applied sciences

MDPI

Article

Research on the Construction of Malware Variant Datasets and Their Detection Method

Faming Lu, Zhaoyang Cai, Zedong Lin *, Yunxia Bao * and Mengfan Tang

College of Computer Science and Engineering, Shandong University of Science and Technology, Qingdao 266590, China; lufaming@sdust.edu.cn (F.L.); touchczy@foxmail.com (Z.C.); tangmf0220@163.com (M.T.)
* Correspondence: jack@sdust.edu.cn (Z.L.); baoyunxia98@163.com (Y.B.)

Abstract: Malware detection is of great significance for maintaining the security of information systems. Malware obfuscation techniques and malware variants are increasingly emerging, but their samples and API (application programming interface) sequences are difficult to obtain. This poses difficulties for the development of malware variant detection models. To address this issue in this paper, we first generated a malware variant dataset using the obfuscation technique based on the disassembly and decompilation of malware. Then, an API call dataset of these malware variants was constructed through sandboxing. Compared to similar work, the malware variants and their obfuscated API call sequences generated in this paper were all runnable. After that, taking a public API call sequence dataset of obfuscation-free malware as input, a BERT (bidirectional encoder representation from transformers) pretrained model for malware detection was constructed. To enhance the ability of this pretrained model to handle obfuscation and variants, in this paper, we used adversarial training to improve the robustness and generalization of the detection model under obfuscation. As the experimental results show, the proposed scheme can improve the classification performance of malware variants under obfuscation. The accuracy of the malware variant classification was close to that of the unobfuscated case.

Keywords: malware detection; bidirectional encoder representation from transformers; adversarial training; intelligent data analysis; AI for systems

Citation: Lu, F.; Cai, Z.; Lin, Z.; Bao, Y.; Tang, M. Research on the Construction of Malware Variant Datasets and Their Detection Method. *Appl. Sci.* **2022**, *12*, 7546. https://doi.org/10.3390/app12157546

Academic Editors: Weipeng Jing, Xin Wang, Bo Peng and Gang Wang

Received: 25 June 2022
Accepted: 25 July 2022
Published: 27 July 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Malware is a malicious program that is harmful to a computer system. It attempts to invade, damage or disable computers, networks, tablets and mobile devices. The purpose is usually to steal, encrypt or delete data, alter or hijack computer core functions, monitor computer activity, or even damage computer hardware. With the rapid development of network technology, the automatic attack of large-scale malware has become the main form of network attack, which not only brings great trouble to ordinary users but also brings great losses to enterprises and government departments. For example, WannaCry [1], which began to erupt in 2017, caused a worldwide disaster. In August 2020, the weekly report on network security information and dynamics released by China's national Internet Emergency Center pointed out that the number of hosts infected with network malware in China had reached 586,000, approximately 527,000 hosts controlled by Trojans or zombies, and approximately 59,000 hosts infected with flying guest worms [2]. The overall amount of malware increased by 43.71% over the same period in 2019 [3], which made enterprises, governments and schools subject to complex and diverse threats.

Generally, malware has symbolic signature features. Therefore, the signature-based malware detection method is the most widely used method in commercial anti-malware programs. However, such a method cannot detect emerging malware and its variants [4]. Attackers also attempt to limit the effectiveness of signature-based analysis by implementing techniques such as obfuscation and packing [5]. To solve this problem, malware

detection technology based on supervised learning technology has been widely studied [6]. These methods can be divided into two categories: static detection and dynamic detection.

The codes of the same malware family usually have a coding similarity. In view of the above similarities, static detection methods analyze the static features extracted from the binary files of malware and then detect malware by comparing the similarities and differences between the static features of malware and normal software. Generally, static methods usually detect malware based on opcode sequences [7], byte sequences [8], or the generated image features [9] of malware. For the detection method based on opcode sequences, the executable PE file can usually obtain the instruction sequence through disassembly and then classify and count the opcode of the instruction according to arithmetic operations, data operations, logical operations, etc., as the classification feature or use the n-gram method to divide the opcode as the classification feature [10]. For the detection method based on a byte sequence, the instruction sequence part of the opcode is usually taken out separately and transformed into digital features. On this basis, n-gram and other methods are used for classification [11]. The malware detection method based on image features regards every 8 bits of the malware byte sequence as a gray value. In this way, a 256-color gray image [12,13] can be generated for the malware, or the gray image can be extracted by processing a variety of byte sequences, and then a three-channel color image [14] can be constructed through the gray image. Then, the classification methods based on image features such as convolutional neural networks are used to detect the malware.

Dynamic malware detection methods mainly record the behavior characteristics of malware by executing malware samples in a virtual environment and determine whether it is malware through the analysis of behavior characteristics. The behavior information includes the behavior log, system call name, context parameters, environment variables, etc. The system API is the interface used to implement the operating system, and malware usually destroys the operating system by calling the system level API. Therefore, the mainstream dynamic detection scheme mainly identifies malware based on the system API call sequence of malware [15]. It usually places the executable PE file in a sandbox such as the Cuckoo Sandbox. Then, a running PE program is monitored, and an API call sequence is obtained. API instructions usually include NtReadFile, CreateDirectory, and NtWriteFile. Based on the API call sequences, malware is classified using methods such as n-gram sequence analysis [16] or behavior diagram analysis [17].

Both static and dynamic methods play an important role in malware detection, but attackers often circumvent malware detection by creating malware variants through packaging or obfuscation. To address the impact of various malware obfuscation techniques on malware detection, in this paper, we first constructed a dataset of unobfuscated and obfuscated malware variants, as well as their API sequences. Then, a malware detection method based on BERT and TextCNN was proposed. Finally, an experimental evaluation on the proposed method was made.

The paper is organized as follows. Section 2 discusses related work in obfuscation technology of static detection and dynamic detection. Section 3 provides a construction method of malware variants and API sequence dataset. Section 4 offers a pretrained model of malware detection based on BERT. Section 5 presents a malware variant detection model based on adversarial training. Section 6 discusses the Experimental evaluation. Finally, Section 7 summarizes this paper.

2. Related Work

There has been a lot of related work on malware static and dynamic detection, which has played an important role in malware detection. Most of the modern anti-malware engines, such as Windows Defender6, Avast7, Deep Instinct D-Client8 and Cylance Smart Antivirus9, are powered by machine learning, making them robust against emerging variants and polymorphic malware [6]. Malware detection also has important applications in Cloud Infrastructure as a Service (IaaS). IaaS is more vulnerable to malware attacks due to exposure to external adversaries [18]. In IaaS, dynamic online malware detection is also

an important application by detecting the fine-grained process system characteristics of the process runtime [19,20].

Meanwhile, to avoid detection, malware manufacturers have developed many malware obfuscation methods, including packing [21], encryption, invalid instruction insertion, instruction replacement and instruction reordering. Since obfuscation methods such as packing and encryption usually do not change malware behavior, they have no impact on the dynamic detection method. This is one of the reasons why dynamic detection has been widely studied. However, instruction insertion, replacement and rearrangement in the malware API call sequences may lead to the failure of dynamic malware detection methods. For example, Rosenberg et al. [22] presented a black-box attack against API call-based machine learning malware classifiers, focusing on generating adversarial sequences combining API calls and static features. It is shown that such an attack is effective against many traditional machine learning classifiers.

To reduce the impact of obfuscation techniques on malware detection, Sihag et al. [23] constructed the document feature table of operation code segments for Android malware, which makes the operation code feature flexible to obfuscation and enhances the flexibility of malware classification by simplifying the semantics of the Dalvik operation code. Yaz et al. [24] used Cuckoo Sandbox to extract the API call sequence of the malware family, constructed a public dataset [25], and classified the malware family through a deep learning model. Oliveira et al. [26] extracted the API call sequence by running benign software and malignant software in the sandbox and processing the API call sequence into a graph structure. On this basis, they proposed a malware detection method based on a depth graph convolution neural network (DGCNN).

The set of obfuscated API sequences is the basis for the abovementioned malware detection methods. However, it is difficult to obtain enough obfuscated API sequences of real malware. To obtain obfuscated samples of API sequences, Banescu et al. [27] carried out the operations of system call insertion, reordering and replacement when running a malware in a sandbox. However, since the altered objects are API sequences, they cannot be restored to the original program. In other words, these sequences are not necessarily the real behavior of malware. They cannot reflect the real behavior of malware variants after obfuscation. Meanwhile, deep learning techniques such as BERT and adversarial training have obvious advantages in dealing with noise problems. However, they have not been used to handle obfuscation problems in the malware detection field.

Based on the above observations, this paper aims to carry out the following work: (1) construction of malware variants and their real API call sequence sets; (2) a malware variant detection method based on BERT and adversarial training; and (3) experimental evaluation of the proposed malware detection method toward malware with and without obfuscation.

3. Construction of the Malware Variant and Its API Call Sequence Dataset

Section 3.1 gives the construction scheme of real and operable malware variants, and Section 3.2 extracts its API sequence to build an API call dataset that can reflect the real behavior of malware variants.

3.1. Construction of Malware Variants

At present, the construction methods of malware variants include packing, encryption, instruction insertion/replacement/disorder, etc. A commonly used packing scheme is to implant a piece of code in the PE program, and the implanted code will be run first at runtime, which is equivalent to hiding the real OEP entry point of the program. Then, the control is returned to the original code so that the original instruction information can be affected, and the malware variant samples can be obtained. Instruction replacement usually refers to the equivalent replacement of instructions, that is, replacing one or more instructions with one or more equivalent instructions. During replacement, it also needs to ensure that it does not affect the normal logic of the program. In this way, it can interfere

with the detection scheme and obtain malware variant samples. The packing scheme usually does not affect the effect of dynamic detection. The malware variant scheme for dynamic detection is more complex, in which the function reference of the IAT table, exception handling during function call and stack balance during API call need to be considered.

To construct a runnable malware variant sample, in this paper, we used x86 disassembly to reverse the target malware, convert the machine code into assembly language, insert the API call sequence into the assembly code, complete the binary rewriting of the PE file, and then obtain the obfuscated sample. This obfuscation scheme maintained the original code execution structure by inserting code rather than obfuscation by inserting after extracting the API sequence so that each malware can be used as a variant of the original malware, which is more in line with the actual application scenario.

Specifically, a binary rewriting obfuscation scheme for the PE program was designed and implemented. The scheme flow is shown in Figure 1. First, a certain number of malware samples were collected through VirusShare [28]. During statistics, it was found that the dataset contained a large number of PDF, HTML and other files. Python-magic was used to filter out the files with PE32 characteristics. After filtering, some C/C++ compiled exe files and some C# compiled exe files could be obtained.

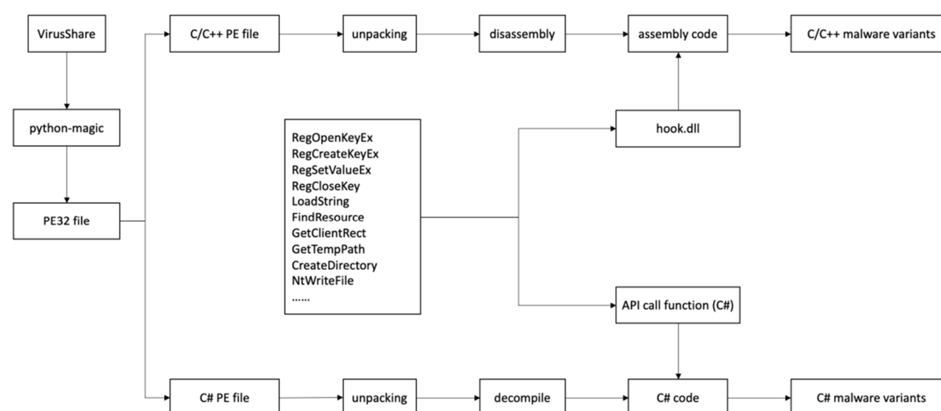

Figure 1. Malware variant construction process.

For C/C++ compiled samples, they first require preprocessing. If the malware has a shell, it needs to be unpacked. If ASLR is enabled in the PE file header, DYNAMIC_BASE needs to be disabled to avoid processing relocations, and NX_COMPAT protection needs to be disabled. This step can be performed using Python's lief library [29] or dnSpy. After that, a new 00 code segment needs to be added by modifying the PE header or using the existing 00 code segment. Usually, a 00 segment of a part of the area will be reserved after each code segment in the PE file. After finding the 00 segment that can be written, one must give RWX permission in this part of the code segment. Through the preprocessing of the PE sample, a segment 00 with executable permission can be obtained, and the assembly code can be written in this part to realize the hook. Since the goal to be achieved is to achieve random calls, it is not guaranteed that these are all the functions that need to be introduced in the IAT table, as well as enough address space for writing, so in this paper we built a general externally imported dll. A function called random API in this dll file was built. RegOpenKeyEx, RegCreateKeyEx, etc., were selected in the API sequence to be inserted. When calling these APIs, it was necessary not to affect the operation of the original program. By dynamically loading this dll into the assembly code, it can prevent the need to rebuild the IAT table to realize dynamic import because there is no relevant API introduced and the number of rewriting 00 segments in the original malware sample is reduced. The functions related to dynamically loading dll are in KERNEL32.dll, which is a dll file that must be referenced to build an application. You only need to use KERNEL32.dll. LoadLibrary and KERNEL32.dll. GetProcAddress two functions to load dll. The specific

hook logic is shown in Figures 2 and 3. Figure 2 shows an assembly instruction sequence, assuming that it is the instruction sequence of the original malware sample and requires an instruction with the address hook 0×01. Figure 3 shows the implementation of the hook. Supposing an API call is implemented at function h_message_box, and Figure 3 shows the API that calls the pop-up window. In the specific implementation in this paper, we moved these function declarations into dll for implementation. Then, a hook function h_0 × 30 was implemented. The purpose of this function is to (1) call the API call function h_message_box completed; (2) execute the instruction of the original hook position; and (3) jump back to the position of the next instruction of the hook instruction. Then, a jump will be made at the position of 0×01, jump to h_0 × 30 and execute the function call. Through the above implementation, we can obtain C/C++ malware variants.

```
0 × 01 call origin_function
0 × 06 push 0
...
0 × 30 dd 00
0 × 31 dd 00
```

Figure 2. Original malware assembly instructions.

```
0 × 01 jmp h_0 × 30; hijacked command
        function h_0 × 02; function declaration h_0 × 02
0 × 06 push 0
...
        function h_0 × 30; function declaration h_0 × 30
0 × 30 call h_message_box
0 × 35 call origin_function
0 × 40 jmp h_0 × 06
...
        function h_message_box; function declaration h_message_box
0 × 42 push 0
0 × 44 push 0
0 × 46 push 0
0 × 48 push 0
0 × 50 call dword ptr [MessageBoxW]
0 × 56 ret
```

Figure 3. Variant malware assembly instructions.

For C# compiled samples, the same unpacking process is carried out first, and C# runs in the.Net framework, so there is a runtime environment, so it will not be directly compiled into assembly code; just use C# decompilation software such as dnspy to disassemble it. Then, select the same API call as above, use C# to complete an API call function, insert the relevant function call in a random position, and recompile it into exe. Through the above implementation, we obtained the C# malware variant.

The logic of the hook function for both approaches is shown in Figure 4. Firstly, a function pointer declaration is defined, and then the relevant API calls are implemented through the function. After that, a caller function is exported in the DLL, in which all the API calls are stored by vector. Both the number of API function calls, and the number of repeated function calls are generated with a uniformly distributed random number generator. The index value of the API function is generated with a uniformly distributed random number generator as well. Finally, the function calls are completed by looping.

```
typedef void (*ptr)(void); // function pointer declaration

void readFileApiCaller();
// ... More ApiCaller function declarations and implementations

void DLL_EXPORT Caller(void){
    vector <ptr> fns;
    fns.push_back(readFileApiCaller);
    // ... Push ApiCaller functions

    int size = fns.size();
    std::default_random_engine random;
    // generate uniformly distributed random numbers
     std::uniform_int_distribution<int> count(3, 6);
     // The number of API function calls is between 3-6
     std::uniform_int_distribution<int> call(1, 3);
     // The number of repeated API calls is between 1-3
     std::uniform_int_distribution<int> pick(0, size - 1);
     // The index value of the function call

    int randomCount = count(random);
    int randomCallCount = call(random);
    for(int i=0; i<randomCount; ++i){
        int randomPick = pick(random); // Pick an API call at random
        for(int k=0; k<randomCallCount; ++k) {
            fns.at(randomPick)(); // API function call
        }
    }
}
```

Figure 4. The logic of the hook function.

Based on the above scheme, the malware variant sample set used in this experiment was constructed. We generated a variant for each malware sample and a total of 1113 malware samples and their variants were obtained. Making malware variants may cause some damage to the program and prevent the malware variants from running properly. In our experiments, there were three hooked malware that caused exceptions during sandbox operation, while most of the malware variants were able to run successfully. In this repository [30], all original malware and malware variants for the experiment are available, as well as the API sequences generated through the sandbox.

3.2. Construction of the Malware Variant API Call Sequence Dataset

Section 3.1 constructed the executable PE file of malware and its variants. This section extracts its API call sequence, constructs an API call sequence set that can truly reflect the behavior of malware after obfuscation, and partially preprocesses the malware tag and sequence set. The extraction of API call sequences depends on automatic testing tools. Cuckoo Sandbox is an open-source automatic malware analysis system that can capture information such as the system API call and its parameter transmission and return value, network traffic data, memory usage records and so on. The overall process of building the API sequence dataset is shown in Figure 5. For a malware, in order to make its variant, the two malware were placed in the sandbox and their call sequences extracted to obtain an original malware API call sequence set and a variant malware API call sequence set. By comparing the classification effects of the two datasets, we could detect whether the variant had a certain interference ability with the classification and how to deal with the interference of the variant malware with the detection.

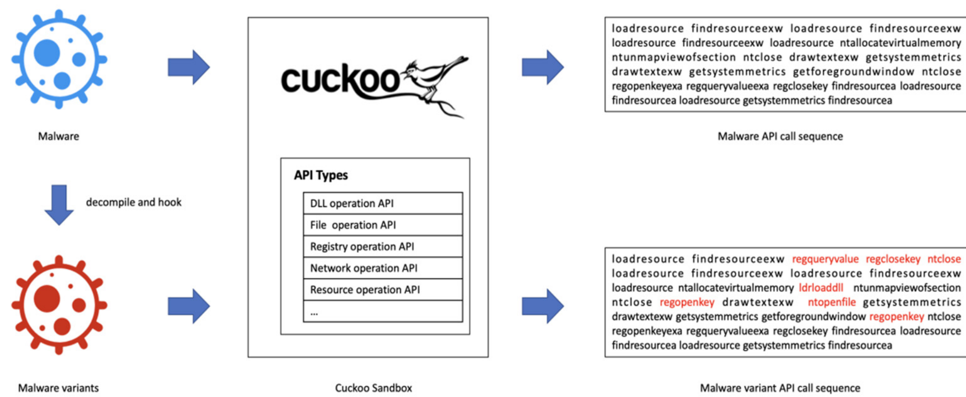

Figure 5. Extraction process of the malware variant API sequence dataset.

For the tag of malware, considering that the existing detection mechanisms are tags for static behavior and that there is no good tag mechanism for the behavior of malware, in this paper, we intended to use more information to construct the tag of malware. Specifically, based on the classification tags of AVClass2 [31], we further combined the information in the malware engine report to generate more accurate tags. AVClass2 was used to analyze the malware sample report of VirusTotal, which is divided into five categories as the multiclass classification labels of malware: normal, downloader, backdoor, keylogger and grayware. For the obfuscated variant malware, the corresponding original malware label is used as the variant malware label. In addition, in the API call sequence extracted by Cuckoo Sandbox, malware will often call the same API repeatedly. Some API calls are expected behavior. For example, WannaCry blackmail malware will repeatedly call the API to read and delete files, and some are unexpected behaviors. For example, exceptions are discarded during the execution of malware, read again and call the relevant API again. To counteract the problem of repeated calls on the dataset and enhance the generalization ability of the model, it was necessary to duplicate the repeated continuous API call sequence.

Finally, since different malware performs different operations, and different operations take different lengths of time, the length of the API sequences we extracted from the different malware in the same time duration was usually different. The longest sequence can contain 30,000 API calls, while the shortest sequence contains only 50. For this problem, the authors of [24] only intercept the front part of the sequence for detection, which may omit the information related to the malicious behavior of malware because the sequence of relevant malicious behavior is not executed. Therefore, in this paper, we adopted a method of subsequence extraction. Every 200 API calls were extracted as a sequence. At the same time, to minimize the lack of semantics, the last 50 API calls of the previous sequence were taken as the first 50 API calls of the next sequence. The principle of dataset entry division is shown in Figure 6. Compared with the method of extracting the first N API calls, the method of constructing subsequences can detect as many sequences as possible to obtain a more complete detection effect. However, this also has some disadvantages because some of the obtained sequences may not have malicious behavior, which may lead to a reduction in detection accuracy.

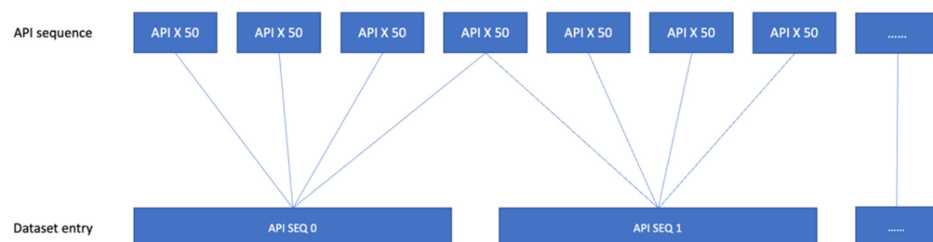

Figure 6. Schematic diagram of API sequence dataset entry division.

After the above processing, the unobfuscated dataset and obfuscated dataset were obtained, and a total of 9123 unobfuscated API sequences and 8264 obfuscated API sequences were obtained [30].

4. Pretrained Model of Malware Detection Based on BERT

4.1. Construction of the BERT Pretrained Model

A pretrained model is an application of transfer learning. It can use as much training data as possible to extract as much common information as possible so that the commonness can be transferred to the model of specific tasks and then fine-tuned with a relatively small amount of annotation data in relevant specific fields so that the downstream tasks can start from commonness. Furthermore, the model only needs to learn the characteristics of the task itself and the knowledge of a specific task. In deep learning models, the dependence on the amount of data is usually high, and a large number of parameters need to be learned. If the amount of data is small, it will inevitably lead to poor model overfitting and generalization. This pretrained model can make full use of a wide range of network resources and does not need to manually mark the data to obtain general knowledge after a large amount of data pretraining. Then, a small amount of annotation data on the target task is used for fine tuning so that the fine-tuned model can deal with the target task well.

BERT is a pretrained language expression model that is trained on a large-scale unlabeled corpus through a two-way transformer encoder to integrate the deep two-way expression of left and right context information to realize the deep two-way representation of sentences to obtain the text vector containing a large amount of semantic information.

When building the BERT pretrained model, it is necessary to establish a word table through word pieces. One of the main implementation methods of word piece is called BPE (byte pair encoding). Generally, the process of BPE can be understood as the re-splitting of a word, which makes the expression of the word more concise and the meaning clearer to reduce the size of the word list without losing semantic features. The word splitting process is shown in Figure 7.

Figure 7. Word splitting process.

The essence of the BERT pretrained model is unsupervised training, so an index is needed to quantify the training effect. BERT will randomly mask out part of the data when initializing the data, then predict the masked language model, and select sentence A and sentence B to predict whether the two sentences are connected, that is, next sentence prediction, to quantify the effect. The specific training process is shown in Figure 8. First, the thesaurus is built through word pieces and API sequences. After building the input vector, it is input into the transformer encoder layer of the BERT model for training, and then the pretrained model is obtained.

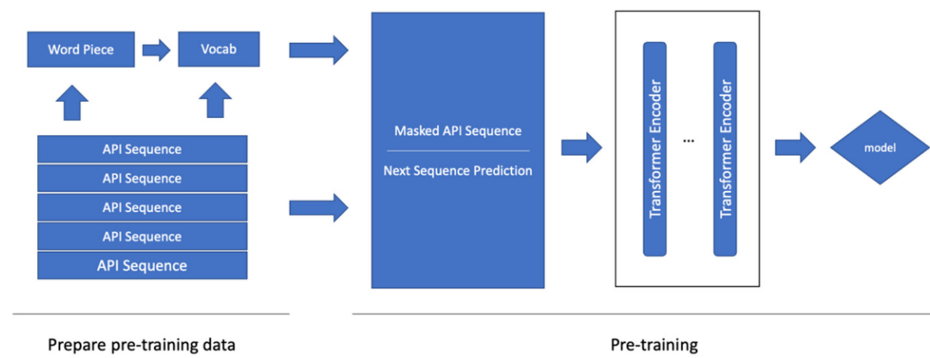

Figure 8. Construction of the pretrained model.

4.2. Construction of the BERT Pretrained Model

With the recent development of natural language processing (NLP) techniques, printable strings became more effective to detect malware [32]. Considering the outstanding performance of the BERT model in the NLP field, in this paper, we used BERT to detect malware.

The classification task based on BERT needs a pretrained model based on a malware API sequence, but there is no public BERT model in the malware field, so the pretraining of the model is needed. In this paper, the open malware API call sequence dataset provided by reference [25] was used for unsupervised training of the pretrained model. The main work was divided into two parts.

The first is to build a thesaurus. When processing pretraining data, many API sequences end with A, W, ExA and ExW. A means using ANSI coding as the text coding of standard input and output streams, W means using Unicode as the coding, Ex means the expansion of parameters and functions, and ExA and ExW are the combination of A, W and Ex. This part of the data can be regarded as the suffix of API call words. The functional semantics represented by relevant APIs are similar, so word pieces can be used to split them. For API words similar to DnsQuery_W, BERT will think _ is the link symbol of a word to represent a phrase, so the relevant underlying information should also be removed. Finally, a thesaurus with a size of 258 is obtained. Then, the representation of token embedding, segment embedding and position embedding is constructed through the thesaurus construction, and the output is the word vector matrix encoded by BERT. The detailed input and output modules are shown in Figure 9.

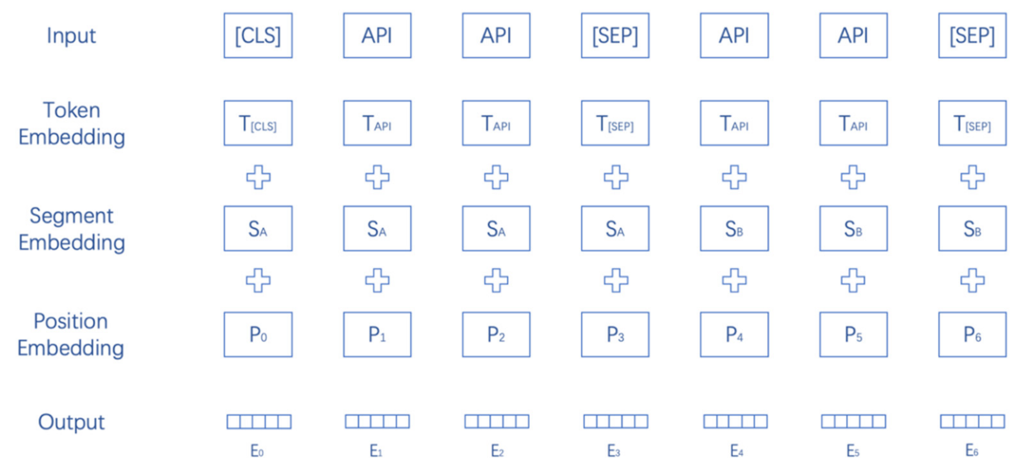

Figure 9. Input and output modules.

Next, returning to the API sequence, remove various special symbols from the sequence, and use spaces as segmentation labels between APIs. Due to the long length of the API sequence, it is necessary to first duplicate the API sequence and then adopt the

method of fixed sequence length. Every 100 APIs are used as a sentence, and each malware sequence is used as the content of an article. Fill the shorter sequence with [PAD] to a fixed length. Finally, [CLS] and [SEP] tags are added at the beginning and end for model training to input the processed sequence into the matching model, and then the input module is built to start input into the pretraining part. Input the constructed input information into the transformer encoder layer for training. Here, six encoder layers were selected for training, and 500,000 iterations were carried out. The final pretrained model had an accuracy of 0.975 in MAS evaluation. BERT is essentially realized through the encoder stack of a transformer, and the core part of the transformer is self-attention. In Figure 10, some API words were randomly selected to represent the weight information of the self-attention of the trained BERT model.

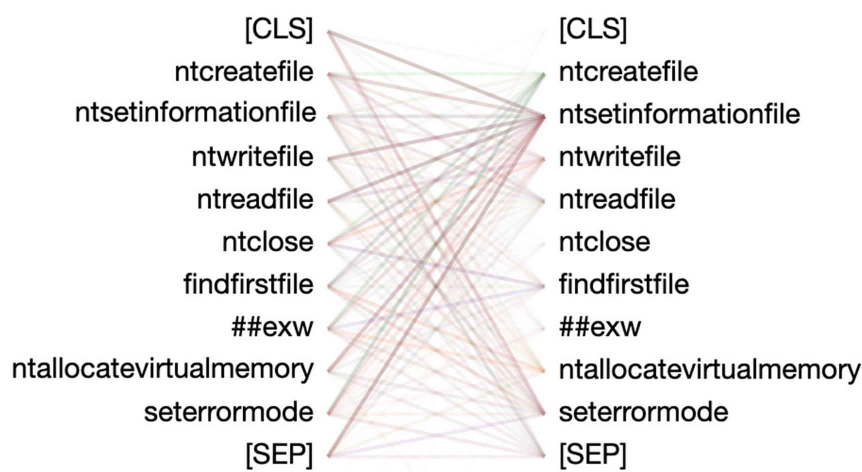

Figure 10. Self-attention weight of API words. The Line weight reflects the attention value, while line color identifies the attention head.

5. Malware Variant Detection Model Based on Adversarial Training

5.1. BERT-TextCNN Classification Model

TextCNN obtains the feature representation of n-grams in sentences through one-dimensional convolution. Its structure is simple. It is an efficient text convolution algorithm that can capture the local structural relationship between adjacent texts. At the same time, the characteristics of convolution make it support parallel operation. In TextCNN, first, the embedded representation of the input statement is obtained through an embedding layer, then the characteristics of the statement are extracted through a revolution layer, and finally the final output is obtained through a fully connected layer.

After obtaining the BERT pretrained model in Section 4.2, this pretrained model could be used for classification. Because the BERT pretrained model is essentially the encode part of the transformer, classification as a downstream task needs to be connected to the decoder part. The common practice is to take the first token position output by the last layer of BERT, that is, the CLS position, as the representation of the sentence, followed by the full connection layer for classification. Because TextCNN can effectively capture the n-gram information in the text sequence, TextCNN is used here as the decoder layer of BERT for classification. The specific classification process of BERT-TextCNN is shown in Figure 11.

In classification, first, the representation of token embedding, segment embedding and position embedding is obtained, and then they are input into the BERT pretrained model. Then, the word vector matrix encoded by BERT can be obtained, where n is the input length and d is the word vector dimension (d = 768). In this paper, three convolution kernels were set, namely, (h = 3, d = 768), (h = 4, d = 768), and (h = 5, d = 768), where h is the height of the convolution kernel, d is the width of the convolution kernel, and the output channel of the convolution kernel is 256. Relu was used as the activation function. After convolution, the maximum pooling operation was carried out through the pooling

layer of TextCNN; that is, for the features obtained after convolution, only the maximum value is taken, the feature information is compressed and retained, and then the pooling results of three convolution cores are connected to obtain a new feature matrix. Finally, the vectors were obtained in the full connection layer for classification.

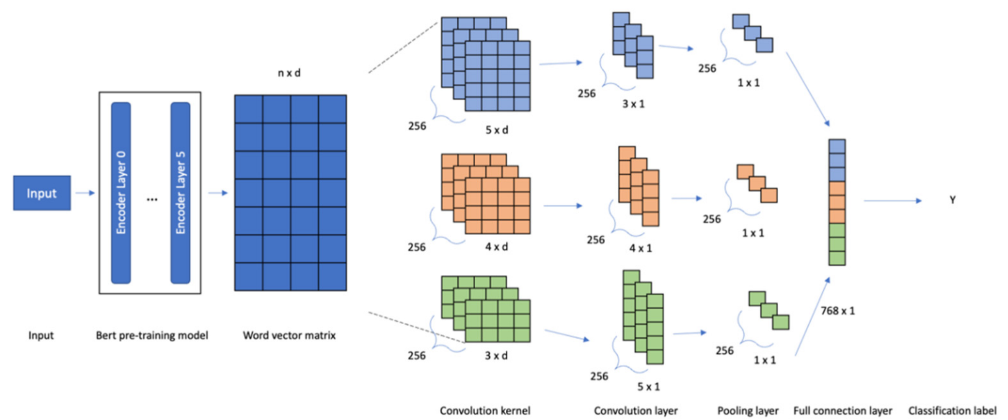

Figure 11. Schematic diagram of the BERT-TextCNN classification process.

5.2. BERT's Adversarial Training

Adversarial training is an important way to enhance the robustness of neural networks. In the process of adversarial training, the samples are mixed with some small disturbances, and then the neural network adapts to this change to have stronger robustness to the original samples. Adversarial attacks are equivalent to finding ways to create more adversarial samples. Malware variants are similar situations. Malware manufacturers fight detection by building more noise or changing some behaviors. Adversarial defense is equivalent to finding ways to enable the model to correctly identify more adversarial samples. Adversarial training is a method of defense. Its basic principle is to construct some adversarial samples by adding disturbances and putting them into the model for training. With the addition of random disturbances, the loss of the sample becomes larger, and training needs to make the loss of the model as small as possible. Thus, the model has stronger robustness and can improve the generalization ability to a certain extent.

The method of constructing adversarial training samples by adding disturbances is easy to realize in the CV field because in the CV field, a continuous RGB value is usually input, that is, the image matrix can be regarded as an ordinary continuous real number vector, while in the NLP related field, the input is usually a discrete word sequence. If the disturbance is directly carried out on the original word text, then the size and direction of the disturbance are of little significance, so the calculated disturbance can be added to the embedding layer [33]. The specific implementation is to find the embedding layer in the defined model in each step of training, obtain the embedding gradient, calculate the disturbance, inject the disturbance into the embedding layer parameter matrix, then let the model train, and delete the disturbance after training. In the adversarial training method, the FGM fast gradient [34] method is used to calculate the disturbance (1):

$$\delta = \epsilon \cdot (g / \parallel g \parallel_2) \tag{1}$$

among:

$$g = \nabla_X (L(f_\theta(X), y)) \tag{2}$$

In other words, let the direction of the disturbance rise along the gradient, which means that the loss increases the most.

6. Experimental Evaluation and Discussion

6.1. Experimental Dataset

According to the construction method of malware variants and the API call sequence dataset in Section 3, taking the malware provided in VirusShare as the original sample, we finally obtained the malware API call dataset in obfuscated and unobfuscated cases. Due to the scattered types of malware in VirusShare, the amount of data in the dataset was unbalanced. To balance the dataset during training, the maximum number of data for each type in the unobfuscated dataset was 1500, and the maximum number of data for each type in the obfuscated dataset was 1000. The number of sequence samples contained in different types of malware is shown in Table 1.

Table 1. Number of API sequence samples of various malware in the dataset.

Type	Unobfuscated Dataset	Obfuscated Dataset
normal	820	350
downloader	1500	1000
backdoor	1060	1000
keylogger	768	835
grayware	1500	1000

Next, binary classification experiments of various kinds of malware were carried out on these two datasets.

6.2. Comparison of Malware Detection Results without Obfuscation

The pretrained model based on BERT was obtained in Section 4.2. Next, a comparative experiment was carried out using random forest, CNN, LSTM and BERT. Table 2 shows the classification effect of the unobfuscated dataset. The behavior of the normal software was relatively fixed. Usually, some system API calls are just generated by user operations. So the normal behavior is relatively fixed and the sequence is relatively short. The classification accuracy was relatively high, as expected. For downloaders, the classification accuracy was relatively low. A possible reason is that it usually performs only some downloading tasks, which are also common operations of ordinary software. Backdoors are relatively easy to recognize, as they usually perform a large number of API calls in the background to steal user data or take control of the user’s machine when the user is inactive. Keyloggers usually record user input only after gaining access to a computer’s external devices. The classification accuracy is usually higher because the behavior is relatively fixed. Graywares are a collection of Spyware, Adware, Dialer, and other malware. One possible reason for the relatively low classification accuracy is that their behavior is diverse making it difficult for the classifier to determine. By comparing the classification effect in random forest, CNN, LSTM and BERT, it can be seen that BERT had relatively good performance in the various classification methods.

Table 2. Classification effect of the unobfuscated dataset.

Method	Normal	Downloader	Backdoor	Keylogger	Grayware	Average
Random forest	0.92	0.71	0.89	0.90	0.82	0.848
CNN	0.95	0.74	0.89	0.91	0.80	0.858
LSTM	0.91	0.74	0.89	0.89	0.81	0.848
BERT	0.94	0.81	0.87	0.90	0.84	0.872

6.3. Comparison of Malware Detection Results with Obfuscation

Table 3 shows the classification effect of the obfuscated dataset. It can be seen that BERT still performed relatively well in various classification methods. By comparing the effects of various classification schemes in Tables 3 and 4 on the unobfuscated dataset and obfuscated dataset, it can be seen that the obfuscated dataset could indeed affect

the classification effect of malware families to a certain extent. Specifically, for decision tree-based random forest methods, the effect of adding obfuscation was relatively low. The classification effect is not strongly related to semantics, so adding confounding factors in semantics has relatively little effect on them. For CNN, the features extracted in the same size window may be biased after adding obfuscation because the window for extracting features was not large. For LSTM and BERT, the classification effect was closely related to the semantics. After adding obfuscation, the semantic dependency became longer, leading to the reduction of classification effectiveness.

Table 3. Classification effect of the obfuscated dataset.

Method	Normal	Downloader	Backdoor	Keylogger	Grayware	Average
Random forest	0.93	0.76	0.86	0.86	0.78	0.838
CNN	0.92	0.76	0.83	0.86	0.79	0.832
LSTM	0.92	0.77	0.83	0.87	0.74	0.826
BERT	0.93	0.78	0.84	0.87	0.80	0.844

Table 4. Effect of BERT-TextCNN joining adversarial training.

Method	Normal	Downloader	Backdoor	Keylogger	Grayware	Average
Adversarial training	0.93	0.78	0.86	0.89	0.82	0.856

To reduce the impact caused by the obfuscation of malware variants, in this paper, we improved the detection effect in the obfuscated dataset of malware variants by adding adversarial training. In Section 5.2, the adversarial training model of BERT-TextCNN was constructed. Table 4 shows the classification effect of this model on various types of malware on the obfuscated dataset. It can be seen that in the classification effect, the malicious family classification of backdoor, keylogger and grayware had improved, and by the average value of classification effect, it can be seen that it had a certain effect on the obfuscated dataset. The classification accuracy of malware is close to that of unobfuscated dataset.

7. Conclusions

To address the impact of various malware obfuscation techniques on malware detection, in this paper, we first constructed a dataset of unobfuscated and obfuscated malware variants, as well as their API sequences. Then, we proposed a malware detection method based on BERT and TextCNN. Finally, an experimental evaluation on the proposed method was made. Specifically, a real and operable malware variant dataset was constructed by binary rewriting first. Then, the original malware and variant malware were placed in the sandbox, and their API call sequences were dynamically extracted to construct the API sequence dataset with and without obfuscation. After that, BERT's pretrained model was trained through a public malware API call sequence dataset. By combining BERT and TextCNN, a new malware detection method was used to perform malware classification. Finally, to improve the malware detection accuracy under obfuscation, adversarial training techniques were adopted in the proposed malware detection schema. The experimental results show that the malware variant with obfuscation constructed in this paper can reduce the classification accuracy of traditional malware detection methods. Meanwhile, the proposed BERT-TextCNN-based malware detection methods combined with adversarial training can eliminate the impact of obfuscation.

There are still many deficiencies in this paper. The input feature information currently considered is only the API sequence part, which is relatively simple. Considering that the parameters, timestamp, API category, call times and other information passed during API call are also important features, we have considered combining these features into our next detection scheme. In addition, FGM computes adversarial interference directly by parameter ϵ, which may not be the best way to generate adversarial samples. Therefore,

a better method for adversarial sample generation needs to be developed in the future. Finally, the method of slicing API call sequences in this paper also needs to be optimized. For example, one may slice the API sequences using the method of concept drift detection in the process mining field [35].

Author Contributions: Conceptualization, F.L.; methodology, F.L.; software, Z.C.; validation, Z.C. and M.T.; formal analysis, Z.L.; investigation, Z.C.; resources, Z.C.; data curation, Z.L.; writing—original draft preparation, Z.C.; writing—review and editing, Y.B.; visualization, Z.C.; supervision, F.L. and Y.B.; project administration, F.L.; funding acquisition, F.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Natural Science Foundation, China (61602279), the Taishan Scholars Program of Shandong Province (No. ts20190936), the Excellent Youth Innovation Team Foundation of Shandong Higher School (2019KJN024), the Postdoctoral Innovation Foundation of Shandong Province (201603056), the Open Foundation of First Institute of Oceanography, China (2018002), and the Distinguished Teachers Training Plan Program of Shandong University of Science and Technology (MS20211102).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The raw virus data collected in this study are available at https://virusshare.com/ (accessed on 19 June 2022). Virus API sequence datasets are available at https://github.com/ocatak/malware_api_class (accessed on 19 June 2022). Datasets for this research can be found in Github: https://github.com/WindrunnerMax/MalwareDataset (accessed on 19 June 2022).

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

References

1. Chen, Q.; Bridges, R.A. Automated Behavioral Analysis of Malware a Case Study of Wannacry Ransomware. In Proceedings of the 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA), Cancun, Mexico, 18–21 December 2017; pp. 454–460.
2. National Internet Emergency Center Network Security Information and Dynamic Weekly Report. Available online: https://www.cert.org.cn/publish/main/upload/File/202008.pdf (accessed on 19 May 2022).
3. Beijing Rising Network Security Technology Co., Ltd. 2020 China Network Security Report. *Res. Inf. Secur.* **2021**, *7*, 102–109.
4. Santos, I.; Brezo, F.; Sanz, B.; Laorden, C.; Bringas, P.G. Using Opcode Sequences in Single-Class Learning to Detect Unknown Malware. *Inf. Secur. IET* **2011**, *5*, 220–227. [CrossRef]
5. Kimmell, J.C.; Abdelsalam, M.; Gupta, M. Analyzing Machine Learning Approaches for Online Malware Detection in Cloud. In Proceedings of the 2021 IEEE International Conference on Smart Computing (SMARTCOMP), Irvine, CA, USA, 23–27 August 2021; pp. 189–196.
6. Aryal, K.; Gupta, M.; Abdelsalam, M. A Survey on Adversarial Attacks for Malware Analysis. *arXiv* **2021**, arXiv:2111.08223.
7. Lu, R. Malware Detection with Lstm Using Opcode Language. *arXiv* **2019**, arXiv:1906.04593.
8. Raff, E.; Barker, J.; Sylvester, J.; Brandon, R.; Catanzaro, B.; Nicholas, C. Malware Detection by Eating a Whole Exe. *arXiv* **2017**, arXiv:1710.09435.
9. Yuan, B.; Wang, J.; Liu, D.; Guo, W.; Wu, P.; Bao, X. Byte-Level Malware Classification Based on Markov Images and Deep Learning. *Comput. Secur.* **2020**, *92*, 101740. [CrossRef]
10. Shekhawat, N.S.; Mathew, R. A Review of Malware Classification Methods Using Machine Learning. In Proceedings of the International Conference on IoT Based Control Networks & Intelligent Systems—ICICNIS 2021, Kottayama, India, 28–29 June 2021.
11. Raff, E.; Zak, R.; Cox, R.; Sylvester, J.; Yacci, P.; Ward, R.; Nicholas, C.; Tracy, A.; McLean, M. An Investigation of Byte N-Gram Features for Malware Classification. *J. Comput. Virol. Hacking Tech.* **2018**, *14*, 1–20. [CrossRef]
12. Bensaoud, A.; Abudawaood, N.; Kalita, J. Classifying Malware Images with Convolutional Neural Network Models. *Int. J. Netw. Secur.* **2020**, *22*, 1022–1031.
13. Fangwei, W.; Guofang, C.; Qingru, L.; Changguang, W. Small Sample Malware Classification Method Based on Parameter Optimization Meta Learning and Difficult Sample Mining. *J. Wuhan Univ.* **2022**, *68*, 17–25. [CrossRef]
14. Chunyu, Y.; Yang, X.; Sicong, Z.; Xiaojian, L. A Malware Classification Method Based on Three Channel Image. *J. Wuhan Univ.* **2022**, *68*, 26–34. [CrossRef]
15. Shankarapani, M.K.; Ramamoorthy, S.; Movva, R.S.; Mukkamala, S. Malware Detection Using Assembly and Api Call Sequences. *J. Comput. Virol.* **2011**, *7*, 107–119. [CrossRef]

16. Eskandari, M.; Khorshidpur, Z.; Hashemi, S. To Incorporate Sequential Dynamic Features in Malware Detection Engines. In Proceedings of the European Intelligence & Security Informatics Conference, Odense, Denmark, 22–24 August 2012.
17. Asrafi, N. Comparing Performances of Graph Mining Algorithms to Detect Malware. In Proceedings of the 2019 ACM Southeast Conference, Kennesaw, GA, USA, 18–20 April 2019.
18. McDole, A.; Abdelsalam, M.; Gupta, M.; Mittal, S. Analyzing Cnn Based Behavioural Malware Detection Techniques on Cloud Iaas. In Proceedings of the International Conference on Cloud Computing, Honolulu, HI, USA, 18–20 September 2020; pp. 64–79.
19. Kimmel, J.C.; McDole, A.; Abdelsalam, M.; Gupta, M.; Sandhu, R. Recurrent Neural Networks Based Online Behavioural Malware Detection Techniques for Cloud Infrastructure. *IEEE Access* **2021**, *9*, 68066–68080. [CrossRef]
20. McDole, A.; Abdelsalam, M.; Gupta, M.; Mittal, S.; Alazab, M. Deep Learning Techniques for Behavioral Malware Analysis in Cloud Iaas. In *Malware Analysis Using Artificial Intelligence and Deep Learning*; Springer: Berlin/Heidelberg, Germany, 2021; pp. 269–285.
21. Muralidharan, T.; Cohen, A.; Gerson, N.; Nissim, N. File Packing from the Malware Perspective: Techniques, Analysis Approaches, and Directions for Enhancements. *ACM Comput. Surv.* 2022, *accepted*. [CrossRef]
22. Rosenberg, I.; Shabtai, A.; Rokach, L.; Elovici, Y. Generic Black-Box End-to-End Attack against State of the Art Api Call Based Malware Classifiers. In *International Symposium on Research in Attacks, Intrusions, and Defenses*; Springer: Cham, Switzerland, 2018; pp. 490–510.
23. Sihag, V.; Vardhan, M.; Singh, P. Blade: Robust Malware Detection against Obfuscation in Android. *Forensic Sci. Int. Digit. Investig.* **2021**, *38*, 301176. [CrossRef]
24. Yazi, A.F.; Çatak, F.Ö.; Gül, E. Classification of Methamorphic Malware with Deep Learning (Lstm). In Proceedings of the 27th IEEE Signal Processing and Applications Conference, Sivas, Turkey, 24–26 April 2019.
25. Catak, F.O.; Yazi, A.F. A Benchmark Api Call Dataset for Windows Pe Malware Classification. *arXiv* **2019**, arXiv:1905.01999.
26. Oliveira, A.; Sassi, R.J. Behavioral Malware Detection Using Deep Graph Convolutional Neural Networks. *TechRxiv* 2019, *preprint*. [CrossRef]
27. Banescu, S.; Wüchner, T.; Guggenmos, M.; Ochoa, M.; Pretschner, A. Feebo: An Empirical Evaluation Framework for Malware Behavior Obfuscation. *arXiv* **2015**, arXiv:1502.03245.
28. Virusshare. Available online: https://virusshare.com/ (accessed on 19 June 2022).
29. Romain, T. Lief—Library to Instrument Executable Formats. April 2017. Available online: https://lief.quarkslab.com/ (accessed on 19 June 2022).
30. Malware Dataset. Available online: https://github.com/WindrunnerMax/MalwareDataset (accessed on 19 June 2022).
31. Sebastián, S.; Caballero, J. Avclass2: Massive Malware Tag Extraction from Av Labels. In Proceedings of the ACSAC '20: Annual Computer Security Applications Conference, Austin, TX, USA, 7–11 December 2020.
32. Mimura, M.; Ryo, I. Applying Nlp Techniques to Malware Detection in a Practical Environment. *Int. J. Inf. Secur.* **2022**, *21*, 279–291. [CrossRef]
33. Miyato, T.; Dai, A.M.; Goodfellow, I. Adversarial Training Methods for Semi-Supervised Text Classification. *arXiv* **2016**, arXiv:1605.07725.
34. Goodfellow, I.J.; Shlens, J.; Szegedy, C. Explaining and Harnessing Adversarial Examples. *arXiv* **2014**, arXiv:1412.6572.
35. Lin, L.; Wen, L.; Lin, L.; Pei, J.; Yang, H. LCDD: Detecting Business Process Drifts Based on Local Completeness. *IEEE Trans. Serv. Comput.* **2020**, *8*, 1. [CrossRef]