Сучасний стан наукових досліджень та технологій в промисловості. 2024. №3 (29)
ISSN 2522-9818 (print)
ISSN 2524-2296 (online)

UDC 004.94 DOI: https://doi.org/10.30837/2522-9818.2024.29.039

A. Nafiiev, A. Rodionov

ARCHITECTURE OF AN AUTOMATED PROGRAM COMPLEX BASED ON A MULTIPLE KERNEL SVM CLASSIFIER FOR ANALYZING MALICIOUS EXECUTABLE FILES

Subject matter. This article presents the development and architecture of an automated program complex designed to identify and analyze malicious executable files using a classifier based on a multiple kernel support vector machine (SVM). **Goal.** The aim of the work is to create an automated system that enhances the accuracy and efficiency of malware detection by combining static and dynamic analysis into a single framework capable of processing large volumes of data with optimal time expenditure. **Tasks.** To achieve this goal, tasks were carried out that included developing a program complex that automates the collection of static and dynamic data from executable files using tools like IDA Pro, IDAPython, and Drakvuf; integrating a multiple kernel SVM classifier to analyze the collected heterogeneous data; validating the system's effectiveness based on a substantial dataset containing 1,389 executable samples; and demonstrating the system's scalability and practical applicability in real-world conditions. **Methods.** The methods involved a hybrid approach that combines static analysis – extracting byte code, disassembled instructions, and control flow graphs using IDA Pro and IDAPython – with dynamic analysis, which entails monitoring real-time behavior using Drakvuf. The multiple kernel SVM classifier integrates different data representations using various kernels, allowing for both linear and nonlinear relationships to be considered in the classification process. **Results.** The results of the study show that the system achieves a high level of accuracy and completeness, as evidenced by key performance metrics such as an F-score of 0.93 and ROC AUC and PR AUC values. The automated program complex reduces the analysis time of a single file from an average of 11 minutes to approximately 5 minutes, effectively doubling the throughput compared to previous methods. This significant reduction in processing time is critically important for deployment in environments where rapid and accurate malware detection is necessary. Furthermore, the system's scalability allows for efficient processing of large data volumes, making it suitable for real-world applications. **Conclusions.** In conclusion, the automated program complex developed in this study demonstrates significant improvements in the accuracy and efficiency of malware detection. By integrating multiple kernel SVM classification with static and dynamic analysis, the system shows potential for real-time malware detection and analysis. Its scalability and practical applicability indicate that it could become an important tool in combating modern cyber threats, providing organizations with an effective means to enhance their cybersecurity.

Keywords: cybersecurity; malware detection; automated program complex; static analysis; dynamic analysis; drakvuf; IDA Pro; multiple kernel.

Introduction

Cyber threats have become increasingly sophisticated, presenting significant challenges to global security. Modern malware uses advanced techniques like code obfuscation, polymorphism, and zero-day exploits to bypass traditional security measures. This escalation necessitates the development of more robust detection systems capable of identifying threats in real time. Traditional antivirus programs, relying on signature-based detection, are inadequate against novel and rapidly changing malware variants. There is an urgent need for automated systems that can effectively analyze executable files without prior knowledge of specific threats, processing large volumes of data efficiently. Integrating machine learning into malware detection shows promise in enhancing accuracy and speed. By combining static and dynamic analysis methods, comprehensive features can be extracted from executable files, enabling the detection of sophisticated malware that traditional methods might miss. However, challenges remain in optimizing these systems for performance and scalability, especially when dealing with the vast and diverse nature of modern malware.

Analysis of publications and problem statement

Study [1] focuses on the use of deep learning for the static analysis of complete executable files. It presents a method that automates feature extraction without the need for manual definition, allowing the system to ingest .exe files and classify them as malicious or benign. This approach employs convolutional neural networks to effectively detect complex patterns and anomalies in binary data indicating maliciousness. However, limitations arise due to the large volumes of data required for training and vulnerability to adversarial attacks. Additionally, the significant computational resources needed restrict its practical deployment.

Another study [2] explores the application of dynamic analysis for automating the feature extraction process, relying on the behavioral characteristics of malware. It utilizes techniques that detect malicious actions based on observations of program execution, providing a deeper level of analysis. Despite its effectiveness, this approach demands substantial computational resources and time, rendering it inefficient for real-time applications where prompt detection is essential.

Recent research [3] introduces a novel approach to malware detection using graph-based features. It employs graphlet frequency distribution as feature vectors for classifying malware, demonstrating improved accuracy over traditional methods. Similarly, study [4] has explored feature selection and learning techniques for graphlet kernels, further advancing malware analysis through machine learning. These methods highlight the potential of graph-based representations in capturing complex structural patterns within malware code.

Furthermore, research [5] investigates the performance overhead of virtual machine introspection using Drakvuf for malware analysis. This study underscores the potential of virtual machine introspection in providing detailed behavioral insights while minimizing system performance degradation. Another work [6] contributes by employing graphlet analysis on complex data, which can be adapted to analyze intricate malware behaviors, offering deeper insights into malware dynamics.

Based on these challenges, the goal of this work is to develop an automated program complex that integrates static and dynamic analysis into a single system using a multiple kernel SVM classifier, thereby enhancing the accuracy and efficiency of malware detection. By creating an automated system capable of analyzing large volumes of data with optimal time expenditure, the proposed solution aims to address the limitations of previous methods. Based on our previous works [7, 8, 9, 10] that theoretically and practically substantiated the use of an SVM classifier, a program complex was developed that integrates the described approaches and algorithms.

Architecture of the program complex

The program complex developed in this work employs a multiple kernel SVM classifier structure, ensuring an effective combination of static and dynamic analysis of executable files. The program complex consists of three interconnected modules that provide data collection, processing, and analysis, forming the necessary foundation for effective malware detection. Each of these modules has its unique parameters set, directly influencing the final decision of the SVM classifier regarding the maliciousness of an executable file [11].

The **Information Extraction Module** consists of four submodules, each focused on a specific type of executable file representation:

1. Binary: This submodule uses IDA Pro and IDAPython to extract byte code from executable files, allowing for low-level information retrieval about the file's structure. [12, 13]

2. Disassembly instructions: Utilizing IDA Pro and IDAPython, this submodule extracts disassembled instructions, enabling the analysis of the program's internal commands. [14]

3. CFG (Control Flow Graphs): Also using IDA Pro and IDAPython, this submodule creates control flow graphs that depict the possible execution paths of the program and its structural connections. [15]

4. Dynamic: The use of Drakvuf for dynamic analysis allows tracking the program's execution in real-time, providing behavioral features essential for identifying malicious actions. [16, 17]

The **Information Processing Module** analyzes and transforms the data obtained from the first module, preparing it for classification. This module optimizes the data for further analysis, ensuring its standardization and normalization, which are necessary for accurate machine learning.

The **Multiple Kernel SVM Classifier Module** classifies executable files as malicious or benign. The integration of features from different sources and their analysis using various SVM kernels ensures high accuracy in malware detection, considering both static and dynamic aspects of the analysis. The architecture of the multiple kernel SVM classifier program complex can be seen in Figure 1. Experiments were conducted on a personal computer with the following specifications: Windows 10 operating system, Intel Core i5-7600 processor, 8 GB of RAM.

ISSN 2522-9818 (print)
ISSN 2524-2296 (online)

The dynamic analysis was performed in a Linux (Ubuntu 22.04) virtual machine environment with Windows 7 operating system, allocated 4 GB of RAM and two processor cores.

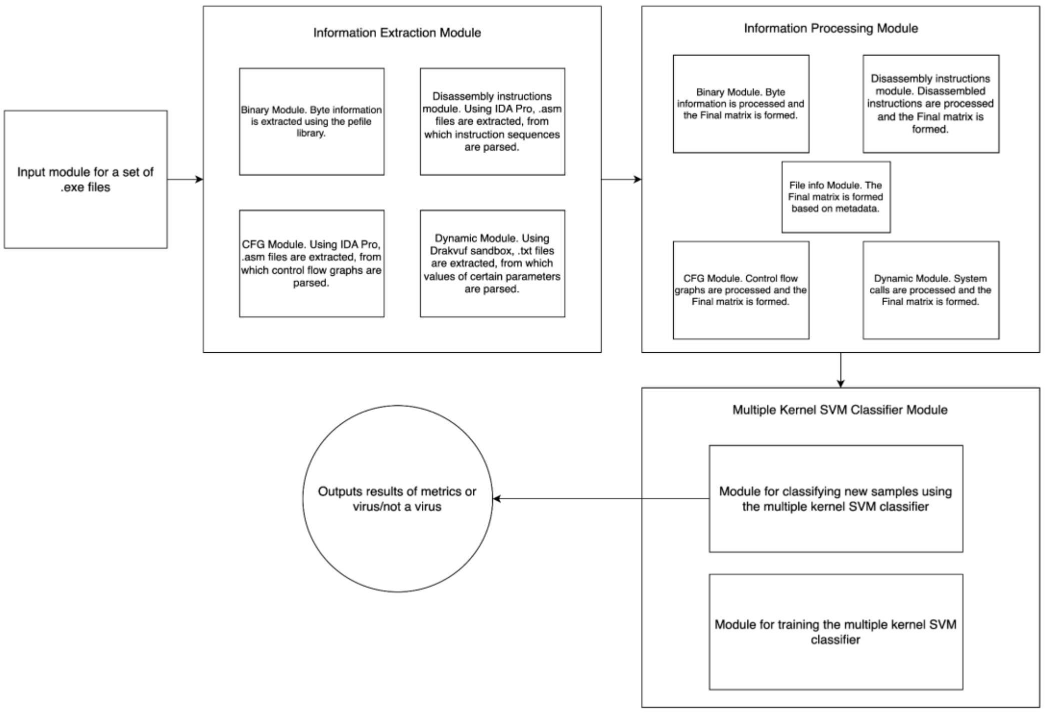

Input module for a set of .exe files
Information Extraction Module
Binary Module. Byte information is extracted using the pefile library.
Disassembly instructions module. Using IDA Pro, .asm files are extracted, from which instruction sequences are parsed.
CFG Module. Using IDA Pro, .asm files are extracted, from which control flow graphs are parsed.
Dynamic Module. Using Drakvuf sandbox, .txt files are extracted, from which values of certain parameters are parsed.
Information Processing Module
Binary Module. Byte information is processed and the Final matrix is formed.
Disassembly instructions module. Disassembled instructions are processed and the Final matrix is formed.
File info Module. The Final matrix is formed based on metadata.
CFG Module. Control flow graphs are processed and the Final matrix is formed.
Dynamic Module. System calls are processed and the Final matrix is formed.
Multiple Kernel SVM Classifier Module
Module for classifying new samples using the multiple kernel SVM classifier
Module for training the multiple kernel SVM classifier
Outputs results of metrics or virus/not a virus

Fig. 1. Architecture of the program complex

Dataset

To validate and train the proposed architecture of the program complex, an own dataset was collected from public sources. Forming the dataset is a fundamental step in creating a machine learning model [18, 19]. For this study, a dataset comprising 1,389 executable files was assembled. This set was divided into training and testing samples. The training sample includes 889 files, with 437 benign and 452 malicious files. The testing set includes 500 files, with 250 benign and 250 malicious files. The ratio of malicious to benign files is approximately 50/50 to achieve more balanced accuracy results for the model. Figure 2 and Figure 3 shows the distribution charts of files by type for each sample.

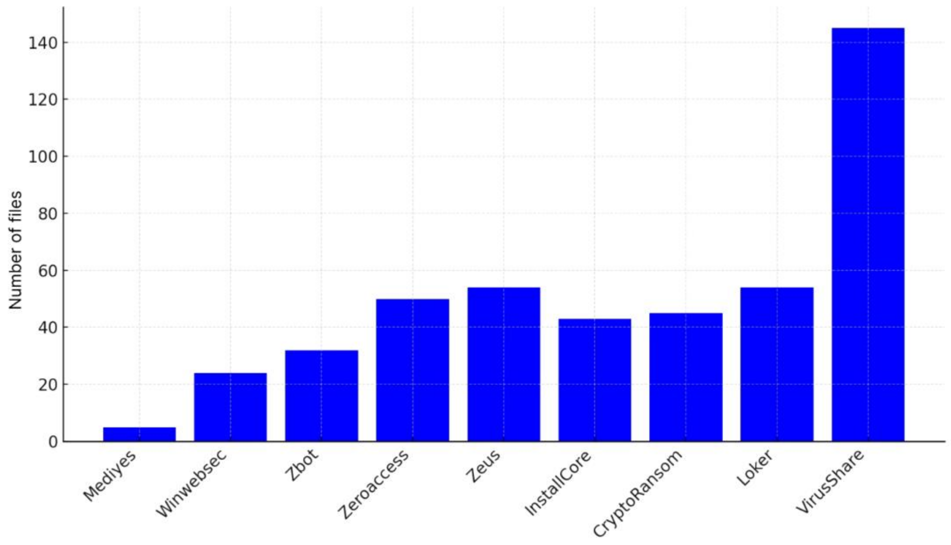


Fig. 2. Infographic of the distribution of malicious files by type in the training set

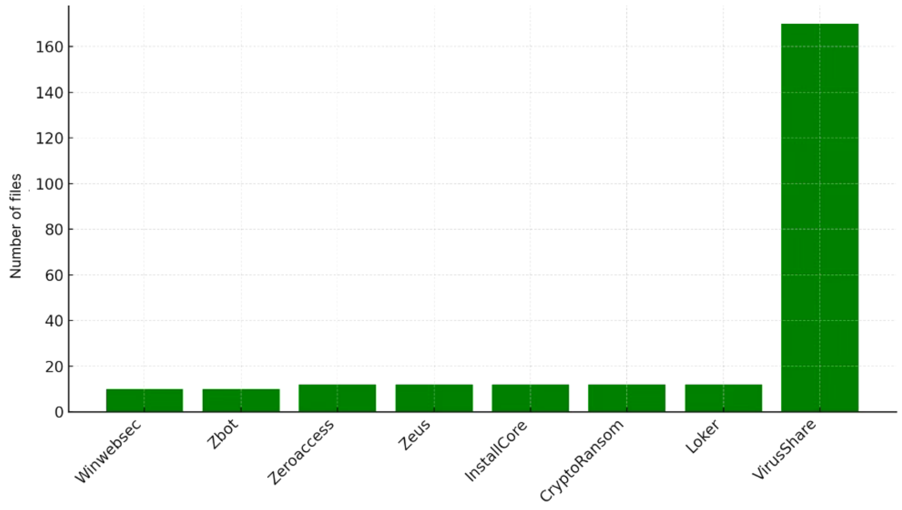


Fig. 3. Infographic of the distribution of malicious files by type in the test set

The malicious files were collected from various sites: "virusshare.com", "malicia-project.com", and "thezoo.morirt.com". The benign files were taken from the installation folders of legitimate software applications across different categories. Additionally, files were sourced from the website "exefiles.com". The "VirusShare" category represents a collection of malicious files of unknown origin, downloaded from the "virusshare.com" resource. The assembled dataset was used for training and testing the program complex, allowing its effectiveness and accuracy to be verified under conditions close to real-world scenarios.

Information extraction module

Automation of static analysis data collection

The automation of the static analysis process in our program complex is a crucial component that allows for the rapid processing of large volumes of executable files. This process is based on using IDA Pro with integrated scripting in IDAPython, which implements a complete analysis cycle for three different representations of an executable file: bytecode, disassembled instructions, and control flow graphs. The static analysis process starts with the launch of the main script "main_static.py", which automatically processes a set of executable files. For each file, the script runs IDA Pro via the command line and applies the "IDAPython_static.py" script, which performs several key tasks:

– Bytecode extraction: The script extracts all the bytecode from the executable file. This code, which is the fundamental representation of the program at the lowest level, provides critical information for further analysis of malicious patterns.

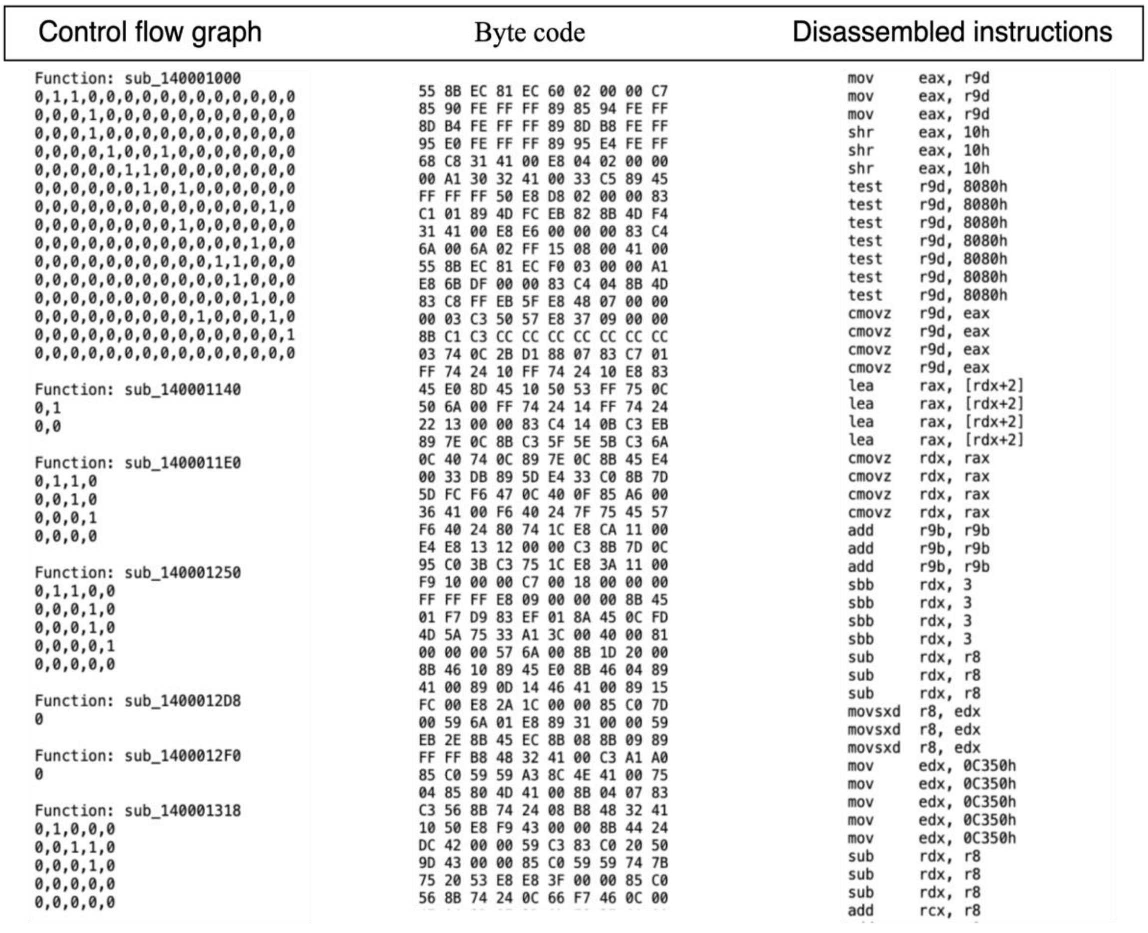
– Disassembled instructions extraction: Using the IDA Pro disassembler, the script transforms the bytecode into disassembled instructions, allowing an understanding of the program's logic and potentially malicious actions.

– Control flow Graph (CFG) generation: The script constructs control flow graphs representing all possible execution paths of the program. These graphs are essential for identifying complex behavioural patterns and dependencies between different code blocks.

All collected information from each executable file is stored as text files, enabling convenient archiving and data analysis during subsequent processing. The recorded data includes raw byte data, disassembled code, and control flow graphs as adjacency matrices. Examples of these text files can be seen in Figure 4. The required time to process one executable file ranges from 4 to 18 seconds, depending on the file itself.

Automation of dynamic analysis data collection

The dynamic analysis process starts with the execution of a control script that orchestrates a sequence of actions to run each executable file in a virtual environment. The system utilizes the Drakvuf tool to monitor system calls, interactions with the file system, registry, network requests, and other critical program behaviour parameters [20]. The control script coordinates the process, ensuring each file runs in isolation, and no processes can affect the host machine or other virtual machines. The analysis of an executable file lasts 120 seconds, during which the running program has time to exhibit its properties. After the execution, Drakvuf logs all relevant events triggered by the executable file and saves this information in text files for further analysis. It should be noted that preparation for analysis and process completion requires additional time. Before running each executable file, it is necessary to prepare the environment: start the virtual machine and create its snapshot. A snapshot is a state capture of the virtual machine at a certain point in time, allowing it to be quickly restored to this state after the analysis. This is particularly important when working with malicious programs, as it ensures that each file runs in the same state, and any system changes caused by the malicious program do not affect subsequent tests. After analyzing the executable file, the virtual machine is destroyed. Overall, starting the virtual machine, creating the snapshot, and deleting the environment after the analysis adds extra time, resulting in a total processing time of one file of 280 seconds. The dynamic analysis system is described in detail in our previous work [8].


Control flow graph

```
Function: sub_140001000
0,1,1,0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,1,0,0,0,0,0,0,0,0,0,0,0
0,0,0,1,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,1,0,0,1,0,0,0,0,0,0,0
0,0,0,0,0,1,1,0,0,0,0,0,0,0,0
0,0,0,0,0,0,1,0,1,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0,0,1,0
0,0,0,0,0,0,0,0,1,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0,1,0,0
0,0,0,0,0,0,0,0,0,0,1,1,0,0,0
0,0,0,0,0,0,0,0,0,0,0,1,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0,1,0,0
0,0,0,0,0,0,0,0,0,1,0,0,0,1,0
0,0,0,0,0,0,0,0,0,0,0,0,0,0,1
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

Function: sub_140001140
0,1
0,0

Function: sub_1400011E0
0,1,1,0
0,0,1,0
0,0,0,1
0,0,0,0

Function: sub_140001250
0,1,1,0,0
0,0,0,1,0
0,0,0,1,0
0,0,0,0,1
0,0,0,0,0

Function: sub_1400012D8
0

Function: sub_1400012F0
0

Function: sub_140001318
0,1,0,0,0
0,0,1,1,0
0,0,0,1,0
0,0,0,0,0
0,0,0,0,0
```

Byte code

```
55 8B EC 81 EC 60 02 00 00 C7
85 90 FE FF FF 89 85 94 FE FF
8D B4 FE FF FF 89 8D B8 FE FF
95 E0 FE FF FF 89 95 E4 FE FF
68 C8 31 41 00 E8 04 02 00 00
00 A1 30 32 41 00 33 C5 89 45
FF FF FF 50 E8 D8 02 00 00 83
C1 01 89 4D FC EB 82 8B 4D F4
31 41 00 E8 E6 00 00 00 83 C4
6A 00 6A 02 FF 15 08 00 41 00
55 8B EC 81 EC F0 03 00 00 A1
E8 6B DF 00 00 83 C4 04 8B 4D
83 C8 FF EB 5F E8 48 07 00 00
00 03 C3 50 57 E8 37 09 00 00
8B C1 C3 CC CC CC CC CC CC CC
03 74 0C 2B D1 88 07 83 C7 01
FF 74 24 10 FF 74 24 10 E8 83
45 E0 8D 45 10 50 53 FF 75 0C
50 6A 00 FF 74 24 14 FF 74 24
22 13 00 00 83 C4 14 0B C3 EB
89 7E 0C 8B C3 5F 5E 5B C3 6A
0C 40 74 0C 89 7E 0C 8B 45 E4
00 33 DB 89 5D E4 33 C0 8B 7D
5D FC F6 47 0C 40 0F 85 A6 00
36 41 00 F6 40 24 7F 75 45 57
F6 40 24 80 74 1C E8 CA 11 00
E4 E8 13 12 00 00 C3 8B 7D 0C
95 C0 3B C3 75 1C E8 3A 11 00
F9 10 00 00 C7 00 18 00 00 00
FF FF FF E8 09 00 00 00 8B 45
01 F7 D9 83 EF 01 8A 45 0C FD
4D 5A 75 33 A1 3C 00 40 00 81
00 00 00 57 6A 00 8B 1D 20 00
8B 46 10 89 45 E0 8B 46 04 89
41 00 89 0D 14 46 41 00 89 15
FC 00 E8 2A 1C 00 00 85 C0 7D
00 59 6A 01 E8 89 31 00 00 59
EB 2E 8B 45 EC 8B 08 8B 09 89
FF FF B8 48 32 41 00 C3 A1 A0
85 C0 59 59 A3 8C 4E 41 00 75
04 85 80 4D 41 00 8B 04 07 83
C3 56 8B 74 24 08 B8 48 32 41
10 50 E8 F9 43 00 00 8B 44 24
DC 42 00 00 59 C3 83 C0 20 50
9D 43 00 00 85 C0 59 59 74 7B
75 20 53 E8 E8 3F 00 00 85 C0
56 8B 74 24 0C 66 F7 46 0C 00
```

Disassembled instructions

```
mov     eax, r9d
mov     eax, r9d
mov     eax, r9d
shr     eax, 10h
shr     eax, 10h
shr     eax, 10h
test    r9d, 8080h
test    r9d, 8080h
test    r9d, 8080h
test    r9d, 8080h
test    r9d, 8080h
test    r9d, 8080h
test    r9d, 8080h
cmovz   r9d, eax
cmovz   r9d, eax
cmovz   r9d, eax
cmovz   r9d, eax
lea     rax, [rdx+2]
lea     rax, [rdx+2]
lea     rax, [rdx+2]
lea     rax, [rdx+2]
cmovz   rdx, rax
cmovz   rdx, rax
cmovz   rdx, rax
cmovz   rdx, rax
add     r9b, r9b
add     r9b, r9b
add     r9b, r9b
sbb     rdx, 3
sbb     rdx, 3
sbb     rdx, 3
sbb     rdx, 3
sub     rdx, r8
sub     rdx, r8
sub     rdx, r8
movsxd  r8, edx
movsxd  r8, edx
movsxd  r8, edx
mov     edx, 0C350h
mov     edx, 0C350h
mov     edx, 0C350h
mov     edx, 0C350h
mov     edx, 0C350h
sub     rdx, r8
sub     rdx, r8
sub     rdx, r8
add     rcx, r8
```

Fig. 4. Fragments of text files of static data types

Information processing module

The information processing module takes a set of text files for each of the four data representations as input. The module's goal is to transform the text files into a format suitable for machine learning, particularly for the SVM classifier. The output consists of five final matrices for each data representation: Binary, Instructions, System calls, CFG, File Info.

The data processing process begins with the automated parsing of text files by running a Python script that uses regular expressions to parse information from the text files and store it in NumPy arrays. This makes it easier to manipulate the data in subsequent processing stages. The resulting arrays store sequences of certain elements such as bytes – [4A, 5D, 00, 6G, …, FF], disassembled instructions – [mov, push, call, …, jm], or system calls – [NtOpenKeyEx, NtDelayExecution, NtAlpcQueryInformationMessage, …, NtTraceEvent]. The next step is to create adjacency matrices that account for interactions between sequential elements. However, first, it is necessary to determine the feature set for each data representation, as the adjacency matrices are built based on this feature set. For the binary representation, the feature set includes all existing bytes, so the adjacency matrix's dimensionality equals 256×256.

For the other two representations, all corresponding NumPy arrays must be analyzed and the number of unique elements: disassembled instructions or system calls, must be counted. It was found that our set of executable files includes 322 unique disassembled instructions and 241 unique system call names. These unique elements form the basis of the corresponding adjacency matrices, from which transition matrices are then built, where each cell stores the probability of transitioning from one byte, instruction, or system call to another. Each transition matrix is then converted into a vector representing a separate row in the final matrix of a specific data representation. This vectorization process reduces the data complexity to a format that can be effectively used for machine learning. The final matrix's dimensionality, which is input to the SVM classifier, can be calculated using the following formula: the number of files * (the number of unique elements)^2.

To process text files with control flow graphs, the graphlet kernel method is used. The graphlet kernel analyzes a graph through its local structures - graphlets, which are subgraphs with a certain number of vertices. We use four-vertex graphlets, such as 4-clique, 4-chordalcycle, 4-tailedtriangle, 4-cycle, 3-star, 4-path. For each executable file, the number of occurrences of each graphlet type is counted, and this data is transformed into a feature set representing the percentage ratio of each graphlet type in the graphs of all executable file functions.

Based on the already processed data, another representation of the executable file is built – File info. Seven key metadata are used, which help gather a broader spectrum of information about the executable file and its potential malicious activity. This metadata includes: file entropy, which indicates the randomness level of the data; file packing to detect code obfuscation; file size; the number of vertices and edges in the control flow graph; and the number of static and dynamic instructions. The data processing for the five data representation types is described in detail in our previous work [10].

Multiple kernel SVM classifier module

The foundation of the software complex for analyzing malicious executable files is a multiple kernel SVM classifier capable of integrating various types of data into a single model. These representations include binary code, disassembled instructions, control flow graph information, dynamic analysis of system calls, and file metadata. Combining this data into a single integrated machine-learning model allows the classifier to consider the diverse aspects and features of each data type during training, significantly improving the accuracy of detecting malicious files. One of the key aspects of the multiple kernel SVM classifier is its ability to combine different types of kernels, allowing the use of both linear and nonlinear relationships between the data. We use Gaussian and polynomial kernels. The multiple kernel SVM classifier module is divided into two submodules. The first submodule handles training the classifier, using the available data to learn and improve the model before applying it in real-world conditions. The input to the submodule consists of five matrices, each corresponding to its data representation. During training, these matrices are combined using kernels, allowing the model to account for various dependencies between the data.This approach ensures high accuracy in classifying malicious programs and the model's adaptability to new samples. For retraining the classifier in case of new samples, an incremental learning approach is used, where new data is added to the existing training dataset, and the model is retrained to account for these new samples. The second submodule classifies new samples using the already trained model. The process of the multiple kernel SVM classifier is detailed in our previous work [10].

Results

This section compares the results of the new model developed in this study (New model) and the old model presented in our previous work [10]. The Old model was trained on the dataset used in the previous work, which included 180 executable files, but testing was carried out on the new test set used in this study. The New model described in this work was trained on a set of 889 files and was also tested on the same test set of 500 executable files

Metrics such as F-score, Precision, Recall, ROC AUC, and PR AUC are used to reflect the models' ability to distinguish between malicious and benign files. Additionally, the throughput of each system is compared, providing insight into efficiency and speed. The metrics results can be seen in Table 1. The training dataset in the new model significantly increased compared to the previous work, likely impacting the overall accuracy improvement. The F-score, Precision, and Recall metrics not only improved but also became more balanced, indicating the new model's ability to effectively recognize both classes (malicious and benign files). The previous model was more prone to identifying malicious files. In this study, the method of distributing types of malicious files for the test and training sets was changed. The test set includes a significant number of 'undefined' malicious files that are not classified into specific families. This gives the model a more balanced and generalized ability to recognize both malicious and benign files, making it more realistic and better at adapting to unknown threats.

In the previous work, the dynamic analysis with Drakvuf Sandbox required manually running each executable file and manually saving the results, taking about eight minutes per file. In the static analysis, files were manually opened in IDA Pro, and text files were saved, taking about three minutes per file. The new program complex with a multiple kernel SVM classifier automated the data collection processes. Dynamic analysis based on Drakvuf now takes 280 seconds per file, and static analysis with IDA Pro processes each file in 4–18 seconds. Automation made the system's throughput two times faster, making it more efficient in detecting cyber threats.

Table 1. *Experiment results. The throughput metric represents the minutes required to extract information from one executable file*

	F-score		Precision		Recall		roc_auc	pr_auc	throughput
	0	**1**	**0**	**1**	**0**	**1**			
Old model	0.8667	0.8732	0.8903	0.8517	0.8443	0.8961	0.9521	0.9542	11
New model	0.9327	0.9351	0.9502	0.9189	0.9160	0.9520	0.9716	0.9723	5

Conclusion

In this study, an automated program complex based on a multiple kernel SVM classifier was developed for analyzing malicious executable files. One of the key results is the notable increase in the system's throughput, specifically the reduction in the time required to analyze each file. Compared to the previous system, where processing each file took an average of 11 minutes, the automated program complex now requires only about 5 minutes per file. This improvement allows for the analysis to be over two times faster, optimizing time and resource utilization. Thanks to the improved data processing speed, our complex can be useful in various real-world application conditions. In cases where a high volume of files needs to be processed, such as in corporate environments or cloud security services, the analysis process can be parallelized if sufficient computational resources are available. For example, if 100 files are received, they can be distributed across multiple instances of the program complex, maintaining an effective analysis time of 5 minutes per file. Additionally, with adequate computational power, the processing speed can be further increased, potentially reducing the time to ~3 minutes per file. By combining the automation of static and dynamic analysis, the program complex based on the multiple kernel SVM classifier reliably detects malicious programs.

References

1. Raff, E., et al. (2018), "Malware Detection by Eating a Whole EXE." *Workshop on Binary Analysis Research (BAR).*
2. Santos, I., et al. (2013), "Opcode Sequences as Representation of Executables for Data-Mining-Based Unknown Malware Detection." *Information Sciences*, vol. 231, pp. 64–82.
3. Tu, K., Li, J., Towsley, D. and Braines, D. (2019), "gl2vec: Learning feature representation using graphlets for directed networks", *Proceedings of the 2019 Workshop on Binary Analysis Research*. DOI: 10.1145/3341161.3342908
4. Aziz, F., Ullah, A. and Shah, F. (2020), "Feature selection and learning for graphlet kernel", *Pattern Recognition Letters*, 140, pp. 45–51. DOI: 10.1016/j.patrec.2020.05.019
5. Paakkola, S. (2020), "Assessing performance overhead of Virtual Machine Introspection and its suitability for malware analysis", *University of Turku*. Available at: https://core.ac.uk/download/pdf/347180664.pdf

6. Khater, I.M., Meng, F., Nabi, I.R. and Hamarneh, G. (2019), "Identification of caveolin-1 domain signatures via machine learning and graphlet analysis of single-molecule super-resolution data", *Bioinformatics*, 35(18), pp. 3468–3474. DOI: 10.1093/bioinformatics/btz951
7. Nafiiev Alan, Kholodulkin Hlib, Rodionov Andrii, (2021) "Comparative analysis of machine learning methods for detecting malicious files". *Theoretical and Applied Cybersecurity*, Vol. 3 No. 1, pp 46–51.
8. Alan Nafiiev, Hlib Kholodulkin, Andrii Rodionov, (2022), "Malware dynamic analysis system based on virtual machine introspection and machine learning methods", *Information Technologies and Security*. *Proceedings of the XXII International Scientific and Practical Conference ITB-2022*. Issue 22: pp 53–58.
9. Nafiiev Alan, Lande Dmytro, (2023), "Malware detection model based on machine learning". *Bulletin of Cherkasy State Technological University*, No. 3, pp. 40–50.
10. Nafiiev Alan, Rodionov Andrii, (2023), "Malware detection system based on static and dynamic analysis using machine learning", *Theoretical and Applied Cybersecurity*, Vol. 5 No. 2, pp. 97–104.
11. Rizvi, S.K.J., Aslam, W., Shahzad, M., Saleem, S. (2022), "PROUD-MAL: static analysis-based progressive framework for deep unsupervised malware classification of windows portable executable", *Complex & Intelligent Systems*, 8(1), pp. 1345–1361. DOI: 10.1007/s40747-021-00560-1
12. Faloutsos, M. (2019), "IDAPro for IoT Malware analysis? ", *Workshop on Binary Analysis Research (BAR)*, Available at: https://escholarship.org/content/qt4rp172kk/qt4rp172kk.pdf
13. Chen, Z., Brophy, E., Ward, T. (2021), "Malware classification using static disassembly and machine learning", *arXiv preprint arXiv:2201.07649*.
14. Talukder, S. (2020), "Tools and techniques for malware detection and analysis", arXiv preprint arXiv:2002.06819, Available at: https://www.researchgate.net/publication/339301928_Tools_and_Techniques_for_Malware_Detection_and_Analysis
15. Aziz, F., Ullah, A. and Shah, F. (2020), "Feature selection and learning for graphlet kernel", *Pattern Recognition Letters*, 140, pp. 45–51. DOI: 10.1016/j.patrec.2020.05.019
16. Singh, S. (2023), "DRAKVUF Malware Sandbox", *World Forum on Engineering and Science*, 5(1), pp. 23–30. DOI: 10.5281/zenodo.5544337
17. Dietz, C., Antzek, M., Dreo, G., Sperotto, A. (2022), "Dmef: Dynamic malware evaluation framework", *International Journal of Information Security*, 21(1), pp. 67–85. DOI: 10.1007/s10207-021-00554-1
18. Sidey-Gibbons, J.A.M. and Sidey-Gibbons, C.J. (2019), "Machine learning in medicine: a practical introduction", *BMC Medical Research Methodology*, 19(1). DOI: 10.1186/s12874-019-0681-4
19. Starink, J.A.L. (2021), "Analysis and automated detection of host-based code injection techniques in malware", *Journal of Computer Virology and Hacking Techniques*, 17(1), pp. 1–12. DOI: 10.1007/s11416-020-00356-0
20. Leszczyński, M. and Stopczański, K. (2020), "A new open-source hypervisor-level malware monitoring and extraction system-current state and further challenges", *Virus Bulletin 2020*, Available at: https://vblocalhost.com/uploads/VB2020-Leszczynski-Stopczanski.pdf (Accessed: 14 July 2024).

Надійшла (Received) 14.07.2024

Відомості про авторів / About the Authors

Нафієв Алан Емінович – Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського", Фізико-технічний інститут, аспірант, Київ, Україна; e-mail: Alan.nafiev@gmail.com; ORCID ID: https://orcid.org/0009-0004-8604-377X

Родіонов Андрій Миколайович – кандидат технічних наук, доцент, Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського", Фізико-технічний інститут, Київ, Україна; e-mail: andrey.rodionov@gmail.com; ORCID ID: http://orcid.org/ 0000-0001-7284-9458

Nafiiev Alan – National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Institute of Physics and Technology, PhD student, Kyiv, Ukraine.

Rodionov Andrii – PhD (Engineering Sciences), Associate Professor, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Institute of Physics and Technology, Kyiv, Ukraine.

АРХІТЕКТУРА АВТОМАТИЗОВАНОГО ПРОГРАМНОГО КОМПЛЕКСУ НА ОСНОВІ БАГАТОЯДЕРНОГО *SVM*-КЛАСИФІКАТОРА ДЛЯ АНАЛІЗУ ШКІДЛИВИХ ВИКОНУВАНИХ ФАЙЛІВ

Тематика дослідження. У статті запропоновано розроблення та архітектуру автоматизованого програмного комплексу, призначеного для ідентифікації та аналізу шкідливих виконуваних файлів за допомогою класифікатора на основі багатоядерного навчання машини опорних векторів (*SVM*). **Мета** – створення автоматизованої системи, що підвищує точність і ефективність виявлення шкідливого програмного забезпечення завдяки поєднанню статичного й динамічного аналізу в єдину структуру, здатну обробляти значні обсяги даних з оптимальними витратами часу. **Завдання статті.** Для досягнення окресленої мети розроблено програмний комплекс, що автоматизує збір статичних і динамічних відомостей із виконуваних файлів за допомогою таких інструментів, як *IDA Pro, IDAPython* і *Drakvuf*; застосовано інтеграцію багатоядерного класифікатора *SVM* для аналізу зібраних різнорідних даних; виконано валідацію ефективності системи на основі значного датасету, що містить 1 389 виконуваних зразків; продемонстровано масштабованість і практичну застосовність системи в реальних умовах. **Методи** передбачали гібридний підхід, що поєднує статичний аналіз – витяг байт-коду, дизасембльованих інструкцій та графів потоку керування за допомогою *IDA Pro* та *IDAPytho*n – з динамічним аналізом, який полягав у моніторингу поведінки в реальному часі за допомогою *Drakvuf*. Багатоядерний класифікатор *SVM* інтегрує різні подання даних, використовуючи різні ядра, що дає змогу брати до уваги як лінійні, так і нелінійні взаємозв’язки в процесі класифікації. **Результати дослідження** продемонстрували, що система досягає високого рівня точності та повноти, про що свідчать ключові метрики ефективності, зокрема *F*-міра 0,93 та значення *ROC AUC* і *PR AUC*. Автоматизований програмний комплекс зменшує час аналізу одного файлу з середніх 11 хв до приблизно 5 хв, що фактично подвоює пропускну здатність порівняно з попередніми методами. Це значне скорочення часу оброблення є критично важливим для впровадження в середовищах, де необхідне швидке й точне виявлення шкідливого програмного забезпечення. Крім того, масштабованість системи дає змогу ефективно обробляти значні обсяги даних, що робить її придатною для реального застосування. **Висновки.** Розроблений у межах цього дослідження автоматизований програмний комплекс демонструє значні поліпшення щодо точності та ефективності виявлення шкідливого програмного забезпечення. Інтегруючи багатоядерну класифікацію *SVM* зі статичним і динамічним аналізом, система виявляє потенціал для аналізу шкідливого ПЗ в реальних умовах. Її масштабованість та практична застосовність свідчать про те, що система може стати важливим інструментом у боротьбі із сучасними кіберзагрозами, надаючи організаціям ефективний засіб для підвищення їх кібербезпеки.

Ключові слова: кібербезпека; виявлення шкідливих програм; автоматизований програмний комплекс; статичний аналіз; динамічний аналіз; *Drakvuf; IDA Pro*; багатоядерне навчання.

Бібліографічні описи / Bibliographic descriptions

Нафієв А. Е., Родіонов А. М. Архітектура автоматизованого програмного комплексу на основі багатоядерного *SVM*-класифікатора для аналізу шкідливих виконуваних файлів. *Сучасний стан наукових досліджень та технологій в промисловості*. 2024. № 3 (29). С. 39–47. DOI: https://doi.org/10.30837/2522-9818.2024.29.039

Nafiiev, A., Rodionov, A. (2024), "Architecture of an automated program complex based on a multiple kernel svm classifier for analyzing malicious executable files", *Innovative Technologies and Scientific Solutions for Industries*, No. 3 (29), P. 39–47. DOI: https://doi.org/10.30837/2522-9818.2024.29.039