DOI: 10.2507/33rd.daaam.proceedings.024

STATIC-ANALYSIS TECHNIQUES OF MALWARE REVERSE ENGINEERING

Zlatan Moric, Loïc Branstett & Robert Petrunic

This Publication has to be referred as: Moric, Z[latan]; Branstett, L[oic] & Petrunic, R[obert] (2022). Static-Analysis Techniques of Malware Reverse Engineering, Proceedings of the 33rd DAAAM International Symposium, pp.0172-0179, B. Katalinic (Ed.), Published by DAAAM International, ISBN 978-3-902734-36-5, ISSN 1726-9679, Vienna, Austria
DOI: 10.2507/33rd.daaam.proceedings.024

Abstract

Network and system security are critical issues of overall Internet security. Scientific papers and popular literature are full of new security issues being published and analysed daily. Due to the rapid proliferation of various types of malware tools that can be used both to create security attacks and to influence their results, traditional analysis methods struggle with the size and scope of samples needed to do proper analysis. For example, a well-disguised malware attack can easily penetrate an environment which is protected by a load balancer and/or Web Application Filter (WAF). In this paper, we will discuss different static-analysis techniques used to reverse engineer executable code to determine if the code is in fact malware.

Keywords: static analysis; malware; reverse engineering; artificial intelligence; dynamic analysis.

1. Introduction

Malware threats assessed by IT security organizations have been increasing in number by more than ten thousand per day. Symantec Internet Security Threat Report reveals that the total number of unique variants of malware in the world in 2011 amounts to around 403 million compared to 286 million variants in 2010. The usage of avoidance techniques such as self-defending code, packing, anti-debugging, and anti-virtualization techniques has led to advanced malware capable of passing the researchers detection barriers and anti-malware. This has increased the potential breach of corporate assets and individual computers. From the aspect of antivirus software companies and researchers, the most challenging are the threats that occur in computer applications because of the unknown vulnerability treat, also known as a zero-day attack.

These attacks take advantage of an application vulnerability unknown to anyone, so antivirus software cannot detect it either. That is why majority of the solutions used today try to go deeper and do the behavioural analysis that is the evolution of antimalware software we have the luxury of using. A reverse engineering approach can be used to detect and analyse malware by extracting data and finding out how it works on infected system. This study aims to compare different reverse engineering techniques to malware analysis. Reverse engineering has been shown to be highly successful in analysing and detecting of malware. However, it causes high complexity. Static analysis should be far more efficient and secure because the characteristics are extracted without actually executing the application [1].

2. Analysis methods

To defend against malware, analysts use different types of analysis: Static-analysis, Dynamic-analysis, and Hybrid-analysis. Each of these analysis types is unique and powerful, but the best way to use it is to combine them together. When combined, they offer more robust and more resilient solution to identify modern malware strains. Static analysis on its own is used to identify malware code by partially translating machine language into a human readable code, while the dynamic analysis tries to understand the malware behaviour while running it in the isolated environment called sandbox. Hybrid analysis is trying to bring the best from both worlds.

2.1. Static analysis

In static analysis, malware characteristics can be collected without execution of the code itself. For Portable Executable (PE) files, which are native to Windows, analysts often use tools that can present tabular views of PE header information, disassemble machine language, extract printable strings, file hashes, etc. Some features commonly investigated as part of static analysis include file hashes, strings, opcode sequences, DLL imports, API calls, and other metadata found in the PE header. In malware visualization, researchers propose to view the contents of software in raw numerical forms, such as binary, decimal, or hexadecimal, and convert these strings of numbers into images.

2.2. Dynamic analysis

In contrast to static analysis, dynamic analysis involves the execution of the malicious code. While this is much harder to scale than static analysis, monitoring how a binary interacts with an infected system can lead to more insights. Analysts are often interested in recording API and system calls, processes, modifications to system files and registries, network communication, etc.

2.3. Hybrid analysis

Both static and dynamic analysis have their limitations when conducted individually. For example, packers that compress software can be used as an obfuscation tool to obscure the contents of an executable. This will often necessitate the manual efforts of a malware analyst to conduct further static analysis on machine code. Dynamic analysis is not always successful since some types of malwares require a certain duration to pass or a specific event to trigger its execution. Hybrid analysis, the combination of both static and dynamic analysis, can lead to more comprehensive views of malicious programs and enable researchers to gather a larger set of features for classification.

3. Technology background

As contrast to the data file that needs to be processed (interpreted) by a program in order to be meaningful, an executable program, executable file, or executable piece of code permits a computer to "perform indicated tasks according to encoded instructions" [2]. Machine code tries to defend itself in a variety of ways, which makes it challenging to analyse.

3.1. Machine code and Assembly language

Machine code, also known as machine language, is the elemental language of computers. It is read by the computer's central processing unit (CPU), it is composed of digital binary numbers, and looks like a very long sequence of zeros and ones.

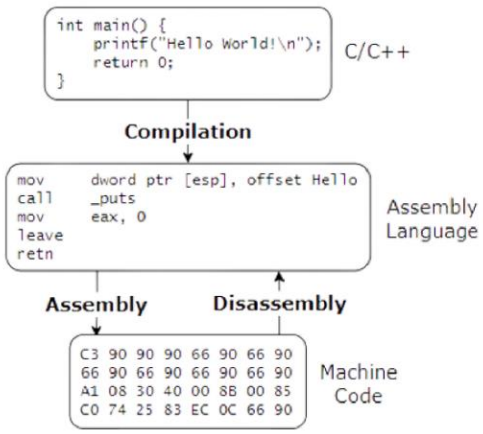

Fig. 1. Compilation and assembly, disassembly [4]

Ultimately, the source code of every human-readable programming language must be translated to machine language by a compiler or an interpreter because binary code is the only language that computer hardware can understand [3]. This transformation is generally called compilation, as it transforms source code to the assembly language understood by the CPU, this assembly language is then transformed into a series of machine instructions (aka machine code). However, static analysis generally requires a higher level of understanding of the code and uses a technique known as disassembly; the process of transforming machine code to a control flow graph of machine instructions, which can be transformed to a very low language like C.

3.2. Code obfuscation and encryption

Since executable files can be analysed by disassembly, new techniques called anti-reverse engineering have been invented to obstruct the process. Not only malware authors but also software companies use them to protect commercial software from being cracked or pirated. Code obfuscation [5] and encryption are two commonly used methods. One common technique used in code obfuscation is Control flow obfuscation. It synthesizes conditional, branching, and iterative constructs that produce valid executable logic, but yield non-deterministic semantic results when de-compiled.

In a managed executable, all strings are discoverable and readable. Even when methods and variables are renamed, strings can be used to locate critical code sections by looking for string references inside the binary. This includes messages (especially error messages) that are displayed to the user. To provide an effective barrier against this type of attack, string encryption can be used to hide strings in the executable and only restores their original value when needed.

```
Invalid Authentication - Try Again => !ù$àç_èè-'("'=):'")'
```

A more extreme version of string obfuscation is complete code encryption. Unlike obfuscation, code encryption packs and encrypts executable files on the disk. The code will be decrypted during the execution. It means that it is nearly impossible to analyse the code just by static disassembly so, we have to rely on execution and reviewing of system logs also.

3.3. Executable format

Executable formats encapsulate system-related information, such as API export and import tables, resources (icons, images, audio, etc.), and the distribution of data and code. This information is critical for malware analysis. Data and executable code are stored in different sections after the headers, depending on their functions.

3.4. Portable executable

Portable Executable (PE) [6] format is a file format for executables, object code, DLLs, and the like used in 32-bit and 64-bit versions of Windows operating systems.

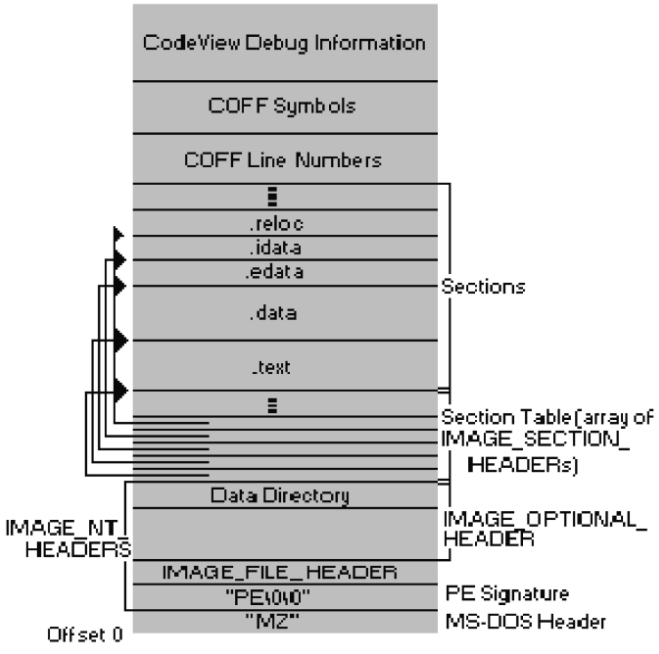

Fig. 2. The PE file format [6]

Executable file on a disk is very similar to what the module will look like after Windows loads it. The Windows loader does not need to work extremely hard to create a process based on the disk file, it just maps the necessary file segments into the virtual address space using the memory-mapped file technique.

3.5. Executable and Linkable Format

Executable and Linkable Format (ELF) format is flexible, extensible, and cross-platform file format. For instance, it supports different endianness and address sizes, so it does not exclude any particular central processing unit (CPU), or instruction set architecture. This has allowed it to be adopted by many different operating systems on many different hardware platforms.

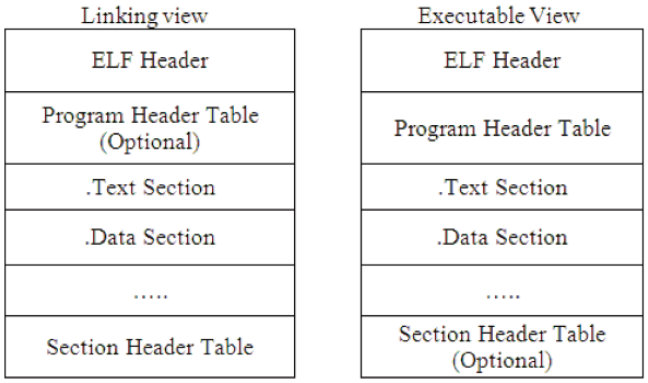

Fig. 3. Executable and linkable format (ELF) structural view [6]

Figure 3 illustrates the ELF file's structure. Every file has an ELF header at the top that describes how the file should be organized. A section header table is required, and a program header table is optional for relocatable files. A program header table and a section header table are both required in the case of an in-executable object file.

4. Analysis procedures

Analysis is the complex process that is trying to get to the conclusion of the sample behaviour, with sample being the malware analysed. Manual analysis is usually slow and heavily dependent on the analyst skills, while automated analysis allows for quick and reproducible results. There are problems with both approaches that can lead to an incomplete analysis when being done manually, or to an overseen parts of the malware code when full automation is used. That is the reason for mixed approach in the same way as explained with malware analysis types, meaning that human interaction is still required to some extent. Machine learning is closing that gap and it is used both by malware writers to hide an evolve malware, and malware analysts to identify the malicious samples and behaviour [6].

4.1. Manual analysis

At this stage, analysts reverse-engineer code using debuggers, disassemblers, compilers, and specialized tools to decode encrypted data, determine the logic behind the malware algorithm, and understand any hidden capabilities that the malware has not yet exhibited. Code reversing is a rare skill set and executing code reversals takes a great deal of time and knowledge. For these reasons, malware investigations often skip this step and therefore miss out on a lot of valuable insights into the nature of the malware.

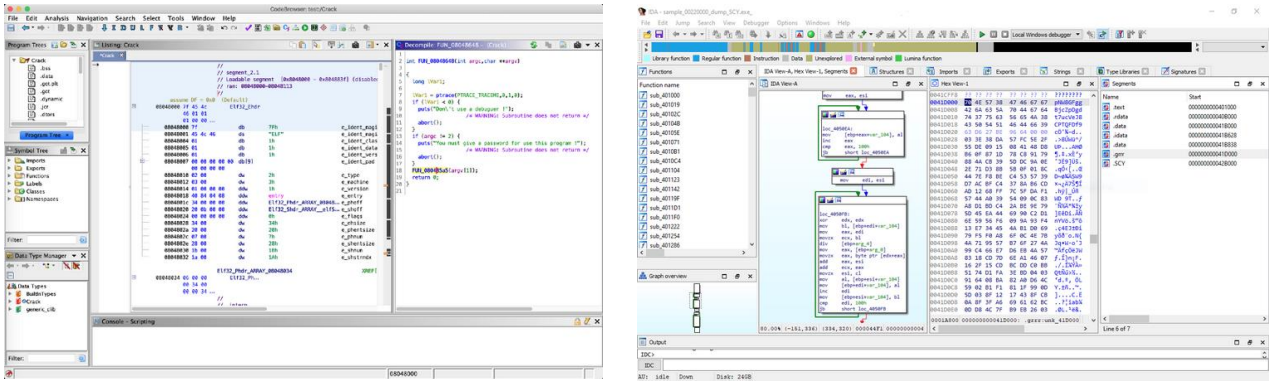

Fig. 4. Sample debugging tools (Ghidra and IDA Pro)

To manually reverse the code, malware analysis tools such as a debugger and disassembler are needed. The skills needed to complete manual code reversing is very important, but also difficult to find. Sometime forensic tools are also used to find hidden artifacts. [8]

4.2. Semi-automated analysis

Semi-automated analysis is a type of analysis that leaves only a small portion of the analysis to humans. The researcher presented an architecture that enables collaborative malware analysis by sharing resources (scripts, executable, ...) and leaving to human analysts only a small and manageable part of the whole effort. [9],

4.3. Automated analysis

The emergence of various malware tactics to get through network-based and host-based security defence's has led to an evolution in malware analysis and detection approaches during the past ten years. [10] Forensics investigators found it extremely challenging to respond promptly due to the rapidly increasing variety and number of malware species. To automate various processes of static and dynamic malware analysis, Machine Learning (ML) aided malware analysis become essential. [11]

One of the simplest ways to assess a suspicious program is to scan it with fully automated tools, which can quickly assess what malware is capable of if it infiltrated the system. This analysis can produce a detailed report regarding the network traffic, file activity, malware access and modified registry keys. Even though a fully automated analysis does not provide as much information as an analyst, it is still the fastest method to sift through large quantities of malware.

4.4. Using Equality

The "equality" technique uses information such as strings, and hashes, ... and compares them to a pre-existing database. Due to its simplicity, it is one of the mostly used techniques on a client, like the anti-malware solution. [12]

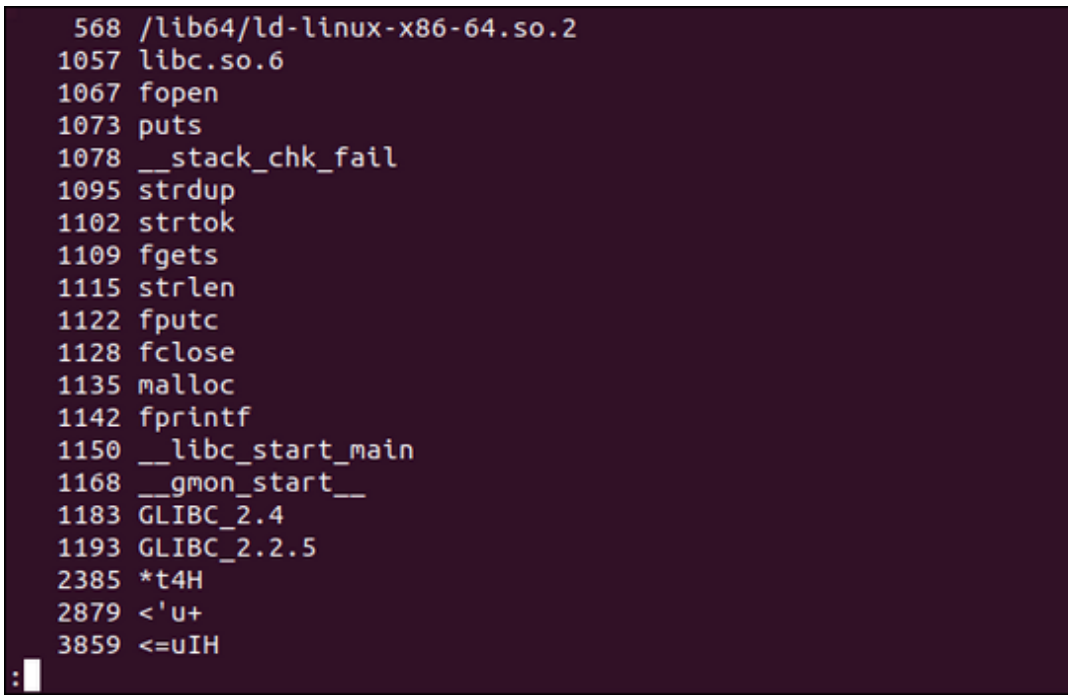

Fig. 5. Symbols of an executable

Most of the programs we have tested had triggers that were equality tests, however malware that checks value ranges may have many more trigger conditions.

4.5. Using Machine Learning

The semi-automated analysis is a type of analysis that leaves only a small portion of the analysis to humans. We can use tools like Android-COCO [13], a supervised approach for detecting Android malware.

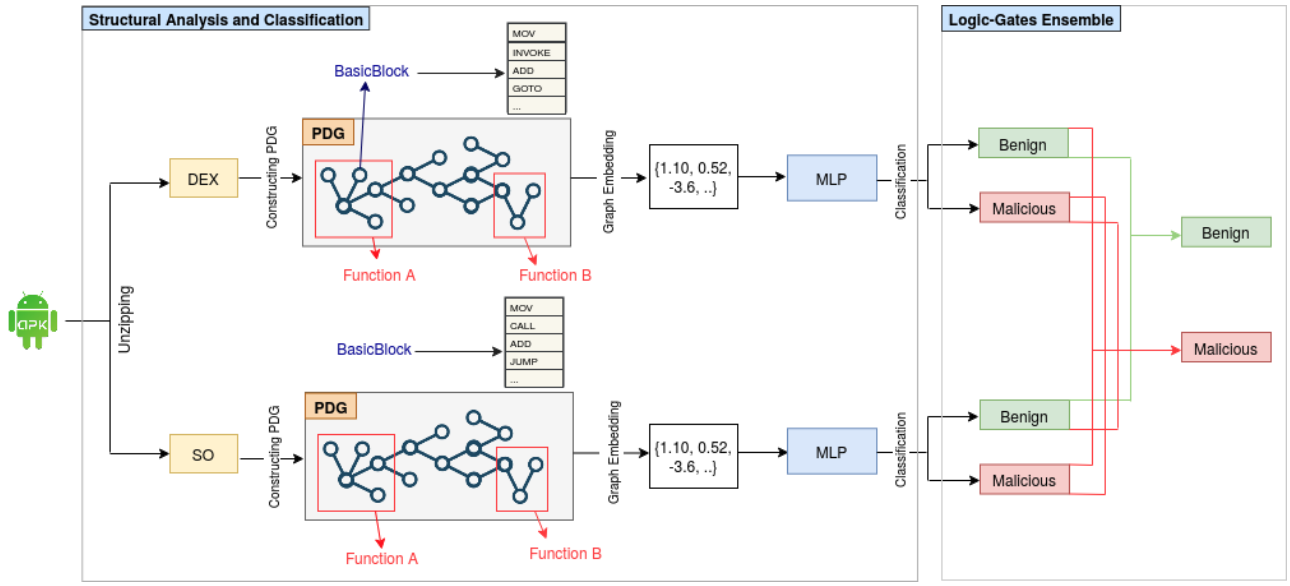

Fig. 6. Android-COCO architecture [13]

It works by automatically transforming the byte and native code to the program dependency graphs (PDG) and uses the embedding of these graphs to classify applications. Large-scale experiments on 100,113 samples (35,113 malware and 65,000 benign) shows that this approach can detect malware applications with an accuracy of 99.86%, significantly outperforming state-of-the-art solutions. [13] We can also use the N-GRAM approach [4]. This novel technique uses an N-GRAM neuronal network with different inputs:

- File Size
- API
- Op Codes
- Dependencies
- Section Size
- Section Permissions
- Content complexity

The new approach features together with a classical machine learning algorithm (Random Forest) present very good accuracy at 99.40%. Higher than the previous 4-grams than only achieve 57.96% accuracy and require a more complex model.

5. Fighting limitations of static analysis

As already mentioned, static analysis is not perfect and it requires a lot of analyst's time and effort, and especially knowledge in the debugging and assembly code analysis. The knowledge of architecture where the application is running is also one of the prerequisites the analyst is supposed to have while working on the static analysis process.

5.1. Lazy loading

Lazy initialization/loading [14] is one of those design patterns which is in use in almost all programming languages. Its goal is to move the object's construction forward in time. It is especially handy when the creation of the object is expensive, and you want to defer it as late as possible, or even skip it entirely. This implies that the object may not be precomputed in the data section like other data. Fighting this during the malware analysis is very hard and close to impossible. The only method that works reliably is to execute the TLS (Thread Local Storage) in a sand-boxed environment, but this defeats the goal of static analysis.

5.2. Name Mangling

Name mangling is a mechanism used by compilers to add additional characters to functions with the same name (function overloading). The goal of name mangling is to avoid any confusion when executing the program and calling a function that may have the same name as another one. [15]

This c++ code:

```
void add(int a, int b){cout << a + b << endl;}
```

, would result in these symbols inside the compiled library or binary:

```
$ nm a.out
[..]
00401382 T __Z3adddd
00401350 T __Z3addii
[..]
```

If name mangling were made the same across compilers, your applications would link to libraries given by other compilers but crash when run because of differences in how objects are put out, multiple inheritance is implemented, virtual function calls are handled, and other factors. Because of this, ARM urges compiler developers to differentiate their name from other compilers for the same platform. Then, rather than at run time, incompatible libraries are found at the time of linking. [16] Name mangling can be minimized. De-mangling identifiers could reveal the function's prototype. This could be used to discover special, undocumented API functions that the company would want to keep as a secret.

5.3. Jump tables

A jump table can be either an array of pointers to functions or an array of machine code jump instructions. If you have a relatively static set of functions (such as system calls or virtual functions for a class), you can create this table once and call the functions using a simple index into the array. This would mean retrieving the pointer and calling a function or jumping to the machine code depending on the type of table used. Recognizing the target addresses of conditional branches implemented via a jump table in a machine-independent manner is one of the core issues with static analysis of binary (executable) code. Without these addresses, it is impossible to fully decode the machine instructions for a specific procedure, which results in inaccurate code analysis. [17]

6. Conclusion

In this paper we have explored different techniques for malware analysis, from manually looking at the assembly code of a program to the usage of artificial intelligence to provide an analysis of a potentially malicious program. One part of the problem was identified as the inefficiency and inability to scale when the traditional methods such as static or dynamic analysis are used. This is partially solved by using the combination of these, and automation or semi-automation on top of it. Even then, the efficiency is still not at the desired level because modern malware is evolving faster than the methods and used techniques. The other part of the problem was identified as the process complexity, and the knowledge the analyst must have for the traditional malware analysis methods. This is again, only partially solved by automation and semi-automation techniques. Based on all of that, we believe that the future of malware analysis will be based on automatic analysis with some degree of machine learning embedded in both static and dynamic analysis techniques.

7. References

[1] Baskaran, B. and Ralescu, A. (2016). A study of android malware detection techniques and machine learning. In: MAICS, pp 15–23.

[2] M.-W. (n.d.), Executable Definition & Meaning - Merriam-Webster. Available from: https://www.merriam-webster.com/dictionary/executable. Accessed: 2022-10-10

[3] Kinder, J. (2010). Static Analysis of x86 Executables Statische Analyse von Programmen in x86 Maschinensprache, Ph.D. Thesis, Technische Universität, Darmstadt

[4] Chen, Z.; Brophy, E. & Ward, T. (2021). Malware Classification Using Static Disassembly and Machine Learning, The 29th Irish Conference on Artificial Intelligence and Cognitive Science 2021, Dublin

[5] Madou, M.; Van Put, L.& De Bosschere, K. (2006). Understanding obfuscated code, IEEE Int. Conf. Progr. Compr., vol. 2006, pp. 268–271

[6] Pietrek, M. (1994). Peering Inside the PE: A Tour of the Win32 Portable Executable File Format, Microsoft Syst. Journal-US Ed., vol. 9, no. 3, pp. 15–38

[7] Ucci, D.; Aniello L.; Baldoni R. (2019). Survey of machine learning techniques for malware amalysis, ScienceDirect, vol. 81, pp. 123–147

8] Moric, Z.; Redzepagic, J. & Gatti, F. (2021). Enterprise Tools for Data Forensics, Ann. DAAAM Proc. Int. DAAAM Symp., vol. 32, no. 1, pp. 98–105, Wiena

[9] Laurenza, G.; Ucci, D.; Aniello, L. & Baldoni, R. (2016). An Architecture for Semi-Automatic Collaborative Malware Analysis for CIs, Proc. - 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Networks, DSN-W 2016, pp. 137–142

[10] Svoboda, J. & Lukas, L. (2019). Sources of Threats and Threats in the Cyber Security, DAAAM Int. Sci. Book, pp. 321–330, Wiena

[11] Shalaginov, A.; Banin, S.; Dehghantanha, A. & Franke, K. (2018). Machine learning aided static malware analysis: A survey and tutorial, Cyber threat intelligence, vol. 70, pp. 7–45, Springer
[12] Sharif, M.; Lanzi, A.; Giffin, J. & Lee, W. (2008). Impeding Malware Analysis Using Conditional Code Obfuscation, Proc. 15th Netw. Distrib. Syst. Secur. Symp. - NDSS, pp. 321–333, Citeseer
[13] Xu, P. & Khairi, A. E. (2021). Android-COCO: Android Malware Detection with Graph Neural Network for Byte- and Native-Code, arXiv:2112.10038v2
[14] Park, H. & Kim, C.(2019). Design of Adaptive Web and Lazy Loading Components for Web Application Development, Journal of the Korea Institute of Information and Communication Engineering, vol. 23, no. 5, pp. 516–522, The Korea Institute of Information and Commucation Engineering
[15] Kefallonitis, F. (2007). Name mangling demystified, pp. 1–21, Accessed: 2022-10-10
[16] Ellis, M. A. & Stroustrup, B. (1990). The Annotated {C++} Reference Manual, Addison-Wesley Longman Publishing
[17] Cifuentes, C. & Van Emmerik, M. (2001). Recovery of jump table case statements from binary code, Science of Computer Programming, vol. 40, no. 2–3, pp. 171–188, Elsevier