Detecting Cryptographic Functions for String Obfuscation

Pierre Mondon
School of Computing
University of Kent, UK
Email: ppm5@kent.ac.uk

Rogério de Lemos
School of Computing
University of Kent, UK
Email: r.delemos@kent.ac.uk

***Abstract*—Analysing complex evasion and obfuscation techniques is crucial for creating more robust defences against malware. String obfuscation is an easy-to-implement technique that hides information, such as domain names, registry keys, etc. Its detection and removal allow malware to be more accurately analysed. This paper proposes a new method for generating detectors for string obfuscation in binary executables. This is achieved by combining features extracted from the assembly of a binary, and its respective control flow graph and the directed graph derived from the control flow graph. Our method generates highly efficient detectors tailored for string obfuscation achieving more than 90% across all evaluation metrics.**

***Index Terms*—malware, string obfuscation, decryption, static analysis**

I. Introduction

The analysis of malware fulfils multiple purposes, such as malware detection, threat hunting, and forensic analysis. Malware authors tend to rely heavily on *obfuscation*[1] since it slows the analysis process and evades some detection mechanisms.

A particular type of obfuscation is *string encryption*. Its objective is to hide the strings used during the program execution, which may include domain names, file names, etc. This process makes it impossible to see the actual string used in the binary code since all strings are encoded or encrypted. For manipulating strings, the program should include a function for decrypting the string before its use. This obfuscation technique embeds in the program, the encrypted strings together with a function that decrypts the strings at run-time, which often relies on a cryptographic or encoding algorithm. Although encoding algorithms, like Base64, are not cryptographic, in this paper, we refer to them as cryptographic algorithms for the sake of readability.

Hence the term *decryption function* refers to the function that implements the algorithm responsible for decrypting the obfuscated strings. Since string obfuscation, according to MITRE ATT&CK [2], is often performed using XOR, Base64 and RC4, this paper takes an innovative approach to the generation of effective and efficient string obfuscation detectors. In particular, our focus is on MS Windows executables generated from C source code, and this contrasts with similar contributions in the area that mainly focus on Android [3, 11, 12]. These approaches either rely on detecting the strings themselves [11, 12] rather than the decryption function, which can be extremely challenging to locate in binary executables, or rely on features that are not present in native binaries (for example, variables type).

The key contribution of this paper is a new method for generating detectors for string obfuscation decryption functions in binary executables. An advantage of our approach is the fact that the dataset creation is guided by the MITRE ATT&CK and follows real APT and malware trends. The generated detectors are machine learning models trained on features extracted by performing static analysis on the assembly of a binary, its respective control flow graph (CFG), and the directed graph derived from the CFG. Moreover, since the proposed approach relies on static analysis, there is no need to execute programs, which is less time consuming [16]. For the generation and evaluation of our detectors, we have created datasets compiled with GCC, Clang and Microsoft Visual C++ (MSVC) compilers capturing different optimisation levels. We show that our approach, under the assumption that no other obfuscation technique is applied, achieves results above 90% across all metrics and outperforms current detection methods.

The rest of the paper is structured as follows. To contextualise our contribution, the next section describes the challenges associated with the detection of decryption functions in string obfuscation. Then, Section III presents some related work. The following section (Section IV) describes how our datasets were generated. Section V details the machine learning method for generating detectors for detecting string decryption functions. Section VI details the experiments performed and the results obtained. Section VII reflects on the perceived limitations of the overall method for generating detectors. Finally, in Section VIII, we present some concluding remarks and potential directions for future research.

II. Cryptographic algorithms in string obfuscation

String obfuscation relies on the use of cryptographic or encoding algorithms for obfuscating strings, which are embedded in the program. In addition, the program should also include a function for decrypting the obfuscated string before its usage. There are three major categories of cryptographic algorithms:

[1]Obfuscation is referred to as any mechanism that aims to harden the process of understanding a program while conserving its logic.

[2]https://attack.mitre.org/

symmetric, asymmetric and hashing. Symmetric algorithms are the most commonly used in string obfuscation according to MITRE ATT&CK. The exception is Base64, which is an encoding algorithm. Our work focuses on two categories (referred to as "techniques" on MITRE ATT&CK). In "Obfuscated Files or Information" (T1027), the most popular algorithms are XOR, Base64, RC4 and AES. "Deobfuscate/Decode Files or Information" (T1140) lists similar algorithms, but with two exceptions. Table I summarises the cryptographic algorithms used for categories T1027 and T1140. The last column of Table I ("String obfuscation") includes the number of times an algorithm is used in T1027 and T1140 when the description specifically mentions "string obfuscation". Both in T1027 and T1140, AES is popular, however, when the focus is made on string obfuscation, AES is rarely used.

Asymmetric algorithms tend to be more complex and less popular in string obfuscation. Hashing algorithms are not usable in string obfuscation since they are destructive.

TABLE I
CRYPTOGRAPHIC ALGORITHMS USED IN STRING OBFUSCATION

Algorithm	T1027*	T1140*	String obfuscation
Base64	73	34	27
XOR	77	47	30
DES/3DES	7	2	2
Custom	16	13	16
RC4	32	9	10
AES	28	26	3
ChaCha	1	0	0
RSA	7	1	1
Caesar	1	0	0
RC5	1	0	0
Salsa20	1	0	0
ECDHP256	1	0	0
RC6	1	0	0
Rijndael	3	2	2
LEA	0	1	0

*T1027: Obfuscated Files or Information
*T1140: Deobfuscate/Decode Files or Information

From this analysis, we conclude that the algorithms mostly used for string obfuscation are XOR, Base64, RC4 and some custom made algorithms. This accentuates the need for detectors specialised in string obfuscation.

III. RELATED WORK

The literature on detecting cryptographic functions specifically designed for string obfuscation is limited. In cryptographic function detection, approaches are inadequate for string obfuscation that often employ simple cryptography algorithms (as shown in section II). On the other hand, most of the work on string obfuscation has been done on Android programs [11, 12]. Hence, the motivation to take a broader view of the literature on both cryptographic function detection and string obfuscation.

A. Cryptographic function detection in binaries

As string obfuscation aims at hiding strings from static analysis, we focus on static approaches for detection. Static approaches for detecting cryptographic functions in binaries have been proposed in the literature [1, 10]. These rely on the ratio of *cryptographic related instructions* since cryptographic functions use arithmetic and bit-shift operations. However, there is no fixed ratio among the different approaches, each proposes a different threshold: 70% [1] and 40% [10]. Although these approaches show good results in complex cryptographic functions, they fail in the case of algorithms that tend to have less arithmetic and bit-shift operations [18].

Jia et al. [5] published a method based on natural language processing for cryptographic function detection. Li et al. [8] also uses graph embedding in deep neural networks. However, dataset labelling relies on the KANAL plugin for PEiD which is not usable for algorithms without specific signatures, such as those used in string obfuscation. This makes it impractical to tailor algorithms that are specific to string obfuscation.

FLARE Obfuscated String Solver [3] is a practical tool that has looked into automatically extracting obfuscated strings from malware in MS Windows binaries using emulation. This tool uses simple rules to rank functions by the likelihood of being cryptographic without a detection mechanism. Moreover, most of the work is done during the dynamic analysis.

B. Android string obfuscation

The majority of the research in string obfuscation is concentrated on Android platforms [3, 11, 12], which is not directly applicable to MS Windows binary executables. However, there are some similarities in the techniques used, such as instruction count and control flow graph. Glanz et al. [3], Mirzaei et al. [11], Mohammadinodooshan et al. [12] have proposed machine learning approaches for detecting string obfuscation that rely on extracting features directly from strings. This is possible in Android because strings are directly accessible. However, in MS Windows binary executables, it is invalid to assume that a string can be statically accessed since its address may be resolved at run-time, and there is no access to variable types, hence for our approach to focus on detecting decryption functions.

IV. DATASET

Since no dataset is publicly available targeting string obfuscation, we created a dataset by gathering source codes in C language from GitHub repositories. The dataset contains two classes of functions: cryptographic and non-cryptographic. The non-cryptographic part contains functions without any cryptographic algorithm implementation. On the other hand, the cryptographic part contains only implementations of cryptographic algorithms represented by the most popular algorithms used in string obfuscation (Section II), namely: XOR, Base64 and RC4.

For the non-cryptographic part, we used the Google Code Jam dataset [4], downloaded from the respective GitHub repository [5]. The Google Code Jam contains several C functions with different programming styles, given that several participants are from various programming backgrounds. Functions in this

[3]https://github.com/mandiant/flare-floss/
[4]https://codingcompetitions.withgoogle.com/codejam/archive
[5]https://github.com/Jur1cek/gcj-dataset

dataset predominantly consist of algorithmic solutions to complex problems, without any cryptographic algorithms, making them ideal non-cryptographic code. The variety in coding styles and complexity levels across different solutions provides a robust base for testing binary analysis methodologies. For the cryptographic part, we created a dataset by gathering C source code implementation of the XOR, Base64 and RC4 algorithms from GitHub.

Similar to Li et al. [8], we leverage cross compilation to enlarge the dataset size. This is convenient since different compilers and compiler optimisations produce different binary code but preserve program semantics. Furthermore, cross compilation increases resistance to compiler optimisations. In our case, this takes the dataset size from 61 cryptographic source code functions to 787 functions in binary code. This is achieved by using three different compilers: GCC with 5 optimisation levels, Microsoft Visual C++ (MSVC) with 2 optimisation levels, and Clang with 5 optimisation levels. For labelling, we appended an identifier to the function names.

Table II shows a breakdown of the dataset used for this research. The numbers differ slightly for each compiler and compiler optimisations within an algorithm category. This is because some source codes do not compile for certain optimisations and compilers inject functions in the binary during compilation. The dataset presents imbalanced classes, which is realistic for this class of problem [15].

TABLE II
GENERATED DATASET FOR EACH COMPILER AND OPTIMISATIONS

		XOR	Base64	RC4	non-crypto
GCC	O0	19	22	20	1067
	O1	19	22	20	1073
	O2	18	21	18	1064
	O3	19	22	20	1071
	Ofast	19	22	20	1062
Clang	O0	19	22	20	4724
	O1	19	22	20	2462
	O2	19	22	20	1743
	O3	19	22	20	1765
	Ofast	19	22	20	1745
MSVC	O0	19	22	19	2529
	O1	19	22	20	2483
	O2	19	22	19	2489
Sub-total		246	285	256	25277
Total			787		25277

V. METHOD FOR DETECTING CRYPTOGRAPHIC FUNCTIONS

In this section, we present our method for detecting cryptographic functions in binary code, which relies on features generated from the assembly code of the function, and its control flow graph (CFG). An overview of the proposed method is presented in Figure 1, which is partitioned into three major steps. *Binary Pre-processing* transforms the binary code into a high-level representation of the functions, that is, assembly code, control flow graph and directed graph. *Feature Extraction* extracts features from the three binary representations. *Cryptographic Function Detection* classifies functions, based on a machine learning model, as either cryptographic or not.

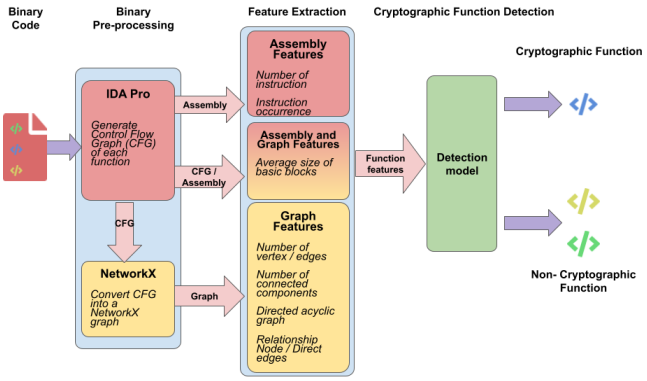

Fig. 1. Method for detecting cryptographic functions

A. Binary pre-processing

Once the binary code input is disassembled, the control flow graph (CFG) for each of its functions can be extracted. A CFG represents the possible execution paths of a function. Its nodes capture basic blocks, grouping contiguous assembly instructions. The directed edges represent the possible jumps in the execution. The entry node is the entry point of the function, and the exit node is the return point of the function. The CFG is then converted into a directed graph from which features associated with general graph theory are extracted.

B. Feature extraction

For each function, features are extracted by analysing the code in terms of its assembly instructions, and the code's respective CFG. Table III describes all the features used for detecting cryptographic functions. The features extracted from the assembly code are named as Asm_*, with the symbol * corresponding to the assembly instruction. Similarly, the features extracted from the directed graph are named as G_*. The last feature type Asm_G_* denotes features extracted from the CFG. This naming convention allows us to identify feature types, and where these are extracted.

The choice of feature from the assembly code was based on field knowledge. Since cryptographic functions perform arithmetic and bit-shift operations on string bytes, the corresponding instructions were selected as features. An exception has been made for the instruction xor: only the non-zero xor instructions are considered since compilers often xor a register with itself to assign the value zero. By filtering the zeroing xor, meaningful xor operations can be captured. In addition, we considered instructions cmp, jmp and push that, respectively, perform a comparison, jump the instruction pointer, and push data to the stack. Another feature extracted from the assembly code is the number of instructions per function (Asm_nbins).

The features extracted from the directed graph were inspired by Liu et al. [9]. We analysed the relevance of these features using a Random Forest classifier and selected the five best features. The feature G_VcEBool represents the ratio between nodes and directed edges. The value taken by this feature can be one of three possibilities: 1 when there are more edges

TABLE III
FEATURE NAMES AND DESCRIPTION

Feature name	Description
Asm_xor	Number of non-zeroing xor in the function
Asm_or	Number of instruction or in the function
Asm_shl	Number of instruction shl in the function
Asm_shr	Number of instruction shr in the function
Asm_sal	Number of instruction sal in the function
Asm_sar	Number of instruction sar in the function
Asm_shld	Number of instruction shld in the function
Asm_rol	Number of instruction rol in the function
Asm_rcl	Number of instruction rcl in the function
Asm_rcr	Number of instruction rcr in the function
Asm_shrd	Number of instruction shrd in the function
Asm_and	Number of instruction and in the function
Asm_add	Number of instruction add in the function
Asm_sub	Number of instruction sub in the function
Asm_mul	Number of instruction mul in the function
Asm_div	Number of instruction div in the function
Asm_imul	Number of instruction imul in the function
Asm_idiv	Number of instruction idiv in the function
Asm_dec	Number of instruction dec in the function
Asm_inc	Number of instruction inc in the function
Asm_ror	Number of instruction ror in the function
Asm_cmp	Number of instruction cmp in the function
Asm_push	Number of instruction push in the function
Asm_jmp	Number of instruction jmp in the function
Asm_nbins	Total number of instructions per function
G_QtyVertex	Number of nodes in the CFG per function
G_QtyEdge	Number of edge in the CFG per function
G_QtyCC	Number of connected components
G_DAG	1 if the first connected component is directed acyclic graph; 0 otherwise
G_VcEBool	Relationship between the numbers of nodes and directed edges
Asm_G_avrblocksize	Average number of instruction per basic block per function

than nodes, 0.5 when there is an equal number of vertices and edges, and 0 when there are more nodes than edges. The subtype G_Qty* represents a quantity of a certain attribute in the graph: G_QtyVertex the number of vertices (nodes), G_QtyEdge the number of edges, and G_QtyCC the number of connected components (CC). G_DAG takes as value 1 if the function does not contain a loop, and 0 otherwise. Only one feature was generated from the CFG (Asm_G_avrblocksize), which captures the average number of instructions per basic block.

C. Function classifier

For classifying a function as either cryptographic or non-cryptographic, we leverage pre-processing techniques with machine learning algorithms for generating a binary classifier. The data model, characterising the function classifier, is generated by extracting the features from the dataset described in section IV. We use pre-processing techniques to re-balance the dataset that is heavily imbalanced between classes. Different techniques have been experimented with, which are further described in section VI. This re-balanced dataset is then used to train a machine learning algorithm, resulting in a model that can detect cryptographic functions.

VI. EXPERIMENTS

To demonstrate the effectiveness of our method for detecting cryptographic functions, in the following, we describe the practical setup of the proposed method and present the results of several experiments using different pre-processing techniques and machine learning algorithms.

A. Experimental setup

Experiments were conducted with an Intel i5-9300H processor and 16 GB RAM. We adopted K-fold cross validation (10 folds) on the dataset described in section IV. A pipeline tool was built to train and test a combination of all the identified pre-processing techniques and machine learning algorithms.

1) Feature extraction: For the implementation, we used IDA Pro [4] for disassembling the binary code and generating the control flow graph (CFG). To transform the CFG into a directed graph, we use NetworkX [6], which facilitates the extraction of graph-based features.

2) Pre-processing for imbalance dataset: For dealing with imbalanced data, different pre-processing techniques were employed, namely, SMOTE [2], SMOTE SVM [13], and Random Undersampling [7]. For their implementation, we leverage the imbalanced-learn [7] library, which contains special implementations for imbalanced datasets. The main parameter is the sampling strategy, which captures the proportion between classes after pre-processing. The choice of SMOTE and SMOTE SVM parameters is: **sampling strategy:** 0.1. For Random Undersampling: **sampling strategy:** auto.

3) Classification algorithm: We have experimented with several machine learning algorithms, namely, Support Vector Machines (SVM), Gaussian Naive Bayes (NB), Multi-layer Perceptron (MLP) and Random Forest (RF). We use for their implementation the Python library Sklearn [14].

B. Evaluation metrics

The imbalanced nature of our datasets led us to choose the metrics used in imbalanced-learn [7] for evaluating our results: *precision*, *recall*, *specificity*, *F1*, and *G-mean* [6]. The *index balance accuracy* (IBA) of the G-mean is calculated with the default Alpha setting of imbalanced-learn by applying equation (1). These metrics were averaged across the 10 folds of the cross validation.

$$G - mean = \sqrt{Recall * Specificity} \quad (1)$$

C. Experimental results

In the following, we present and analyse results obtained by applying different machine learning algorithms. The results of the experiments are then analysed in terms of pre-processing techniques, and these are compared in the context of different machine learning algorithms.

1) Results for Random Forest: For Random Forest, the following parameters were used: **number of trees**: 200; **criterion**: entropy; **class weight** (automatically adjust weights inversely proportional to class frequencies in the input data for each bootstrap sample of each tree): balanced sub-samples; **max features** (the number of features potentially used in each tree split is the total number of features): None.

The results, presented in Table IV, are high across all metrics for all pre-processing techniques except from Random Undersampling for which precision and F1 score lower. This is explained by a high number of false positives which means a non-cryptographic function classified as a cryptographic function. Out of SMOTE, SVM SMOTE and No pre-processing,

[6]https://github.com/networkx/networkx
[7]https://imbalanced-learn.org/

we have stable results across all metrics ranging from 94% to 99+%. SVM SMOTE outperforms No pre-processing by 1% on average across all metrics.

Random Forest is a suitable algorithm as it provides good results across all metrics. Furthermore, SMOTE SVM can be used to improve the results.

TABLE IV
EXPERIMENTAL RESULTS FOR DIFFERENT MACHINE LEARNING ALGORITHMS AND PRE-PROCESSING TECHNIQUES

	pre-processing technique	precision	recall	specificity	F1	G-mean
Multi Layer Perceptron	No Pre-processing	96.60	96.06	99.89	96.29	97.95
	Under Sampling	58.37	96.31	97.84	72.62	97.07
	SMOTE	94.38	95.31	99.80	94.63	97.51
	SMOTE SVM	95.28	94.92	99.85	95.05	97.34
Random Forest	No Pre-processing	94.58	96.19	99.92	96.87	98.04
	Under Sampling	58.90	98.86	97.83	73.74	98.34
	SMOTE	96.85	96.57	99.90	96.70	98.22
	SMOTE SVM	97.37	97.33	99.92	97.34	98.61
Support Vector Machines	No Pre-processing	45.33	95.05	96.42	61.35	95.73
	Under Sampling	44.10	95.05	96.22	60.19	95.63
	SMOTE	48.55	94.54	96.87	64.11	95.69
	SMOTE SVM	48.38	94.28	96.86	63.92	95.56
Gaussian Naive Bayes	No Pre-processing	37.38	56.41	97.04	44.84	73.87
	Under Sampling	29.90	57.03	94.77	35.78	73.34
	SMOTE	35.42	58.06	96.70	43.93	74.82
	SMOTE SVM	37.20	59.20	96.89	45.59	75.60

2) Results for Support Vector Machines (SVM): The following parameters were used: **class weight** (automatically adjust weights inversely proportional to class frequencies in the input data): balanced; **linear kernel**: linear; **gamma scale**: scale.

Table IV demonstrates that this algorithm performs above 94% for each pre-processing technique on three of the five metrics: recall, specificity, and G-mean. However, this algorithm yields high false positives regardless of the pre-processing technique used as precision and F1 score shows.

Regarding pre-processing techniques, using either SVM or SMOTE SVM lowers the false positive rate, which raises precision and F1 score by 3-4%. Still, SVM is not a very good algorithm for our proposed method since it produces a larger number of false positives even when improved by pre-processing techniques.

3) Results for Multi-Layer Perceptron (MLP): The following parameters were used: **activation**: Relu; **solver**: Adam; **learning rate**: constant; **learning rate init**: 0.001; **epochs**: 200.

The results related to this algorithm are shown in Table IV. These are excellent on metrics across all pre-processing techniques except Random Undersampling since it presents a high false positive rate. The other three pre-processing techniques have stable results across all metrics ranging from 94% to 99+%. No pre-processing is slightly better than SMOTE and SVM SMOTE by less than one per cent on average.

MLP effectively and consistently delivers excellent results across all metrics. No pre-processing is necessary as neither SMOTE nor SMOTE SVM enhances the results and would significantly reducing the training time.

4) Results for Gaussian Naive Bayes : The following parameters were used **var smoothing**: $1e^{-9}$; **priors**: None.

The results for the Gaussian Naive Bayes algorithm are presented in Table IV. Although this algorithm performs poorly on most metrics, we included the results for comparison and transparency.

5) Results and discussion: The results display variations depending on the machine learning algorithm for generating the function classifier. Notably, Random Forest and Multi-Layer Perceptron (MLP) demonstrate high efficacy. Regarding pre-processing techniques, the results slightly lean towards SMOTE SVM, but more importantly, rule out Random Undersampling.

When assessing the optimal combination of pre-processing techniques and machine learning algorithms, we computed the average and median values for all metrics associated with each pre-processing technique. This allowed us to identify the most effective combination for our method. When considering the Random Forest algorithm, SVM SMOTE emerges as the most effective technique, boasting a mean score of 98.81% and a median score of 99.84%. In contrast, when employing MLP without pre-processing, the median score is 99.78%, with a mean score of 98.31%. These findings indicate a combined use of Random Forest with SVM SMOTE for our approach.

D. Comparison

As mentioned in section III, most approaches in the literature fall short when considering string obfuscation. Still, we implemented the approach by Matenaar et al. [10] that claims to detect arbitrary cryptographic algorithms in binary code. They propose heuristics based on the ration of *cryptographic related instructions*. However, this method does not provide any positive results for string obfuscation since RC4, XOR and Base64 are simple and contain few operations, the ratio of cryptographic instructions is low. This furthers the motivation for this paper providing a method specialised for string obfuscation.

VII. LIMITATIONS

This section discusses the threats to the validity of our research according to the categories listed by Wohlin et al. [17].

Internal validity. In our study, a potential threat is the overfitting of the detectors towards a certain programming style. This threat has been avoided by collecting data from various repositories and authors. We included multiple implementations of an algorithm, when possible, to reduce that threat.

Construct validity. In our study, a threat could be our dataset misrepresenting string obfuscation. To mitigate this, we have used guidance from MITRE ATT&CK to identify the most used cryptographic algorithms in string obfuscation. Another potential threat is that we rely on external software to disassemble binaries and generate their respective control flow graphs.

External validity. In our study, despite our efforts, we still have a relatively small dataset which may be seen as a limitation compared to real-world applications. Also, we do not consider any other types of obfuscation on top of the string obfuscation.

VIII. Conclusions and future work

This paper has proposed a new static method for detecting cryptographic functions in binary code for string obfuscation. This method uses machine learning algorithms, leveraging features extracted from the assembly code, its associated control flow graph and the directed graph. As part of our study, we have examined the most prevalent cryptographic algorithms used for string obfuscation by analysing MITRE ATT&CK obfuscation techniques. Based on these findings, we have created a dataset containing binary code examples of these cryptographic algorithms. The results of the experiments have shown that the generated detectors can achieve a high level of efficacy: across all metrics, the generated detectors are specifically tailored for string obfuscation algorithms in binary executables.

There are several initiatives for future research. First, we need to identify a method that generates generic detectors capable of handling a broader spectrum of cryptographic algorithms. Additionally, eliminating the assumption that no other obfuscation techniques have been applied to the binary code is crucial, as it will enhance the method's applicability to real-world malware. One possibility for removing this limitation is creating a dataset containing multiple layers of obfuscation.

Availability

All the source codes and datasets can be found here: https://github.com/pmondon/string-obfuscation-detection

References

[1] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 621–634.

[2] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[3] L. Glanz, P. Müller, L. Baumgärtner, M. Reif, S. Amann, P. Anthonysamy, and M. Mezini, "Hidden in plain sight: Obfuscated strings threatening your privacy," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp. 694–707.

[4] hex rays, "Ida pro," https://hex-rays.com/ida-pro/.

[5] L. Jia, A. Zhou, P. Jia, L. Liu, Y. Wang, and L. Liu, "A neural network-based approach for cryptographic function detection in malware," *IEEE Access*, vol. 8, pp. 23 506–23 521, 2020.

[6] M. Kubat, S. Matwin *et al.*, "Addressing the curse of imbalanced training sets: one-sided selection," in *Icml*, vol. 97, no. 1. Citeseer, 1997, p. 179.

[7] G. Lemaître, F. Nogueira, and C. K. Aridas, "Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning," *Journal of Machine Learning Research*, vol. 18, no. 17, pp. 1–5, 2017.

[8] X. Li, Y. Chang, G. Ye, X. Gong, and Z. Tang, "Genda: A graph embedded network based detection approach on encryption algorithm of binary program," *Journal of Information Security and Applications*, vol. 65, p. 103088, 2022.

[9] H. Liu, C. Guo, Y. Cui, G. Shen, and Y. Ping, "2-spiff: a 2-stage packer identification method based on function call graph and file attributes," *Applied Intelligence*, vol. 51, no. 12, pp. 9038–9053, 2021.

[10] F. Matenaar, A. Wichmann, F. Leder, and E. Gerhards-Padilla, "Cis: The crypto intelligence system for automatic detection and localization of cryptographic functions in current malware," in *2012 7th International Conference on Malicious and Unwanted Software*. IEEE, 2012, pp. 46–53.

[11] O. Mirzaei, J. de Fuentes, J. Tapiador, and L. Gonzalez-Manzano, "Androdet: An adaptive android obfuscation detector," *Future Generation Computer Systems*, vol. 90, pp. 240–261, 2019.

[12] A. Mohammadinodooshan, U. Kargén, and N. Shahmehri, "Robust detection of obfuscated strings in android apps," in *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, ser. AISec'19. New York, NY, USA: Association for Computing Machinery, 2019, p. 25–35.

[13] H. M. Nguyen, E. W. Cooper, and K. Kamei, "Borderline over-sampling for imbalanced data classification," *International Journal of Knowledge Engineering and Soft Data Paradigms*, vol. 3, no. 1, pp. 4–21, 2011.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

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

[16] P. Shijo and A. Salim, "Integrated static and dynamic analysis for malware detection," *Procedia Computer Science*, vol. 46, pp. 804–811, 2015.

[17] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*, 1st ed. Springer: Springer, 2012.

[18] C. Zarate, S. Garfinkel, A. Heffernan, S. Horras, and K. Gorak, "Analysis of the Use of XOR as an Obfuscation Technique in a Real Data Corpus," in *Advances in Digital Forensics X*, G. Peterson and S. Shenoi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, vol. 433, pp. 117–132, series Title: IFIP Advances in Information and Communication Technology.