1016/j.gltp.2021.01.004, 1st International Conference on Advances in Information, Computing and Trends in Data Engineering (AICDE - 2020).

Silberschatz Abraham, G.B.P., 2018. Operating System Concepts, tenth ed. Wiley, p. 1259.

Singh, J., Singh, J., 2020. Detection of malicious software by analyzing the behavioral artifacts using machine learning algorithms. Inf. Softw. Technol. 121, http://dx.doi.org/10.1016/j.infsof.2020.106273.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R., 2014. Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. 15 (1), 1929–1958.

Stenne, M.A., 2021. Quick introduction to Windows API. https://users.physics.ox.ac.uk/~Steane/cpp_help/winapi_intro.htm. (Accessed on 14 May 2021).

Suaboot, J., Tari, Z., Mahmood, A., Zomaya, A.Y., Li, W., 2020. Sub-curve HMM: A malware detection approach based on partial analysis of API call sequences. Comput. Secur. 92, 1–15. http://dx.doi.org/10.1016/j.cose.2020.101773.

Sukul, M., Lakshmanan, S.A., Gowtham, R., 2022. Automated dynamic detection of ransomware using augmented bootstrapping. In: 2022 6th International Conference on Trends in Electronics and Informatics. ICOEI, pp. 787–794.

Sun, Z., Rao, Z., Chen, J., Xu, R., He, D., Yang, H., Liu, J., 2019. An Opcode sequences analysis method for unknown malware detection. In: Proceedings of the 2019 2nd International Conference on Geoinformatics and Data Analysis. pp. 15–19. http://dx.doi.org/10.1145/3318236.3318255.

Tekerek, A., Yapici, M.M., 2022. A novel malware classification and augmentation model based on convolutional neural network. Comput. Secur. 112, 102515.

Tien, C.-W., Chen, S.-W., Ban, T., Kuo, S.-Y., 2020. Machine learning framework to analyze iot malware using elf and opcode features. Digit. Threat. Res. Pract. 1 (1), 1–19.

Tirumala, S.S., Valluri, M.R., Nanadigam, D., 2020. Evaluation of feature and signature based training approaches for malware classification using autoencoders. In: 2020 International Conference on COMmunication Systems and NETworkS. COMSNETS, pp. 1–5. http://dx.doi.org/10.1109/COMSNETS48256.2020.9027373.

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, pp. 101–105. http://dx.doi.org/10.1109/IESYS.2017.8233569.

Udayakumar, N., Anandaselvi, S., Subbulakshmi, T., 2017. Dynamic malware analysis using machine learning algorithm. In: 2017 International Conference on Intelligent Sustainable Systems. ICISS, IEEE, pp. 795–800. http://dx.doi.org/10.1109/ISS1.2017.8389286.

Uppal, D., Sinha, R., Mehra, V., Jain, V., 2014. Exploring behavioral aspects of API calls for malware identification and categorization. In: 2014 International Conference on Computational Intelligence and Communication Networks. pp. 824–828. http://dx.doi.org/10.1109/CICN.2014.176.

Vemparala, S., Troia, F.D., Visaggio, C.A., Austin, T.H., Stamp, M., 2019. Malware detection using dynamic birthmarks. arXiv:1901.07312.

Vukotić, V., Raymond, C., Gravier, G., 2016. A step beyond local observations with a dialog aware bidirectional GRU network for spoken language understanding. In: Interspeech.

Wang, J., Bao, R., Zhang, Z., Zhao, H., 2022. Rethinking textual adversarial defense for pre-trained language models. IEEE/ACM Trans. Audio Speech Lang. Process. 30, 2526–2540.

Wüchner, T., Cisłak, A., Ochoa, M., Pretschner, A., 2019. Leveraging compression-based graph mining for behavior-based malware detection. IEEE Trans. Dependable Secure Comput. 16 (1), 99–112. http://dx.doi.org/10.1109/TDSC.2017.2675881.

Xiaofeng, L., Fangshuo, J., Xiao, Z., Shengwei, Y., Jing, S., Lio, P., 2019. ASSCA: API sequence and statistics features combined architecture for malware detection. Comput. Netw. 157, 99–111. http://dx.doi.org/10.1016/j.comnet.2019.04.007.

Xue, J., Wang, Z., Feng, R., 2022. Malicious network software detection based on API call. In: 2022 8th Annual International Conference on Network and Information Systems for Computers. ICNISC, IEEE, pp. 105–110.

Yaqub, M., Feng, J., Zia, M.S., Arshid, K., Jia, K., Rehman, Z.U., Mehmood, A., 2020. State-of-the-art CNN optimizer for brain tumor segmentation in magnetic resonance images. Brain Sci. 10 (7), 427.

Yesir, S., Soğukpinar, İ., 2021. Malware detection and classification using fasttext and BERT. In: 2021 9th International Symposium on Digital Forensics and Security. ISDFS, IEEE, pp. 1–6.

Yuan, S., Wu, X., 2021. Deep learning for insider threat detection: Review, challenges and opportunities. Comput. Secur. 104, 102221. http://dx.doi.org/10.1016/j.cose.2021.102221.

Zelinka, I., Amer, E., 2019. An ensemble-based malware detection model using minimum feature set. MENDEL 25 (2), 1–10. http://dx.doi.org/10.13164/mendel.2019.2.001.

Zhang, S.-H., Kuo, C.-C., Yang, C.-S., 2019. Static PE malware type classification using machine learning techniques. In: 2019 International Conference on Intelligent Computing and Its Emerging Applications. ICEA, IEEE, pp. 81–86. http://dx.doi.org/10.1109/ICEA.2019.8858297.

Pascal Maniriho received his B.Tech with Honors in Information and Communication Technology from Umutara Polytechnic, Rwanda, and a Master's degree in Computer Science from Institut Teknologi Sepuluh Nopember (ITS), Indonesia, in 2013 and 2018, respectively. He has been working in academia in Information Technology since 2019. He is currently pursuing his Ph.D. degree in cybersecurity at La Trobe University, Australia. His research interests include malware detection, data theft prevention, information security, machine learning and deep learning.

Abdun Naser Mahmood received the B.Sc. degree in applied physics and electronics, and the M.Sc. (research) degree in computer science from the University of Dhaka, Bangladesh, in 1997 and 1999, respectively, and the Ph.D. degree from the University of Melbourne, Australia, in 2008. He is currently an Associate Professor with the Department of Computer Science, School of Engineering and Mathematical Sciences, La Trobe University. His research interests include data mining techniques for scalable network traffic analysis, anomaly detection, and industrial SCADA security. He is a senior member of the IEEE.

Dr. Mohammad Jabed Morshed Chowdhury is currently working as Associate Lecturer at La Trobe University, Melbourne, Australia. He has earned his Ph.D. Candidate at Swinburne University of Technology, Melbourne, Australia. He has earned his double Masters in Information Security and Mobile Computing from Norwegian University of Science and Technology, Norway and University of Tartu, Estonia under the European Union's Erasmus Mundus Scholarship Program. He has published his research in top venues including TrustComm, HICSS, and REFSQ. He is currently working with Security, Privacy and Trust. He has published research work related to blockchain and cybersecurity in different top venues.