AN OVERVIEW OF MALWARE ANALYSIS METHODS FROM THE PERSPECTIVE OF COMPUTER SYSTEM ARCHITECTURE

Toshikatsu Hosokawa

GWID: G49220573

CSCI6461_10

Professor Morris Lancaster

March 15, 2015

1 Introduction

Until early 2000s, most malware were created for fun. A typical motivation for malware writers to create malware was to demonstrate their skill. Therefore, in order to attract many people’s interest, those malware typically had the functionality to spread widely and quickly in the network [1]. Those malware sometimes caused a huge disruption of the network. However, since an anti-virus vender could quickly obtain the information about those malware and upgrade their products, those malware could be removed rather easily in a timely manner [1].

Nowadays, the monetary profit which malware can bring has become one of the main purposes for creating malware [1]. Stolen data which are collected by malware, such as email and credit card accounts, are traded in the underground black-market. In addition, infected computers which can be remotely controlled are available for rent in order to perform malicious activities, such as sending spam emails and conducting Distributed Denial Service (DDos) attacks [2].

This shift of the motivation for malware creation also caused the change in the behavior and characteristics of malware. Today, the persistency of malware is crucial for malware writers in order to obtain as much monetary profit as possible from their malware. Therefore, those malware have the functionality to hide their existence in order to prevent being detected and deleted by virus scanners. As a result, it has become difficult to detect malware and create effective countermeasures for them. Furthermore, in the underground black-market, tools for creating custom-made malware are sold as well, which make it possible to create new malware without programing skills. Consequently, the number of malware reported to anti-virus venders is increasing. In order to deal with this serious situation on malware threats, it is essential to develop efficient and effective methods to analyze malware.

Malware writers implement a variety of techniques in order to create sophisticated malware that can accomplish their malicious intent effectively. For example, some malware can intercept invocations of Windows API and system calls and filter the result in order to hide its existence. Some malware exploits the rich instruction set that is available on some computer architectures in order to change its binary representation by replacing instructions [3]. Furthermore, a recent study suggests that parallel computing using Graphics Processing Unit (GPU) is possibly exploited by malware writers to create more sophisticated malware [4].

On the other hand, analysts use their knowledge of computer system architecture in order to develop countermeasures for malware. For example, in order to decode the binary executables which are obfuscated by replacing instructions, analysts should understand the Instruction Set Architecture (ISA), and in order to analyze encrypted (packed) malware, some analysis tool exploits the memory management system. In addition, a recent study shows that GPU computing can accelerate analysis process of malware [5].

As seen above, the arm-race between analysts and malware writers can be seen as a competition to utilize the knowledge of computer system architecture. Therefore, malware analysis is an interesting topic from the perspective of computer system architecture. Thus, this paper intends to overview the evolution of methods of malware analysis and to describe how the knowledge of computer system architecture is used to develop the analysis methods and create sophisticated malware. In addition, this paper intends to introduce the recent studies on the possible impact caused by General-purpose computing on GPU (GPGPU) regarding malware analysis or creation.

The rest of this paper is organized as the following: Section 2 provides a brief explanation of types of malware. Section 3 overviews the basics of malware analysis methods. Section 4 elaborates a variety of techniques which malware writers implement on their malware in order to evade the analysis and/or detection. Section 5 describes countermeasures for the techniques which were mentioned in the previous section. Section 6 introduces recent studies regarding possible applications of GPGPU for malware analysis and creation.

2 Types of malware

Malware is defined as “software that fulfills the deliberately harmful intent of an attacker” [6]. Malware can be categorized into several types according to their characteristics. Those types are worm, virus, Trojan horse, spyware, bot and rootkit. A brief description of them is as the following [2, 3]:

Worm: A worm is a malicious program which replicates itself and spreads to the network. Some worm does not do any harmful actions other than spreading to the network. However, even this action caused various troubles, such as saturating the capacity of the network and exhausting the memory of the infected computer.

Virus: A virus is a malicious program which attaches itself on other program. When the user executes the program infected by a virus, the virus is activated and starts malicious actions, such as deleting user data, encrypting user data and downloading spyware.

Trojan horse: A Trojan horse is a malicious program which disguises itself as a useful program, and after the user installs the program, it conducts various malicious actions in the background such as capturing the user’s passwords.

Spyware*:* A spyware is a program which collects user’s data without user’s authorization. Those data could be personal and confidential, such as a history of visited web pages and information of bank accounts.

Bot: A bot is a malicious program which provides the functionality to remotely control the infected computer to the bot master. Infected computers could compose a network (botnet) and the network is used for malicious actions, such as sending spam emails and launching Distributed Denial Service (DDos) attacks.

Rootkit*:* A rootkit is a program which can be used for hiding certain processes or programs. Rootkits can be used benevolent purposes such as hiding analysis environment from malware. However, many malware use rootkit techniques to hide their existence.

3 Malware analysis methods

Today, virus scanners mainly depend on the signatures (a set of the pattern of binary to identify a specific malware). Typically, the signatures are created manually by analysts. Prior to create a signature for a certain binary executable, analysts should analyze the sample's behavior and determine whether the sample is malware or not.

The signature-based malware detection is not effective against malware which has the functionality to transform itself, such as polymorphic malware and metamorphic malware, because after the transformation, the signature cannot match for the malware. Hence, techniques which can detect malware independently from the signatures are also required. In order to develop such techniques, it is essential to analyze the behavior and characteristics of malware.

Generally speaking, there are two types of analysis. These are static analysis and dynamic analysis. The following subsection provides a brief description of those analysis methods.

3.1 Static analysis

Static analysis is a method to analyze malware without its execution. In this methods, analysts read the binary representation or the source code of malware in order to analyze the behavior of malware. In many cases, the source code of malware is not available for analysts, therefore static analysis is conducted by using the binary representation of malware. Typical analysis flow is as the following [7]:

Disassembling the binary executable of the malware

Creating a Control Flow Graph (CFG)

Identifying the malicious behaviors of the malware based on the assembly code and the CFG

Since static analysis does not depend on the actual behavior which the malware shows during the run-time, the analysis possibly examines a broad behavior of the malware. However, several techniques to thwart static analysis exist, such as encrypting the binary executable and inserting irrelevant instructions into the binary executable. Recently, malware writers tend to implement those techniques on their malware. Although some improved static analysis techniques have been proposed in order to analyze binary executables which implement the anti-analysis techniques [3], in most cases, it is difficult to examine the behavior of such malware alone with static analysis.

3.2 Dynamic analysis

Dynamic analysis is a method which analyzes malware while it is executed. Typically, the samples are executed under a limited environment, such as emulators or virtual machines in order to prevent negative affect on the real system. Dynamic analysis can be performed manually with assistance of tools, such as debuggers and system monitor programs provided by operating systems (e.g., Windows Task Manager). However, in order to address the increase of malware newly reported, automated dynamic analysis techniques are essential. In typical automated dynamic analysis systems, firstly, the malware sample is executed on the emulator or the virtual machine. Secondly, the system monitors and records its activities that are relevant to identify the behavior of the malware sample. Such activities possibly contain creating and modifying files, creating new processes, changing the Windows registry, memory accesses and opening the network connections. This monitoring can be performed by using techniques such as function call monitoring which is mentioned below. Lastly, after this logging process, the system generates the report of the malware's behavior.

Major techniques of dynamic analysis are as the following [7]:

Function Call Monitoring: Through monitoring the functions which the malware sample calls, analysts can obtain the information regarding the malware's behavior. Among those functions calls, monitoring the invocation of Application Programing Interface (API) and system calls provides valuable information for analysts. For example, in order to open the connection to the Internet, malware which runs in user-mode must use API, and in order to create/modified files, the malware must use system calls. Typically, injecting hooking functions to the malware executable enables the function call monitoring.

Function Parameter Analysis: The parameters which are passed to the functions provide valuable information along with monitoring the function calls. For example, if the value returned by *CreateFile* system call is passed to *WriteFile* system call as a parameter. This most likely means that the malware sample created a new file and wrote some data on the file.

Dynamic analysis is effective to analyze malware that uses obfuscation techniques and/or encryption. For example, since the malware samples must decrypt their selves at the run-time, the encryption cannot thwart dynamic analysis. In addition, there are variety of tools which automatically conduct dynamic analysis and generate the reports. However, the crucial drawback of dynamic analysis is that it cannot examine the behaviors of malware thoroughly. Some malware only shows their malicious behavior under a particular condition, such as the date of its execution, existing certain files, receiving a certain command from the Internet. In that case, dynamic analysis can detect only a part of malicious features or cannot detect them at all.

4 Anti-analysis techniques

As described in Section 3, static analysis and dynamic analysis are effective techniques to identify malware’s behavior and characteristics. However, today, it is not unusual that malware writers implement techniques to prevent those analysis. Those techniques contain obfuscation, encryption, analysis environment detection and logic bombs [7]. Some of these techniques are not necessarily used just for evading the analysis. For example, some malware transforms its binary representation by encryption and/or obfuscation in order to hinder the signature-based detection conducted by virus scanners. As mentioned in Section 1, the persistency has become an important factor for malware. Therefore, malware writers try to improve the persistency of their malware through implementing these techniques. The following subsection describes common anti-analysis techniques.

4.1 Obfuscation

Obfuscation is a technique which prevents analysis through reducing readability of the binary executable on purpose. The technique is also used in order to prevent been detected by virus scanners, because the signature of the malware changes through the obfuscation process. The common obfuscation techniques are as the following [8]:

Dead-code insertion: Dead-code insertion is an obfuscation technique which inserts instructions to the binary executable in a way the insertion does not affect its functionality (e.g., inserting nop instructions).

Register reassignment: Register reassignment is an obfuscation technique which changes the assignment of registers of the binary executable in a way its functionality remains same.

Instruction substitution: Through replacing instructions of the binary executable by other instructions which have same functionality, malware writers can make the malware's structure difficult to understand for analysts.

Code transposition: Code transposition is an obfuscation technique which reorders the instructions of the binary executable without changing its functionality.

4.2 Encryption

Through encrypting the binary executable, malware writers can thwart static analysis, because the encrypted binary cannot be disassembly correctly. Such malware has a decryption routine besides the encrypted code. When it is executed, the decryption routine reconstructs the program and copies it into the memory. Therefore, the decrypted version of the malware only exists during its run-time. Today it is usual to use an executable compression program (Packer) for the encryption. With Packer, malware writers easily create the encrypted malware. On the other hand, dynamic analysis is applicable to such binary executables, since it is not affected by the binary representation before the run-time.

4.3 Polymorphism and metamorphism

Polymorphism and metamorphism are techniques to transform malware. Those techniques make use of obfuscation and/or encryption. Polymorphic malware and metamorphic malware are the malware which implement those techniques. Those malware are typically difficult to analyze by static analysis. In addition those malware can evade the signature-based detection effectively. Their characteristics are as the following [7]:

Polymorphic malware: Polymorphic malware encrypts itself by a different key each time it copies itself. Therefore, the encrypted portion of the malware is always different. In addition, the malware transforms its decryption routine. However, once it is executed, the decrypted code of the malware is identical for all variants.

Metamorphic malware: Metamorphic malware, differs from polymorphic malware, transforms the whole code by using sophisticated obfuscation techniques. Thus, it is more difficult for virus scanners to detect this type of malware.

4.4 Analysis environment detection

Some malware has the functionality which detects the existence of analysis environment. Those malware can hide their malicious behavior when they are executed under such environment. This technique can prevent dynamic analysis, because even though the binary is executed, no malicious behavior can be observed. Malware could use the following techniques to detect the analysis environment [7]:

Monitoring loaded processes: Malware can obtain the list of the loaded processes by using API calls. If the list contains the processes of analysis environment. Malware could cease its malicious activities.

Checking code integrity: In order to monitoring API calls and system calls invoked by the malware, dynamic analysis systems typically modify the malware by injecting hooking functions before executing the malware. Therefore, the malware can detect the analysis environment through checking its integrity. For example, if the malware stores the hash value of its own, since a hash value computed from the modified malware differs from the original hash value, the malware can easily tell the breach of the integrity.

Monitoring machine–specific registers: In order to tracing instructions of the malware, dynamic analysis systems might set the trap flag in x86 EFLAGS register. Therefore, the malware could detect the analysis environment through checking the value of the register by using system calls.

4.5 Logic bombs

Similar to analysis environment detection, some malware shows malicious behavior only under a certain condition, such as the date of its execution, existing certain files, existing the Internet connection [7]. For example, Michelangelo virus shows its malicious behavior only on March 6th (the birthday of Michelangelo) [6]. This techniques can prevent dynamic analysis, because if the condition does not hold, malware will not show any malicious activities.

5 Counter measures

Section 4 elaborated a variety of methods which malware writers implement on their malware in order to prevent the analysis and/or detection. This section introduces countermeasures for those techniques.

5.1 Semantic-Aware malware detection

As mentioned in Section 4.1, static analysis become difficult when the malware is obfuscated. In addition, the signature-based detection is not effective for malware that transforms itself by obfuscation, because the detection methods heavily rely on the syntactic representation of malware.

Chirstodorescu *et al*. [3] proposed a method for malware detection which are resilient to common obfuscation techniques. Their method is possibly enable to detect and/or analyze obfuscated malware statically. This section explains their methods briefly, and then discusses the limit of their method. A brief summary of the method is as the following [3]:

The methods is used templates in order to detect malicious behavior of malware instead of its signatures. The templates represent certain activities which many malware have in common, such as self-decryption and sending spam emails. The templates are abstracted so that they are not sensitive to the syntactic representation of malware. Figure 1 illustrates the concept of the templates. Figure 1(b) is an example of an instructions sequence of malware, as seen in Figure 1(a), the information such as memory address and the assignment of registers are abstracted in the template. Figure 1(c) shows the correspondence of the abstracted information between Figure

1(a) and Figure 1(b). According to their paper, this template-based approach shows resilience to common obfuscation techniques.

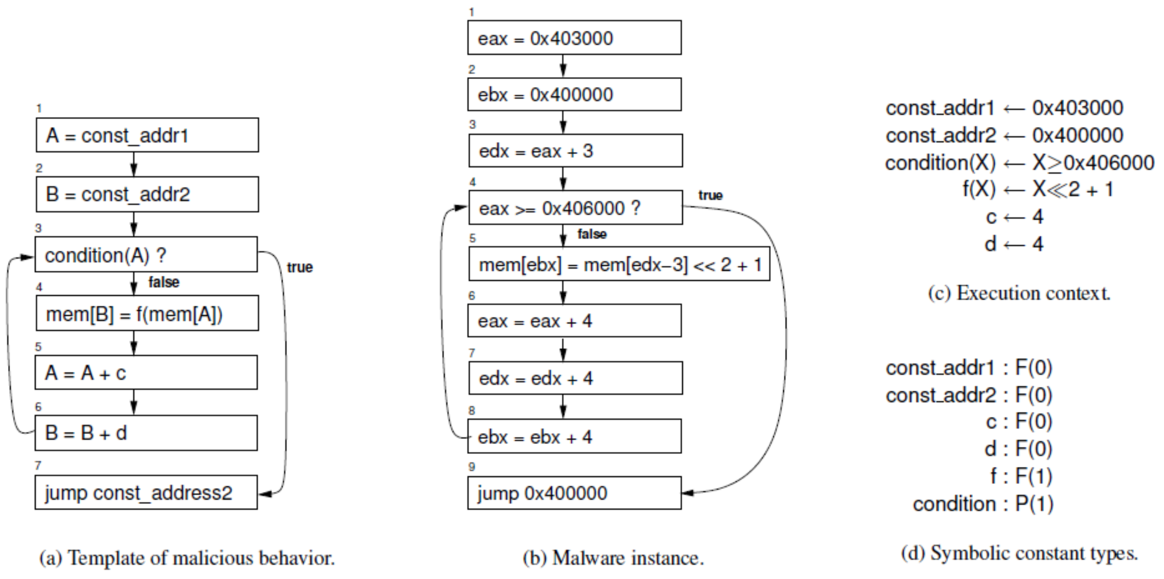

Figure 1. Example of template [3]

As seen in the summary above, the template-based approach possibly enable to detect obfuscated malware. However, Moser *et al*. in [9] introduced more sophisticated obfuscation techniques which can deter this detection method, and they argued that theoretically there are a limit for static analysis. Furthermore, in order to create the templates, the method relies on disassembling techniques, but obfuscated and/or encrypted malware is, in many cases, difficult to disassemble correctly. Therefore, alone with static approach, it is difficult to analyze and/or detect sophisticated malware.

5.2 Unpacking

As mentioned in Section 4.2, today, it is usual that malware writers encrypt (pack) their malware by using Packer. In order to conduct static analysis, it is essential for analyst to decrypt (unpack) the packed malware. If the malware is packed by the Packer commonly used, it is not difficult to unpack the packed executable. However, if the malware writers use an original packing routine, it is difficult to unpack the malware. There are some systems which enable to unpack such executables are presented. This subsection provides a brief explanation of such systems, using Renovo [10] and Saffron [11] as examples.

Most of the unpacking systems make use of the fact that the packed executable necessarily writes its unpacked code in the memory and moves the instruction pointer to the unpacked code. Figure 2 shows how a packed executable works [10]. When the packed executable is executed, firstly the executable is loaded in the memory, secondly the executable starts its unpacking routine and writes the unpacked code in the memory and finally moves the instruction pointer to the unpacked code [10]. Therefore, the memory regions which is indicated as black rectangles in Figure 2 hold the unpacked code of the malware.

Renovo [10] is one of the systems which provides a generic unpacking functionality. Renovo is implemented on an emulator engine. In order to identify the memory regions which hold the unpacked code, Renovo uses the shadow memory which is a copy of the actual memory. Renovo can monitor the malware’s memory access by using the shadow memory.

On the other hand, Saffron [11] uses different approach to monitor the malware’s memory access. Figure 3 shows the format of Page Table Entry (PTE), Saffron sets User/Supervisor flag of PTE so that user-mode programs cannot access the memory. Therefore, when the malware under analysis tries to write the unpacked code, a page-fault occurs, and by hooking the page-fault handler, Saffron can identify the memory regions that hold the unpacked code.

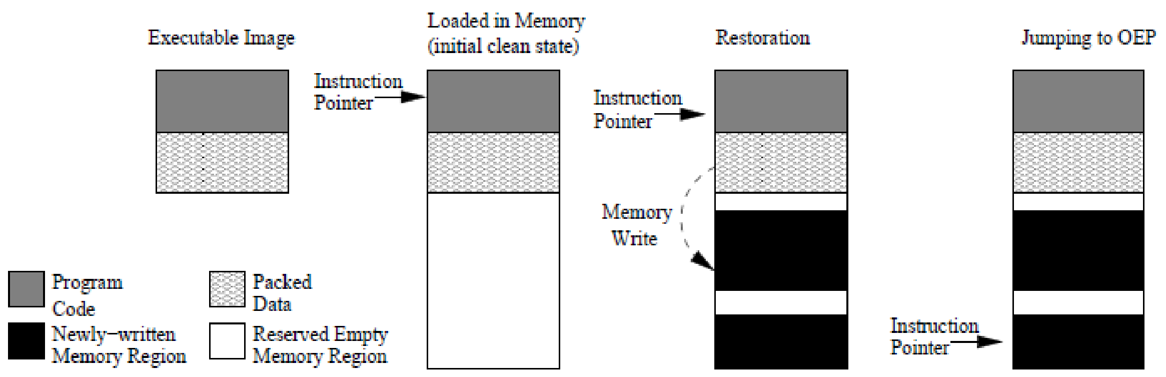

Figure 2. Behavior of packed executable [10]

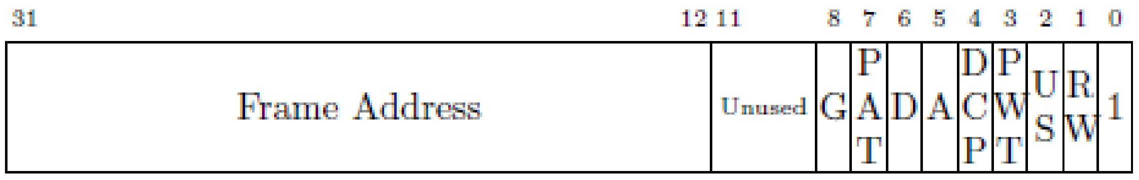

Figure 3. Format of Page Table Entry [12]

As mentioned above, there are effective systems to unpack the packed malware. However, most of the systems cannot unpack multi-packed malware so far [7]. Today multi-packed malware is not unusual. Hence, it is required to improve the unpacking systems.

5.3 Hiding analysis environment

Malware possibly detects the existence of the analysis environment by using a variety of techniques mentioned in Section 4.4. This functionality of malware is a great threat for dynamic analysis. This section introduces two major techniques which conceal the analysis environment from malware. The first technique uses rootkits to hide the analysis processes, and the second one uses a shadow memory and modifies PTE to circumvent the integrity check of malware. A brief explanation of these techniques is as the following [7]:

Concealing analysis processes: As mentioned in Section 4.4, malware can use API to check the existence of analysis processes. However, through using rootkit techniques, the analysis processes can be concealed from the malware. A rootkit is a program which can hook API calls and filter the result. For example, when the malware calls the API which returns the list of all processes currently running, the analysis system hooks the API call and sanitizes the result so that the analysis processes cannot be observable from the malware.

Circumventing the integrity check: As mentioned in Section 4.4, malware can check its integrity by using its hash value. In order to circumvent the integrity check, some analysis system holds the unmodified image of the malware in a shadow memory. In addition, the analysis system sets “not present” flag of the PTE which corresponds to the memory region of the malware (the flag is the least significant bit of PTE, see Figure 3). When the malware tries to read the memory region, a page-fault occurs, and the analysis system hooks the page-fault handler and returns the unmodified image of malware from the shadow memory. The Figure 4 illustrates this flow.

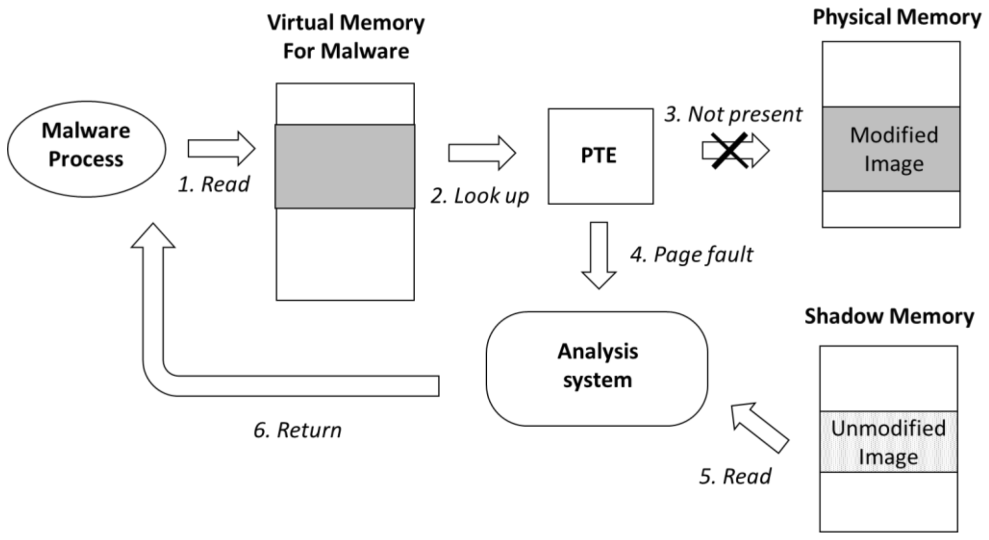

Figure 4. Circumventing the integrity check (created in reference to [7])

As mentioned above, some effective techniques to prevent malware from detecting the analysis environment are presented. However, malware possibly uses other information for the detection. For example, malware could monitor the user action, since automated dynamic analysis systems most likely behave differently compared with the computers operated by human users [7]. Therefore it is required that improving existing analysis system in order to address a variety of the techniques which malware writers could implement.

5.4 Multi-path exploration

Moser *et al*. [6] proposed an effective countermeasure for logic bombs. In typical scheme of logic bombs, the malware acquires data from outside, such as the current date or the user input, by using API or system calls. The malware uses the data for its control-flow decisions. The tool presented by Moser *et al*. makes use of this characteristic of logic bombs.

At first, the tool executes the malware on its emulator, and monitors the behavior of the malware during a certain period of time. During the monitoring, the tool takes snapshots for all branching points (the points which the malware changes its behavior based on the data from outside). After a timeout, the system restarts from one of the branching points. This tool can rewrite the data used for branching so that the malware takes the other path at the branching point. Repeating this until all paths are taken, the tool can analyze the behavior of the malware thoroughly.

6 Malware and GPU

In the previous sections, this paper provided a comprehensive overview of the evolution of malware analysis and creation. Finally, this section introduces the possible impact of General-purpose computing on GPU (GPGPU) as a new trend regarding malware. This section explains the concept of GPGPU and modern GPU architecture as a prerequisite. After that, this section introduces the possible GPU-assisted malware [4] and an acceleration technique of malware analysis using GPUs [5]

6.1 GPGPU

Historically speaking, GPUs were utilized for solely graphic computing. Therefore, old GPUs had limited functionality and programmability. However, according to the expansion of the video game industry, GPUs have extended their computational power and functionality [4]. For example, in 2007 *GeForce 8800 Ultra* had only 128 cores (shaders), on the other hand, in 2014 *GeForce GTX Titan Z* has 5760 cores [13]. In addition, the core can be used for general-purpose computing as with CPU.

A distinctive difference regarding the architecture between CPU and GPU are the number of its cores. Today CPUs typically have 2 – 8 cores, in contrast, modern GPUs have thousands of cores. Therefore, GPUs are good at conducting parallel computing. Today the platform for utilizing GPUs for general-purpose computing is provided, such as Compute Unified Device Architecture (CUDA). Therefore, it is possible to write programs which use the massive GPU’s computational power. Consequently, GPGPU has gained much attention due to its possibility of broad applications.

6.2 GPU and CUDA architecture

CUDA is a platform for GPGPU provided by NVIDIA. As a prerequisite of the following subsection, this subsection presents a brief description of GPUs (NVIDIA's models) and CUDA architecture. Figure 5 illustrates a general architecture of GPUs. GPUs have several Stream Multiprocessors (SMs), and each SM is composed of several Stream Processors (SPs). Each SM perform instructions in a SIMD (Single Instruction Multiple Data) manner [14]. The number of SM and SP varies according to types of GPU. As mentioned in the previous subsection, modern GPUs have thousands of SPs.

In CUDA programing model, a function executed on the GPU is called a kernel. The kernel is divided into threads (a unit of function calculated on each SP) in order to perform computations in a parallel manner [14]. CUDA provides users with a variety of APIs which can use GPUs' computational functions. Therefore, today people can write programs which utilize the GPU's massive computational powers.

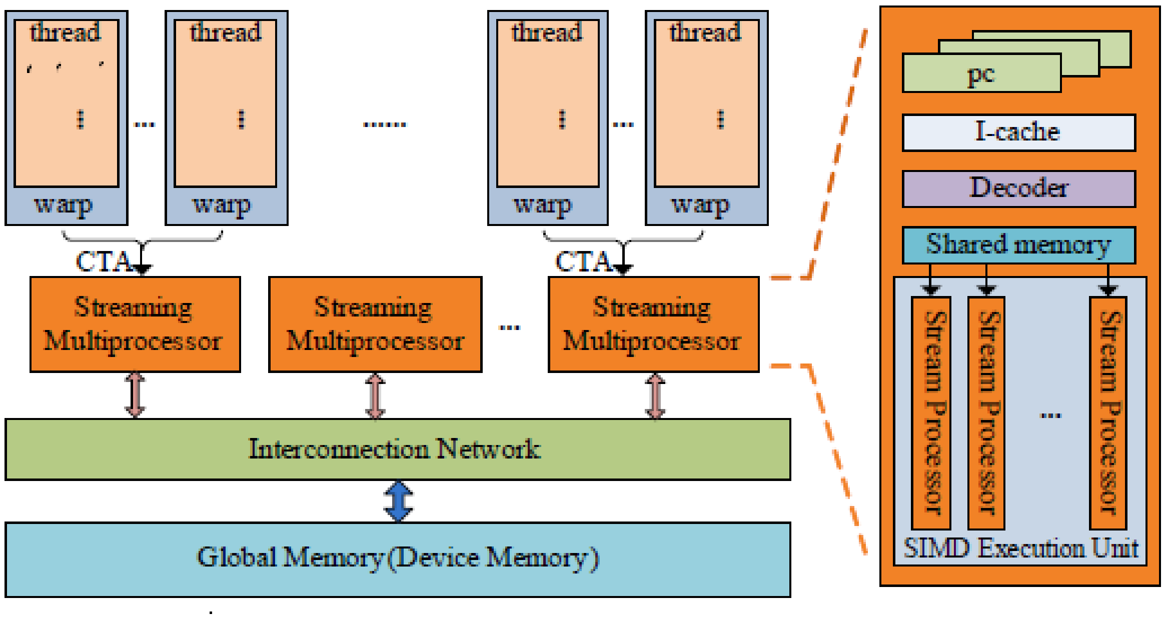

Figure 5. GPU architecture [14]

6.3 Threats of GPU-assisted malware

GPGPU enables people to create programs which utilize the GPU's computational power. It is possible that malware writers try to write malicious code for GPUs [4]. Vasiliadis *et al*. in [4] warned the possible threats of malware which utilizes GPUs for improving their capability to evade the analysis or detection. According to [4], there are two possible GPU's applications which malware writers could attempt. These are self-unpacking malware and run-time polymorphic malware. The brief description of these malware is as the following [4]:

Self-unpacking malware: As mentioned in Section 4.2, to pack malware is one of the most common techniques to evade the analysis and detection. In order to address the problem of packed malware. Some virus scanners attempts to unpack the malware by unpacking routine of the virus scanner. After finishing unpacking, the virus scanner perform its signature-based detection routine. So far, virus scanners use CPU-based unpacking routines. However, malware writers could implement an unpacking routine which requires the GPU's parallel computing power. If so, the virus scanner's CPU-based unpacking routine most likely cannot unpack the malware because of lack of computational power. This can nullify the virus scanner's detection functionality.

Furthermore, the malware which has a GPU-based unpacking routine can evade the existing dynamic analysis tools. For example, as mentioned in Section 5.2, Renovo [10] have the functionality to retrieve the unpacked code through executing the malware on its emulator. However, Renovo currently supports only Instruction Set Architecture (ISA) of CPUs. Thus, it is not applicable to the malware. In addition, most of the existing dynamic analysis tools rely on the single-step execution, therefore, inherently cannot handle the paralleled execution of GPUs.

Run-time polymorphic malware: Even though the GPU-based unpacking routine is complicated, in the end, the malware have to unpack itself and copy its unpacked code in the memory. Therefore, even if existing dynamic analysis tools or virus scanners cannot unpack the malware, analysts can retrieve the unpacked code through executing the malware on the real system. However, a run-time polymorphic malware can effectively evade such analysis or detection. The malware unpacks its code only partly at a time. In addition, it re-packs the unpacked code after the code is executed. As a result, only a small part of the unpacked malicious code exists in the host's memory at a time. This makes the analysis or detection more difficult.

Run-time polymorphic malware is itself not a novel type of malware. However, through utilizing GPU, malware writers possibly create more robust run-time polymorphic malware. For example, if the malware stores its decryption key not in the main memory but in the GPU's memory, the analysis or detection will become more difficult.

As seen above, the existing analysis or detection techniques have been built for the IA-32 architecture. As a result, those techniques cannot handle the malware which utilizes the GPUs architecture so far. As Vasiliadis *et al*. suggested in [4], it is needed to improve the existing techniques or developing new techniques in order to address these upcoming threats.

6.3 Accelerating malware detection by using GPU

The previous subsection elaborated the possible threats which GPGPU would bring. On the other hand, this subsection introduces a recent study regarding a possible application of GPU computing for malware analysis and detection. Gupta *et al*. showed that by utilizing GPU computing, an existing packer detection system (SPADE) can be improved [5]. Prior to explain their study, as a prerequisite, the following part provides a brief description of SPADE [15]:

SPADE: SPADE is a signature-based Packer detection system provided by Naval *et al*. The system can identify the Packer by which the malware is packed with good accuracy. In order to generate the signature for Packers, SPADE needs a training set which is composed of many malware samples packed by various Packers. After inputted the samples, SPADE searches the similar pattern among them and generate the signatures.

As mentioned in the brief description above, SPADE is a powerful system which can address the threats of packed malware. However, since SPADE uses an algorithm which requires expensive computational cost in order to generate the signatures, the long computation time was its major drawback [5]. In order to solve this drawback, Gupta *et al*. proposed an improved SPADE (P-SPADE) which utilizes GPU computing [5]. According to their experiment, the computation time of SPADE has been improved by a factor of 49.91 at most.

7 Conclusion

The malware analysis methods can be divided into two methods, these are static analysis and dynamic analysis. Although both methods have advantages and disadvantages, considering the increasing number of malware newly reported to anti-virus venders, automated dynamic analysis systems are effective. As mentioned in Section 4, malware writers have implemented a variety of techniques in order to prevent the analysis and detection. Especially, packed malware poses a great threat for static analysis, and logic bombs possibly nullify dynamic analysis. Although a variety of countermeasures for sophisticated malware have been presented as mentioned in

Section 5, either static analysis or dynamic analysis cannot complete the analysis alone. Therefore, analysts have to use both methods according to the type of the malware under analysis.

Furthermore, a recent study suggests that the prevalence of GPGPU might bring new threats, that is, the possible advent of GPU-assisted malware [4]. Most existing malware analysis and detection systems are made for the IA-32 architecture. Considering the fact that malware writers have been implementing new techniques to improve the robustness of their malware. It is plausible that they create malware which exploits the GPUs architecture. Therefore, in order to address such new threats, to improve the existing analysis systems or to develop new analysis systems are required.

8 References

[1] Ross J. Anderson. 2008. *Security Engineering: A Guide to Building Dependable Distributed Systems* (2 ed.). Wiley Publishing.

[2] C. Wueest, “Underground black market: Thriving trade in stolen data, malware, and attack services” http://www.symantec.com/connect/blogs/underground-black-market-thriving-trade-stolen-data-malware-and-attack-services, Dec. 2014.

[3] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. 2008. A semantics-based approach to malware detection. *ACM Trans. Program. Lang. Syst.* 30, 5, Article 25 (September 2008), 54 pages.

[4] G. Vasiliadis, M. Polychronakis, S. Ioannidis, "GPU-assisted malware," *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on* , vol., no., pp.1,6, 19-20 Oct. 2010.

[5] N. Gupta, S. Naval, V. Laxmi, M.S. Gaur and M. Rajarajan, "P-SPADE: GPU accelerated malware packer detection," *Privacy, Security and Trust (PST), 2014 Twelfth Annual International Conference on* , vol., no., pp.257,263, 23-24 July 2014.

[6] A. Moser, C. Kruegel and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," *Security and Privacy, 2007. SP '07. IEEE Symposium on*, vol., no., pp.231,245, 20-23 May 2007.

[7] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2008. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.* 44, 2, Article 6 (March 2008), 42 pages.

[8] Ilsun You and Kangbin Yim, "Malware Obfuscation Techniques: A Brief Survey," *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on* , vol., no., pp.297,300, 4-6 Nov. 2010.

[9] A. Moser, C. Kruegel, E. Kirda, "Limits of Static Analysis for Malware Detection," *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, vol., no., pp.421,430, 10-14 Dec. 2007.

[10] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. 2007. Renovo: a hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode* (WORM '07). ACM, New York, NY, USA, 46-53.

[11] D. Quist and V. Smith. "Covert debugging circumventing software armoring techniques." *Black hat briefings USA* (2007).

[12] C. Merimovich, “IA-32 Paging” http://www2.mta.ac.il/~carmi/Teaching/OS/xv6/IA32-Paging.pdf, n.d.

[13] C. DeFanti, K. Yuh and B. Yuan “CS179: GPU Programming Lecture 1: Introduction”

http://courses.cms.caltech.edu/cs101gpu/2014/lectures/179lec1.pdf, 2014.

[14] Z. Yu, L. Eeckhout, N. Goswami, T. Li, L. John, H. Jin, C. Xu and J. Wu, "GPGPU-MiniBench: Accelerating GPGPU Micro-Architecture Simulation," *Computers, IEEE Transactions on* , vol.PP, no.99, pp.1,1.

[15] Smita Naval, Vijay Laxmi, M. S. Gaur, and P. Vinod. 2012. SPADE: Signature based PAcker DEtection. In *Proceedings of the First International Conference on Security of Internet of Things* (SecurIT '12). ACM, New York, NY, USA, 96-101.