Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186-5140
Volume 14, Number 1, pages 10–17, January 2025

Building an Automated Deobfuscation System that Integrates Multiple Deobfuscation Tools (preliminary version)

1st Kohei Arai
Kogakuin University
Tokyo, Japan
em24003@ns.kogakuin.ac.jp

2nd Ryotaro Kobayashi
Kogakuin University
Tokyo, Japan
ryo.kobayashi@cc.kogakuin.ac.jp

***Abstract*—The main purpose of program obfuscation is to protect the security of a program's source code and prevent its misuse. However, attackers can also exploit obfuscation to make analysis and detection of their own programs more difficult. Therefore, it is necessary to deobfuscate the source code during analysis; however, the numerous obfuscation methods available make deobfuscation challenging. In this paper, we aim to automate a system that encompasses deobfuscation tools for multiple methods. Our system employs a detection tool for malicious JavaScript files, a deobfuscation tool for JavaScript, a deobfuscation tool for XOR obfuscation, and a deobfuscation tool for Portable Executable (PE) files in order to achieve detection and deobfuscation. From our results, we confirm the detection of malicious JavaScript files and the deobfuscation of obfuscated files, while for PE files, we verified deobfuscation at the same level as the source code. The implementation of this system enables efficient deobfuscation of numerous obfuscated programs.**

***Index Terms*—Deobfuscation, Automated System, XOR, Packer, JavaScript**

I. INTRODUCTION

Obfuscation involves concealing a program's source code and making reverse engineering difficult. Its primary purpose is to protect the security of the source code and prevent its misuse. However, attackers can also exploit obfuscation to make analysis and detection of their own programs more difficult, necessitating deobfuscation by code analysts. Various obfuscation techniques exist, with packers and XOR being commonly used in many programs. Notable packers include UPX and ASPack. When unpacking libraries are unavailable or proprietary, however, manual deobfuscation is required. There are two main types of deobfuscation: static and dynamic. Examples of static deobfuscation include using unpackers, Large-Language Models (LLM), or compilers, while dynamic deobfuscation typically involves using memory dumps obtained during program execution. An example of the manual dynamic deobfuscation process for a program obfuscated with a packer is as follows:

1) Use static analysis tools to identify program Entry Points (EP);
2) Identify instructions in the packed program to transfer control to the Original Entry Point (OEP);
3) Set a Breakpoint (BP) at an instruction identified by the debugger and execute the program sequentially up to that instruction;
4) Dump process memory to disk and rebuild the Import Address Table (IAT);
5) Use the tool to scan process memory, then search for IAT; and
6) Obtain and apply the list of import functions, after which deobfuscation is complete

While manual dynamic unpacking of packers allows for precise deobfuscation, it involves complex tasks such as identifying the EP and setting BPs, as well as the need for a dynamic analysis environment. Considering these factors, the deobfuscation of numerous programs is impractical. Research has been conducted on static deobfuscation using LLMs and compilers [1–8], as well as on dynamic deobfuscation and has targeted executable files such as Portable Executable (PE) files and codes such as JavaScript [9–11]. Additionally, researchers have developed systems for deobfuscation and there are several existing deobfuscation tools [12–14]. However, each system or tool supports different obfuscation methods and input formats, thus requiring transformation of the program into the format compatible with each tool. In practice, adapting to the usage methods, input formats, and construction environments of various tools is not straightforward when deobfuscating programs.

In this paper, we aim to automate a deobfuscation system that encompasses these tools. Considering that most recent malware targets Windows, we focus on PE files. We also address the detection and deobfuscation of malicious files embedded in JavaScript within HTML emails and web pages. The target obfuscation methods include PE file packing, JavaScript obfuscation, and XOR. XOR obfuscation is used because of its reversibility and difficulty of decryption, making it a suitable target for system-based deobfuscation. Our system employs a virtual environment, using Docker for static deobfuscation and VirtualBox for dynamic deobfuscation.

The structure of this paper is as follows. First, related research is discussed in Section II, while Section III describes the positioning of this study within the field. Section IV

presents the proposed system. Sections V and VI cover the implementation of the proposed system and the results of testing, respectively. Based on these, Section VII provides the conclusion and future prospects.

II. Related Work

In this paper, we propose an automated system for deobfuscating programs. This system integrates multiple tools to achieve deobfuscation. In this section, we review existing research on static and dynamic deobfuscation and discuss previous work on deobfuscation systems. Finally, we position our research within this context.

A. Related Work on Static Deobfuscation

Herrera et al. proposed Safe-Deobfs, a deobfuscation tool for JavaScript [1]. This tool achieves deobfuscation through static analysis based on compiler techniques. It performs constant and value replacement, removal of redundant codes, function inlining, and string decoding. The input is a JavaScript file, and deobfuscation is accomplished by optimizing the input code. In addition, there have been studies on static deobfuscation [2–8]. These applied reduction algorithms to obfuscated programs in order to transform the semantics of the programs. As a result, they reduce redundant codes, streamline control flow, and replace constants and values, thereby deobfuscating the program. These studies on static deobfuscation mainly use algorithms based on LLMs and compilers.

B. Related Work on Dynamic Deobfuscation

Martignoni et al. achieved deobfuscation of packed programs by scanning memory pages for a certain period after detecting dangerous system calls [9]. They defined dangerous system calls as actions like creating or setting registry keys and assumed the period until the next call as the boundary of the unpacking process, thereby estimating the unpacking activity. Tang et al. and Udepa et al. achieved deobfuscation by first performing taint analysis through dynamic execution of the program and then conducting static analysis based on the utilized flows [10,11]. By combining dynamic execution and static analysis in this way, it is possible to deobfuscate without relying on patterns in the obfuscation methods.

C. Related Work on Deobfuscation Systems

Hajarnis et al. proposed a system that integrates static deobfuscation using deobfuscation tools and the detection of malicious JavaScript files [12]. The targeted obfuscations included multiple packers for PE files and XOR. The system achieved deobfuscation using unipacker and XORSearch. Choi et al. proposed a comprehensive system for packing detection, unpacking, and verification [13]. For detection, the system sequentially checks the presence of the EP section and the presence of signatures and then examines write attributes and entropy. For deobfuscation, it uses static deobfuscation if an unpacking library is provided, and dynamic deobfuscation if not. The targeted obfuscation is for PE files, whereas the static deobfuscation supports only UPX. Menguy et al. proposed a search-based black-box deobfuscation tool [14], improving the stability of the search method, as well as the success rate and efficiency, compared to tools developed in prior research [15]. Deobfuscation was achieved by transforming the input functions into semantically equivalent code.

D. Related Work on Detection of Malicious Files in JavaScript

Fass et al. proposed a tool for detecting malicious JavaScript files [17]. This tool identifies JavaScript using analysis based on an abstract syntax tree and classifies malicious files by extracting n-gram features.

III. Positioning of This Paper

In the existing research described in Section II-A, obfuscation of JavaScript and PE files was addressed using compilers and LLMs, achieving deobfuscation by transforming obfuscated code into semantically equivalent code. However, static code transformation for programs is time-consuming, making it impractical when considering deobfuscation of numerous programs.

In Section II-B, deobfuscation was achieved by utilizing the behavior observed during the dynamic execution of the program. The advantage of using dynamic deobfuscation is that it does not depend on specific patterns of obfuscation methods.

In Section II-C, a deobfuscation system was realized by integrating existing deobfuscation tools and detection tools. In Hajarnis et al.'s research, a system integrating static deobfuscation with unipacker, detection of malicious JavaScript files, and XOR-based deobfuscation was constructed. However, the deobfuscation target is limited to packers supported by unipacker. In addition, for JavaScript files, the existing research described in Section II-D was utilized, providing for only malicious file detection.

Choi et al. constructed a system encompassing multiple detection methods, static and dynamic deobfuscation using unpacking libraries, and verification. However, the supported unpacking library is limited to UPX. While dynamic deobfuscation is also possible, making it widely applicable to various obfuscation methods, it is limited to PE file obfuscation.

Menguy et al. constructed a black-box deobfuscation tool to achieve static code transformation on functions. However, since the input is a function, the analyst needs to handle the processing themselves. The process of extracting each function from the program is time-consuming and depends on the accuracy of the tools and methods used, thus posing a challenge.

Therefore, we construct a comprehensive system that includes XOR deobfuscation, detection and deobfuscation of malicious JavaScript files, and both static and dynamic deobfuscation of PE files. Compared to the existing research described in Section II-C, our system integrates various deobfuscation tools for different obfuscation methods, which is the novel contribution of this paper. Furthermore, by using a program as the input to the system and minimizing the need for manual processing by the analyst, we make the system

user-friendly. By unifying the tools to accept programs as input and enabling the deobfuscation of a wide range of obfuscation techniques, we establish the utility of this study.

IV. Proposed System

In this study, we propose an automated deobfuscation system that integrates various tools, including a detection tool for malicious JavaScript files, a deobfuscation tool for JavaScript, a deobfuscation tool for XOR obfuscation, and both static and dynamic deobfuscation tools for PE files. Table I lists the tools used in this system.

TABLE I
List of Tools Used

Tool	URL of the Tools
JAST	https://github.com/Aurore54F/JaSt
js-beautify	https://github.com/beautifier/js-beautify
JStillery	https://github.com/mindedsecurity/JStillery
XORSearch	https://blog.didierstevens.com/programs/xorsearch
unipacker	https://github.com/unipacker/unipacker
mal_unpack	https://github.com/hasherezade/mal_unpack

Additionally, the workflow of the system is described as follows, with an overview of the system illustrated in Figure 1.

1) About file type detection;
2) Processing for JavaScript files (JAST, js-tools);
3) Run XORSearch on PE files (XORSearch);
4) Processing for PE files (Detect Packer); and
5) Deobfuscate PE files (unipacker, mal_unpack);

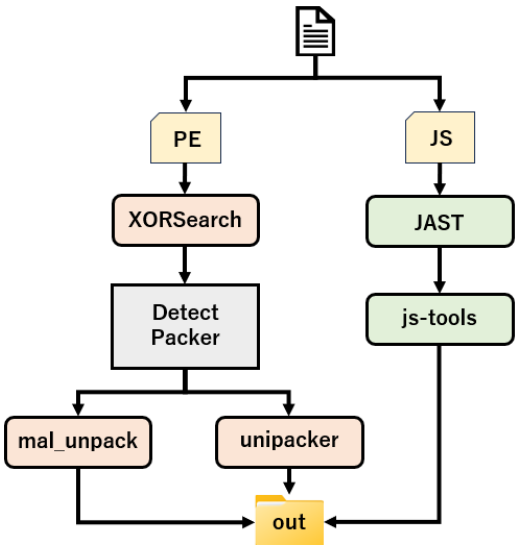

Fig. 1. Flowchart of the system.

The input to this system can be either a PE file or a JavaScript file. The system automatically detects and deobfuscates malicious files for the input file. It is assumed that the PE files and JavaScript files inputted into the system are obfuscated. The detailed procedures for each stage of the process will be described in the following sections.

A. About File Type Detection

The deobfuscation system starts by placing the target files in a folder and running a script. Since the processes for PE files and JavaScript files are different, it is necessary to identify and classify the format of the input files. For file format identification, this system uses Magika, a file format identification tool provided by Google, which utilizes an AI model that has been shown to score higher than other identification tools [18]. While general identification tools recognize the source code of programs in various languages as text files, Magika can classify them by language. The classified PE files and JavaScript files are then processed accordingly.

B. Processing for JavaScript Files

For JavaScript files, the system performs both malicious file detection and deobfuscation. The former is conducted using JAST, a detection tool developed by Fass et al. For the latter, commonly used JavaScript deobfuscation tools such as js-beautify and JStillery are employed. The system inputs a folder containing JavaScript files into JAST, which outputs a report of the detection results for malicious files. For obfuscated files, the system proceeds with deobfuscation using both js-beautify and JStillery.

For unknown malicious files, JAST can be tuned and reconfigured to enable their classification. Since many malicious JavaScript files are obfuscated to hide the program's contents, deobfuscation is essential. This approach automates the detection of malicious JavaScript files and the deobfuscation of JavaScript files, thereby enhancing the overall effectiveness of the system.

C. Run XORSearch on PE Files

For PE files, the system performs deobfuscation of XOR obfuscation and outputs the findings as a report. The tool used for this purpose is XORSearch, which can perform a brute-force search of the input file to extract keywords of interest. Since attackers tend to conceal URLs within programs, the default keyword for investigation in this system is "http". However, as the target keywords may vary depending on the analyst, the system allows the modification of keywords via options, enabling the investigation of any specified string.

D. Processing for PE Files

For PE files, the system detects obfuscation methods. Deobfuscation for PE files involves using unipacker for static deobfuscation and mal_unpack for dynamic deobfuscation. However, unipacker supports only a limited number of packers: ASPack, FSG, MEW, MPRESS, PEtite, and UPX. The detection of these packers is achieved using the analysis tool Detect It Easy (DIE). DIE can analyze the contents of executable files and determine whether they are packed, provided the packer is known. Since all packers supported by unipacker are known packers, DIE enables the identification of samples suitable for static deobfuscation by recognizing these specific packers.

E. Deobfuscate PE Files

We perform both static unpacking using unipacker and dynamic unpacking using mal_unpack on PE files. Based on the results of the processing conducted in Section IV-D, if unipacker detects a corresponding packer, then static unpacking is executed; if no packer is detected, then dynamic deobfuscation using mal_unpack is executed. The static analysis is performed on Docker and, upon completion of the unpacking, the unpacked executable file is output. Since the dynamic unpacking is executed on VirtualBox, it is necessary to send the specimen to be unpacked and, once the obfuscation removal for each specimen is complete, the unpacked executable file is output. This process enables the selection of specimens for static unpacking and the automation of both static and dynamic unpacking.

V. Implementation

In this section, we construct the release system based on the content described in the previous section. In this system, the processing of JavaScript files and PE files is virtualized using Docker and the dynamic release of PE files is virtualized using VirtualBox. The detailed flowchart of this system is shown in Figure 2. By starting the script placed on the host machine, the sample zip file placed on the host machine is extracted, and each stage described in Section IV is automatically executed by the script. First, PE files and JavaScript files are classified using Magika, and if they are identified as the target file format, they are stored in the respective data folders. Since this system targets PE files and JavaScript files, files of other formats are excluded at this point. Next, the processing for JavaScript files is executed, with the detection of malicious files by JAST and the deobfuscation by js-beautify and JStillery being performed in sequence. The deobfuscation of JavaScript files outputs the results of each tool, so the processing for JavaScript files outputs three files: the malicious file list by JAST and the deobfuscation results of each tool for the JavaScript files. After the processing for JavaScript files is completed, XORSearch is executed on the PE files and a report of the strings obfuscated by XOR is generated. For the deobfuscation of PE files, DIE is used to identify the packer corresponding to unipacker, and static unpacking is performed. The analysis results for each corresponding packer are output with the packer's name, and the decision to execute static unpacking is based on these results. For samples not supported by unipacker, dynamic unpacking is performed by executing mal_unpack. Since dynamic unpacking is performed on VirtualBox, the virtual environment is started, dynamic unpacking is executed after sending the sample; once the unpacking is completed, the unpacked sample is sent, and the virtual environment is stopped and restored. This process is automated by the script. In addition, when automating dynamic unpacking, it is necessary to consider cases where the sample does not run in the virtual environment. This is addressed by specifying a timeout duration with the mal_unpack option.

This system automatically performs the unpacking for PE files and the detection and deobfuscation for JavaScript files. If the sample zip file does not contain the target files, then the operations at each stage are omitted, improving the efficiency of the system. The construction environment of this system is shown below.

- Host OS: Ubuntu-22.04-amd64
  - CPU: Intel Core i3-1115G4
  - Memory: DDR4-3200 32 GB
- Docker: Ubuntu-22.04
- VBox: Windows10-1507
  - Number of CPU Processors: 2 core
  - Memory: 4 GB

VI. Results and Discussion

In this section, we verify the usefulness of the system described in Section V by inputting obfuscated samples and executing the deobfuscation system. The input to this system consists of obfuscated PE files or JavaScript files and the output includes deobfuscated PE files, deobfuscated JavaScript files, and a list of malicious JavaScript files detected. If XOR obfuscation is present, then a list of the corresponding locations is output. Additionally, PE files are evaluated using static analysis results before and after deobfuscation.

A. Verification Details and Conditions

Here, we describe the verification and its conditions. The samples used for verification are malicious JavaScript files, obfuscated JavaScript files, and obfuscated PE files. The verification for JavaScript uses JavaScript obfuscation tools such as JavaScript Obfuscator and Closure Compiler. For obfuscated PE files, we use samples obfuscated by both methods supported by unipacker and those not supported by unipacker. The applied obfuscation methods include ASPack, FSG, MEW, MPRESS, PEtite, UPX, Amber, EnigmaVB, PECompact, and Yoda Protector. Except for UPX, the unpacking libraries for these obfuscation methods are either commercial or do not exist. Additionally, for the evaluation of deobfuscated samples obtained from this system in this paper, PE files, being executable files, are evaluated based on the results of surface analysis and static analysis. In this paper, we use Capa, a capability detection tool for executable files; Stella, a tool that creates a list of readable strings by risk level of attacks; and Ghidra, a reverse engineering tool, as evaluation tools. This allows us to visualize and evaluate the changes in internal information, readable strings, and code before and after deobfuscation. The verification results of this system are described below.

B. Detection of Malicious Files for JavaScrtipt

The detection results for malicious JavaScript files are described below. Malicious and benign JavaScript files were inputted into the system and the classification of malicious files was verified. Since the detection of malicious files in this system uses the verification tool from the existing research by Fass et al., detailed verification results are omitted.

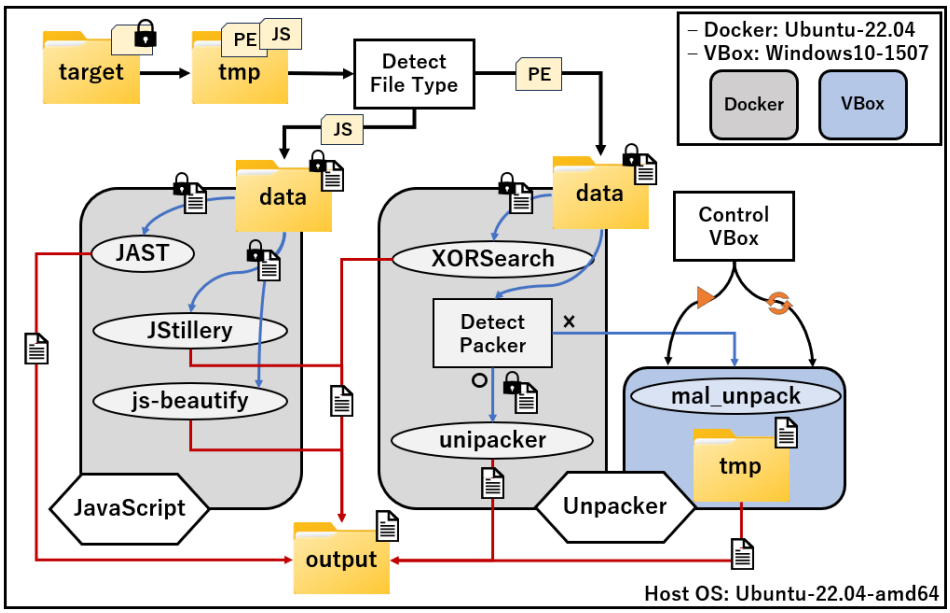

Fig. 2. System implementation.

C. Deobfuscate JavaScript Files

In this section, JavaScript obfuscation was applied to simple code using JavaScript obfuscation tools, JavaScript Obfuscator and Closure Compiler. The deobfuscation results for JavaScript are shown below, with the code before deobfuscation in 1 and the code after deobfuscation in 2:

Code 1. JavaScript Code before Deobfuscation

```
'use strict';function hello(a){alert("
    Hello, "+a)}hello("New user");
```

Code 2. JavaScript Code after Deobfuscation

```
'use strict';
function hello(a){
        alert("Hello, "+a)
}
hello("New user");
```

D. Obfuscation with XOR

Some of the information obtained by applying XORSearch in this system is as follows. The keyword “http” was used for the XORSearch verification, and samples that did not include the keyword in their readable strings were used as the target:

- http://crl.microsoft.com/XXX/MicCodSi
- http://www.microsoft.com/XXX/MicCodSigPCA_08
- http://www.microsoft.com/XXX/MicrosoftTimeSt
- http://crl.microsoft.com/XXX/microsoft

E. Static Deobfuscation of PE Files

The information obtained through the application of unipacker in this system is as follows. As mentioned earlier, we use the results of static analysis before and after deobfuscation for evaluation. The following are the analysis results obtained using the executable capability detection tool, Capa, and the surface analysis tool, Stella:

- Analysis by Capa before Static Deobfuscation
  - no capabilities found
- Analysis by Capa after Static Deobfuscation
  - no capabilities found
- Analysis by Stella before Static Deobfuscation
  - File: CompareFileTim 4UserDe
- Analysis by Stella after Static Deobfuscation
  - File: ComapareFileTime, ReadFile, WriteFile, GetPrivateProfileIntW, CreateFileA, ...

Additionally, the results of analysis using Ghidra for the program after deobfuscation and the source code before obfuscation are shown in Figure 3 and Figure 4, respectively.

F. Dynamic Deobfuscation of PE Files

The information obtained through the application of mal_unpack in this system includes the following. The evaluation method is as previously described, and the analysis results from Capa and Stella are shown below:

- Analysis by Capa before Dynamic Deobfuscation
  - no capabilities found
- Analysis by Stella before Dynamic Deobfuscation
  - Reg: null
- Analysis by Stella after Dynamic Deobfuscation
  - Reg: RegCloseKey, RegOpenKeyExA, RegCreateKeyW, RegDeleteValueW, ...

The analysis results by Capa after deobfuscation are presented in Table II. Additionally, the analysis results by Ghidra are shown in Figures 5 and 6, respectively.

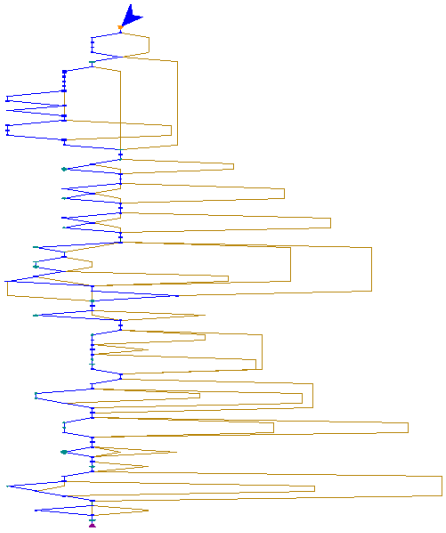

Fig. 3. Results of Ghidra's analysis of the source code before obfuscation with UPX.

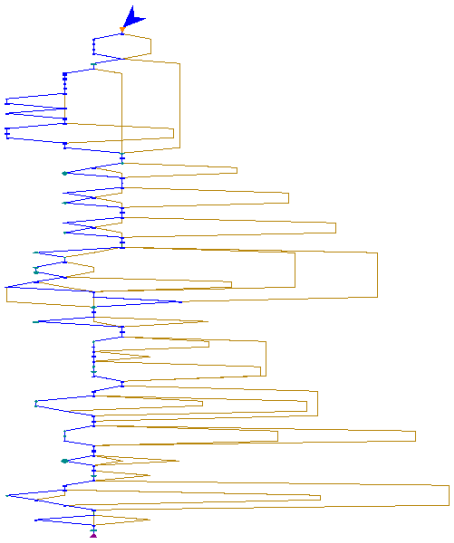

Fig. 4. Result of analysis by Ghidra after static deobfuscation.

G. Regarding the Processing Time of This System

Table III shows the total processing time for deobfuscation in this system. Here, the number of samples is varied as 5, 10, 50, and 100, and the processing time by the system is measured. The breakdown of each sample set consists of 1 JavaScript file, 2 samples supported by unipacker, and 2 samples not supported by unipacker but supported by mal_unpack. The number of samples is increased without changing the proportion of each type. The measurement interval is from the moment the script execution starts to the moment it finishes. Additionally, Table 2 shows the processing time for each tool when the number of samples is set to 50. The measurement interval here is from the moment each script starts execution to the moment the script finishes. Processing of the sample zip files and data transfer are not included in the measurement interval.

TABLE II
ANALYSIS BY CAPA AFTER DYNAMIC DEOBFUSCATION

ATT&CK Tactic	ATT&CK Technique
COLLECTION	Clipboard Data T1115 Video Capture T1125
DEFENSE EVASION	Hide Artifacts::Hidden Window T1564.003 Modify Registry T1112
DISCOVERY	Account Discovery T1087 File and Directory Discovery T1083 Query Registry T1012

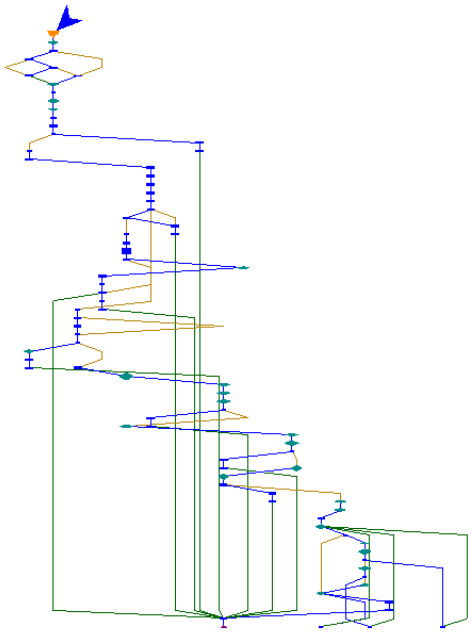

Fig. 5. Results of Ghidra's analysis of the source code before obfuscation with PECompact.

H. Discussion

Based on the verification results, we discuss the findings in this section. Regarding the detection of malicious JavaScript files, we have confirmed the classification by JAST. The JavaScript code used for the deobfuscation verification in this paper is simple. The deobfuscation tools js-beautify and JStillery only applied indentation and code formatting, as shown in Code 1 and Code 2. Additionally, deobfuscation restoring the source code from obfuscation that converts the code content to ASCII codes was not applied. Therefore, the deobfuscation of JavaScript in this system is an act of enhancing readability by inserting indentation and formatting the code.

Regarding the static deobfuscation of PE files, this system uses unipacker. For the verification of static deobfuscation, samples obfuscated with packers supported by unipacker were used. The static analysis results using Capa were not detected. This is because unipacker's static deobfuscation identified the state as still being packed, causing the tool to malfunction. In the surface analysis by Stella, comparison of the information

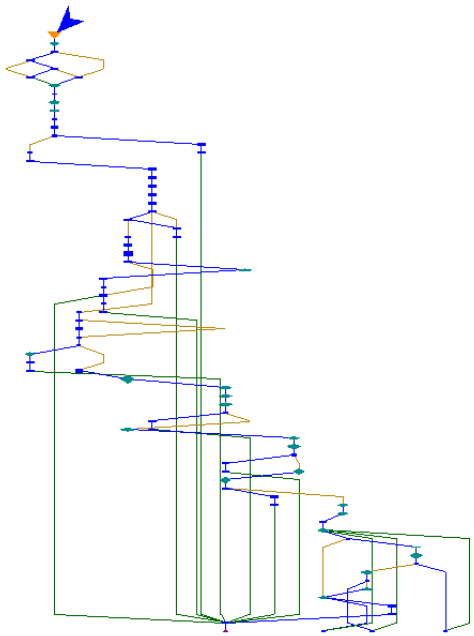

Fig. 6. Result of analysis by Ghidra after dynamic deobfuscation.

TABLE III
THE PROCESSING TIME OF THIS SYSTEM.

Number of Samples	1st Time(s)	2nd Time(s)	3rd Time(s)
5	424	416	427
10	851	851	849
50	4,240	4,285	4,241
100	8,479	8,495	8,435

obtained before and after static deobfuscation indicates that the number of readable strings increased after deobfuscation. Additionally, comparison of the main function of the program after dynamic deobfuscation and the source code before obfuscation shown in Figures 3 and 4 indicates that the code flow is equivalent. From these observations, it can be concluded that while static deobfuscation with unipacker is possible, additional processing of the sample is necessary after unipacker's deobfuscation since it is identified as being in a packed state. On the other hand, for surface analysis aimed at obtaining surface information and static analysis for reverse engineering, unipacker's static deobfuscation is effective for deobfuscation. Therefore, processing appropriate to the purpose of deobfuscation is required. Regarding the dynamic deobfuscation of PE files, this system uses mal_unpack. For the verification of dynamic deobfuscation, samples obfuscated with methods that unipacker does not support were used. The static analysis results using Capa were detected after deobfuscation, whereas they were not detected before deobfuscation. This indicates that dynamic deobfuscation can correctly identify the sample as being in an unpacked state after deobfuscation. In the surface analysis by Stella, comparison of the information obtained before and after dynamic deobfuscation clearly suggests that the number

TABLE IV
THE PROCESSING TIME OF THIS SYSTEM.

Tool	1st Time(s)	2nd Time(s)	3rd Time(s)
js-scirpts	3.492	4.167	4.389
unipacker	3,784.241	3,785.224	3,782.465
mal_unpack	451.974	495.222	452.954

of readable strings increases after deobfuscation. Additionally, comparison of the main function of the program after dynamic deobfuscation and the source code before obfuscation shown in Figures 5 and 6 indicates that the code flow is equivalent. From these observations, it can be concluded that dynamic deobfuscation with mal_unpack is possible and can achieve a level of deobfuscation equivalent to that of the source code. The processing times for deobfuscation in this system can be observed in Table III and Table IV. However, since a quantitative evaluation of each processing time cannot be conducted, these results should be regarded as merely indicative.

VII. CONCLUSION

In this paper, we proposed an automated system that encompasses the detection and deobfuscation of malicious JavaScript files, deobfuscation of XOR-obfuscated data, and both static and dynamic deobfuscation of PE files. For the detection of malicious JavaScript files, regular updates of the training data are necessary to handle unknown malicious files. Regarding the deobfuscation of JavaScript, the tools used in this system can handle deobfuscation techniques that remove indentation or convert codes to redundant forms. However, they have difficulty deobfuscating techniques that convert codes to ASCII codes. For the deobfuscation of PE files, the system achieves both static and dynamic deobfuscation. However, with static deobfuscation using unipacker, the deobfuscated samples are often still identified as being packed, necessitating further processing that does not affect static analysis.

Future challenges include the tools and processes used for the deobfuscation of JavaScript and the handling of samples after static deobfuscation using unipacker. In this study, Windows 10-1507 was used for the dynamic deobfuscation environment. However, since some malware depends on specific software or versions within PE files, constructing environments with multiple versions and software could increase the number of samples amenable to dynamic deobfuscation. Furthermore, the deobfuscation process and the quantitative evaluation of the deobfuscation results produced by various tools have not yet been conducted, making this a challenge for future work. Additionally, as malware exists not only in PE files but also in ELF files compatible with Linux, we also aim to address obfuscation techniques for ELF files.

ACKNOWLEDGMENTS

This work was supported in part by JSPS KAKENHI Grant Number 23H03396.

REFERENCES

[1] A. Herrera, "*Optimizing away javascript obfuscation*," in Proc. 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 215-220, 2020.

[2] Y. Guillot, et al., "*Automatic binary deobfuscation*," Journal in Computer Virology, vol. 6, pp. 261-276, 2010.

[3] R. T. Shirazi, et al., "*DoSE: Deobfuscation based on semantic equivalence*," in Proc. 8th Software Security, Protection and Reverse Engineering Workshop (SSPREW-8), pp. 1-12, 2018.

[4] M. A. Lachaux, et al., "*DOBF: A deobfuscation pre-training objective for programming languages*," in Proc. Advances in Neural Information Processing Systems, vol. 34, pp. 14967-14979, 2021.

[5] B. Yadegari, et al., "*A generic approach to automatic deobfuscation of executable code*," in Proc. Symposium on Security and Privacy, pp.674-691, 2015.

[6] J. H. Suk, et al., "*SCORE: Source code optimization & reconstruction*," in IEEE Access, vol. 8, pp. 129478-129496, 2020.

[7] P. Garba, et al., "*Software deobfuscation framework based on llvm*," in Proc. 3rd ACM Workshop on Software Protection (SPRO), pp. 27-38, 2019.

[8] M. Talukder, et al., "*Analysis of obfuscated code with program slicing*," in Proc. International Conference on Cyber Security and Protection of Digital Services (Cyber Security), pp. 1-7, 2019.

[9] L. Martignoni, et al., "*OmniUnpack: Fast, generic, and safe unpacking of malware*," in Proc. 23rd Annual Computer Security Applications Conference (ACSAC), pp. 431-441, 2007.

[10] Z. Tang, et al., "*SEEAD: A semantic-based approach for automatic binary code de-obfuscation*," in Proc. Trustcom/BigDataSE/ICESS, pp. 261-268, 2017.

[11] S. K. Udupa, et al., "*Deobfuscation: Reverse engineering obfuscated code*," in Proc. 12th Working Conference on Reverse Engineering (WCRE), p. 10, 2005.

[12] K. Hajarnis, et al., "*A Comprehensive solution for obfuscation detection and removal based on comparative analysis of deobfuscation tools*," in Proc. International Conference on Smart Generation Computing, Communication and Networking (SMART GENCON), pp. 1-7, 2021.

[13] M. J. Choi, et al., "*All-in-one framework for detection, unpacking, and verification for malware analysis*," Security and Communication Networks, vol. 2019, article id 5278137, pp. 1-16, 2019.

[14] G. Menguy, et al., "*Search-based local black-box deobfuscation: understand, improve and mitigate*," in Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 2513-2525, 2021.

[15] T. Blazytko, et al., "*Syntia: Synthesizing the semantics of obfuscated code*," in Proc. 26th Usenix Security Symposium (USENIX Security), pp. 643-659, 2017.

[16] D. Binkley, et al., "*ORBS: Language-independent program slicing*," in Proc. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), pp. 109-120, 2014.

[17] A. Fass, et al., "*JAST: Fully syntactic detection of malicious (Obfuscated) javascript*," in Proc. 15th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), pp. 303-325, 2018.

[18] Google, "*Google Open Source Blog*," https://opensource.googleblog.com/2024/02/magika-ai-powered-fast-and-efficient-file-type-identification.html (Accessed 2024-05-20).