Journal of Theoretical and Applied Information Technology
31 August 2012. Vol. 42 No.2

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

A NEW GENERIC TAXONOMY OF MALWARE BEHAVIOURAL DETECTION AND REMOVAL TECHNIQUES

[1]LEE LING CHUAN, [2]MAHAMOD ISMAIL, [3]CHAN LEE YEE, [4]KASMIRAN JUMARI
[1,3]PhD Student, Department of Electrical, Electronic and System Engineering, National University of Malaysia, Malaysia
[2,4]Professor, Department of Electrical, Electronic and System Engineering, National University of Malaysia, Malaysia
E-mail: [1]lclee_vx@f13-labs.net , [2]mahamod@eng.ukm.my , [3]chanleeyee@f13-labs.net , [4]kbj@eng.ukm.my

ABSTRACT

Modern malware has become a major threat to today's Internet communications. The threat can infiltrate hosts using a variety of methods, such as attacks against known software vulnerabilities, hidden functionality in regular programs, drive-by download from unsafe web sites, and so forth. Matching a file stream against a known virus pattern is a fundamental technique for detecting viruses. With the popularity and variety of malware attack over the Internet, computer virus protection companies need to constantly update new virus signatures in their virus definition databases. However, the increasing size of the signature database can only detect known virus but cannot defend against new variants of malware. In this paper, we present an overview of the detection of modern malware focuses on suspect behavioural patterns. Contrary to classical heuristic engines which focus on the detection of encrypted malware samples, we integrate a known packer detector as well as unpacking routines to circumvent the protection techniques used by most of the modern malware. We believe that many obfuscated techniques used by malware authors are available on the Internet. More precisely, the use of known packer removals would strip out the packer protection with our dedicated decryption routines. Our apprehensive program is based on the integration of both static heuristic and emulator approaches; however, they do not necessarily have to serve as a complement for each other. Static heuristic scanner involves static extraction, which is relying on byte signature to identify a dedicated viral signature. Emulator can execute the arbitrary code from the instance and would trace the instance's body code in a virtual environment. It can be used to combat any protection code, regardless of the complexity of the protection algorithm. Fragments of virus body could be detected while the execution is in a decrypted virus body. Lastly, we present experimental results that indicate our proposed technique can provide good performance against obfuscated malware. Through this study, we hope to help security researchers understand our defence approach and give some directions for future research.

Keywords: *Static Analysis, Dynamic Analysis, Heuristic, Emulator, Malware*

1. INTRODUCTION

Malicious software is a generic term to denote any unwanted software designed to perform an unauthorized process that will have adverse impact on the availability, integrity or confidentiality of a system. Over the past decades, the battle between defensive and offensive in the world of virology has never ceased. Many avenues of research has been done with regards to the manner of detecting computer virus, yet the use of signature is still the most common detection method today. Modern malware detection uses different data extraction method from the malware body including patterns with or without wildcards, checksums, behaviour patterns, file geometry, and statistic distribution of code instructions [1]. In an attempt to defeat detection engines, malware authors have evolved the infection, replication and spreading of mechanism codes. The malicious program is devised over every possible way to evade the detection engine. Such techniques include encryption, obfuscation, packing, entry point obscuring and more [9]. In the early days, encryption scheme is a common key to protect the innards of an instance's malicious executable.

Later, encrypted virus evolved from simple encryption to more sophisticated self-defence mechanisms to give all the creations a better survival rate. A virus has the ability to modify its viral code and alter its appearance at each infection. Among all the techniques, polymorphism [11] and metamorphism [10] are certainly the most advanced defence mechanisms for malware.

Polymorphic viruses are designed to conceal their potential signatures by obfuscating the entire code. It mutates or changes their appearance by generating multiple unique encryption methods to encrypt the virus body. A number of distinct decryptors are generated which allow the viruses to change their decryptor code from generation to generation. As opposed to polymorphism, metamorphic viruses change their internal structure but all are functionally equivalent in each generation. Both techniques help in changing the virus signatures to avoid signature based detection. Consequently, form-based detection relies on signature which is no longer reliable in detecting the well-design of both polymorphic and metamorphic malware. In spite of the fact that different obfuscation techniques have been used to protect the malware instance's innards, most packer algorithms are available from the Internet; for instance, Ultimate Packer for eXecutables (UPX) [12], ASPack [13], WWWPACK [14] and so forth. Paradoxically, many malware that appear today are repacked version with common packers but effectively evades from the detection of Antivirus software [25].

The current trend in the anti-malware community is to devise the next generation of viral code detectors over semantic aspects [24]. The motivation of this work is to develop a standalone heuristic engine for detection of obfuscated and new variants of malware. In fact, the peculiarity of the majority of virus that appears today is the repacked version of old malware. Our approach is to subvert the protection mechanisms and bypass the defences of malware. We propose a combination of known packer removal and heuristic malware scanning engine, both statically and dynamically for analysing the creation's structure, its behaviour and other attributes. The approach of this known packer removal module is based on a pre-defined packer signature. The idea is to acquaint the packer of a malicious program; an automatic component will then extract the obfuscated part and invoke the scanning engine against the real malware body. The identification and extraction of packer feature would accelerate the scanning process though it requires a human expert and time to forge a reliable signature and extract the program. Our scanning engine can fall into two categories: static and dynamic. The primary difference between the two categories is that the static heuristic technique does not execute the code being analysed. The scanner will study the suspicious program in a hexadecimal format and compare it to the code of known viruses and virus-like activities. If the code matches the code of known viruses or virus-like activities, the file is flagged and the user alerted. However, on the condition that no viral-like activity is detected, an emulator technique will copy parts of an application's program code into a safe emulation buffer and emulate the execution. If any suspicious actions are detected during the "execution", the object will flag as malicious.

In summary, this paper is to demonstrate the ability to develop competitive heuristic scanning for malicious codes at a much lower cost. Towards this end, we make several contributions. We proposed a design of malware signature database that accelerates the process of malware detection. The database uses multiple parts of malware pattern to be matched in sequence for virus detection. The proposed method can reduce the overall size of database and accelerate the pattern matching process. Instead of using the full text signature, malware patterns are partially selected for the matching process. We then proposed a combination of a known packer detector and removal module with both static heuristic and emulator module. The packer detector is devised based on a signature approach to automate the process of identifying and extracting the hidden code of packed executable files. The proposed method can accelerate the implementation of malware detection process and reduce the size of malware signature database. In fact, a single malware signature is capable in detecting a whole variant set of a virus family. Finally, we propose a design of the next generation of malware detector over static heuristic and emulator engine corresponding to a future threat that most malware detection software must deal with. The primary goal of the proposed method is to deal with obfuscated and new variants of malware. The design of automatically executing arbitrary program is in a safe and isolated environment. A region of malicious code is identified by tracing an instance's executable program dynamically based on a basic block approach. Our design relies on disassembling the analysis code dynamically and performing just-in-time compilation [26] targeted for the host CPU.

This article is articulated according to the following structure. Section 2 describes related work. A brief overview of scanning engine is presented in section 3. Then, section 4 introduces the design of malware signature database where the design and implementation detail of our proposed system will be discussed. Both static and dynamic system architecture will be discussed in this section. Section 5 provides an experimental evaluation of its effectiveness. Finally, section 6 briefly concludes and outlines future work.

2. RELATED WORK

This section briefly reviews the background and related work to this project. Although virus and malware detection has been studied for years, many modern malware programs are still able to evade the existing malware detectors [27].

Obfuscation is a common method that transforms the true purpose of original program code into a misleading or unreadable form, in hopes of hiding the program's true intentions. According to a report, more than 92% of malware files are runtime packed [2]. In particular, the obfuscation malware is the very first problem that a malware analysis should be addressed. If an obfuscated malware instance cannot be unpacked, the analysis of the program will only view the obfuscated block as non-instruction data. There are systems which perform automated unpacking processes for program executable files using different tactics. Renovo [3] uses a dynamic approach to monitor the execution of given program in an emulated environment. The run-time execution and memory writes are tracked in such a manner as to determine that the execution in the memory region is newly generated; it will then extract the executable program in the memory region. The approach of PolyUnpack software [4] to automatically extract the original hidden code is based on the observation of sequence instructions of packed executable. It disassemblies the binary instructions and executes the instructions until a code section is reached.

Malware detection can occur before, or after the malicious code is loaded into the memory. Thus, the detection approach can be categorized into static and dynamic strategies. Sung presents a robust malware detection technique using API. The method is called Static Analyzer of Vicious Executables (SAVE) [5], with emphasizes on detecting metamorphic and polymorphic malware. The approach of SAVE in detecting malware is based on the sequence of API calls. The detection decision is made based on the comparison of this sequence to a known malware sequence. Clam AntiVirus [6] also known as ClamAV, is an open source and cross-platform antivirus software designed for detecting malicious threats. It provides file format detection, packer unpacking support, and multiple signatures for detecting viral code. The signature can be divided into three types: basic patterns, regular expression and MD5 checksums [7]. The virus signature is updated very frequently and as of October 11, 2011, contained 1,044,387 virus signatures.

3. HEURISTIC ENGINE DESCRIPTION

The modern scourge of malicious program is greatly exacerbated by the implementation of effective protections. These protections frequently use obfuscation techniques, such as packing malware programs with software armouring or packers to generate several variants of malware programs. Due to this tendency, the number of virus signatures will increase very fast, thus requiring higher computational resource consumption. The idea of identifying known packer signature at the entry point of every scanning target file can accelerate the scanning process and reduce the size of virus signature database.

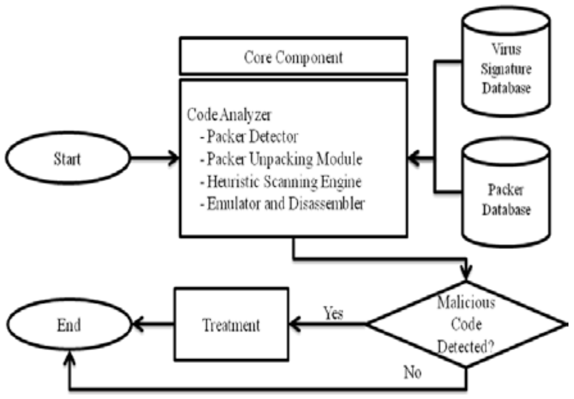


Figure 1. Malware Scanner Architecture

Our approach of malware and virus scanner architecture, shown in Figure 1, is based on the observation of a code analyser. It is divided into known packer detector, packer unpacking module and heuristic scanning engine. Obfuscated code or packer [8] is a technique commonly used to hinder malware code analysis via reverse engineering. As more and more new malwares are packed or encrypted, they mutate themselves as they spread around so that no two copies will share the same virus signature. To prevent any obfuscated code from posing obstacles to the scanning and detection module, an automated process for identifying and extracting the hidden-code bodies is proposed. We devised an algorithm to identify if a program applies any obfuscation mechanism. Packer detector function is built on top of scanning core components and it is developed to analyse a malware instance file, and determines if any packer has been applied. Our approach begins by detecting any packer applied for the instance malware files based on the packer signature detection at entry point. The entry point is the first instruction that the pointer is pointed to, which is intended as a destination of a long jump. A module for automating the process of extracting the hidden-code by using the known decryption algorithm and obtaining the original-code bodies of the program is executed if any known packer is detected.

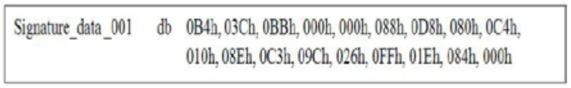
The idea of our malware scanner is to ensure the engine locates any viral code statically and dynamically. The static heuristic technique is devised to analyse a program's code statically without actually executing it. This method typically relies on our virus database which scans for viral codes by searching of predetermined malware patterns. The process begins at the program's entry point. Unfortunately, the analysis process will become difficult if any binary is intentionally designed to thwart the code analysis of static analysis approach. Thus, the ability of the emulator to execute a program code has the significant advantage to combat with this malware's protection.

As mentioned earlier, it is common for malware writers to use code obfuscation techniques to hinder the actual viral code. The problem can be solved by using our known packer remover function. In some cases, our dedicated decryption routines are unable to find the known decryption algorithm to decrypt the program; thus, the design of emulation solves the problem. In contrast to static technique, emulator analyses an executable's inner code during run-time in a controlled environment. Every protected malware code should eventually be decrypted and executed in memory, regardless of the sophistication of the obfuscated algorithm or multiple encryption layers that have been implemented. Emulator works by attempting to execute the binary in our emulated environment and eventually virus could be detected after the virus body has been decrypted.

4. THE DESIGN OF MALWARE SIGNATURE DATABASE

Heuristic scanning is a malware analysis process that looks for "viral-like" activity. Such activities include overwriting or moving benign program's entry point in memory, attempts to infect and evade detections by writing viral code to system files, modifying interrupts vectors, and so forth. Unlike traditional signature detection, the verification of either benign or malicious of an executable is based on behavioural signature but not simple byte patterns. Behavioural signature is a program with distinct syntaxes that have identical malware behaviour capture signature. With the design of malware behaviour signature, the ability of detection is no longer a single piece of malware program but a whole class of malware coming from a common strain.

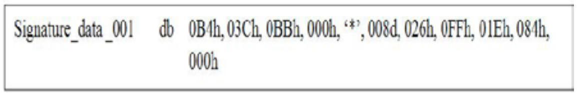
There are two methods for recognizing various program behaviours. One is by maintaining a large database of byte sequences of signatures. Figure 2 shows the pattern signature that we have defined. Here, the signature is compared to full and the malware instance is flagged as infection if the entire text of "viral-like" pattern is exactly matched.



```
Signature_data_001    db    0B4h, 03Ch, 0BBh, 000h, 000h, 088h, 0D8h, 080h, 0C4h,
                            010h, 08Eh, 0C3h, 09Ch, 026h, 0FFh, 01Eh, 084h, 000h
```

Figure 2. Full Pattern of Malware Signature

The second method is capable of optimizing the computer performance by comparing malware instance's code with arbitrary fragments of the "viral-like" patterns; thus each fragment is separated by an arbitrary wildcard ('*'). Wildcard strings make it possible to skip bytes and to employ regular expressions.

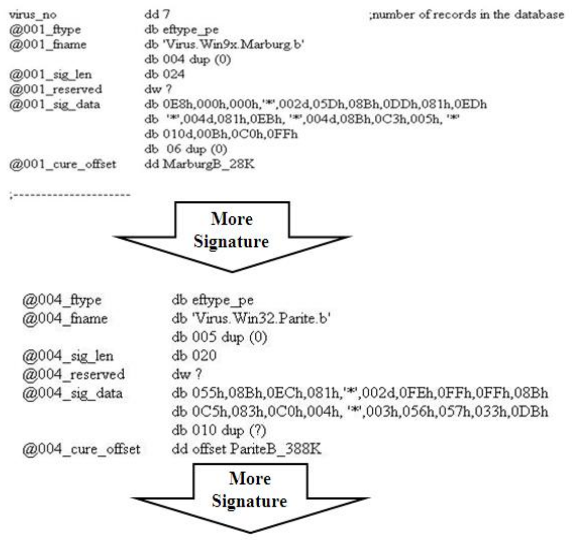


```
Signature_data_001    db    0B4h, 03Ch, 0BBh, 000h, '*', 008d, 026h, 0FFh, 01Eh, 084h,
                            000h
```

Figure 3. Fragmented Pattern of Malware Signature

The signature shown in Figure 3 represents the same virus sample as the earlier pattern signature in Figure 2. The difference between both of them is the pattern signature of Figure 3 which uses wild card regular expression to divide the signature resulting in two segments. The intention is to reduce the amount of states needed to be tracked. As shown in the example, upon a hit of 0B4h, 03Ch, 0BBh, 000h for a malware instance reported; the appearance of 026h, 0FFh, 01Eh, 084h, 000h is only possible after the skipping of 8 bytes distance from the first segment. The value of byte after (*) wild card is arbitrary; it indicates distance in bytes between two segments.

Our approach of malware and virus scanner detection engine comprises the scanning engine module and the malware signature database. Both of the modules work together and are inseparable. Generally, the design of our signature database is very volatile. The main goal of this volatility is to ensure that new signatures can be updated in the future.



```
virus_no            dd 7                                   ;number of records in the database
@001_ftype          db eftype_pe
@001_fname          db 'Virus.Win9x.Marburg.b'
                    db 004 dup (0)
@001_sig_len        db 024
@001_reserved       dw ?
@001_sig_data       db 0E8h,000h,000h,'*',002d,05Dh,08Bh,0DDh,081h,0EDh
                    db  '*',004d,081h,0EBh, '*',004d,08Bh,0C3h,005h, '*'
                    db 010d,00Bh,0C0h,0FFh
                    db  06 dup (0)
@001_cure_offset    dd MarburgB_28K

;--------------------
```

More
Signature

```
@004_ftype          db eftype_pe
@004_fname          db 'Virus.Win32.Parite.b'
                    db 005 dup (0)
@004_sig_len        db 020
@004_reserved       dw ?
@004_sig_data       db 055h,08Bh,0ECh,081h,'*',002d,0FEh,0FFh,0FFh,08Bh
                    db 0C5h,083h,0C0h,004h, '*',003h,056h,057h,033h,0DBh
                    db 010 dup (?)
@004_cure_offset    dd offset PariteB_388K
```

Figure 4. Virus Signature in Malware Signature Database

Figure 4 depicts a sample of virus signatures in our malware signature database. The database used seven entities to store the **virus_no**, **ftype**, **fname**, **sig_len**, **reserved**, **sig_data**, and **cure_offset**. As shown in Figure 4, besides **virus_no**, the rest of the entity will be defined with a series of prefix numbering identification. **virus_no** is the entity that displays the total number of virus signatures inside the database. Currently, only 6 malware signatures have been generated and more signatures will be added in the future. Consider the fourth group of virus signature, **@004_ftype** and **@004_fname**, both represent the type of executables file and name of the malware instance, respectively. **eftype_pe** represents PE file format. **@004_sig_len** specifies the total length of malware signature. In addition, malware signatures were stored in the most efficient Opcode data type (**@004_sig_data**) rather than human readable format (for example assembly language). Our approach of **@004_cure_offset** will trigger the scanner to PariteB_388K cure function if the infection of Virus.Win32.Parite.B was detected. **@004_reserved** takes no action and is reserved for future usage.

5. SYSTEM ARCHITECTURE

System architecture, described with more details in Figure 5, uses the combination of known packer removal, static heuristic and emulator for detecting malware. The scanner is initialized by reading the information about scanning path directory and determining the total number of executable files that needs to be scanned. The *Information Collective* observes the intended actions of a program including file type, file system, file size and most importantly is determining the entry point of each executable file. The information will then be inherent to the static heuristic and emulator scanner function.

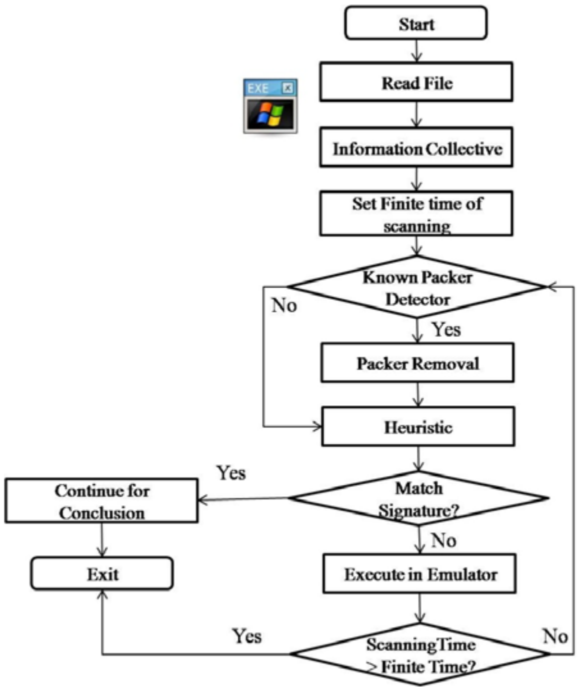


Figure 5. Flowchart describing the overall system architecture

Both static heuristic and emulator are devised to analyse any given executable file in a finite time, during which it must conclude that the program is benign or not. The maximum time limit for the scanning engine is important, a particular malware might not be detected if the allowed time is too short, which is diametrically opposed to the longer finite time that will deteriorate the average speed of the emulator. The idea of maximum allowed scanning time is to prevent our scanning resources from exhaustion, and also to avoid the scan to remain in an infinite loop while analysing a file.

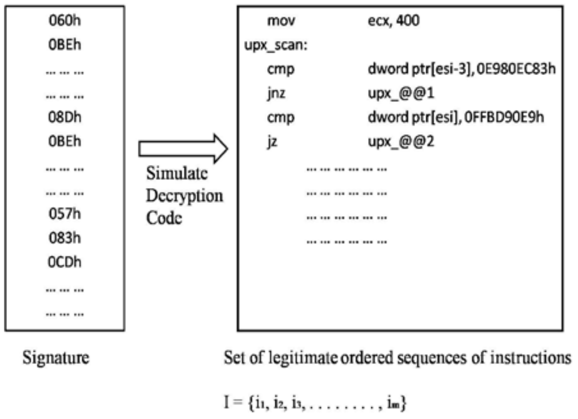
As mentioned earlier, most obfuscated techniques used by malware authors are from known packers. *Emulator scanner* is capable of unpacking obfuscated executables in memory by executing the instance code in the virtual buffer. One drawback to the manner is that code simulation might be too slow if the decryption loop is complex. Particularly, when the malware instance uses common packer to

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

obscure the malicious portion, code emulation may be too slow to decrypt the decryption loop to its core instruction set. The approach of known packer removal can accelerate the scanning process by detecting and removing any known packer which begins at the common entry-point and reveals the real intention of malicious code instead of consuming computer time and performance to emulate and decrypt garbage instructions. Static heuristic scanner is devised based on an analysis of file format and instance code fragment by comparing to virus "pattern". The word "pattern" refers to the hexadecimal string in a virus signature.

5.1 SYSTEM ARCHITECTURE

Known packer unpacking works by attempting to detect and simulate the decryptor to decrypt known obfuscation packer used by malware authors within our pre-defined virtual buffer. A fragment of obfuscation code is used to be part of the signature to detect if a binary contains packed-code. When the obfuscation code is detected, the known decryption algorithm is executed to deobfuscate the decryption code from the executable itself within our virtual environment.These dedicated decryption routine approaches provide better performance compared to the classical emulator technique that execute every instruction in memory. Unfortunately, our signature database that detects packer is very limited and needs to be updated in the future for it to detect packed binary instances.



```
060h
0BEh
... ... ...
... ... ...
08Dh
0BEh
... ... ...
... ... ...
057h
083h
0CDh
... ... ...
... ... ...
```

```
    mov         ecx, 400
upx_scan:
    cmp         dword ptr[esi-3],0E980EC83h
    jnz         upx_@@1
    cmp         dword ptr[esi],0FFBD90E9h
    jz          upx_@@2
                ... ... ... ... ... ...
                ... ... ... ... ... ...
                ... ... ... ... ... ...
                ... ... ... ... ... ...
```

$I = \{i_1, i_2, i_3, \ldots\ldots\ldots, i_m\}$

Figure 6. On the left, example opcode signature of UPX packer. On the right, a subsequence of order instruction im comprises partially of the unpacking function of UPX packer.

The idea of an approach to automate the unpacking process is to identify a composition of sets of ordered sequences of instructions that is able to extract the hidden-code bodies of an instance malware. Figure 6 shows the overall process of a known packer unpacking function. Let the tuple *I*= *{i1, i2, i3, . . . , im}* be a set of ordered sequences to unpack a dedicated pack. As the executable instance is paused at an instance's entry point, our scanner uses a signature database to determine if an executable file contains packed-code. If a match is detected, a set of order sequences of unpack instructions, *im* is executed in our virtual environment to clarify the context of an executable file.

5.2 STATIC HEURISTIC SCANNING

The right side of Figure 7 shows the overall flow of static heuristic scanning architecture. As illustrated in the figure, the first component performs *information collective* where the intended actions of Windows binary file can be observed by the implementation of PE parse. The PE parse transforms the Portable Executable (PE) [15] binary file and collects the scanning instruction and required information including target file permission, path information and file extension. The information collective flow consists of 5 steps as shown on the left side of Figure 7. The initialization function begins by displaying the virus detection toolkit information and its scanning file option. The *command line arguments* function will call GetCommandLine Win32 API function [16] to collect and receive the instruction of scanning option and determine the instruction action. The *prepare drive path* and *search for file* modules call GetCurrentDirectory [17], FindFirstFile [18], and FindNextFile [19] Win32 API functions to collect information about the scanning target. This includes scanning path, filename and total number of files. Lastly, *process file* will identify the target's file size, file permissions and file type or file extension. The next heuristic scanning component embeds the matching algorithm used to compare the executable file to the malicious behaviour signatures. The executable file either labelled as benign or malicious is dependent on the result.

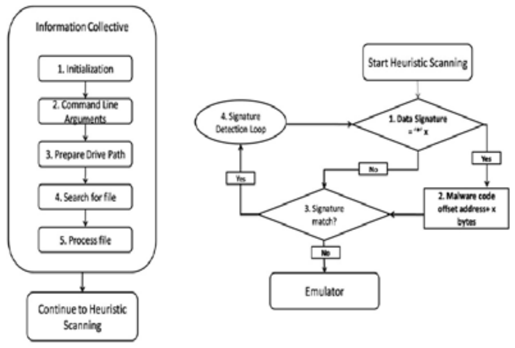


Figure 7. Flowchart describing the overall process of static heuristic scanning

Heuristic scanning methodology performs in a manner that uses search and discover operation to look for certain instructions or commands within a program that is not found in typical application programs. To reduce memory consuming due to a large number of states, a simple displacement gap pattern is supported in our heuristic scanning module. It is a full pattern or rule which consists of a sequence of one or more segments separated by a gap with arbitrary bytes of length. Like common class of string matching methods, the scanning method tracks a finite automaton constructed from the set of patterns. The tracking reads only one character in the text per iteration. As shown in Figure 7, the operation of heuristic scanning will jump into a scanning loop until scanning process is completed. The operation proceeds according to the following steps:

Step 1, *Data Signature='*'x*. The process begins by detection of character asterisk (*) wild cards. Symbol (x) represents a gap which contains arbitrary byte values between two segments that was predefined by antivirus analysts. On the condition that the scanning pattern's character is not equal to character asterisk (*), it will jump to Step 3 to perform the signature matching with the database signature. On the contrary, if the scanning process matches the character asterisk (*) wild card, it will proceed to step 2.

Step 2, Malware Code Offset address+ x bytes. The scan pointer will move to the next pattern segment with a predefined length of gap. The process will proceed to step 3.

Step 3, Signature match. This stage performs string pattern matching process with database signature. Upon a hit of signature match reported, the process will jump to step 4 to prepare for the next scanning loop. However, if no match is reported, the heuristic scanning process will stop and the remaining incomplete scanning target file will be passed to the emulator module. The emulator is a safe virtual environment used to monitor the running code. Details of the emulator will be explained in the next section.

Step 4, Signature Detection Loop. The scan pointer will shift to the next character and the scanning approach will continue by returning to step 1.

5.3 EMULATION EXECUTION FLOW

Emulation is a dynamic malware analysis process. It identifies common malicious activities via emulating the instructions of malware executable program. With the design of a safe and isolated architecture set on a host platform, a just-in-time binary execution could be performed. Our approach of emulation is to ensure no damage is done to the host machine; thus, a specific target platform to simulate the application level instruction and system call Interface is proposed. To emulate every instance's instructions and observe its execution, the CPU emulation is devised to be the core of our emulator. The CPU (Control Processor Unit) [20] is designed as the central part of machinery. It controls a computer by performing most of the calculations and the hard work of a system. Figure 8 illustrates the structural relationship among the emulator's components.

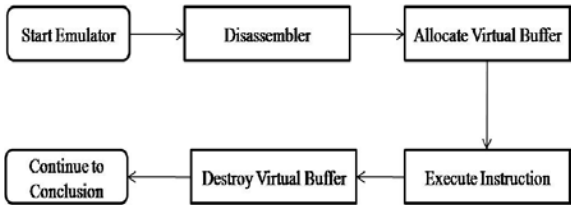


Figure 8: The overall Emulation Execution Flow

The emulator engine always begins with the Disassembler component [21]. The component is used to parse an instance's CPU instruction into its corresponding assembly code and use it to emulate the instance's execution inside a safe virtual buffer environment. Each instruction will be decoded to fetch the needed instruction type, length and operands. In order to transform a byte stream of opcode from a test case into assembly instructions, raw instructions phase is devised to determine total bytes of opcode that could be broken down for a single instruction. As shown in Figure 9, a stream of opcode is transformed into a list of much smaller, yet raw and groups of bytes at raw instructions phase. Each raw instruction is then parsed into a line of command and numbers of assembly program.

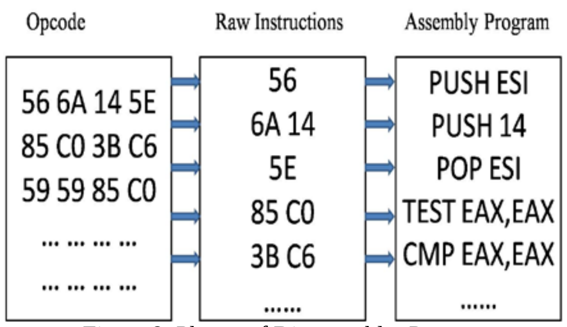


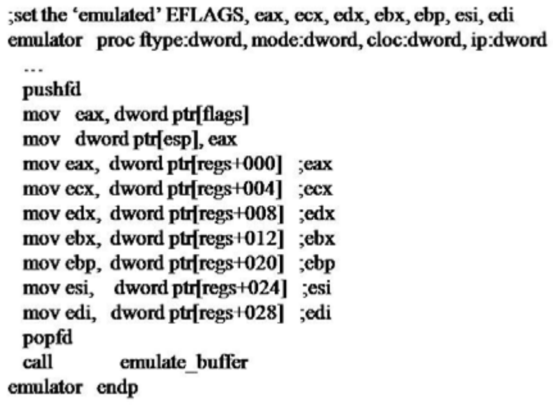
Figure 9: Phases of Disassembler Process

After the disassembly process, the translated program should be executed in a safe virtual environment. However, emulating a contiguous bunch of instruction assembly code is computationally expensive and not efficient in detecting malware signature. Our approach of solving the problem is by determining a contiguous block of code which has a single entry point and a single exit point at both the beginning and the end of the block, participate the code into blocks of instructions and store the entry point of each block in a table. More precisely, each instruction is converted into a basic block approach. This way, execution and signature matching will need to be performed within each block.

Typically, the task of allocating a virtual environment requires translating any arbitrary code of a given program into a design code sequence which has functionalities equivalent to the original. This is to ensure that the code could correctly run on our designed environment without damaging the host machines. The platform-independent's translating code is devised by using an intermediate language manner; thus, the possible CPU registers including operators, operand types and combinations of these would have to support the inside of the component for basic execution environment. As far as the CPU registers are concerned, there are eight general purpose registers (EAX, EBX, ECX, EDX, EBP, ESP, ESI and EDI) on a regular basis [22]. Each register also has a specific purpose, depending on the type of instruction currently being executed. The EAX register is commonly used as a default for addition and multiplication instructions. The ECX register is commonly used as a counter for looping, the ESP register is used to point to the last item on the stack and so forth. There are also special use registers, which have a particular purpose. The segment registers (CS, DS, ES, FS, GS, and SS) [22] are used to describe different segments of memory. The bits in the EFLAGS registers [22] are used for two purposes: to represent the outcome of computations and to control the operation of the CPU.

As mentioned earlier, a virtual environment is essentially a list of virtual CPU register that can be called. Thus, all possible operators and operand types should be ready to be translated using intermediate language. This can be done by creating a list of virtual CPU register to perform the corresponding instructions. To allocate virtual registers, the current original CPU registers including original general purpose register and the EFLAGS registers will be saved at a temporary allocated memory. This is to ensure that the original operators and operands are able to transfer back to the original once the execution is complete.

As soon as the original register is saved, the defined virtual CPU will transfer to the real CPU register. The execution of the target sequence will call the defined virtual CPU without access to the original register. Every execution is done in this virtual register instead of the real one; therefore, no damage will be caused by the execution. In Figure 10, the parameter of the virtual CPU registers, namely, [regs+000], [regs+004], [regs+008], [regs+012], [regs+020], [regs+024], and [regs+008] are stored into the EAX, ECX, EDX, EBX, EBP, ESI and EDI respectively. The function eventually calls *emulate_buffer* function to execute arbitrary code once the virtual environment is ready.



```
;set the 'emulated' EFLAGS, eax, ecx, edx, ebx, ebp, esi, edi
emulator   proc ftype:dword, mode:dword, cloc:dword, ip:dword
   ...
   pushfd
   mov   eax, dword ptr[flags]
   mov   dword ptr[esp], eax
   mov eax,  dword ptr[regs+000]   ;eax
   mov ecx,  dword ptr[regs+004]   ;ecx
   mov edx,  dword ptr[regs+008]   ;edx
   mov ebx,  dword ptr[regs+012]   ;ebx
   mov ebp,  dword ptr[regs+020]   ;ebp
   mov esi,   dword ptr[regs+024]   ;esi
   mov edi,  dword ptr[regs+028]   ;edi
   popfd
   call          emulate_buffer
emulator   endp
```

Figure 10: Translation of CPU register in Virtual Environment

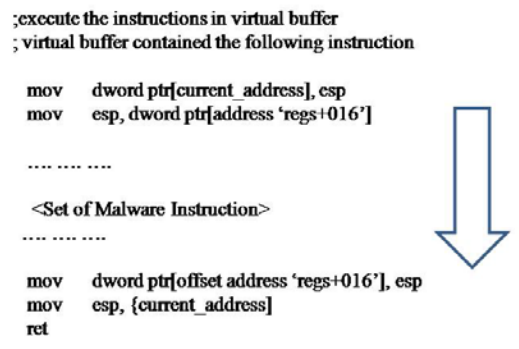
The stack is typically used to store local variables, as well as parameters passed in to the function. Our problem is that the value in the ESP register may change during the function's execution. Referencing values on the stack becomes rather difficult and complex; therefore, no alterations will be done for the ESP register in this function but alterations will be done for the later executed instruction function. Figure 11 illustrates the execution of every translated code that is manipulated by the ESP register. Prior to executing the arbitrary code instructions in the virtual system, the current ESP register must be saved onto a temporary address. This is to ensure the pointer of the function's execution can point back to the original after the execution. The activity that occurs in this function is to copy the defined virtual ESP register, which is the virtual address of [regs+016] to the current value of the stack pointer (ESP). While the preparation of virtual buffer environment is ready, the execution of translated of malware block code can be performed.

Journal of Theoretical and Applied Information Technology
31 August 2012. Vol. 42 No.2
© 2005 - 2012 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Table 1: Feature Analysis for detecting obfuscated virus and malware

Virus / Malware	Detection (√ or X)	Emulation Detection Time (Milliseconds)	Emulation Detection and Cure Time (Milliseconds)	File Size (KB)
Virus.Win9x.Marburg.b	√	47	63	28
Virus.Win32.Kriz.4029	√	125	140	470
Virus.Win32.Funlove.4070	√	16	18	61
Worm.Win32.QAZ	√	16	17	118
Virus.Win32.Parite.b	√	31	32	338

At the beginning of the execution, the current address would be at the entry point of an instance execution. While the scanning and detecting malware process is finished within the block of code, the current address will be updated to the destination pointer's instruction at the end of the block. This emulator process will only be considered to have completed if either a malware signature is detected or maximum allowed scanning time for a file has elapsed.



```
;execute the instructions in virtual buffer
; virtual buffer contained the following instruction

  mov    dword ptr[current_address], esp
  mov    esp, dword ptr[address 'regs+016']

  .... .... ....

  <Set of Malware Instruction>
 .... .... ....

  mov    dword ptr[offset address 'regs+016'], esp
  mov    esp, {current_address]
  ret
```

Figure 11: Fragment code of the execution of an arbitrary code in virtual environment

During execution, the translated code has to check whether an existing block consists of malicious code. The virtual buffer of emulator would be destroyed if any malware signature has been detected or the maximum allowed time for analysis time has elapsed. All original register saved on the stack must be restored before handling a pointer to conclusion.

6. EXPERIMENTAL ANALYSIS

In this section, we present the experimental analysis of the malware detection engine. The heuristic scanner is installed on a fresh VMware virtual machine of Windows operating system and a snapshot is taken. After each execution of an instance executable (either benign or malicious), the original snapshot will be reverted back to the parent image. The VMware software is chosen as the test platform mainly for two reasons: the first is the capability to revert the Operating System back to a clean state in case of any malware infection, and the second is to prevent any infection infecting the real machine.

6.1 HEURISTIC-BASED DETECTION RESULTS

The heuristic-based detection software detects malware based on malicious code behaviour. This is useful particularly when it is confronted with sophisticated obfuscating malware. To validate this, five different species of obfuscation virus and malware from VX Heaven [23] have been collected. We tested our approach on the following malware: Virus.Win9x.Marburg, Virus.Win32.Kriz, Virus.Win32.Funlove, Worm.Win32.QAZ and Virus.Win32.Parite.b. The detailed results are presented in Table 1 (√ indicates detection, X indicates failure to detect).

6.2 TESTING ON NEW VARIANTS

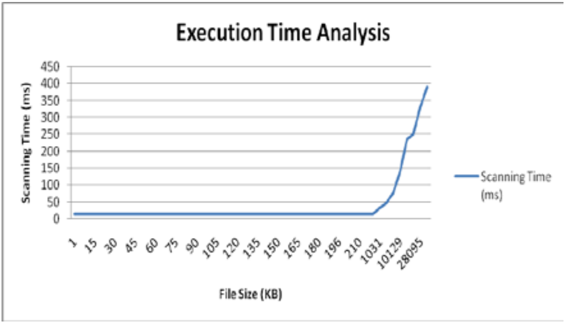


Figure 12: Execution Time Analysis

In this section, the performance result of our malware detection engine is reported. The required time to classify an instance as benign or malicious is tested. Figure 12 shows the execution time analysis graph where the x-axis represents the binary file size and the y-axis represents the execution time in milliseconds. As illustrated in the figure, for file sizes smaller than 1M, the execution time is almost constant. For file sizes bigger than 1M, the scanner took more time to finish its operation.

To test the effectiveness of the malware detector, we gathered 100 Windows binary programs from our fresh installation of Windows operating system and each file with the average size of 5KB. The experiment was conducted by incrementally choosing higher number of executable sample such as 10, 20, 30 and so on up to 100. The evaluation results are presented in Figure 13. The scanner took more time to finish its operation as the number of files increased.

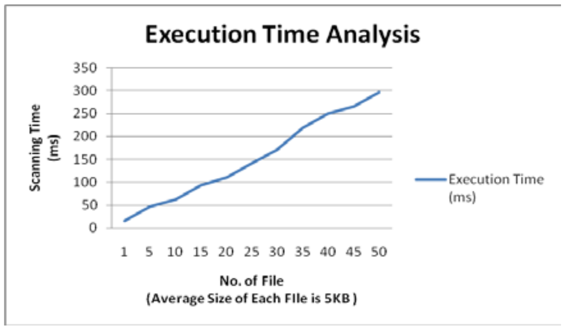


Figure 13: Execution Time Analysis

7. CONCLUSION AND FUTURE WORK

In this paper, we have proposed an approach of a generic malware detection engine based on the integration of known packer removal and both static heuristic and emulator techniques. The proposed scheme provides implicit robustness against most protection technique implemented by malware authors especially repacked virus in the purpose of evading the detection of antivirus. The important feature of our detection engine is that it can be both statically and dynamically detection based. We performed experiments to test our scanning engine on an extensive executable dataset. The results of our experiment show that the scanning is able to perform well and the detection accuracy is different depending on malware type. However, the limitation of this system is the lack of malware signature in our database. This can be overcome by creating signatures in our engine. However, this will require human expertise, is time consuming and most importantly, it is a continuous task.

REFRENCES:

[1] E.S.Adrian, "Defeating Polymorphism Beyond Emulation", *Virus Bulletin Conference 2005*, Dublin, Ireland, October 5-7, 2005, pp.40-48.

[2] M. Morgenstern, and T. Brosch, "Runtime Packer: The Hidden Problem", Black Hat USA 2006, Las Vegas, USA, July 29 – August 3, 2006, pp.40-48, http://www.blackhat.com/presentations/bh-usa-06 BH-US-o6-Morgenstern.pdf.

[3] M.G. Kang, P. Poosankam, and H. Yin, "Renovo: A Hidden Code Extractor for Packed Executables", *Proceeding of the 2007 ACM Workshop on Recurring Malcode*, NY, USA, October 29-November 02, 2007, pp.46-53.

[4] P. Royal, M. Halpin, D. Dagon, R. Edmonds and W. Lee, "PolyUnpack: Automating the Hidden-Code Exteaction of Unpack-Executing Malware", *Proceeding of the 22nd Annual Computer Security Applications Conference*, DC, USA, 2006, pp.269-278.

[5] A.H. Sung, "Static Analyzer of Vicious Executables (SAVE)", *Computer Security Applications Conference*, Socorro, NM, USA, December 6-10, 2004, pp.326-334.

[6] Clam AntiVirus Website [online]. Available: http://www.clamav.net/lang/en/

[7] J Ho., and G. Lemieux "PERG: A Scalable Pattern-Matching Accelerator". CMC Microsystems and Nanoelectronics Research Conference, Ottawa 2008, pp. 29-32.

[8] J. M. Aquilina, E. Casey, and C H. Malin. Malware Forensics: Investigating and Analyzing Malicious Code. Syngress, USA, 2008. pp. 283-378

[9] W. Wong, and M. S.tamp, "Hunting for Metamorphic Engines", *Journal in Computer Virology 2(3)*, 2006, pp. 211-229

[10] J. Borello, and L. Mé, "Code Obfuscation Techniques for Metamorphic Viruses", *Journal in Computer Virology*, *4(3)*, 2008, pp. 211-220

[11] B. Bayoglu, and I. Sogukpinar, "Polymorphic Worm Detection Using Token-Pair Signatures", *Proceedings of the 4th International Workshop on Security, Privacy and Trust in Pervasive and Ubiquitous Computing*, 2008. pp. 7-12

[12] Ultimate Packer for eXecutables, http://upx.sourceforge.net/

[13] ASPACT, http://www.aspack.com/

[14] WWWPACK, http://www.wwpack32.venti.pl/wwpack.html

[15] M. Pietrek, "An In-Depth Look Into the Win32 Portable Executable File Format", MSDN Magazine, February 2002

[16] Microsoft MSDN, http://msdn.microsoft.com/en-us/library/windows/desktop/ms683156(v=vs.85).aspx

[17] Windows, Dev Center – Desktop, http://msdn.microsoft.com/en-us/library/windows/desktop/aa364934(v=vs.85).aspx

[18] Windows, Dev Center – Desktop, http://msdn.microsoft.com/en-us/library/windows/desktop/aa364418(v=vs.85).aspx

[19] Windows, Dev Center – Desktop, http://msdn.microsoft.com/en-us/library/windows/desktop/aa364428(v=vs.85).aspx

[20] J. L. Hennessy, and D. A.Patterson, "Computer Architecture: A Quantitative Approach, Third Edition", Morgan Kaufmann Publishers, San Francisco, USA, 2003. pp. 678-778

[21] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of Executable Code Revisited". *Proceeding of 9th Working Conference on Reverse Engineering (WCRE)*, 2002. pp. 45–54.

[22] R. B. Blunden, "The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System", Wordware, USA, 2009. pp 54-56.

[23] VX heavens, http:vx.netlux.org

[24] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray, "A Semantics-Based Approach to Malware Detection", *34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007)*, Nice, France, January, 2007. pp.377-388.

[25] F. Guo, P. Ferrie, and T. Chiueh, "A Study of the Packer Problem and its Solutions", *11th Symposium on Recent Advances in Intrusion Detection (RAID)*, Boston, MA, September 2008. pp. 98-115.

[26] M. O. Myreen, "Verified Just-In-Time Compiler on x86", *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, 45(1)*, 2010. pp. 107-118

[27] J. Canto, M. Dacier, and E. Kirda, and C. Leita, "Large Scale Malware Collection: Lessons Learned". *IEEE SRDS Workshop on Sharing Field Data and Experiment Measurements on Resilience of Distributed Computing Systems*, Naples, Italy, October 2008.