electronics

MDPI

Article

A Dynamic Analysis Data Preprocessing Technique for Malicious Code Detection with TF-IDF and Sliding Windows

Mihui Kim * and Haesoo Kim

School of Computer Engineering & Applied Mathematics, Computer System Institute, Hankyong National University, Jungang-ro, Anseong-si 17579, Gyeonggi-do, Republic of Korea; ww232330@hknu.ac.kr
* Correspondence: mhkim@hknu.ac.kr

Abstract: When using dynamic analysis data to detect malware, time-series data such as API call sequences are used to determine malicious activity through deep learning models such as recurrent neural networks (RNN). However, in API call sequences, APIs are called differently when different programs are executed. To use these data as input for deep learning, preprocessing is performed to unify the size of the data by adding dummy zeros to the data using the zero-padding technique. However, when the standard deviation of the size is significant, the amount of dummy data added increases, making it difficult for the deep learning model to reflect the characteristics of the data. Therefore, this paper proposes a preprocessing technique using term frequency–inverse document frequency (TF-IDF) and a sliding window algorithm. We trained the long short-term memory (LSTM) model on the data with the proposed preprocessing, and the results, with an accuracy of 95.94%, a recall of 97.32%, a precision of 95.71%, and an F1-score of 96.5%, showed that the proposed preprocessing technique is effective.

Keywords: malware; dynamic analysis; TF-IDF; sliding window; preprocessing

Citation: Kim, M.; Kim, H. A Dynamic Analysis Data Preprocessing Technique for Malicious Code Detection with TF-IDF and Sliding Windows. *Electronics* **2024**, *13*, 963. https://doi.org/10.3390/electronics13050963

Academic Editor: Simeone Marino

Received: 14 January 2024
Revised: 26 February 2024
Accepted: 27 February 2024
Published: 2 March 2024

1. Introduction

Malware detection techniques [1] are mainly divided into static and dynamic approaches. In general, detection through static analysis is used; however, static analysis is a method of analysis that uses opcodes, strings, and import address tables (IATs), without running the software. The static analysis makes it difficult to detect such malware when obfuscation and packing [2] are applied to the code or compilation process. This can be solved by dynamic analysis, which runs files in a virtual environment to analyze changes in the system. In this case, dynamic analysis approaches utilize time-series data, such as API call sequences, to determine the presence of maliciousness using deep learning models like recurrent neural networks (RNNs) [3]. Deep-learning models require input data of the same size for training. However, in the API call sequence, APIs are called different numbers of times when different programs run, ranging from as few as ten times to millions of times. To use these data as input for deep learning, the commonly used zero-padding technique [4] is applied to adjust the data to a fixed size. However, when there is a significant standard deviation in the data size, that is, when the difference between the smallest and largest data points is significant, the amount of dummy data increases, making it challenging for the deep learning model to capture the characteristics of the data. Therefore, a technique that reduces the loss of the time-series information of API call sequences while preprocessing the data to a fixed size is essential.

Accordingly, to convert dynamic analysis data to a fixed size, our previous study [5] used the term frequency–inverse document frequency (TF-IDF) [6] to map the APIs that determine benign and malicious API call sequences to high values and meaningless ones to low values. The preprocessing technique was performed by converting the data to a fixed size using a sliding window algorithm. In this paper, we describe the preprocessing in

detail and improve the preprocessing technique by computing the term frequency–inverse document frequency (TF-IDF) for category information and APIs and multiplying the results. We evaluate the performance of the proposed technique by comparing it with other methods.

The remainder of this paper is organized as follows: Section 2 introduces the related work on converting dynamic analysis data to a fixed length, and Section 3 describes the proposed preprocessing technique. Section 4 analyzes the experimental results of the proposed preprocessing method, and Section 5 concludes the paper.

2. Related Work

In [7], a technique was proposed that uses API2Vec embedding [8] and bidirectional long short-term memory (BiLSTM) [9] to reduce the dimensionality of information from data of varying lengths and extract the functional characteristics of APIs through convolutional operations. The extracted features are then transformed into temporal information. Additionally, for each pair of API operations and categories, the study extracts encoded operations and category vectors using an embedding layer and extracts similar features between APIs using BiLSTM. The extracted information is then integrated, and the weights are calculated using an attention layer. Finally, a Fully Connected Layer is used to determine the presence of malicious behavior.

In [10], API call sequences shorter than a certain length are padded and encoded into integers for conversion into fixed lengths. The API vectors for each sequence are then calculated using an embedding layer. Subsequently, a 1-D Convolutional Neural Network (CNN) [11] is used to extract the features of the sequences and reduce dimensionality through Max Pooling. This method converts API call sequences of varying lengths into fixed lengths and maps high-level API features to low dimensions using a bidirectional gated recurrent unit (BiGRU) [12] model for prediction.

In [13], the authors proposed preprocessing techniques using sequence-based embeddings and name-based embeddings. Sequence-based embeddings use Skip-Gram, a model from Word2Vec. Skip-Gram is a low-level dimensional vector embedding model that uses a central word in a one-hot vector to predict peripheral words. For name-based embedding, they calculate the frequency of each API through TF-IDF and perform malware detection using the CNNs-BiGRU model using the data embedded through Skip-Gram.

In [14], a preprocessing method of truncating data above a certain length and padding data under a certain length is adopted as a preprocessing method to propose an Android malware detection model using an RNN-based model.

However, ref. [7] requires multiple techniques, two deep learning models for dimensionality reduction and feature extraction, and layers for integrating features and calculating weights, resulting in significant overhead. The work in [10] incurs a large overhead in mapping sequences of excessive length to the vector space through the embedding layer and requires significant preprocessing before dimensionality reduction. A malware detection framework proposed in [13] extracts time-series information and API name information, respectively, which occupies more vector space for embedding, and uses a one-layer DNN to extract embedding data, which is resource-intensive for preprocessing. Ref. [14] showed that slicing and padding techniques can also achieve meaningful detection performance through certain length-specific experiments; however, considering that the maximum length of data is about 4000, the accuracy may decrease as the length of discarded data increases. Therefore, there is a need for preprocessing techniques that can reflect the information in the API with low overhead and minimal loss of information.

In this paper, we propose preprocessing techniques using TF-IDF for weight calculation and data reduction techniques using a sliding window to address the excessive overhead incurred by separate deep learning models and mapping large amounts of data to vector space.

3. Proposed Method

In this section, we describe the proposed preprocessing techniques. The proposed preprocessing techniques utilize TF-IDF and sliding windows. The TF-IDF calculates weights by considering the importance of words in a document. The TF represents the frequency of a specific API occurrence in a particular API call sequence, whereas the IDF is the inverse of the document frequency (DF). The DF refers to the number of API call sequences in which a specific API appears among all API call sequences. For example, if there were four API call sequences and a specific API existed in the first and second sequences, the TF of that API would be 40 in the first sequence and 200 in the second, whereas the DF would be 2. Using TF-IDF, we can calculate weights that consider the importance of APIs that frequently appear in the overall API call sequence and assign higher weights to APIs that appear frequently in specific API call sequences. A sliding window is a technique in which a fixed-size window moves and calculates the values within the window based on the data. Fixed-length data can be generated by dividing data of different sizes into a fixed number of windows and calculating only one value for each window.

Algorithm 1 demonstrates the proposed preprocessing procedure. The inputs consist of API call sequences (A), categories (C), and target lengths for transformation (N). The output is the preprocessed data ($W = w_1, w_2, w_3, \ldots, w_N$) with the target length. This algorithm comprises three main steps: (**a**) Weight calculation using TF-IDF (lines 1–3). The first step in this algorithm involves converting API call sequences and categories into meaningful numerical values. The algorithm uses the TF-IDF method, a popular technique in information retrieval, to calculate the weights. This represents a significant departure from the common approach of mapping API call sequences and categories to word dictionaries. This method effectively quantifies the importance of words in APIs and categories. (**b**) Preprocessing to separate the data into window units (lines 4–11). The second step involves subdividing the data into distinct window units. This is achieved using the sliding window technique, which allows the analysis of subsets of data. Each window unit represents a fixed data length. (**c**) Calculation of the final value based on the data from each window (lines 12–14). The final step of the algorithm involves the calculation of the final value. This value is derived from the data contained within each window.

Lines 1–2: Calculate the weights of the API call sequences and categories (W_A, W_c) for malicious and benign files using TF-IDF.

Table 1 provides examples of the input and output for process a, and the input and output data are arbitrary data used to understand Algorithm 1, not actual data.

Table 1. An example of inputs and outputs for step (**a**).

Step	Inputs	Outputs
(**a**)	$A = [API_1,\ API_2,\ API_3, \ldots, API_n]$ $C = [Cat_1,\ Cat_2, Cat_3, \ldots, Cat_n]$	W_{AC} = [0.23, 0.27, 0.19, 0.19, . . ., 0.49]

Process a calculates the TF-IDF by taking the input API call sequence ($[API_1,\ API_2, API_3, \ldots, API_n]$) and categories ($[Cat_1,\ Cat_2, Cat_3, \ldots, Cat_n]$) and outputs the result by multiplying the two values, resulting in [0.23, 0.27, 0.19, 0.19, . . ., 0.49].

Table 2 illustrates the process of calculating the TF-IDF in step (**a**). The TF calculates the APIs for each API call sequence in the input. In the input API call sequence data A_1, there are one each of API_6, API_4, and API_7; so, the TF values of the APIs in A_1 are 1 for API_4, API_6, and API_7. The DF represents the number of documents in which APIs are used. The IDF is obtained by dividing the DF value by the total number of documents and then taking the logarithm. In the entire input dataset, API_1 is used once by A_4, and API_4 is used twice by A_1 and A_2. API_1 has a DF value of 1, and API_4 has a DF value of 2. The IDF is calculated using Formula (1), which is 0.6931471806 for API_1 and 0.2876820725 for API_4.

Algorithm 1: Proposed TF-IDF and sliding window calculation algorithm

Input: API call sequences A; Categories C; Target length for transformation N
Output: Preprocessed data W
/*Calculate weights with TF-IDF*/
1 $W_A \leftarrow A$ calculated from TF-IDF
2 $W_C \leftarrow C$ calculated from TF-IDF
3 $W_{AC} \leftarrow W_A * W_C$
/*Preprocess before sliding window calculation*/
4 $W_n \leftarrow$ (length of W_{AC})$/N$
5 Convert W_n to integer
6 **if** (length of W_{AC}) mod N = 0 **then**
7 $W \leftarrow$ list chunk with W_{AC} and W_n
8 **end**
9 **else**
10 $W \leftarrow$ list chunk F with W_{AC}, W_n and N
11 **end**
/*Calculate the final value for each window*/
12 $W_{avg} \leftarrow$ Calculate the average of each window in W
13 $W \leftarrow$ Calculate the final values in W using W_{avg}
14 **return** W

$$ln\left(\frac{API}{1+df(API_n)}\right), \tag{1}$$

where API is the total number of API call sequences, and $df(API_n)$ is the DF value of the API.

Table 2. Examples of the TF-IDF calculation in the API.

	Example
Inputs	$A_1 = [API_6, API_4, API_7]$ $A_2 = [API_5, API_4, API_7]$ $A_3 = [API_2, API_5, API_3, API_5]$ $A_4 = [API_8, API_9, API_1]$
TF	$TF_1 = \{API_4 : 1,\ API_6 : 1,\ API_7 : 1\}$
	$TF_2 = \{API_4 : 1,\ API_5 : 1,\ API_7 : 1\}$
	$TF_3 = \{API_2 : 1,\ API_3 : 1,\ API_5 : 1\}$
	$TF_4 = \{API_1 : 1,\ API_8 : 1,\ API_9 : 1\}$
DF	$API_1 : 1, API_2 : 1, API_3 : 1, API_4 : 2, API_5 : 2, API_6 : 1, API_7 : 2, API_8 : 1, API_9 : 1$
IDF	$\{API_1 : 0.693147, API_2 : 0.693147,\ API_3 : 0.693147,$ $API_4 : 0.287682, API_5 : 0.287682, API_6 : 0.693147,$ $API_7 : 0.287682, API_8 : 0.693147,\ API_9 : 0.693147\}$
TF-IDF	$A_1 = [0.693147,\ 0.287682,\ 0.287682]$ $A_2 = [0.693147,\ 0.287682,\ 0.287682]$ $A_3 = [0.693147,\ 0.575364,\ 0.693147,\ 0.575364]$ $A_4 = [0.693147,\ 0.693147,\ 0.693147]$

The TF-IDF is obtained by multiplying the TF value by the IDF, resulting in W_A. If the TF-IDF is calculated for API_4, the TF value is 2, and the IDF value is 0.2876820725. Multiplying the two values, 0.5753641449, yields the TF-IDF value for API_4. The same process is performed for the categories, and the two values are multiplied to obtain W_{AC}.

Line 3: Multiply the two weights (W_A, W_c) calculated in Step 2 to obtain (W_{AC}).

Lines 4–5: Divide the length by the target length (N) and convert it into an integer. The window size (W_n) is determined during this step.

Lines 6–8: This divides the data into W_n units. When obtaining W_n, if the result of dividing the length of the API call sequence by the target length is an integer, the entire dataset is divided into W_n (List_chunk).

Algorithm 2 is a implementation of the List_chunk function. This necessitates two primary inputs: a list of weights, denoted as W_{AC}, and the size of the window, termed W_n. The W_{AC} list of weights is computed in line 2 of Algorithm 1: The final output provided by Algorithm 2 is a list in which each dataset is partitioned into windows with a size specified by W_n. First, variable L_c is initialized as a list. The algorithm then uses a loop structure. This loop iterates over the entire W_{AC} list, with each iteration advancing by a step size equivalent to W_n.; the algorithm is designed to extract a specific portion of the W_{AC} list during each iteration. This extraction process is performed using list indexing, which extracts elements as step sizes (W_n) at every step. Once these elements are extracted, the algorithm adds them to the list L_c.

Algorithm 2: Implementing List_chunk

Input: List of calculated weights in line 2 of algorithm 1 W_{AC}, Size of window W_n
Output: Chunked list L_C
1 Initialize with a list of L_C
2 **for** $i \leftarrow 0$ **to** length of W_{AC} **and** increment steps are W_n **do**
3 Extract size of W_n elements as list from W_{AC} and add to L_C
4 **end**
5 **return** L_C

Lines 9–11: When N is 10, and the length of the API call sequence is 43, W_n becomes 4. At this point, the resulting data length becomes 11, which differs from the target length. Therefore, a different approach must be applied, which is the process considered for List_chunk_F (line 10 in Algorithm 1).

In List_chunk_F, if the final data length exceeds the value of N, the window size for the subsequent windows increases by one. Specifically, when N is 10, and the length of the API call sequence is 43, the size of the first and seventh windows' W_n is 4, and from the eighth to the tenth windows, the W_n size becomes 5.

$$len(API\ Call\ Sequence) - W_n * N, \tag{2}$$

Equation (2) is used in List_chunk_F to calculate the window from which the size of W_n increases. $len(API\ Call\ Sequence)$ is the length of the API call sequence. When $len(API\ Call\ Sequence)$ is 43, N is 10, W_n is 4, and the result for (2) is 3. With 10 windows, data with a length of 10 are generated when the value becomes 5 by adding 1 to the eighth window, which is the third window from the end.

Algorithm 3 is a Python-based implementation of the function List_chunk_F. This function takes the following three inputs: The first is a list of weights, denoted as W_{AC}, calculated in line 2 of Algorithm 1. The second input is the window size, which is represented by variable W_n. The final input is the target length for the transformation, denoted as N. The output is a list in which each data point is separated into windows of a specified size. When the algorithm begins to execute, it sets two variables: First, L_c, is initialized as an empty list. The second variable, *idx*, is initialized with a value of zero. The algorithm initiates a loop in which size is determined by the length of the W_{AC} list. The step size of this loop is W_n. In each loop cycle, the N-*idx* value is compared with the value calculated using (2). If the result is false, then the algorithm executes the List_chunk operation, and *idx*, the index value, is increased by one. If the result is true, the algorithm increases the window size (W_n) by 1 and breaks the loop. Subsequently, in a new loop, the starting value for the range function in this new loop is determined by multiplying the window size (W_n) by the stored index value (*idx*), and the step size is increased by one from the current W_n value.

Algorithm 3: Implementing List_chunk_F

Input: List of calculated weights in line 2 of algorithm 1 W_{AC}, Size of window W_n, Target length for transformation N
Output: Chunked list L_C
1 Initialize with a list of L_C
2 Initialize idx
3 **for** $i \leftarrow 0$ **to** length of W_{AC} **and** increment steps are W_n **do**
4 **if** $N - idx$ = (length of $W_{AC} - W_n * N$) **then**
5 $W_n \leftarrow W_n + 1$
6 **Break**
7 **end**
8 Extract size of W_n elements as list from W_{AC} and add to L_C
9 $idx++$
10 **end**
11 **for** $i \leftarrow (W_n + 1) * idx$ **to** length of W_{AC} **and** increment steps are W_n **do**
12 Extract size of W_n elements as list from W_{AC} and add to L_C
13 **end**
14 **return** L_C

Table 3 provides examples of the inputs and outputs for process b. Process b represents the process of converting the input to N = 10. If the input length is 20, W_n is 2, and List_chunk is used because the input divided by N is a natural number. Given an input of [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20], the output will be [[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12], [13, 14], [15, 16], [17, 18], [19, 20]]. If the input has a length of 21, then W_n is 2. Because the value obtained by dividing 21 by 10 is not an integer, the result is the output using List_chunk_F. When the input is [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21], (2) is calculated as 21–2 × 10, which equals 1. Applying lines 3–8 of Algorithm 3, the loop is executed until the condition "10 − idx" becomes 1. When idx is 8, L_c becomes [[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12], [13, 14], [15, 16], [17, 18]]. When idx is 9, the condition in line 4 of Algorithm 3 becomes true; therefore, W_n increases by 1 and the first loop terminates. In addition, because $(W_n - 1) \times idx$ is 18, the second loop starts from the 18th data point of W_{AC} with a step of 3. Therefore, the final output after the last loop is [[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12], [13, 14], [15, 16], [17, 18], [19, 20, 21]].

Table 3. Example inputs and outputs for step (**b**).

Step	Inputs	Outputs
(**b**) (List_chunk)	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20] N = 10	[[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12], [13, 14], [15, 16], [17, 18], [19, 20]]
(**b**) (List_chunk_F)	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21] N = 10	[[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12], [13, 14], [15, 16], [17, 18], [19, 20, 21]]

Line 12: Calculate the average of each window ($W_{avg} = w_{1avg}, w_{2avg}, w_{3avg}, \ldots, w_{Navg}$).
Line 13: Use W_{avg} to calculate the value of the N(th) window.

$$w_{Nv} = \begin{cases} w_{Nmax}\ if\ number\ of\ s > w_{Navg} \\ w_{Nmin}\ if\ number\ of\ s > w_{Navg} \\ w_{Navg}\ if\ number\ of\ s = w_{Navg} \end{cases} \quad (3)$$

where w_{Nv} is the final value of the Nth window, w_{Nmax} is the maximum value of the Nth window, w_{Nmin} is the minimum value of the Nth window, w_{Navg} is the average value of the Nth window, and s is the preprocessed API in line 3.

Equation (3) is the process of selecting the representative values for each window. If more values are significant than the average (w_{Navg}) of the window length (W_n), the maximum value (w_{Nmax}) is selected from that window. On the other hand, if there are smaller values than the average, the minimum value (w_{Nmin}) is chosen. This process generates a dataset of API call sequences with N instances, each of length.

Table 4 provides examples of inputs and outputs for process (**c**). Process c calculates the average (W_{avg}) for each window of the input data [[1, 2, 3], [3, 4, 10], [5, 6, 4], [7, 8, 1], [9, 10, 4], [7, 11, 12], [10, 13, 14], [11, 15, 16], [4, 17, 18], [19, 20, 21]], where the window average values for the input data are [2, 5.67, 5, 5.33, 7.67, 10, 12.3, 14, 13, 20]. The outputs W are determined based on W_{avg} using process (2). Therefore, the results based on the inputs are [2, 3, 5, 8, 10, 12, 14, 16, 18, and 20].

Table 4. Example inputs and outputs for step (**c**).

Step	Inputs	Outputs
(**c**)	[[1, 2, 3], [3, 4, 10], [5, 6, 4], [7, 8, 1], [9, 10, 4], [7, 11, 12], [10, 13, 14], [11, 15, 16], [4, 17, 18], [19, 20, 21]]	[2, 3, 5, 8, 10, 12, 14, 16, 18, 20]

4. Performance Evaluation

4.1. Experimental Environments

The experiment was run on an Intel Xeon(R) Silver 4215R CPU @ 3.20 GHz CPU, 256 GB RAM, NVIDIA RTX A6000 GPU, Python 3.7.13, TensorFlow 2.7.0, Scikit-learn 1.0.2, NumPy 1.21.6, Pandas 1.3.5.

4.2. Experimental Datasets

The experimental data used in this study were raw PE files [15] provided by Practical Security Analytics for security and AI research purposes. Among the provided 201,549 samples, the dynamic data of benign and malicious files were extracted using the Cuckoo Sandbox [16] for 5 days. The total number of data points used was 16,590, with 9756 benign and 6834 malicious samples. The training, validation, and testing ratios were set as 6:2:2. The hash values of the software used to extract the dynamic data of benign and malicious files are available in the GitHub repository [17]. Figure 1 presents a table showing the number of data for each data length.

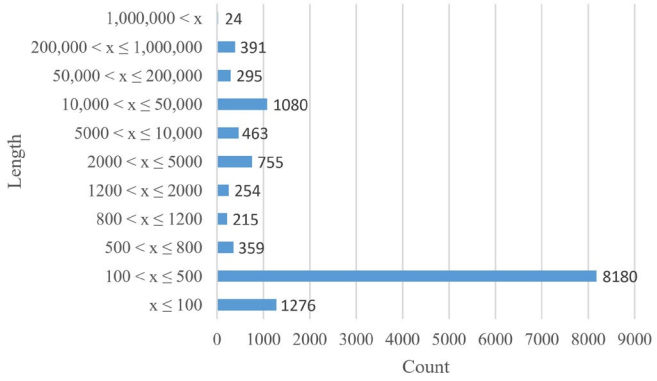

Figure 1. Number of data by length.

4.3. Experimental Model

To evaluate the performance of the proposed model, we used long short-term memory (LSTM) in our experiments. The LSTM is a model that improves the problem of RNN, which has the problem of gradient vanishing when the length of data is long. Therefore, we chose the LSTM among the RNNs that can learn the sequence information of an API call sequence, which is a set of APIs called in the order in the system. Table 5 shows the structure and parameters of the LSTM model we used.

Table 5. Structure and parameters of the LSTM model used.

Layers	Parameters		Values	Outputs
Input Layer			1, 800	1, 800
Bidirectional Layer	LSTM	units	128	256
Batch Normalization Layer				256
Dense Layer 1	units		128	128
	activation		Rectified Linear Unit (ReLU)	
Dense Layer 2	units		1	1
	activation		ReLU	

4.4. Experimental Results

The experimental results are presented in terms of Accuracy, Precision, Recall, and the F1-Score, which are performance metrics that show how well a classification model has learned. Accuracy is the percentage of the total data that the model correctly classified as benign or malicious. For example, for data with a 1:99 ratio of benign or malicious classes, the accuracy would be 99% if all predictions were malicious, which is not an accurate measure of performance on data with an unbalanced ratio of classes. The precision is the percentage of data that the model classifies as malicious that are malicious. The precision metric shows how many times the model classified benign data as malicious. The recall is important in malware detection because it shows the percentage of malicious data classified by the model as malicious. A higher number of the recall indicates a lower probability of false positives since malicious data are more harmful to the system if they are classified as benign. The precision and the recall are a trade-off. There is a metric, the F1-score, to evaluate this trade-off relationship. The F1-score, which is the harmonic mean of the precision and the recall, can be used to evaluate the disadvantage of the accuracy by considering the degree of imbalance in the experimental data [18].

Figure 2 illustrates the performance comparison of the proposed preprocessing technique when the transformed length (N) was set to 800. The performance was evaluated using an LSTM model on different types of preprocessed data: slicing all data with a length of 800 or above (T1), selecting similar-length API call sequences and padding/slicing the data (T2), using the technique proposed in [5] (T3), and applying the preprocessing technique proposed in this study (T4). For T2, API call sequences with lengths similar to the specified length of 800 were selected, and balanced numbers of malicious (1935) and benign (1952) data samples were extracted. Compared with T1, the proposed technique showed an increase in the accuracy (9.31%), recall (11.86%), precision (2.69%), and F1-score (7.42%). Compared with T2, there was an increase in the accuracy (4.45%), recall (2.82%), precision (7.51%), and F1-score (5.26%). Compared with T3, there was an increase in the accuracy (0.55%), recall (1.3%), and F1-score (0.43%) but a decrease in precision (0.41%). Overall, most techniques exhibited improved performance.

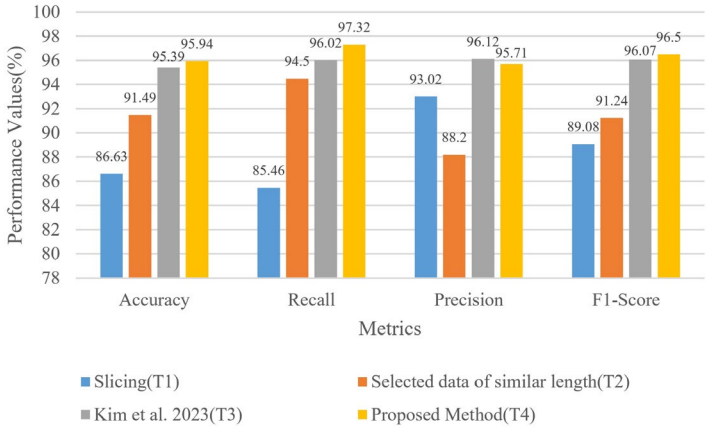

Figure 2. Comparison performance table by method [5].

In terms of accuracy, the proposed method outperformed the other methods, but the number of malicious and normal data differed by about 3000; so, the accuracy is not reliable as a performance indicator. The recall showed that the proposed method detected malicious files with an improvement of up to 11.86%, and the F1-score showed an improvement of up to 7.42%, indicating that the proposed method can contribute to better detection performance than the other methods in an environment with an unbalanced amount of data between classes.

T1 involved padding all the data with a length below and discarding all the data with a length above, resulting in the smallest overhead. T2 involved selecting data with similar lengths N and reducing the standard deviation to reflect the data characteristics in the model; however, not all data could be used. T3 is a preprocessing technique proposed in [5] that uses TF-IDF and APIs. T4 is an extension of T3, which includes the category information of the APIs and List_chunk_F in Algorithm 1, Step 8. T1 required only padding and data removal, resulting in the lowest overhead. T2 had a slightly higher overhead than T1 because it required reading all the data from start to finish to measure the length while extracting data with similar lengths. T3 incurred overhead during the calculation of TF-IDF, which involved determining the number of APIs in each API call sequence and the number of documents in which each API was used. The sliding-window calculations in T3 required an overhead equivalent to that of the window size. T4 had a slightly higher overhead than T3 because it included the calculation of TF-IDF for categories and the computation of (2). However, while the TF values were calculated for each API call sequence, the IDF values, which required reading all documents simultaneously, were computed only for the training data, resulting in varying overhead levels.

5. Conclusions

In this study, to convert API call sequences, which are dynamic analysis data of different lengths, to fixed lengths according to the characteristics of deep learning, we propose a technique that calculates the weights of APIs and categories through TF-IDF and preprocesses the data into a specified length using a sliding window. The proposed preprocessing technique showed higher accuracy and lower false-positive rates than the other techniques. The proposed technique can be applied to dynamic analysis data used to detect obfuscated and packed malware and can be used for detection by minimizing the loss of time-series information in dynamic data of different lengths. However, the data from the unselected APIs were lost when calculated through the sliding window. Therefore, in the future, we would like to develop a technique that calculates the representative value of a window while considering the data within the window.

Author Contributions: M.K. and H.K. completed this work. H.K. evaluated the proposed technique. M.K. supervised to design and develop the proposed technique in this work and guided this whole work as a corresponding author. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) [No. 2018R1A2B6009620].

Data Availability Statement: The PE Malware Machine Learning dataset is available at https://practicalsecurityanalytics.com/pe-malware-machine-learning-dataset/ (accessed on 26 December 2023).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Gopinath, M.; Sethuraman, S.C. A Comprehensive Survey on Deep Learning Based Malware Detection Techniques. *Comput. Sci. Rev.* **2023**, *47*, 100529.
2. O'Kane, P.; Sezer, S.; McLaughlin, K. Obfuscation: The Hidden Malware. *IEEE Secur. Priv.* **2011**, *9*, 41–47. [CrossRef]
3. Jordan, M.I. Serial Order: A Parallel Distributed Processing Approach. In *Neural-Network Models of Cognition: Biobehavioral Foundations*; Elsevier: Amsterdam, The Netherlands, 1997; pp. 471–495.
4. Hu, B.; Lu, Z.; Li, H.; Chen, Q. Convolutional neural network architectures for matching natural language sentences. *arXiv* **2015**, arXiv:1503.03244.
5. Kim, H.; Kim, M. Dynamic Analytic Data Preprocessing Techniques for Malware Detection. In Proceedings of the Annual Conference of Korea Information Processing Society Conference, Busan, Republic of Korea, 2–4 November 2023; KIPS: Seoul, Republic of Korea, 2023; pp. 131–133.
6. Ramos, J. Using tf-idf to determine word relevance in document queries. In Proceedings of the First Instructional Conference on Machine Learning, Piscataway, NJ, USA, 3–8 December 2003; pp. 29–48.
7. Zhang, S.; Wu, J.; Zhang, M.; Yang, W. Dynamic Malware Analysis Based on API Sequence Semantic Fusion. *Appl. Sci.* **2023**, *13*, 6526. [CrossRef]
8. Almeida, F.; Xexéo, G. Word embeddings: A survey. *arXiv* **2019**, arXiv:1901.09069.
9. Hochreiter, S.; Schmidhuber, J. Long short-term memory. *Neural Comput.* **1997**, *9*, 1735–1780. [CrossRef] [PubMed]
10. Maniriho, P.; Mahmood, A.N.; Chowdhury, M.J.M. API-MalDetect: Automated malware detection framework for windows based on API calls and deep learning techniques. *J. Netw. Comput. Appl.* **2023**, *218*, 103704. [CrossRef]
11. O'Shea, K.; Nash, R. An introduction to convolutional neural networks. *arXiv* **2015**, arXiv:1511.08458.
12. Chung, J.; Gulcehre, C.; Cho, K.; Bengio, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv* **2014**, arXiv:1412.3555.
13. Zhang, Y.; Yang, S.; Xu, L.; Li, X.; Zhao, D. A Malware Detection Framework Based on Semantic Information of Behavioral Features. *Appl. Sci.* **2023**, *13*, 12528. [CrossRef]
14. Feng, R.; Lim, J.Q.; Chen, S.; Lin, S.; Liu, Y. SeqMobile: An Efficient Sequence-Based Malware Detection System Using RNN on Mobile Devices. In Proceedings of the 2020 25th International Conference on Engineering of Complex Computer Systems (ICECCS), Singapore, 28–31 October 2020; pp. 63–72.
15. PE Malware Machine Learning Dataset. Available online: https://practicalsecurityanalytics.com/pe-malware-machine-learning-dataset/ (accessed on 26 December 2023).
16. Cuckoo Sandbox—Automated Malware Analysis. Available online: https://cuckoosandbox.org/ (accessed on 26 December 2023).
17. GitHub Repository. Available online: https://github.com/haesookimDev/TFIDFSlidingwindow/blob/main/data/data_name.csv (accessed on 15 February 2024).
18. Hicks, A.S.; Strümke, I.; Thambawita, V.; Hammou, M.; Riegler, A.M.; Halvorsen, P.; Parasa, S. On evaluation metrics for medical applications of artificial intelligence. *Sci. Rep.* **2022**, *12*, 5979. [CrossRef]