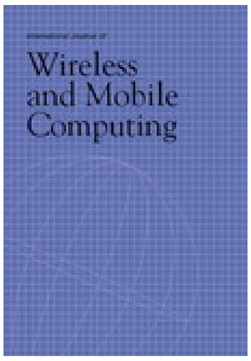


International Journal of Wireless and Mobile Computing

ISSN online: 1741-1092 - ISSN print: 1741-1084
https://www.inderscience.com/ijwmc

Memory-efficient detection of large-scale obfuscated malware

Yueming Wang, Meng Zhang

DOI: 10.1504/IJWMC.2023.10061556

Article History:

Received:	08 November 2022
Last revised:	30 January 2023
Accepted:	22 February 2023
Published online:	07 February 2024

Memory-efficient detection of large-scale obfuscated malware

Yueming Wang and Meng Zhang*

College of Computer Science and Technology,
Jilin University,
Changchun, Jilin, China
Email: wangym19970623@outlook.com
Email: zhangmeng@jlu.edu.cn
*Corresponding author

Abstract: Obfuscation techniques are frequently used in malicious programs to evade detection. However, current effective methods often require much memory space during training. This paper proposes a machine-learning-based solution to the malware detection problem that consumes fewer memory resources. We use hash and sparse matrix to build a text bag of words to reduce memory usage during training. Experiments show that our approach reduces the memory footprint by 95% when using 110,000 text data for confusion recognition training compared to the existing model. In the de-obfuscation step, our method improves the recognition accuracy of the import table function by 40%. Our model achieves shallow memory usage during confusion recognition training and enhances the accuracy of imported table recognition. Additionally, the confusion recognition accuracy is only about 10% lower than the confusion recognition model before the improvement.

Keywords: malware; Naïve Bayes; algorithm.

Reference to this paper should be made as follows: Wang, Y. and Zhang, M. (2024) 'Memory-efficient detection of large-scale obfuscated malware', *Int. J. Wireless and Mobile Computing*, Vol. 26, No. 1, pp.48–60.

Biographical notes: Yueming Wang received his BE degree in Computer Science from Jilin University in 2020. Currently, he is a third-year Graduate student in the College of Computer Science and Technology at Jilin University. His research interests include malware detection and network security.

Meng Zhang received his PhD degree in Computer Science from Jilin University in 2003. Currently, he is a Professor at the College of Computer Science and Technology at Jilin University. His research interests include stringology, network security and computational biology.

1 Introduction

Obfuscation technology is frequently used in malicious programs to evade detection by traditional anti-virus software, and signature-based or string-based detection programs often fail to detect such programs. While string obfuscation is commonly employed in malicious programs, regular developers also use it to prevent reverse-engineering of their programs. With the unregulated nature of the Internet and the ease of access to compilers, software developers can distribute their programs online anywhere. Although Microsoft has built the Microsoft App Store into the Windows operating system, this does not prevent users from installing applications from other sources nor control the impact of malicious programs on the Windows operating system.

In the current scenario, protection against malicious programs heavily relies on the anti-virus program installed on the user's computer. However, many anti-virus programs use signatures or string-based detection, making it challenging to identify obfuscated malicious programs. The analysis of obfuscated malicious programs is more complex and time-consuming than traditional malicious programs, prosing a challenge for anti-virus software vendors.

To address the growing threat of obfuscated malicious programs, obfuscated malware detection has become a highly active research field. Neural networks and data mining schemes are currently the most effective methods to combat obfuscated malicious programs.

The data mining method for detecting obfuscated malware has been employed by Ali and Soomro (2018), who used an efficient mining method based on the PSO selection technique to analyse and detect obfuscated malware. Darem et al. (2021) proposed an approach that leverages OpCode-level features and deep learning to detect obfuscated malicious programs.

Zhu et al. (2021) proposed a novel neural network scheme. This task-aware meta-learning-based Siamese neural network can detect obfuscated malware with high accuracy, even with only one or a few training samples. The method uses entropy features of each malware signature and image features as task input. However, this classification-based approach requires substantial computational resources during the training process, significantly demanding computer performance.

Traditional machine-learning techniques are often preferred over neural networks for malware detection due to their lower resource consumption and satisfactory accuracy (Anderson et al., 2017). For example, Shang et al. (2018) used the Naive Bayes algorithm to detect Android malware and achieved promising results. Similarly, Mohammadinodooshan et al. (2019) used the Naive polynomial Bayes algorithm to model the varying lengths of 235 languages in Wikipedia. They achieved high accuracy, indicating that Naive Bayes can outperform other machine learning algorithms in malware detection under certain circumstances. However, the method proposed by Mohammadinodooshan et al. (2019) required excessive memory during training, making it impractical for some security engineers to training their models.

Additionally, the authors differ on the treatment of Wikipedia data before training. Mohammadinodooshan et al. (2019) removed all symbols, including commas, periods and spaces, which the authors contend are crucial language features. The authors argue that preserving punctuation in string data obtained in natural contexts is more realistic.

The obfuscation of malicious programs is a crucial pre-processing step before security engineers' analysis and remains a complex and contentious issue. Dychka et al. (2018) proposed a program utilising a value-state dependency graph. Meanwhile, Kochberger et al. (2021) presented a comprehensive survey of existing research on the automatic de-obfuscation of virtualisation protectors and introduced a novel method. These works highlight the potential of virtualisation-based automatic de-obfuscation in specific scenarios. However, applying this approach to non-obfuscated programs may inadvertently 'de-obfuscate' and obscure the program.

Existing methods for detecting obfuscated malicious programs can achieve relatively high accuracy, but a common limitation is their high resource consumption. For instance, Mohammadinodooshan et al. (2019) used a Naive Bayesian method to identify obfuscated strings, but we found that their approach had a large memory footprint. We attempted to reproduce their experiment with 60 MB of training text on a computer with 32 GB of memory and encountered an out-of-memory error. Similarly, Zhu et al. (2021) introduced an obfuscated malicious program identification method using a neural network, which necessitates a GPU with robust computing capabilities and substantial memory resources. Consequently, the resource-intensive nature of existing methods poses challenges for processing large training sets.

In practical production environments, security engineers often encounter the requirement of analysing the attack logic of malicious programs and designing detection rules after identifying an obfuscated program as malicious. However, existing classification detection methods have limited utility in fulfilling these requirements.

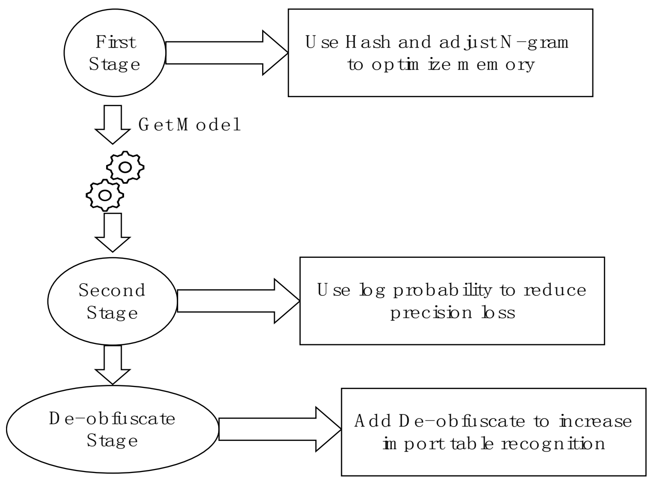
To address these limitations, this study proposes a novel pre-processing method for obfuscated programs. Our objective is to develop a technique that can accurately identify obfuscated programs and attempt to de-obfuscate them in a memory-efficient manner during training. To achieve this goal, we divide the problem into two components: detection and de-obfuscation. The main contributions of our approach are as follows:

1 Our approach to confusion detection builds upon the work of Mohammadinodooshan et al. (2019). To address the high memory usage during training, we improved their method by using a hash word bag technique and reducing the n–gram size, resulting in a 95% reduction in memory usage. This allowed us to train our obfuscated string detection model on large-scale data sets. In the data selection stage, we used the log probability of the predicted output instead of the probability to minimise accuracy loss.

2 We have added a de-obfuscation part based on obfuscation detection. We use the changes before and after the import table as indicators for the de-obfuscation of the detected obfuscated samples. After de-obfuscation, the average accuracy of the content extracted from the import table increased by 40%.

The remainder of this paper is structured as follows: Section 2 provides background knowledge on obfuscation and de-obfuscation techniques. Section 3 presents our design of the experimental protocol. Section 4 introduces the testing procedure and results. In Section 5, we discuss our effects and plan directions for future work. The full text is summarised in section six.

Figure 1 The innovation of our method


First Stage
Use Hash and adjust N-gram to optimize memory
Get Model
Second Stage
Use log probability to reduce precision loss
De-obfuscate Stage
Add De-obfuscate to increase import table recognition

2 Background

This section will introduce some basic knowledge about obfuscation and then discuss general de-obfuscation methods.

2.1 Obfuscation

To reverse the code produced by the compiler is a complex and time-consuming process. If the program undergoes obfuscation resistance analysis, the difficulty of program analysis will be further increased. Here are some situations where obfuscation may be used.

- *Malicious programs*: To evade detection by security software and analysis by security engineers, evil program creators often use this technique to enable them to steal data or achieve other purposes.
- *Intellectual property protection*: Many commercial programs use this technique to prevent unauthorised reverse analysis.
- *Digital rights management (DRM)*: This is the main area in which obfuscation techniques are currently used, and DRM often uses obfuscation techniques to protect encryption keys and protocols.

Obfuscation techniques can be divided into two categories: data-based obfuscation techniques and control-based obfuscation techniques. These techniques are combined in the actual analysis process.

In this paper, we mainly discuss the application of obfuscation in malicious programs.

A study of obfuscated malware detection (Zhu et al., 2021) has shown that malware authors apply different obfuscation techniques on the generic feature of malware to create new variants to avoid detection. Besides, Darem et al. (2021) shown that the usage of obfuscation on malware has become prevalent, most of the common anti-malware products cannot detect these malwares, especially when the new malware is different from the malware before. The worst influence (Ali et al., 2018) of the usage of obfuscation on malware is that obfuscation disturbs computer scientists to judge the function of software examples. As cryptocurrency mining is becoming increasingly popular (Hong et al., 2018), obfuscation is used for covering crypto-jacking attacks or drive-by mining (Konoth et al., 2018). Some obfuscation technology (Rusak et al., 2018) is also used for making malicious PowerShell.

To detect whether a program is obfuscated, you can check whether it is an obfuscated string by extracting a part of the string in the program. The principle of this method is that there is usually a string in a standard program. If the string is extracted and it is found that the string is an obfuscated string (the obfuscated string is a program code or a string that is encrypted in some way and then rendered into a human-unreadable string that has no practical meaning in natural language), then the program is confused. In general, since the program without string obfuscation is easy to extract essential information from the plaintext string in the reverse process, it is not meaningful to use other obfuscation methods in this case; it can be judged whether it is obfuscated by detecting the obfuscated string.

The study by Mohammadinodooshan et al. (2019) proposed a method for string detection on Android. Their approach is based on the Naive Bayes theory and uses the bag-of-words method to extract string features. After extracting the string features of the training set, the corresponding prediction data will be obtained from the input string of the test set. A *c*-value will be received by calculating the predicted data list with the improved formula. Store *c*-values of different lengths in different languages and set a threshold for each *c*-value. When predicting a confused string, it will be compared with the *c*-value, and if it is lower than this threshold, it will be judged as a confused string.

2.2 Overview of de-obfuscation techniques

For programs, obfuscation is a form of transformation. Therefore, the essence of de-obfuscation is the inverse transformation of the obfuscation process using software analysis techniques.

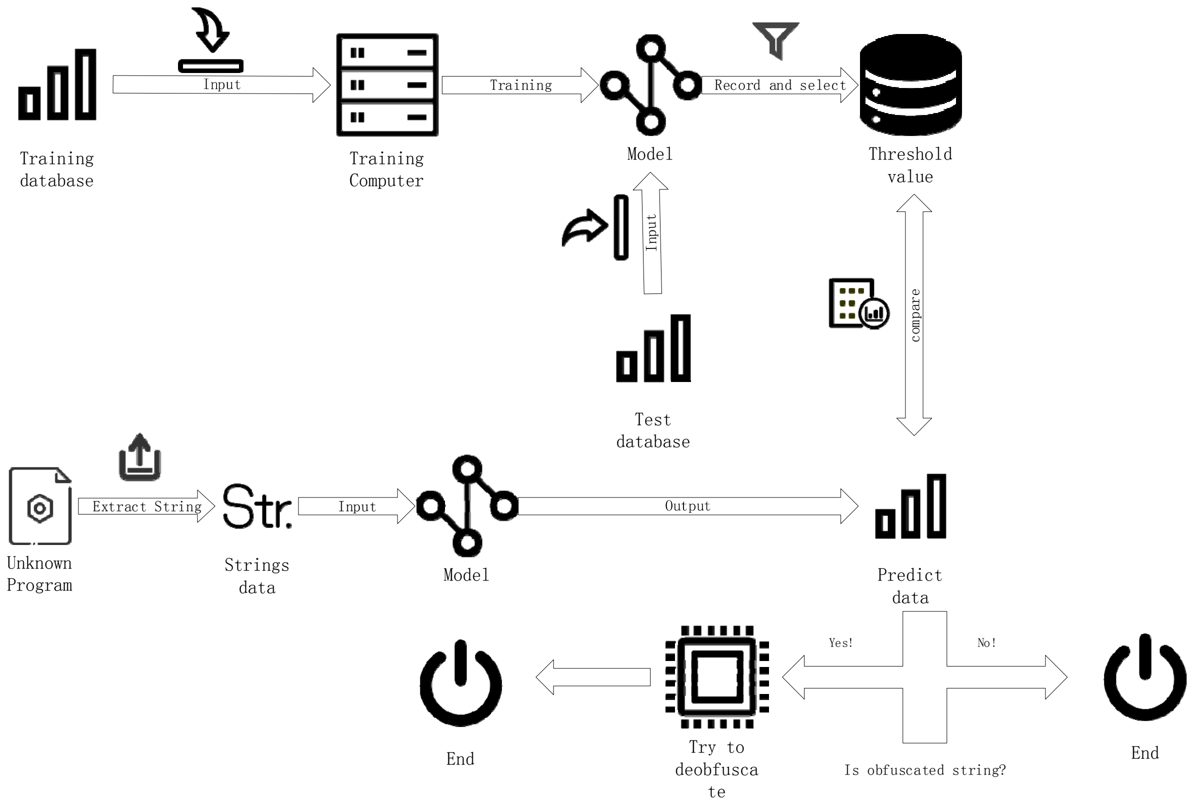
The current terminology in the literature of various research fields about de-obfuscation is not uniform, and the research process is relatively scattered. A reasonably consistent standard or definition in the field of de-obfuscation cannot be found.

The widely used technology in de-obfuscation technology is software analysis technology. According to common classification, software analysis techniques can be divided into static and dynamic analysis (Smaragdakis and Csallner, 2007). Analysing a program is determining whether the program satisfies a particular property. Fully recovering an obfuscated program is nearly impossible statistically; only an approximation of the semantics of a program can be made. Program analysis is also described as an excess approximation and insufficient approximation. For the program's dynamic and static analysis, the difference between static and dynamic analysis is not as big as imagined. According to Ernst (2003), static and dynamic analysis have synergy and duality. Synergy performs in static and dynamic analyses are often performed together, while duality is the opposite.

3 Method design

The objective of our method is to build a low-resource obfuscated program identification and (trial) processing process and improve the accuracy of malicious obfuscated program identification.

Figure 2 The flow chart of our method



The whole process is shown in the figure above. We roughly divide confusion recognition into two processes, training and recognition recording process.

In the training process, we train an existing machine learning model based on the Naive Bayes method. Since the differences between different languages are very large, we decided to build different models for different languages. It is expected to use 235 types in Wikipedia Language to build 235 models.

After building the machine learning model, we will use the model and input the test data set to identify different languages. The Naive Bayesian model will output a list of possibilities. The maximum value in the prediction list must tend to be concentrated in a certain area, so a threshold can be set to define the obfuscation procedure. In addition, considering that there is a huge difference in the distribution of the maximum value of the list of strings of different lengths in the same language, thresholds are set for different lengths.

After completing the obfuscation identification, we consider adding a de-obfuscation solution. According to the survey, obfuscation can significantly reduce the accuracy of malicious program identification. Therefore, we will add a de-obfuscation scheme to prepare for possible subsequent identification of malicious programs.

The specific details of our method are presented below.

3.1 Obfuscation recognition model design

Our innovation for the first part of confusion identification is to adjust the n-gram parameters and use sparse matrices and hashes. Doing so can make the memory occupied by the model training as small as possible, making the model suitable for large-scale training data sets.

Our method is based on Naive Bayes, which is a machine learning algorithm that can be used to construct classifiers. In recent years, Naive Bayes has handled imbalanced data such as fake news and spam detection (Granik and Mesyura, 2017). Naive Bayes is a conditional probability model: when faced with a problem that requires classification, the *n* features or independent variables of the problem can be abstracted into the following vector:

$$x = \left(x_1, ..., x_n\right) \tag{1}$$

For a class variable y, the probability that x supports y can be expressed as:

$$p\left(y|x_1, ..., x_n\right) \tag{2}$$

y can be calculated as:

$$\left(y|x_1, \ldots, x_n\right) = \frac{p\left(y\right) p\left(x_1, \ldots, x_n \mid y\right)}{p\left(x_1, \ldots, x_n\right)} \tag{3}$$

In natural scenarios, we use the following formula:

$$\hat{y} = arg \max_{y} p\left(y\right) \prod_{i=1}^{n} p\left(x_i|y\right) \tag{4}$$

Alternatively, we can use the posterior maximum likelihood (MAP) estimate $p\left(y\right)$. It is the relative probability of class y in the training set and $p\left(x_i \mid y\right)$ is x_i belongs to class y

probability. The main difference between the different Naive Bayes classifiers is the assumptions they make when dealing with $p(x_i \mid y)$.

From the above formula, we can easily conclude that if we have several classes $m = (m_1, \ldots, m_n)$, and a feature f waiting to be classified, we can calculate the probability that f belongs to each model. We name it $p(f|m_i)$ and compare these probabilities to choose the maximum value. It is worth noting that these probabilities are not 'absolute' probabilities but some 'relative probabilities'. In other words, these probabilities cannot represent accurate probabilities but can only mean that a feature is more inclined to belong to a specific model than other models. This is an essential feature of Naive Bayes methods: Naive Bayes methods are suitable for classification but not for estimation (Zhang, 2004).

Polynomial Bayes is one of two classic Naive Bayes variants in text classification, and since it works well for text classification, we also take the form of Multinomial Bayes.

We use the following formula:

$$C(x) = \log P_{\max}(x) - \frac{\sum_{i=1}^{N} \log P(L_i \mid x)}{N} \tag{5}$$

$P_{\max}(x)$ is the maximum value of the list of Naive Bayes predictors. A machine learning model trained in many languages can give a list of predicted values. The number of list entries depends on the number of languages used to train the machine-learning model. The number of list entries is the value of N. In this case, the value of $\sum_{i=1}^{N} \log P(L_i \mid x)$ is easy to understand and can be calculated by adding all the values in the list of predicted values. Therefore, the formula can also be replaced by the following:

$$C(x) = \log P_{\max}(L_i \mid x) - \frac{\sum_{i=1}^{N} \log P(L_i \mid x)}{N} \tag{6}$$

Note that our method uses log probabilities instead of the probabilities themselves, since some of the possibilities in the list of predicted values are small, using log probabilities reduces the error.

For the second part, we collect the maximum value in the list of predictions output by the naive Bayes model at recognition for each language input. We also collect the results for different lengths for each language. After statistics, the fifth percentile (in all statistics, about 95% of the values are greater than this value and 5% less than this value) is used as the threshold for each length of each language. Regarding the length of strings, we found that the accuracy of strings with a length of less than 10 is very low in actual recognition, so it is meaningless, and the increase in the recognition value of strings with a length of more than 80 is minimal. We determined the threshold for the length of 10–80 strings. In the natural environment, the maximum value of the recognition results for strings longer than 80 will be compared with the threshold of 80 for predicting language.

3.2 De-obfuscation model design

To improve the accuracy of subsequent possible automatic malicious program identification processes, our method attempts to de-obfuscate the programs identified by applying the obfuscation technique after identifying the obfuscation. After we proposed this method, we were worried that the de-obfuscation of non-malicious programs might cause copyright-related problems. After actual investigation, we found that companies or individuals capable of applying the obfuscation method to protect software generally use a set of obfuscation developed by themselves. Algorithms or procedures; and malicious programs or exploit programs to obtain maximum benefits from zero-day vulnerabilities in the shortest time, malicious program writers often apply open-source obfuscation solutions to save program writing time. Therefore, we also decided to design our method using an open-source de-obfuscation scheme. Of course, this method may still have errors, but after applying the wrong de-obfuscator, the complete program instructions cannot be obtained, so it does not cause copyright problems.

Our innovation in de-obfuscation model design is as follows:

a) Applying the model to automatic identification of obfuscation followed by de-obfuscation.

b) A multiple de-obfuscation scheme is proposed for programs that apply multiple obfuscations.

There are many kinds of open-source de-obfuscators, and the key lies in how to efficiently and automatically apply these de-obfuscators.

The easier-to-deploy solution is to use all de-obfuscators to traverse the obfuscated program. However, the advantage of this method is that it is easy to implement, and the disadvantage is also obvious: it is very inefficient. Further research found that the characteristics of the obfuscated strings in the programs that applied different obfuscation methods differed. Based on this point of view, our approach uses de4dot's obfuscated string feature comparison to determine which obfuscator or obfuscation method is used by the obfuscation program and uses the de-obfuscator supported by its open-source plugin for de-obfuscation. Our approach divides de-obfuscators into two broad categories: static de-obfuscation and dynamic de-obfuscation. This classification is due to the high similarity between the feature strings extracted from the statically obfuscated program; similarly, the strings extracted from the dynamically obfuscated program also have more similarities. Although it has been recognised twice, this can reduce the accuracy of the recognition.

However, this process still has a problem: some malicious programs may apply two or more kinds of obfuscation. We adopted a solution to detect and attempt to de-obfuscate only one type of obfuscation at a time, using the de-obfuscation scheme multiple times for the obfuscated program. This is because, in a program that adopts multiple obfuscation schemes, if the de-obfuscation is performed simultaneously, it may cause the problem of changing the original semantics of the program or making the program unable to run.

In addition, considering the personal needs of security engineers and to make our method more flexible, in addition to the default automatic de-obfuscation mode, a manual mode option should also be provided so that security engineers know the obfuscation method applied by the obfuscation program. A specialised de-obfuscator can be used.

After de-obfuscation, our method is designed to output a de-obfuscated program that can be distinguished from the original program and analysed by security engineers.

3.3 Analysis and innovation

Our method mainly proposes a pre-processing scheme for obfuscating malicious programs: by extracting and identifying obfuscated strings, we can determine whether there is obfuscation in the program and try to de-obfuscate the obfuscated program. Based on this, if subsequent malicious programs need to be identified, the de-obfuscated program samples can make the following evil program detection parts more accurate.

Our innovation lies in:

a) *N*-gram is a language model based on the assumption that the appearance of the *N*-th word is only related to the previous *N*-1 words and not related to any other words. The probability of the entire sentence is each product of word occurrence probabilities. Therefore, the larger the value of *N*, the larger the memory space occupied during training. Based on this theory, adjusting the *N*-gram to 1 optimises the memory footprint during training.

Although the accuracy rate before taking 40 strings is significantly lower according to Figure 5, and the training time becomes longer, our method still lowers the resource threshold for training. This method can be used for lower-performing hardware that does not render out-of-memory errors. In addition, we combined a de-obfuscation scheme to try to de-obfuscate the programs identified using obfuscation to provide preparations for the subsequent identification of malicious programs.

Specifically, we use *N*-grams to build a bag-of-words model. For an *n*-gram, suppose it uses a text containing *t* characters, where $n, t \in N$ has $t - n + 1$ strings, where each A string requires *n* units of space. Therefore, the total space required for this *n*-gram is $(t - n + 1) \cdot n$ which simplifies to $-n^2 + (t + 1)n$. To choose the value of *n* in an *n*-gram model, it is necessary to find the suitable trade-off between the stability of the estimate and its appropriateness. This means that triples (i.e., triples of words) are a common choice for large training corpora (millions of words), while bigrams are often used for smaller corpora. Based on this, *n*-grams alone may not require much space and even training with very large corpora may only require hundreds of megabytes of memory. However, the actual situation is more complicated. In the whole training process of using *n*-gram to build a naive Bayesian model, a range value of n is often taken; that is to say, all integer value length strings before the maximum value $n_{\max}$ are extracted; in this case, as the *n*-gram maximum value increases, the memory footprint is

$$\sum_{i=1}^{n} - n^2 + (t + 1)n \tag{6}$$

At the cost of this increased memory footprint, the gain in accuracy is not significant. In contrast, in the recognition process, the accuracy rate that can be improved by taking a longer obfuscated string is far greater than the accuracy rate that can be improved by adjusting the Naive Bayesian machine learning training model. Still, the more resources consumed are the opposite. Therefore, we decided to use a unigram model in the training phase to maximise the optimised memory footprint; the upper limit of *n* in *n*-grams is 1.

b) Use hash and sparse matrix to build text word storage to optimise the storage space of words.

An *n*-gram is the first step in building a bag-of-words model, followed by storing words in the text. If the method of converting the words in the text into a word frequency matrix is used, the problem described above still exists. When the vocabulary is very large, the memory footprint of the dictionary is quite large. Therefore, you can use a hash and store the compiled matrix with a sparse matrix, which can solve this problem very well. Compared with the original word frequency matrix, the disadvantage of the hash method is that there is no way to achieve inverse transformation due to the use of an index and not putting the dictionaries into memory, so the established model cannot examine the influence of each feature on the model. However, this is not a problem in our method, which is dedicated to identifying obfuscating procedures and is not concerned with factors affecting classification confounding.

c) Add de-obfuscation to the method to improve the recognition accuracy of the imported table.

The import table is one of the crucial references for identifying malicious programs. After obfuscation, the PE analysis program may not recognise the import table. A typical example is when the program is packed, the execution logic of the program will be changed to 'decompress' the original program in memory first. Therefore, for the PE analysis program, the packaged import table is changed to the import table of a decompression program, and the original program is only the data segment of the current decompression program. To sum up, the obfuscated program cannot identify malicious programs well due to the change of the import table. After de-obfuscation, it is equivalent to manually executing the 'decompression' process. After the software shell is taken out, the program is restored, and the import table is also restored to its original state. Under the circumstance that the de-obfuscation is relatively thorough, the malicious program identification based on the import table can obtain the same effect as before the obfuscation.

4 Experiments and results

This section will show the experimental environment we used, including the software and hardware conditions we used. We will describe our experiments, evaluate our results, compare our results with other people's work, and analyse the reasons for the differences.

4.1 Experimental environment

The experimental environment we use is Windows 11 Pro for WorkStation version 21H2. CPU: Intel Core I7-9750H GPU: NVIDIA GeForce RTX 2060 for notebook. Memory: 32 GB 2667 MHZ

Our method is based on the obfuscated string recognition part implemented by the HashVectoriser class of the python open-source machine learning library SciKit-learn (Pedregosa et al., 2011). We use 500 lines of characters in 235 languages extracted from Wikipedia for the data set, for a total of 115,000 lines of data and the balanced WiLi data set (Thoma, 2018). For the de-obfuscation part, we use a de4dot-based solution.

4.2 Experiment content

We conducted four experiments, in these experiments, we choose M's method and the SVC method as the control group. The SVC method is a traditional machine learning method, it performs well in text classification missions. Besides, the SVC method could output probability as Naïve Bayes, so we can use the same method to assess their result.

The first experiment will be to build multiple language models, and we will record and compare the memory usage of different methods.

The second experiment will use the modified method and test set to select the fifth percentile and give the overall memory footprint of the modified form. We do not provide the memory footprint before the second stage of improvement because the memory footprint of the method before the improvement is too large, and there is no way to complete the first stage.

The third experiment uses the obfuscated string to test the effect of the trained model which is trained in the first stage, and we will provide different methods' accuracy.

The fourth experiment uses the obfuscated and de-obfuscated program import table to compare entries with the original program import table. Since the import table is an essential reference for identifying malicious programs, the use of the change in the import table can show that the de-obfuscation is effective for subsequent malicious programs. It improved program recognition.

For each string that is not obfuscated in the training and test sets, we use the data from the Wi-Li data set (Thoma, 2018) for input and train models of different lengths and languages. In detail, we use the training set data to generate 80-length models in 235 languages. We take the first x characters in the sentence for different string lengths in the same language. The reason why we take the string in this way is as follows: because in the natural environment, the string before obfuscation has its actual meaning, and the programming language, such as C-like statements, these programming statements have extensively borrowed the usage habits of English, so this value is closer to the natural environment.

For obfuscated strings, our point is that since these obfuscated strings are also derived from natural language, we decided to use the original natural language obfuscated strings instead of the combined obfuscated strings.

Since the number of obfuscated strings that can be extracted from existing programs is too small and not universal, we use the following four schemes to generate obfuscated strings as a test set (Mohammadinodooshan et al., 2019):

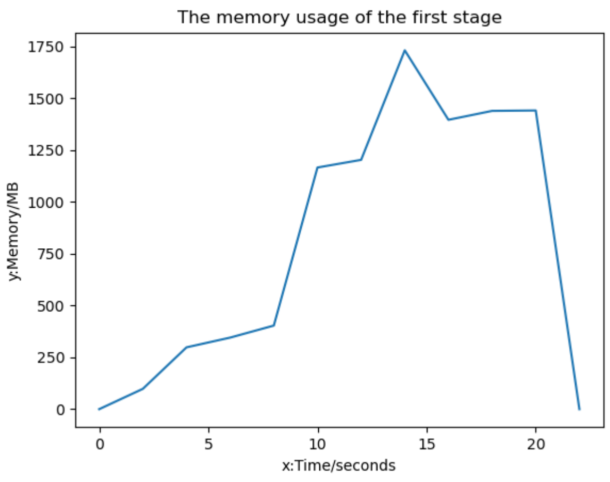
- *Base64 (B64)*: Base64 encoding is often used to obfuscate the encoding of malicious programs. Base64 encoding itself is excellent for use as an obfuscation as well as an encryption scheme. Therefore, we apply the actual Base64 encoding to the not obfuscated string to generate the obfuscated string.
- *Rotation cipher (ROT)*: ROT cipher is a variant of Caesar cipher, and its essence is still a permutation cipher. Our method uses ROT13 for implementation.
- *Fully Uniform Random Sampling (FU)*: This method is not inherently obfuscated; this method is a combination of character sequences sampled from the entire Unicode character using uniform random sampling, Representing a self-developed form of obfuscation.
- *Linguistic Uniform Random Sampling (LU)*: This is a variant of completely uniform random sampling, which solves the problem of an entirely consistent random sample biased towards languages with many letters. When generating this string, we randomly choose a language in the data set, choosing a character with average probability.

For the final stage of obfuscation and de-obfuscation testing, we will use the UPX encryption algorithm as the obfuscation algorithm to test the effect of obfuscation and de-obfuscation on imported tables.

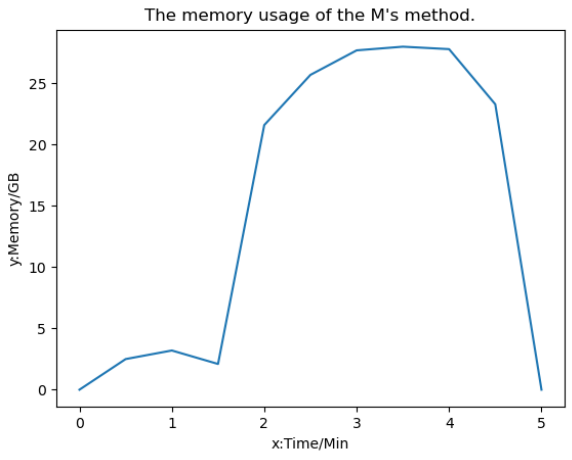
4.3 Results evaluation

Figure 3 shows the memory usage of our method and other methods.

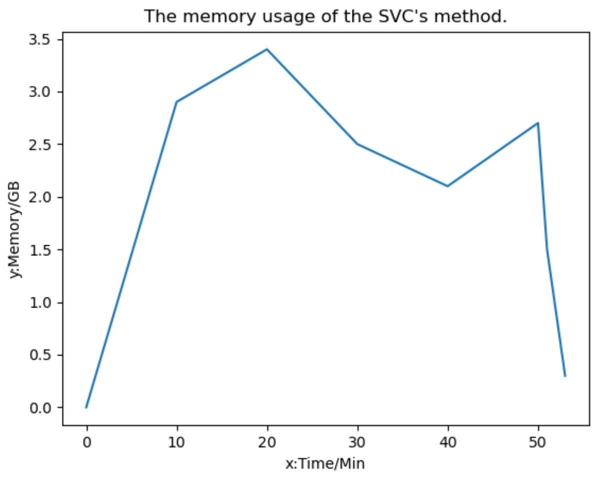
Figure 3 The first stage of memory usage of our method (b) The first stage of memory usage of M's method (c) The first stage of memory usage of SVC



(a)



(b)



(c)

Since our method refers to the work of Mohammadinodooshan et al. (2019), we try to reproduce the first stage training experiment of Mohammadinodooshan et al. (2019). However, it is not entirely successful because our memory is only 32 GB, and the program still needs more memory to complete the training. We used about 60 MB of text data for training during this process. In a natural production environment, 32 GB is already a vast memory and a 60 MB data set is common in the natural environment. Such memory usage is unreasonable. We, therefore, improve upon the method of Mohammadinodooshan et al. (2019) by trying to minimise memory to make the training program adaptable to as many computer environments as possible.

We compared the memory usage of the above three methods in our experiments. Figure 3(b) shows the memory usage during training without our method optimisation in the first training phase: it can be observed that during the training process, the peak memory usage reached about 27 GB, and the memory usage plummeted to 0 after about 4–5 minutes. This is not because the training process had ended; when the memory descended, we observed that the experimental program reported an out-of-memory error and exited. The out-of-memory error message shows that 19.3 GB of memory is still required to continue the experiment. Therefore, this method does not take up to 27GB at most. Still, because our experimental physical machine has only 32 GB of memory, after removing the memory occupation of the system and other processes, the free memory is only 27–28 GB; it may require 57 GB or even more memory to complete the experiment.

Figure 3(c) shows the memory usage of the SVC method during the first stage of training. Although this method does use not so much memory, the training process continues for about 53 minutes.

Figure 3(a) shows our method's memory occupied in the first training stage. It can be observed that the peak memory occupancy during the training process is only about 1.8 GB, and the training task is completed in only about 20 seconds.

The reason for this vast difference is as we conjectured earlier: *N*-gram-based triplet models perform well when training large corpora, and the space complexity increases linearly; however, in this case, the word frequency matrix is constructed. Since the word frequency matrix's space complexity increases squarely with the vocabulary increase, the memory occupied by the final program will be very large under the superposition of the two factors.

After we adopted our experiments to use sparse matrices and unary tuples, the space complexity dropped dramatically, and the training program completed the task in a fraction of the time. Although using a sparse matrix will make it impossible to use some features of the original word frequency matrix, and replacing triples with a tuple will also cause a decrease in the final recognition accuracy, we will not use the discarded features of the word frequency matrix in this article. And the slight reduction in accuracy reduces more than ten times the program memory usage. Our method can now run on personal computers with relatively small memory, which is an increase compared to training methods that only run on large servers. Therefore, we believe that the price is worth it. Moreover, the reduced accuracy can be compensated by taking longer strings during the recognition process, which will be explained in the following paragraphs.

Table 1 shows the training time of all experiments. The SVC second stage training uses 13.2 hours and our method only uses 7.5 hours

Table 1 The peak memory of our method and another method

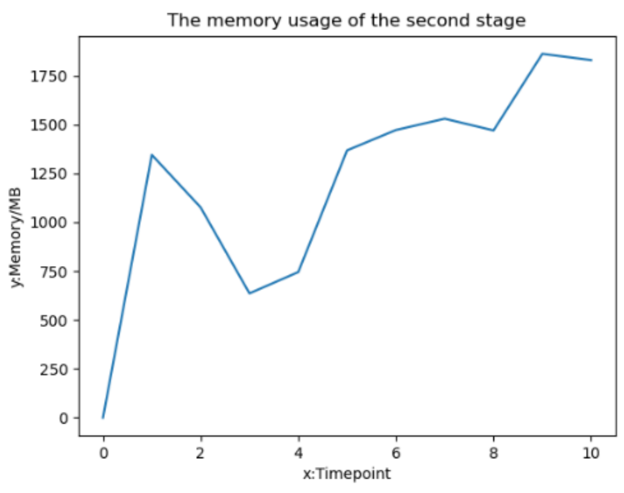
Method	*Max memory*	*Training time*
M's first stage	27 GB	5 Min(With Error)
Our first stage	1.75 GB	30 sec
SVC first stage	3.4 GB	53 Min
M's second stage	–	–
Our second stage	1.8 GB	7.5 hours
SVC second stage	2.4 GB	13.2 hours

Figure 4(a) shows the memory usage in selecting the 5th percentile using the improved method and the test set. Since memory usage fluctuates significantly at this stage, we randomly selected the distribution during the experiment. Of course, these ten-time points are evenly distributed over the entire period of the experiment. It can be observed in the figure that the peak memory usage is around 1.8 GB.

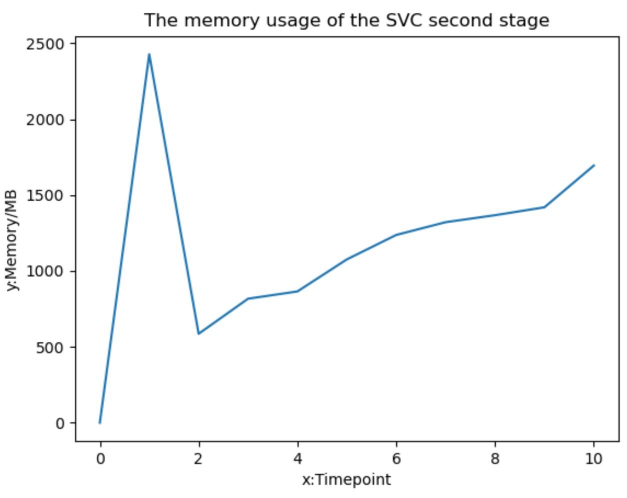
According to Figure 4(b) and Table 1, the SVC second stage training uses more memory the first time and uses nearly memory in the latter half.

Figure 5(a) is the result of obfuscation recognition using the trained model and data, and we use four encoding methods to replace obfuscation. From Figure 5(a), we can easily observe that the longer the string used for recognition, the higher the recognition accuracy. Therefore, the loss of precision caused by using one-tuples instead of triples can be solved by taking longer strings.

Figure 4 (a) The second stage of memory usage of our method (b) The second stage memory usage of the SVC method

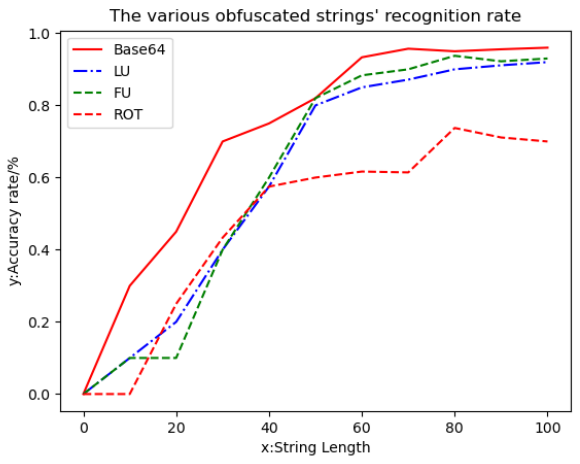


(a)

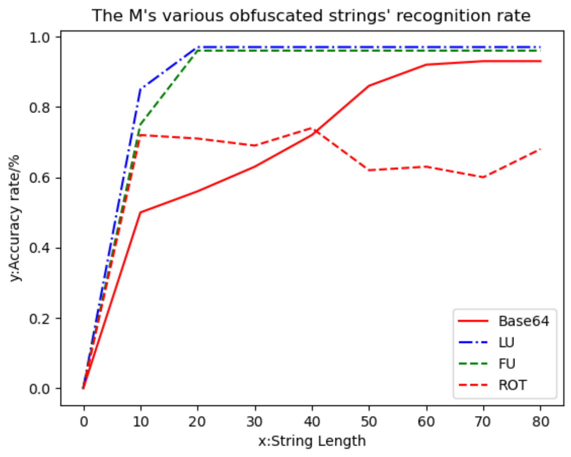


(b)

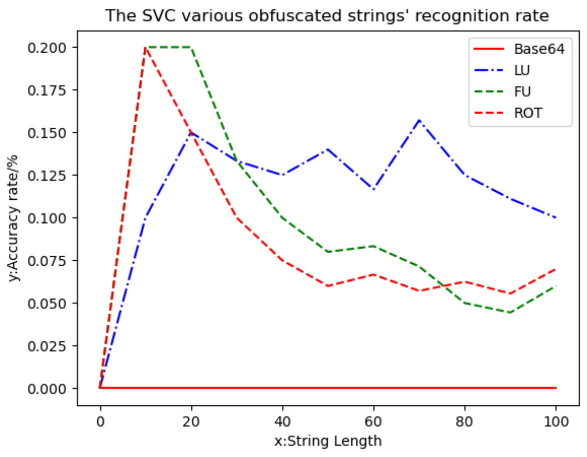
Figure 5 (a) The various obfuscated strings' recognition rate in different string lengths of our method (b) The various obfuscated strings' recognition rate in different string lengths of M's method (c) The various obfuscated strings' recognition rate in different string lengths of SVC's method (see online version for colours)



(a)



(b)



(c)

For the four obfuscation methods, it can be observed from Figure 5(a) that for Base64 encoding, utterly random sampling, and unified language random sampling, when the string length is more than 60, the accuracy rate can reach more than 80%. The effect is the worst. After analysis, the reason is that the rotating password equivalently changes the frequency of occurrence of characters; that is, the number of characters with higher frequency remains unchanged, and only the type of characters is changed. It is equivalent to changing the arrangement rules of characters, and the machine learning model may recognise them as different languages in the same language family. Hence, the recognition effect is relatively poor.

Figure 5(b) shows the various obfuscated strings' recognition rates of M's method. Our method's recognition is slightly lower than their method. But it is necessary to reduce about 95% of training process memory.

Figure 5(c) shows the obfuscated strings' recognition rate of the SVC method. Although the SVC method performs well on normal text classification missions, it performs extremely badly on obfuscated strings classification. And this model classifies all of the Base64 strings to other normal languages. We check the SVC model's output probabilities and we found SVC model output very high predicted probability for Base64 examples to classify them to other language classes.

To study the effect of obfuscation and de-obfuscation on the import table, we used the UPX encryption algorithm to obfuscate and de-obfuscate the test program and observe the changes in the import table before and after the process. To test the program, we used a Crackme program. Before starting, record the 52 entries of the import table. After obfuscation with UPX 3.95, 14,690 entries were extracted, but all were junk data. After de-obfuscation, 1076 entries are extracted, of which the first 1024 entries are garbage entries with no practical significance, and the last 52 entries are consistent with the previously extracted 52 import entries. This shows that de-obfuscation can indeed restore the entries of the import table. Although there may be garbage data and garbage instructions, these garbage data and garbage instructions are easy to deal with. After de-obfuscation, the critical import table entries have been exposed. It can become an essential feature for subsequent malicious program identification.

5 Discussion and future work

This section discusses some of the limitations of our suspected obfuscator pre-processing scheme and the methods used to evaluate it. We also outline directions for future work.

The first problem is that we are committed to optimising the memory usage during model training for obfuscated string recognition, which slightly degrades the model's performance. It is necessary to extract longer strings to compensate for the accuracy when identifying strings. This is because the higher the value of the n-gram during the actual experiment, the higher the accuracy the model can achieve, which also requires more memory. To achieve the highest optimisation level, our method sets the value of n-gram to the default value of 1. Although a high level of memory optimisation is completed, part of this memory optimisation comes at the cost of wasting CPU performance by using multi-layer loop structures in the program, which makes the program less efficient and therefore takes longer to train. Therefore, there is still room for optimisation in terms of time complexity, and we will consider how to optimise time complexity while maintaining a low memory footprint in future work.

The second problem lies in de-obfuscation. We construct a de-obfuscation scheme by integrating open-source de-obfuscators. After designing the system, we found that the difficulty lies in efficiently identifying the used obfuscator according to the features of the extracted obfuscated strings, especially after the integrated de-obfuscator reaches a higher order of magnitude. We can only incorporate a limited number of de-obfuscators in the current experimental stage for experiments. When the obfuscation features are relatively straightforward, such as some codes will add unique symbols or symbol sequences, there will be almost no false positives. Suppose some kinds of obfuscated features are similar, and there is no particular symbol sequence. In that case, there will be a very high false positive rate, and since the search cost will become expensive in the case of a large number of de-obfuscators, it will be a wasted in this case more time; in addition, due to the use of the wrong de-obfuscator processing after false positives, the processed program may still look like an obfuscated program, which will cause an infinite loop of the program in this case. In this case, manual intervention by a security engineer may be required to see if the program is applying multiple layers of obfuscation or is stuck in an infinite loop of errors.

The third problem is that we used real strings encrypted with various methods, not real obfuscated malicious program strings. Therefore, in the future stage, it can be considered to add real-world obfuscated strings for training and recognition.

6 Summary

Obfuscated malicious programs are threats to the internet. Malicious programs using obfuscation technology achieve high concealment and high-attack feasibility. We improved the Bayesian-based strategy for identifying obfuscated programs that require less training memory than the previous model. We added a process to this model to de-obfuscate the recognised obfuscated programs. This can improve the accuracy of program import table recognition. Using the WiLi data set (Thoma, 2018), we evaluate our countermeasures' memory footprint and accuracy when applied to confusion recognition training. Our evaluation results show that the longer the string is, the higher the recognition accuracy of confusion is; the highest can reach more than 80%, which is 10% lower than the previous model, but the memory usage is reduced by about 95%. We used obfuscation and the change of the import table after de-obfuscation to prove that de-obfuscation can improve the recognition accuracy of the import table by about 40%, and the import table is an essential basis for identifying malicious programs. This suggests that our countermeasures can be integrated into obfuscating the identification of malicious programs.

References

Abraham, I., Malkhi, D., Nayak, K., Ren, L. and Yin, M. (2020) 'Sync Hotstuff: simple and practical synchronous state machine replication', *IEEE Symposium on Security and Privacy (SP)*, IEEE, USA.

Ali, Z. and Soomro, T.R. (2018) 'An efficient mining based approach using PSO selection technique for analysis and detection of obfuscated malware', *Journal of Information Assurance and Cyber Security*, pp.1–13.

Anderson, H.S., Kharkar, A., Filar, B. and Roth, P. (2017) 'Evading machine learning malware detection', *Black Hat*, pp.1–6.

Beckman, L., Haraldson, A., Oskarsson, Ö. and Sandewall, E. (1976) 'A partial evaluator, and its use as a programming tool', *Artificial Intelligence*, Vol. 7, No. 4, pp.319–357.

Boyer, R.S., Elspas, B. and Levitt, K.N. (1975) 'SELECT – a formal system for testing and debugging programs by symbolic execution', *ACM SigPlan Notices*, Vol. 10, No. 6, pp.234–245.

Cheng, B., Ming, J., Fu, J., Peng, G., Chen, T., Zhang, X. and Marion, J-Y. (2018) 'Towards paving the way for large-scale windows malware analysis: generic binary unpacking with orders-of-magnitude performance boost', *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp.385–411.

Cousot, P. and Cousot, R. (1977) 'Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints', *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*.

Darem, A., Abawajy, J., Makkar, A., Alhashmi, A. and Alanazi, S. (2021) 'Visualization and deep-learning-based malware variant detection using OpCode-level features', *Future Generation Computer Systems*, Vol. 125, pp.314–323.

Dychka, I., Tereikovskyi, I., Tereikovska, L., Pogorelov, V. and Mussiraliyeva, S. (2018) 'Deobfuscation of computer virus malware code with value state dependence graph', *International Conference on Computer Science, Engineering and Education Applications*, Springer.

Ernst, M.D. (2003) *Static and Dynamic Analysis: Synergy and Duality*, WODA.

Fass, A., Backes, M. and Stock, B. (2019) 'Hidenoseek: Camouflaging malicious javascript in benign ASTs', *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp.1899–1913.

Gligor, V.D. (2019) 'Winning against any adversary on commodity computer systems', *Proceedings of the 1st ACM Workshop on Workshop on Cyber-Security Arms Race*, pp.1–2.

Granik, M. and Mesyura, V. (2017) 'Fake news detection using naive Bayes classifier', *IEEE First Ukraine Conference on Electrical and Computer Engineering (UKRCON)*, IEEE, UKraine.

Guarnieri, M., Köpf, B., Morales, J.F., Reineke, J. and Sánchez, A. (2020) 'Spectector: principled detection of speculative information flows', *IEEE Symposium on Security and Privacy (SP)*, IEEE, USA.

Harilal, A., Toffalini, F., Castellanos, J., Guarnizo, J., Homoliak, I. and Ochoa, M. (2017) 'Twos: a dataset of malicious insider threat behavior based on a gamified competition', *Proceedings of the International Workshop on Managing Insider Security Threats*, pp.45–56.

Hong, G., Yang, Z., Yang, S., Zhang, L., Nan, Y., Zhang, Z., Yang, M., Zhang, Y., Qian, Z. and Duan, H. (2018) 'How you get shot in the back: a systematical study about cryptojacking in the real world', *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp.170–1713.

Ju, A. and Wagner, D. (2020) 'E-ABS: extending the analysis-by-synthesis robust classification model to more complex image domains', *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*, pp.25–36.

Kasturi, R.P., Sun, Y., Duan, R., Alrawi, O., Asdar, E., Zhu, V., Kwon, Y. and Saltaformaggio, B. (2020) 'TARDIS: rolling back the clock on CMS-targeting cyber attacks', *IEEE Symposium on Security and Privacy (SP)*, IEEE, USA.

Kochberger, P., Schrittwieser, S., Schweighofer, S., Kieseberg, P. and Weippl, E. (2021) 'SoK: automatic deobfuscation of virtualization-protected applications', *Proceedings of the 16th International Conference on Availability, Reliability and Security*, pp.1–15.

Kolesnikov, V., Nielsen, J.B., Rosulek, M., Trieu, N. and Trifiletti, R. (2017) 'DUPLO: unifying cut-and-choose for garbled circuits', *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*.

Konoth, R.K., Vineti, E., Moonsamy, V., Lindorfer, M., Kruegel, C., Bos, H. and Vigna, G. (2018) 'MineSweeper: an in-depth look into drive-by cryptocurrency mining and its defense', *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. pp.1714–1730.

Korczynski, D. and Yin, H. (2017) 'Capturing malware propagations with code injections and code-reuse attacks', *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp.1691–1708.

Koruyeh, E.M., Shirazi, S.H.A., Khasawneh, K.N., Song, C. and Abu-Ghazaleh, N. (2020) 'Speccfi: mitigating spectre attacks using cfi informed speculation', *IEEE Symposium on Security and Privacy (SP)*, IEEE, pp.39–53.

Kuhn, C., Beck, M. and Strufe, T. (2020) 'Breaking and (partially) fixing provably secure onion routing', *IEEE Symposium on Security and Privacy (SP)*, IEEE, pp.168–185.

Kurth, M., Gras, B., Andriesse, D., Giuffrida, C., Bos, H. and Razavi, K. (2020) 'NetCAT: practical cache attacks from the network', *IEEE Symposium on Security and Privacy (SP)*, IEEE, USA.

Labaca-Castro, R., Biggio, B. and Dreo Rodosek, G. (2019) 'Poster: attacking malware classifiers by crafting gradient-attacks that preserve functionality', *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp.2565–2567.

Landau, S. (2020) 'Categorizing uses of communications metadata: systematizing knowledge and presenting a path for privacy', *New Security Paradigms Workshop*, ACM, USA.

Lee, J., Nikitin, K. and Setty, S. (2020) 'Replicated state machines without replicated execution', *IEEE Symposium on Security and Privacy (SP)*, IEEE, pp.1–16.

Liu, Z., Chen, C., Zhou, J., Li, X., Xu, F., Chen, T. and Song, L. (2017) 'Poster: neural network-based graph embedding for malicious accounts detection', *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp.2543–2545.

Mohammadinodooshan, A., Kargén, U. and Shahmehri, N. (2019) 'Robust detection of obfuscated strings in android apps', *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, pp.25–35.

Murty, M.N. and Devi, V.S. (2011) *Pattern Recognition: An Algorithmic Approach*, Springer Science & Business Media.

Naderi-Afooshteh, A., Kwon, Y., Nguyen-Tuong, A., Razmjoo-Qalaei, A., Zamiri-Gourabi, M-R. and Davidson, J.W. (2019) 'Malmax: multi-aspect execution for automated dynamic web server malware analysis', *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp.1849–1866.

Naor, M., Pinkas, B. and Ronen, E. (2019) 'How to (not) share a password: privacy preserving protocols for finding heavy hitters with adversarial behavior', *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp.1369–1386.

Niakanlahiji, A., Pritom, M.M., Chu, B-T. and Al-Shaer, E. (2017) 'Predicting zero-day malicious IP addresses', *Proceedings of the Workshop on Automated Decision Making for Active Cyber Defense*, Dallas, USA.

Niaki, A.A., Cho, S., Weinberg, Z., Hoang, N.P., Razaghpanah, A., Christin, N. and Gill, P. (2020) 'ICLab: a global, longitudinal internet censorship measurement platform', *IEEE Symposium on Security and Privacy (SP)*, IEEE, USA.

Novo, C. and Morla, R. (2020) 'Flow-based detection and proxy-based evasion of encrypted malware c2 traffic', *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*, pp.83–91.

Oak, R., Du, M., Yan, D., Takawale, H. and Amit, I. (2019) 'Malware detection on highly imbalanced data through sequence modeling', *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, pp.37–48.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R. and Dubourg, V. (2011) 'Scikit-learn: Machine learning in Python', *Journal of Machine Learning Research*, Vol. 12, pp.2825–2830.

Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J. and Cavallaro, L. (2018) 'Enabling fair ML evaluations for security', *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp.2264–2266.

Pierazzi, F., Pendlebury, F., Cortellazzi, J. and Cavallaro, L. (2020) 'Intriguing properties of adversarial ml attacks in the problem space', *IEEE Symposium on Security and Privacy (SP)*, IEEE, pp.1332–1349,

Raff, E. and Nicholas, C. (2017) 'Malware classification and class imbalance via stochastic hashed LZJD', *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pp.111–120.

Raff, E., Sylvester, J. and Nicholas, C. (2017) 'Learning the pe header, malware detection with minimal domain knowledge', *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pp.121–132.

Raff, E., Zak, R., Lopez Munoz, G., Fleming, W., Anderson, H.S., Filar, B., Nicholas, C. and Holt, J. (2020) 'Automatic YARA rule generation using biclustering', *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*, pp.71–82.

Rivera, E., Tengana, L., Solano, J., Castelblanco, A., López, C. and Ochoa, M. (2020) 'Risk-based authentication based on network latency profiling', *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*, pp.105–115.

Roundy, K.A., Mendelberg, P.B., Dell, N., McCoy, D., Nissani, D., Ristenpart, T. and Tamersoy, A. (2020) 'The many kinds of creepware used for interpersonal attacks', *IEEE Symposium on Security and Privacy (SP)*, IEEE, pp.1–18.

Rusak, G., Al-Dujaili, A. and O'Reilly, U-M. (2018) 'AST-based deep learning for detecting malicious powershell', *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*.

Schindler, P., Judmayer, A., Stifter, N. and Weippl, E. (2020) 'Hydrand: Efficient continuous distributed randomness', *IEEE Symposium on Security and Privacy (SP)*, IEEE, USA.

Schlögl, A. and Böhme, R. (2020) 'eNNclave: offline inference with model confidentiality', *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*, pp.93–104.

Schwartz, E.J., Cohen, C.F., Duggan, M., Gennari, J., Havrilla, J.S. and Hines, C. (2018) 'Using logic programming to recover c++ classes and methods from compiled executables', *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp.1–16.

Shang, F., Li, Y., Deng, X. and He, D. (2018) 'Android malware detection method based on naive Bayes and permission correlation algorithm', *Cluster Computing*, Vol. 21, No. 1, pp.955–966.

Shaw, T., Arrowood, J., Kvasnicka, M., Taylor, S., Cook, K. and Hale, J. (2017) 'Poster: evaluating reflective deception as a malware mitigation strategy', *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pp.2575–2577.

Shumailov, I., Zhao, Y., Mullins, R. and Anderson, R. (2020) 'Towards certifiable adversarial sample detection', *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*, pp.13–24.

Smaragdakis, Y. and Csallner, C. (2007) 'Combining static and dynamic reasoning for bug detection', *International Conference on Tests and Proofs*, Springer, pp.11–6.

Solano, J., Lopez, C., Rivera, E., Castelblanco, A., Tengana, L. and Ochoa, M. (2020) 'SCRAP: synthetically composed replay attacks vs. adversarial machine learning attacks against mouse-based biometric authentication', *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*, pp.37–47.

Thoma, M. (2018) 'The WiLI benchmark dataset for written language identification', *arXiv preprint arXiv:1801.07779.*

Torroledo, I., Camacho, L.D. and Bahnsen, A.C. (2018) 'Hunting malicious TLS certificates with deep neural networks', *Proceedings of the 11th ACM workshop on Artificial Intelligence and Security*, pp.64–73.

Tran, C., Champion, K., Forte, A., Hill, B.M. and Greenstadt, R. (2020) 'Are anonymity-seekers just like everybody else? An analysis of contributions to Wikipedia from Tor', *IEEE Symposium on Security and Privacy (SP)*, IEEE, USA.

Valizadeh, S. and Van Dijk, M. (2019) 'Malpro: a learning-based malware propagation and containment modeling', *Proceedings of the ACM SIGSAC Conference on Cloud Computing Security Workshop*, pp.45–56.

Van Bulck, J., Moghimi, D., Schwarz, M., Lippi, M., Minkin, M., Genkin, D., Yarom, Y., Sunar, B., Gruss, D. and Piessens, F. (2020) 'LVI: hijacking transient execution through microarchitectural load value injection', *IEEE Symposium on Security and Privacy (SP)*, IEEE, USA.

Varshney, G., Misra, M. and Atrey, P.K. (2017) 'Detecting spying and fraud browser extensions: short paper', *Proceedings of the Multimedia Privacy and Security*, pp.45–52.

Verwer, S., Nadeem, A., Hammerschmidt, C., Bliek, L., Al-Dujaili, A. and O'Reilly, U-M. (2020) 'The robust malware detection challenge and greedy random accelerated multi-bit search', *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*, pp.61–70.

Wang, L., Xu, D., Ming, J., Fu, Y. and Wu, D. (2019) 'MetaHunt: towards taming malware mutation via studying the evolution of metamorphic virus', *Proceedings of the 3rd ACM Workshop on Software Protection*, pp.15–26.

Wang, T. (2020) 'High precision open-world website fingerprinting', *Symposium on Security and Privacy (SP)*, IEEE.

Yakura, H., Shinozaki, S., Nishimura, R., Oyama, Y. and Sakuma, J. (2018) 'Malware analysis of imaged binary samples by convolutional neural network with attention mechanism', *Proceedings of the 8th ACM Conference on Data and Application Security and Privacy*, pp,127–134.

Yang, C., Wen, Y., Guo, J., Song, H., Li, L., Che, H. and Meng, D. (2018) 'A convolutional neural network based classifier for uncompressed malware samples', *Proceedings of the 1st Workshop on Security-Oriented Designs of Computer Architectures and Processors*, pp.15–17.

Yoshida, K. and Fujino, T. (2020) 'Disabling backdoor and identifying poison data by using knowledge distillation in backdoor attacks on deep neural networks', *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*, pp.117–127.

You, W., Zhang, Z., Kwon, Y., Aafer, Y., Peng, F., Shi, Y., Harmon, C. and Zhang, X. (2020) 'Pmp: cost-effective forced execution with probabilistic memory pre-planning', *IEEE Symposium on Security and Privacy (SP)*, IEEE, USA.

Yu, H., Nikolić, I., Hou, R. and Saxena, P. (2020) 'Ohie: blockchain scaling made simple', *IEEE Symposium on Security and Privacy (SP)*, IEEE, pp.90–105.

Zhang, H. (2004) 'The optimality of Naive Bayes', *American Association for Artificial Intelligence*, Vol. 1, No. 2, pp.1–6.

Zhang, Y. and Rasmussen, K. (2020) 'Detection of electromagnetic interference attacks on sensor systems', *IEEE Symposium on Security and Privacy (SP)*, IEEE, pp.1–14.

Zhu, J., Jang-Jaccard, J., Singh, A., Watters, P.A. and Camtepe, S. (2021) 'Task-aware meta learning-based Siamese neural network for classifying obfuscated malware', *arXiv preprint arXiv:2110.13409.*