Malware Detection with Sequence-Based Machine Learning and Deep Learning

William B. Andreopoulos

Abstract In this chapter, we review sequence-based machine learning methods that are used for malware detection and classification. We start by reviewing the datatypes extracted from code: static features and dynamic traces of program execution. We review recent research that applies machine learning on opcode and API call sequences, call graphs, system calls, registry changes, information flow traces, as well as hybrid and raw data, to detect and classify malware. With a focus on metamorphic malware, we discuss Hidden Markov Models (HMMs) and Long Short-Term Memory (LSTM) networks. We describe their input formats, such as one-hot encoding and vector embeddings, the architecture of the machine learning models, the training process, and the output formats. Finally, we discuss commercial and open-source tools that are used for data extraction from software.

1 Introduction

Malware is software that is designed to disrupt or damage computer systems. According to Symantec, more than 669 million new malware variants were detected in 2018, which was an increase of more than 80% from 2017, with such trends continuing into 2019 and 2020 [41, 42]. Every day, there are at least 350,000 instances of new malware being created and detected. Additionally, 81% of all ransomware infections target businesses and organizations, making malware infections very costly. Malware and web-based attacks are the two most costly attack types—companies spent an average of US $2.4 million in defense in 2018–2019 [30, 48]. Clearly, malware detection is a critical task in computer security.

Malware detection can be based on static or dynamic software features, or a combination of those, or raw data. Static features are extracted from static files, while dynamic features are extracted during code execution or emulation. Static approaches often use features such as calls to external libraries, strings, and byte

W. B. Andreopoulos (✉)
Department of Computer Science, San Jose State University, San Jose, USA
e-mail: william.andreopoulos@sjsu.edu

M. Stamp et al. (eds.), *Malware Analysis Using Artificial Intelligence and Deep Learning*, https://doi.org/10.1007/978-3-030-62582-5_2

sequences for classification. Other static approaches extract higher level information from binaries, such as sequences of API calls or opcode information.

Signature-based detection that uses static data is widely used within commercial antivirus software. While this method is used widely in commercial antivirus tools and is capable of detecting specific malware families efficiently, it fails to detect new malware. Therefore, modern antivirus tools go beyond static signature-based detection and can detect unknown malwares more accurately using dynamic data. Using dynamic data, it is able to detect unknown malwares according to their behavior [10, 16].

Sequential features extracted from malware source code analysis have been used for the classification of malware with deep learning approaches. Sequences used in malware analysis have been used for LSTMs, as well as HMMs. Both features and sequences can be extracted by performing either static or dynamic analysis. There are many tools that can extract either or both data types. We provide an overview of these tools later in this chapter.

This chapter starts with describing the distinction between static and dynamic data for malware detection. Then we give an overview of the data type representation, recent malware detection methods, and tools for extraction of static and dynamic data. Finally, we describe how sequence-based deep learning algorithms can be used for malware detection.

2 Data Extraction

2.1 Static Data

Machine learning models for malware detection and classification can be trained on static features or attributes that are extracted from executable files [13]. Static analysis involves analyzing the malware software or code without actually executing the program. Static analysis involves disassembling software and representing some of its attributes as features for input to a machine learning tool [16]. Approaches to perform static analysis usually employ the executable binary file, while others use the source code file. Examples of static features include opcodes, API calls, control flow graphs, and many others. Specific features for training the machine learning model include extracting opcode sequences after disassembling the binary file, or extracting the control flow graph from the assembly file, extracting API calls from the binary, as well as extracting byte code sequences from the binary executable file [2, 9, 21, 28, 38, 44] (Figs. 1 and 2).

```
mov     ax,0000h
add     [0CB12h],c1
push    ds
add     [si+0D09h],dh
and     [bx+si+4C01h],di
push    sp
push    7219h
and     [bx+si+71h],dh
```

Fig. 1 Static opcode sequence example. Consider an example based on the assembly code snippet is shown above; the following sequences of length 2, also named bi-grams or 2 grams, can be generated: s1 = (mov, add), s2 = (add, push), s3 = (push, add), s4 = (add, and), s5 = (and, push), s6 = (push, push), and s7 = (push, and). Because most of the common operations that can be used for malicious purposes require more than one machine code operation, this example uses sequences of two opcodes, instead of individual opcodes [35]

2.2 *Dynamic Data*

Dynamic Analysis is the analysis of a software's behavior that is performed while executing the program. Some of the data that can be obtained through dynamic analysis are API calls, system calls, instruction traces, taint analysis, registry changes, memory writes, and information flow tracking. Dynamic analysis uses the tasks

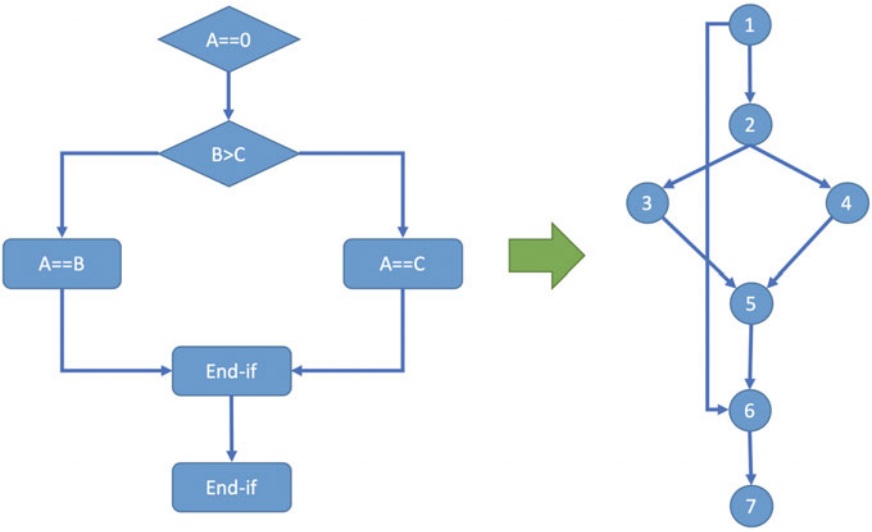

Fig. 2 Static control flow graph (static): A Control Flow Graph (CFG) is the graphical representation of control flow or computation during the execution of programs or applications. Control flow graphs are mostly used in the static analysis, as they can accurately represent the flow inside of a program unit

performed by a program while it is being executed in a virtualized environment [16]. Dynamic analysis is known to provide more accurate malware behavior detection results, but the data can be time-consuming to extract. There are several techniques and approaches followed to perform dynamic analysis.

As an example, this is an API system call extracted dynamically [16]:

```
fork(); getpid(); ioctl(); read(); write(); wait(); exit();
```

In order to input this to a sequence-based neural network we can represent it in one-hot encoded format. One-hot encoding is created by replacing the ith system call with an n-vector of zeros and a '1' in the ith position.

2.3 Hybrid Analysis

Besides static and dynamic analysis techniques, another analysis technique is the hybrid analysis of malware, which combines the advantages of both static and dynamic analyses [34]. Hybrid analysis technique is the combination of static and dynamic analysis techniques. In hybrid analysis, static analysis is done before the execution of a program and then dynamic analysis is done selectively based on the information obtained from static analysis. Dynamic analysis can be tedious due to multi-path execution. Static analysis can be used to selectively choose the path of execution for dynamic analysis. Often hybrid analysis results in increased accuracy and efficiency [34].

2.4 Alternative Approaches That Use Raw Data

In many cases the extraction of these sequences and features for LSTMs and HMMs can be costly, so approaches using raw bytes are preferred, if comparable accuracy can be obtained. For example, byte n-grams have been successfully used as features [43]. Also, it is possible to treat executable files as images and apply image analysis techniques. Images of malware executables have previously been used for classification using convolutional neural networks [25].

2.5 Evaluation of Malware Detection Accuracy

For API call sequences and opcode sequences, three test case scenarios have been implemented in previous work:

- The first case uses dynamic analysis data alone for both training and testing.
- The second case uses dynamic analysis data for training and static analysis data for testing.

- The third case uses static analysis data alone for both training and testing.

It is widely known that dynamic analysis data is more accurate and is therefore popular for training purposes. On the other hand, using dynamic data for training and testing is less efficient due to the complexity of extracting data in a dynamic fashion from software.

3 Recent Research Examples

In this section, we give an overview of recent research on malware detection and classification.

In [51], a system is proposed that extracts the API sequences from a Portable Executable (PE) file format. Then Objective-Oriented Association (OOA) based mining is done for malware classification. The system parses PE files and generates OOA rules efficiently for classification using FP Growth and a frequent-pattern tree. The system was tested on a large collection of PE files obtained from the anti-virus laboratory of KingSoft Corporation to compare various malware detection approaches. The accuracy and efficiency of the OOA system outperformed anti-virus software, such as Norton AntiVirus and McAfee VirusScan, as well as previous data mining-based detection systems that employed Naive Bayes, Support Vector Machine (SVM), and Decision Tree techniques.

In [31], malware is analyzed by abstracting the frequent itemsets in API call sequences. The authors focused on the usage of frequent messages in API call sequences, They hypothesized that frequent itemsets consisting of API names and/or API arguments could be valuable for identifying the behavior of malware. The authors clustered a dataset of malware binaries, demonstrating that using the frequent itemsets of API call sequences can achieve high precision for malware clustering while reducing the computation time.

In [30], a kernel object behavioral graph is created and graph isomorphism techniques and weighted common graph technique are used to calculate the hotpath for each malware family. And the unknown malware is then classified into whichever malware family has similar hotpaths.

In [12], dynamic instruction sequences are logged and are converted to abstract assembly blocks. Data mining algorithms are used to build a classification model using feature vectors extracted from the above data. For malware detection, the same method is used and scored against the classification model.

In [3], the authors propose a malware detection technique that uses instruction trace logs of executables collected dynamically. These traces are then constructed as graphs. The instructions are considered as nodes and the data from the instruction trace is used to calculate the transition probabilities. Then a similarity matrix is generated between the constructed graphs using different graph kernels. Finally, the constructed similarity matrix is input to an SVM for classification.

In [47], the authors presented a malware detection technique using dynamic analysis where fine-grained models are built to capture the behavior of malware using system calls information. Then they use a scanner to match the activity of any unknown program against these models to classify them as either benign or malware. The behavior models are represented in the form of graphs. The vertices denote the system calls and the edges denote the dependency between the calls where the input of one system call (vertex) depends on the output of another system call (vertex).

In [1], the authors propose a run-time monitoring based malware detection tool that extracts statistical features from malware using spatio-temporal information from API call logs. The spatial information is the arguments and return values of the API calls and temporal information is the sequence of the API calls. This information is used to build formal models that are fed to standard machine learning algorithms for malware detection.

From all the research given above, it is evident that dynamic analysis is a good source of information for malware behavior. Even though producing dynamic data incurs an execution overhead, a more accurate model can be obtained from dynamic analysis than using static analysis alone.

3.1 Hybrid Analysis

Hybrid analysis tools are developed for using the accuracy benefit that dynamic analysis offers and the static analysis' advantage of time complexity.

HDM Analyzer uses both static analysis and dynamic analysis techniques in the training phase and performs only static analysis in the testing phase. By combining static and dynamic analyses, HDM Analyzer achieved a better accuracy and time complexity than static and dynamic analysis methods alone [18]. The authors extracted a sequence of API calls for dynamic analysis, which is one of the most effective features for describing the behavior of a program.

In [9], the authors propose a framework for classification of malware using both static and dynamic analysis. They define the features or characteristics of malware as Mal-DNA (Malware DNA). Mal-DNA combines static, dynamic, and hybrid characteristics. Besides extracting the static features of malware, they extract dynamic data with debugging based behavior monitoring. Then they classify malware using machine learning algorithms.

In a slightly modified version, [19] apply n-grams method to the API calls extracted and use the above as a feature set for malware classification. Application Programming Interface (API) call sequences are commonly used features in intelligent malware detection systems. An API call sequence captures the activities of a program and, hence, it is useful data for mining of malicious behavior. Different order of each API call in a sequence may mean a different behavior model. Therefore, the order of API calls is an important issue to analyze malware behavior. The paper proposes a feature extraction approach for modeling malware behavior that extracts

API call sequences by dynamic analysis of executing programs. The novelty of the approach is utilizing n-grams to preserve the order of API calls.

In [17], a set of program API calls is extracted and combined with the control flow graphs (CFGs) to obtain a new representation model called API-CFG, where API calls form the edges in the control flow graph. This API-CFG is trained by a learning model and used as a classifier during the testing stage for malware detection. The behavior of a program is represented by a set of API calls. Therefore, a classifier can be employed to construct a learning model with a set of programs' API calls. Finally, an intelligent malware detection system is developed to detect unknown malwares automatically. This approach is capable of classifying benign and malicious code with high accuracy. The results show a statistically significant improvement over n-grams-based detection method.

In [23], dynamic malware detection is done using registers values set analysis. In this paper, a novel method is proposed based on similarities of binaries behaviors. At first, run-time behavior of the binary files is found and logged in a controlled environment tool. The approach assumes that the behavior of each binary can be represented by the values of memory contents in its run-time. That is, values stored in different registers while the malware is running in the controlled environment can be a distinguishing factor to discriminate it from those of benign programs. Then, the register values for each Application Programming Interface (API) call are extracted before and after API is invoked. After that, the changes of registers values throughout the executable file are traced to create a vector for each of the values of EAX, EBX, EDX, EDI, ESI, and EBP registers.

In [32], a new runtime kernel memory mapping method called allocation-driven mapping is introduced, which identifies dynamic kernel objects, including their types and lifetimes. The method works by capturing kernel object allocation and deallocation events. A benefit of kernel-based malware analysis includes providing a temporal view of kernel objects by performing a temporal analysis of kernel execution. Their system includes a temporal malware behavior monitor that tracks malware behavior by the manipulation of dynamic kernel objects. Allocation-driven mapping is shown to reliably analyze malware behavior by guiding the analysis only to the events relevant to a malware attack.

In [36], the API call sequences and assembly code are combined and a similarity-based matrix is produced that determines whether a portion of code has traces of a particular malware. This research showed good results by using API call sequences and Opcode sequences to give a good description of the behavior of a malware.

An orthogonal approach to the monitoring of function calls during the execution of a program, is the analysis of how the program processes data. The goal of information flow tracking is to propagate and track "taint-labeled" data throughout the system, while a program manipulating this data is executed. The data that should be tracked is specifically marked (tainted) with a corresponding label. Assignment statements, for example, usually propagate the taint-label of the source operand to the target [16].

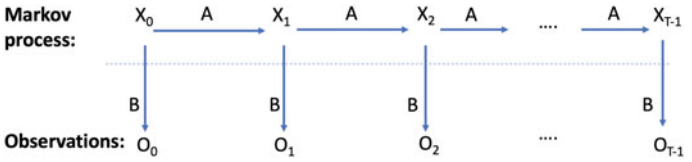

Fig. 3 Illustration of a generic Hidden Markov Model [40]

4 HMM Architecture

The HMM is based on augmenting the Markov chain. A Markov chain is a model that represents the probabilities of sequences of random variables, called states, each of which can take on values from some set. These sets can be words, or tags, or symbols representing anything, like the alphabet. In the case of malware, the sets of values may be opcodes or API calls.

A Markov chain makes an assumption that for predicting the future in a sequence, the current state is all that matters. The states before the current state only impact the future via the current state. For instance, to predict the next word you could consider the current word, but you should not examine previously seen words. Similarly, to predict if the states in a sequence produced by a piece of software constitute malware, you could use the states immediately preceding a state, but not states seen in the distant past.

HMMs have been used for malware detection. A Hidden Markov Model (HMM) is a machine learning model to represent probability distributions over a sequence of observations [22]. The HMM satisfies the markov property, i.e., the current state t is dependent only on $t-1$ and is independent of all states prior to $t-1$. A HMM is motivated by the idea of considering both observed events (such as words that we see in the input) and hidden events (such as part-of-speech tags) that we think of as causal factors in our probabilistic model. An HMM is specified by the following components:

- $Q = q_1 q_2 \ldots q_N$ a set of N states
- $O = o_1 o_2 \ldots o_T$ a sequence of T observations, each one drawn from a vocabulary $V = v_1, v_2, \ldots, v_V$
- $A = a_{11} \ldots a_{ij} \ldots a_{NN}$ a transition probability matrix A, each a_{ij} representing the probability of moving from state i to state j, s.t. $P_{j=1..N} a_{ij} = 1 \forall i$
- $B = b_i(o_j)$ a sequence of observation likelihoods, also called emission probabilities, each expressing the probability of an observation o_j being generated from a state i
- $\pi = \pi_1, \pi_2, \ldots, \pi_N$ an initial probability distribution over states. π_i is the probability that the Markov chain will start in state i. Some states j may have $\pi_j = 0$, meaning that they cannot be initial states.

4.1 Training for Malware Detection

The basic steps followed for performing malware detection using HMM are as follows. First, we select the observation data that the model should be trained for. In this case, the observed sequences represent software states that may originate from malware data. The observed sequences can be API calls sequence or opcode sequences. A model is trained with the above-observed sequences. After convergence, we get an accurate model that best fits the observed sequences. Next, we score a set of malware and benign files against the trained model. If the scores are higher than a predetermined threshold, the scores above the threshold can be classified as files from the malware family and ones that are below the threshold can be classified as files from the benign family [5] (Fig. 3).

4.2 Metamorphic Malware Detection

The use of static data is insufficient when dealing with the advanced malware obfuscation techniques such as code relocation, mutation, and polymorphism [11]. In [37], an opcode-based software similarity measure was developed, showing excellent results for metamorphic malware detection and classification. In [14], the metamorphic malware detection is done based on function call graph analysis. In [29], multiple sequence alignment algorithms from bioinformatics lead to viral code signatures that generalize successfully to previously known polymorphic variants of viruses.

The detection of metamorphic malware became more effective due to the application of Markov models and HMMs to malware detection. Profile Hidden Markov Models (PHMMs) are known for their success at detecting relations between DNA and protein sequences. When applied for malware detection it has been found that PHMMs can effectively detect metamorphic malware [40] and HMMs have also been successful in this regard. In [5], HMMs were used for malware classification. The HMM clustering results classify the malware samples into their appropriate families with good accuracy, providing a useful tool in malware analysis and classification.

5 LSTM Architecture

Long Short-Term Memory networks—usually just called "LSTMs"—are a special kind of Recurrent Neural Network (RNN), capable of learning long-term dependencies in sequences of events. They work well on a large variety of problems. Besides malware detection, they have also been widely used for sequence classification in other fields such as text mining and biology.

In this section, we analyze how LSTM sequence-based deep learning methods may be used for malware classification [24]. While static signature-based malware

detection methods are quick, static code analysis can be vulnerable to code obfuscation techniques. LSTMs offer the benefit that they don't rely on static analysis and can analyze a short snapshot of the runtime behavior. Behavioral data collected during file execution is more difficult to obfuscate but takes a long time to capture (typically up to 5 min). This often means that the malicious payload has likely already been delivered by the time it is detected.

In [39], LSTMs were applied to microarchitectural event traces captured through on-chip hardware performance counter (HPC) registers. The proposed LSTM approach achieved up to 11% higher detection accuracy compared to other sequence-based classification, such as HMM-based approaches in detecting obfuscated malware.

References [27, 46, 49] used sequences of API system calls for training an LSTM. In [27], they trained the LSTM model to learn from the most informative of sequences from the API-dataset based on their relative ranking as determined by Term Frequency–Inverse Document Frequency (TF–IDF) recommended features. They were able to achieve accuracy as high as 92% in detecting malware and benign code from an unknown test API-call sequence.

Reference [8] extracted 3-grams of opcode sequences and API call sequences. They then used attention mechanisms to identify API system calls that are more important than others for determining whether a file is malicious. They report this approach gave an accuracy that was 12% and 5% higher than conventional malware detection models using convolutional neural networks and skip-connected LSTM-based detection models, respectively.

Reference [39] used dynamic data in the form of microarchitectural event traces captured through on-chip hardware performance counter (HPC) registers. They combined this with localized feature extraction from image binaries corresponding to the application binaries. Using this advanced approach, an accuracy of 94% and nearly 90% is achieved in detecting normal and metamorphic malware created through code relocation obfuscation technique.

An LSTM variation is to use coupled forget and input gates. This variation on the LSTM is called the Gated Recurrent Unit, or GRU [7]. It combines the forget and input gates into a single "update gate." It also merges the cell state and hidden state. Instead of separately deciding what to forget and what to add new information to, GRUs make those decisions together. GRUs only forget when they are going to input something in its place. GRUs only input new values to the state when they forget something older. GRUs have also been used for malware detection [6].

5.1 LSTM Training

Input format: A single training data element consists of the label and an input word: xseq—a subsequence of a fixed size sampled randomly from the full original sequence. Reference [45] used one-hot encoding of logged API call sequences reflecting process behavior. One-hot encoding is a method for transforming categor-

ical data to numerical data by representing the ith categorical value from the universe as a numerical vector of zeros and a '1' in the ith position.

Reference [50] compared opcode embedding against one-hot encoding methods. One-hot encoding is simple to use and with the rather small number of Android opcodes, the sparseness of one-hot encoding does not cause a negative impact on efficiency. Another method is learning opcode embedding from data samples. Using opcode embedding achieves better malware detection results than one-hot encoding since opcode embedding captures the opcode semantic information better compared to one-hot encoding. The embedding idea comes from word vector learning in NLP, such as word2vec. Opcode embedding helps to learn the semantic information of opcode sequences and mine for malicious behaviors [50].

Reference [26] also input word embeddings derived from opcode sequences to LSTMs for malware detection and malware classification. Their evaluation results showed their proposed method can achieve an average AUC of 0.99 and an average AUC of 0.987, respectively.

Generally, an LSTM or RNN takes an input sequence of a fixed size for training or classification. An input sequence that is of a shorter size than the input layer of the LSTM can be padded with special characters [6]. Otherwise, it can either be trimmed or subsampled to derive samples of the desired size.

Reference [33] used trimming of sequences to detect whether or not an executable is malicious based on a short snapshot of behavioral data. They collected ten numerical machine activity data metrics (e.g., CPU and memory usage) as feature inputs, which are continuous numeric values, allowing for a large number of different machine states to be represented in a small vector of size 10. They used an ensemble of RNNs to build an RNN model able to predict whether an executable is malicious or benign within the first 5 s of execution with 94% accuracy. This was one of the first works to predict malicious code during execution. Previous dynamic analysis research collected data for around 5 min per sample [33].

In some works training the LSTM involved a subsampling of sequences from the original sequence. Specifically, training on a set of labeled sequences occurs by subsampling a number (say, 50–100) of short fixed-length sequence samples. Then training happens on each sample. For instance, [4] used samples extracted from Windows files. They trained a multiclass classification RNN, more specifically a LSTM on the dataset. This model for analyzing unstructured data was tested on unseen programs and the accuracy reached 67.60%, including six classes with five different types of malware.

The subsampling is chosen to ensure a fair representation of smaller and larger sequences. The number (m) of fixed-length subsequences sampled from each executable code sample should be proportional to the logarithm or square root of the sequence length, such that longer sequences will contribute more samples, but do not overwhelm the training. Each sample is a different subsequence xseq associated with the originating sequence. The sequence xseq can be one-hot encoded if the number of possible values is small.

Output format: The output from the final neuron is in a range from 0 to 1. This value is used to discriminate between a positive or negative classification value; in

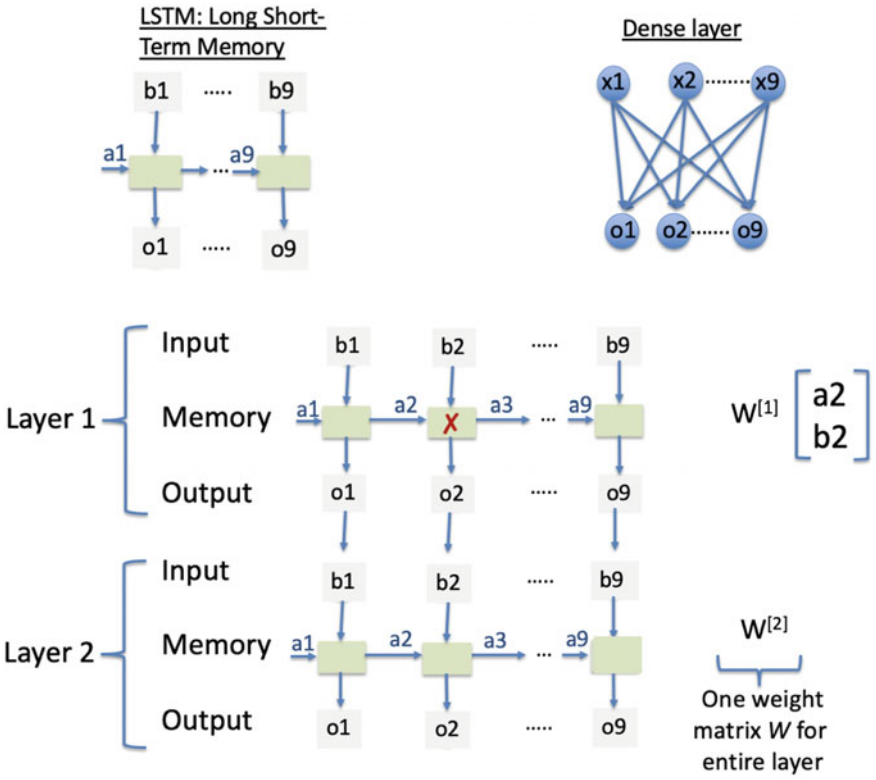

Fig. 4 An LSTM and a layered LSTM architecture

other words, if it is predicted to be malware or not malware. A neural network is a function

$$F(x_{seq}, x_f|\theta) = y. \tag{1}$$

that accepts sequences of points x_{seq} and the feature vector x_f. The function F also depends implicitly on the DL model parameters θ, which are determined during the training process. The output of the neural network, score y, is computed using the softmax function, which ensures that y satisfies $y \in [0, 1]$. By convention, the higher the score is for the sequence, the more likely the sequence is to be a true malware (Fig. 4).

Gradient Descent and Backpropagation

The training cycles are typically repeated for a specified number of epochs (such as 30 training cycles): The loss (error) is computed as target—actual output. E is the loss overall, computed by averaging loss over all instances of the training set.

The gradient of the loss is computed at the position we end up at. For neuron o_j at layer j:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}} \qquad o_j = \varphi(\text{net}_j) = \varphi\left(\sum_{k=1}^{n} w_{kj} o_k\right).$$

During gradient descent the weights are adjusted by the learning rate η:
∇w_{ij} = the product of $-\eta$ and the gradient of loss
∇w_{ij} changes w_{ij} such that E decreases in next epoch
Backpropagation then adjusts all weights from outer to the inner layers.
As hyperparameters, the model can be trained with a binary cross-entropy loss function and Adam optimizer. Usually, there are dropout layers to reduce overfitting.

Another possible architecture involves a hybrid of LSTM and a dense neural network. As input in this case, a single training data element consists of the label and two input words: x_{seq}—a subsequence of a fixed size sampled randomly from the full original sequence and x_f—a vector containing features extracted from the full sequence. The values of the feature vector x_f can be normalized to be bound within $[-1, 1]$. This architecture takes the sequences as input on one branch; and the features as matrices or vectors in another branch [24, 25] (Fig. 5).

6 Tools

6.1 IDA Pro

IDA Pro is a popular disassembler for generating assembly language source code from executables. It can also be used as a debugger. IDA Pro can be used to generate .asm files from which opcodes and windows API calls can be extracted. Also, IDA Pro is useful for collecting the instruction trace logs of executables.

6.2 OllyDbg

OllyDbg is a 32-bit disassembler and debugger, which is second best to IDA Pro. OllyDbg has limited features compared to IDA Pro.

6.3 Ether

Ether is an open-source tool for malware analysis that has been developed via hardware virtualization extensions and resides completely outside of the target OS. This disables the detection of guest software components. Many recent viruses can detect a debugger or a virtual environment during execution. Ether malware analysis tool overcomes this problem and hence provides a benefit as a tool for malware detection [15].

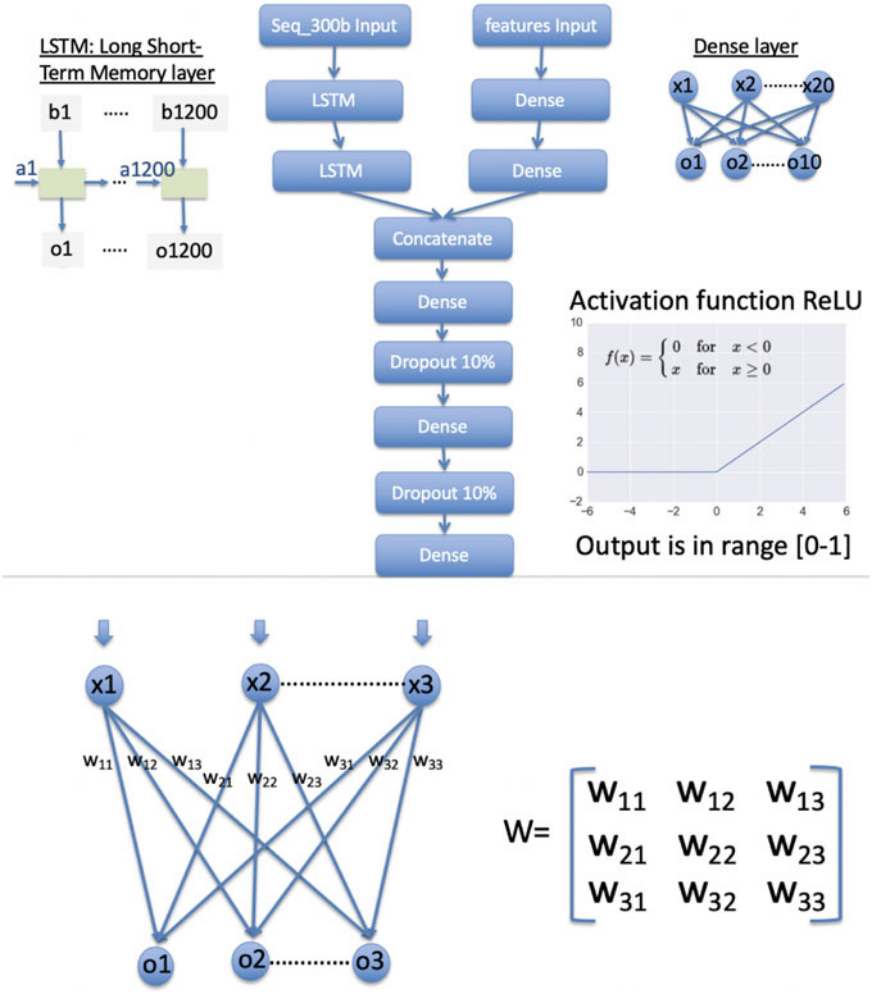

Fig. 5 A hybrid LSTM and dense architecture. A neuron in the dense layer has all-to-all connectivity and connections are described as a two-dimensional numerical matrix. Dropout layers are used to reduce overfitting

6.4 API Logger

API Logger is a tool that logs all API calls that meet the restrictions of the inclusion and exclusion lists. The inclusion list specifies which libraries need to be included and exclusion list specifies which libraries or functions can be ignored [20].

6.5 WinAPIOverride

WinAPIOverride is an advanced API monitoring software. Its main distinction is the ability to manually extract API calls by deciding the flow of the program during execution. In that sense, it fills the gap between classic API monitoring software and debuggers.

6.6 API Monitor

API Monitor is a software tool that also helps in monitoring and controlling API calls made by applications of processes. This tool needs to be run inside a virtual machine to analyze a malware and cannot be run in a sandboxed environment (www.rohitab.com/apimonitor. Accessed 07/14/2020).

6.7 BSA

Buster Sandbox Analyzer (BSA) is a tool that can decide if processes exhibit malicious activities based on dynamic analysis. In addition to analyzing the behavior of running processes, BSA keeps track of the changes made to the system, such as registry changes. The tool runs inside a Sandbox that protects the system from getting infected while executing the malware. BSA can generate API trace calls for win32 executables. Other tools similar to BSA that are useful for tracing in sandboxed environments are CWSandbox and Norman Sandbox (http://bsa.isoftware.nl. Accessed 07/14/2020).

7 Conclusion

In this chapter, we have compared data representations for static and dynamic datasets that can be used to classify and detect malware. The extracted data is input to a sequence-based machine learning or deep learning tool. Sequence-based machine learning provides a non-signature-based malware detection method that can effectively classify new and unknown types of malware, as well as metamorphic malware. Both static and dynamic datasets contain data types that are sequence-based. Using dynamic data offers a benefit over static data for detecting obfuscated code. The data can be trained upon and classified by sequence-based machine learning tools, such as HMMs and LSTMs. The sequence-based approaches are in contrast to training upon and classifying using feature-based machine learning tools, such as dense or convolutional neural networks, which often employ images or raw data from malware.

References

1. Ahmed, Faraz, Haider Hameed, M. Zubair Shafiq, and Muddassar Farooq. 2009. *Using spatio-temporal information in API calls with machine learning algorithms for malware detection*, 55. New York City: ACM Press.
2. Alqurashi, Saja, and Omar Batarfi. 2016. A comparison of malware detection techniques based on hidden Markov model. *Journal of Information Security* 07 (03): 215–223.
3. Anderson, Blake, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. 2011. Graph-based malware detection using dynamic analysis. *Journal in Computer Virology* 7 (4): 247–258.
4. Andrade, Eduardo de O, José Viterbo, Cristina N. Vasconcelos, Joris Guérin, and Flavia Cristina Bernardini. 2019. A model based on lstm neural networks to identify five different types of malware. *Procedia Computer Science* 159: 182–191.
5. Annachhatre, Chinmayee, Thomas H. Austin, and Mark Stamp. 2015. Hidden Markov models for malware classification. *Journal of Computer Virology and Hacking Techniques* 11 (2): 59–73.
6. Athiwaratkun, B, and J. W. Stokes. 2017. Malware classification with lstm and gru language models and a character-level cnn. In *2017 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, 2482–2486.
7. Cho, Kyunghyun, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 1724–1734, Doha, Qatar. Association for Computational Linguistics.
8. Choi, Sunoh, Jangseong Bae, Changki Lee, Youngsoo Kim, and Jonghyun Kim. 2020. Attention-based automated feature extraction for malware analysis. *Sensors* 20 (10): 2893.
9. Choi, Y.H, B.J. Han, B.C. Bae, H.G. Oh, and K.W. Sohn. 2012. Toward extracting malware features for classification using static and dynamic analysis. In *IEEE conference publication*.
10. Christodorescu, M, S Jha, S A Seshia, D Song, and R E Bryant. 2005. *Semantics-aware malware detection*, 32–46, IEEE.
11. Christodorescu , Mihai, and Somesh Jha. 2003. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th conference on USENIX security symposium - volume 12*, SSYM'03, 12. USA: USENIX Association.
12. Dai, Jianyong, Ratan Guha, and Joohan Lee. 2009. Efficient virus detection using dynamic instruction sequences. *Güncel Pediatri* 4 (5).
13. Damodaran, Anusha, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H. 2017. Austin, and Mark Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques* 13(1): 1–12.
14. Deshpande, Prasad. 2013. Metamorphic detection using function call graph analysis.
15. Dinaburg, Artem, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. *Ether: Malware analysis via hardware virtualization extensions*, 51. New York City: ACM Press.
16. Egele, Manuel, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2012. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys* 44 (2): 1–42.
17. Eskandari, Mojtaba, and Sattar Hashemi. 2012. A graph mining approach for detecting unknown malwares. *Journal of Visual Languages and Computing* 23 (3): 154–162.
18. Eskandari, Mojtaba, Zeinab Khorshidpour, and Sattar Hashemi. 2013. Hdm-analyser: A hybrid analysis approach based on data mining techniques for malware detection. *Journal of Computer Virology and Hacking Techniques* 9 (2): 77–93.
19. Eskandari, Mojtaba, Zeinab Khorshidpur, and Sattar Hashemi. 2012. *To incorporate sequential dynamic features in malware detection engines*, 46–52, IEEE.
20. Fasikhov, R. The api logger tool. http://blackninja2000.narod.ru/rus/api_logger.html. Accessed 14 July 2020.
21. Gandotra, Ekta, Divya Bansal, and Sanjeev Sofat. 2014. Malware analysis and classification: A survey. *Journal of Information Security* 05 (02): 56–64.

22. Ghahramani, Zoubin. 2001. An introduction to hidden Markov models and bayesian networks. *International Journal of Pattern Recognition and Artificial Intelligence* 15 (01): 9–42.
23. Ghiasi, Mahboobe, Ashkan Sami, and Zahra Salehi. 2012. *Dynamic malware detection using registers values set analysis*, 54–59, IEEE.
24. Hr, Sandeep. 2019. *Static analysis of android malware detection using deep learning*, 841–845, IEEE.
25. Jain, Mugdha, William Andreopoulos, and Mark Stamp. 2020. Convolutional neural networks and extreme learning machines for malware classification. *Journal of Computer Virology and Hacking Techniques*.
26. Lu, Renjie. 2019. Malware detection with lstm using opcode language. ArXiv:abs/1906.04593.
27. Mathew, J, and M A Ajay Kumara. 2020. API call based malware detection approach using recurrent neural network – LSTM. In *Intelligent systems design and applications*, *Advances in intelligent systems and computing*, eds. Abraham, Ajith, Aswani Kumar Cherukuri, Patricia Melin, and NiketaEditors Gandhi, vol. 940, 87–99. Springer International Publishing.
28. Moser, Andreas, Christopher Kruegel, and Engin Kirda. 2007. *Limits of static analysis for malware detection*, 421–430, IEEE.
29. Naidu, Vijay, Jacqueline Whalley, and Ajit Narayanan. 2017. Exploring the effects of gap-penalties in sequence-alignment approach to polymorphic virus detection. *Journal of Information Security* 08: 296–327.
30. Park, Younghee, Douglas S. Reeves, and Mark Stamp. 2013. Deriving common malware behavior through graph clustering. *Computers and Security* 39: 419–430.
31. Qiao, Yong, Yuexiang Yang, Lin Ji, and Jie He. 2013. *Analyzing malware by abstracting the frequent itemsets in API call sequences*, 265–270, IEEE.
32. Rhee, Junghwan, Ryan Riley, Xu Dongyan, and Xuxian Jiang. 2010. Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory. In *Recent advances in intrusion detection*, *Lecture notes in computer science*, eds. Somesh Jha, Robin Sommer, and Christian Kreibich, vol. 6307, 178–197. Berlin: Springer.
33. Rhode, Matilda, Pete Burnap, and Kevin Jones. 2018. Early-stage malware prediction using recurrent neural networks. *Computers and Security* 77: 578–594.
34. Roundy, Kevin, A., and Barton P. Miller. 2010. Hybrid analysis and control of malware. In *Recent advances in intrusion detection*, *Lecture notes in computer science*, eds. Somesh Jha, Robin Sommer, Christian Kreibich, vol. 6307, 317–338. Berlin: Springer.
35. Runwal, Neha, Richard M. Low, and Mark Stamp. 2012. Opcode graph similarity and metamorphic detection. *Journal in Computer Virology* 8 (1–2): 37–52.
36. Shankarapani, Madhu K., Subbu Ramamoorthy, Ram S. Movva, and Srinivas Mukkamala. 2011. Malware detection using assembly and api call sequences. *Journal in Computer Virology* 7 (2): 107–119.
37. Shanmugam, Gayathri, Richard M. Low, and Mark Stamp. 2013. Simple substitution distance and metamorphic detection. *Journal of Computer Virology and Hacking Techniques* 9 (3): 159–170.
38. Shijo, P.V., and A. Salim. 2015. Integrated static and dynamic analysis for malware detection. *Procedia Computer Science* 46: 804–811.
39. Shukla, Sanket, Gaurav Kolhe, Sai Manoj P D, and Setareh Rafatirad. 2019. Stealthy malware detection using rnn-based automated localized feature extraction and classifier. In *2019 IEEE 31st international conference on tools with artificial intelligence (ICTAI)*, 590–597, IEEE.
40. Stamp, M. A revealing introduction to hidden Markov models. tutorial. www.cs.sjsu.edu/~stamp/RUA/HMM.pdf. Accessed 14 July 2020.
41. Symantec. Symantec Internet security threat report (ISTR) Volume 23. Technical report, Symantec, 03 2018.
42. Symantec. Symantec Internet security threat report (ISTR) Volume 24. Technical report, Symantec, 02 2019.
43. Tabish, S. Momina, M. Zubair Shafiq, and Muddassar Farooq. 2009. Malware detection using statistical analysis of byte-level file content. In *Proceedings of the ACM SIGKDD workshop on cybersecurity and intelligence informatics - CSI-KDD '09*, eds. Chen, Hsinchun, Marc Dacier,

Marie-Francine Moens, Gerhard Paass, and Christopher C. Yang, 23. New York City: ACM Press.
44. Le Thanh, Hieu. 2013. Analysis of malware families on android mobiles: detection characteristics recognizable by ordinary phone users and how to fix it. *Journal of Information Security* 04 (04): 213–224.
45. Tobiyama, S, Y. Yamaguchi, H. Shimada, T. Ikuse, and T. Yagi. 2016. Malware detection with deep neural network using process behavior. In *2016 IEEE 40th annual computer software and applications conference (COMPSAC)*, vol. 2, 577–582.
46. Vinayakumar, R, K P Soman, Prabaharan Poornachandran, and S Sachin Kumar. 2018. Detecting android malware using long short-term memory (lstm). *Journal of Intelligent and Fuzzy Systems* 34 (3): 1277–1288.
47. Wang, Xiaofeng. 2009. Effective and efficient malware detection at the end host. In *USENIX security symposium*, 351–366.
48. Wong, A. Symantec internet security threat report highlights. www.techarp.com/cybersecurity/2019-symantec-istr-highlights/. Accessed 14 July 2020.
49. Xiao, Xi, Shaofeng Zhang, Francesco Mercaldo, Guangwu Hu, and Arun Kumar Sangaiah. 2017. Android malware detection based on system call sequences and lstm. *Multimedia Tools and Applications* 78 (4): 1–21.
50. Yan, Jinpei, Yong Qi, and Qifan Rao. 2018. Lstm-based hierarchical denoising network for android malware detection. *Security and Communication Networks* 1–18: 2018.
51. Ye, Yanfang, Dingding Wang, Tao Li, Dongyi Ye, and Qingshan Jiang. 2008. An intelligent pe-malware detection system based on association mining. *Journal in Computer Virology* 4 (4): 323–334.