Obfuscated Malicious Code Detection with Path Condition Analysis

Wenqing Fan
Communication University of China, Beijing, China
Email: fanwenqing@cuc.edu.cn

Xue Lei and Jing An
Information Security Center, Beijing University of Posts and Telecommunications National Engineering Laboratory for Disaster Backup and Recovery, Beijing University of Posts and Telecommunications, Beijing, China
Email: leixue@bupt.edu.cn, an_jing@bupt.edu.cn

Abstract—Code obfuscation is one of the main methods to hide malicious code. This paper proposes a new dynamic method which can effectively detect obfuscated malicious code. This method uses ISR to conduct dynamic debugging. The constraint solving during debugging process can detect deeply hidden malicious code by covering different execution paths. Besides, for malicious code that reads external resources, usually the detection of abnormal behaviors can only be detected by taking the resources into consideration. The method in this paper has better accuracy by locating the external resources precisely and combining it with the analysis of original malicious code. According to the experiment result of some anti-virus software, our prototype system can obviously improve the detection efficiency.

Index Terms—Malware Detection; Malicious Code Detection; Code Obfuscation

I. INTRODUCTION

To fight against the booming malicious software, many security software have done researches in detecting malicious code. Currently, three methods are most commonly used: scanning of static signature [1], behavior-based detection [2] and emulation-based detection [3]. The combination of the three can detect most traditional malicious software.

However, as the technology adopted by attackers becomes more and more advanced, malicious software now appears as malicious code more frequently. Malicious code is not a standalone executable file but a piece of code segment. Compared with traditional malicious software, malicious code is easier to be obfuscated transferred through network, and take advantages of vulnerabilities of software or system. Malicious attackers disguise malicious code specifically to bypass the existing protection methods. First, the obfuscated code has no constant static signature [4]. Also, the code jump and exception handling routines at ASM code level will further increase the difficulty of detecting behavior signature. And in virtual machines environment like VMware, only valid operations will be executed or the software will quit directly [5].

Researchers have proposed a series of novel detection methods in previous researches. Dawn Song used pure software VM to resist code obfuscation and avoid being detected by malicious code [6]. Engin Kirda mixed static and dynamic behavior technology to cover more behavior patterns of malicious code by adding behavior signature library [7]. Yanick Fratantonio utilized the debugging function of CPU to do dynamic debugging to improve efficiency in software VM [8]. This paper describes a new method on the basis of above three. This method includes path condition driven detection technology and also enhance the I/O supervision of software. It has the following advantages:

More hidden paths of malicious code will be triggered by analyzing path conditions and constraint solving. Malicious code usually use constraint jump or hide its behavior [9]. This paper can drive malicious code to explore different branches based on constraint solving result, and hence improve efficiency.

The supervision of software I/O can detect and locate malicious code more thoroughly. Some malicious code will the malicious part of code in external resources, and only read them when executing. The method in this paper can actively load external resources through I/O APIs to reach a better result.

On top of the previous two steps, the method will mine the signature of malicious code behavior and system call sequence, and detect the malicious code.

The paper is organized as follows. Section 2 presents some related works and discusses challenges of malicious detection when dealing with external resources. Section 3 shows our method for obfuscated malicious code detection. In Section 4, experimental results are discussed and conclude with Section 5.

II. RELATED WORKS

Since malicious code is easier to hide and exploit various vulnerabilities, more and more attackers are using files containing such code or web page with malicious script to attack others [10]. Currently, static and dynamic analyses are being used to detect such code.
A. Static Detection Methods

Static analysis tries to extract static signature by disassembling software [11]. However, Andreas Moser and others pointed out that static analysis is not enough to deal with variations of malicious code [12]. As they said, static analysis can be evaded by code obfuscation. Opaque constants are difficult to be solved in static analysis. As the same time, obfuscating transformation methods are deployed in malicious code.

![Example](image1)

**Figure 1. Example**

![Control-Flow Graph](image2)

**Figure 2. Control-Flow Graph**

As shown in Figure 1, 2 and 3, Control-flow obfuscation like control-flow flattening [13] will thoroughly change the Control-Flow Graph of malicious code. As shown in the example, the control-flow graph will be more complicated. Even more, malicious attacker can insert a lot of unreachable block to confuse static analysis tools and thwart two main static disassembly algorithms: linear sweep and recursive traversal [4]. So, the focus of research has been gradually changed to dynamic debugging and other methods.

B. Dynamic Detection Methods

Michalis Polychronakis has obtained impressive progress by debugging malicious code dynamically in simulated environment [3]. Since then, the dynamic debugging analysis has been developing rapidly. Dawn Song and others designed a malicious code analysis platform based on this method [6]. A common feature of these methods is that they will first execute malicious code in simulated software environment and then analyze the result. In this situation, the malicious attacker discovers that the software simulator can't simulate all real CPU instructions and by using some special instructions the detection mechanism based on it can be bypassed [14]. To solve this problem, Yanick Fratantonio used CPU trap flag in real environment to do single-step debugging, while Kevin Z. Snow used hardware simulator to simulate the real environment.

Compared with static analysis, dynamic analysis has two main defects: low path coverage rate, low execution efficiency. The two methods above take care of execution efficiency with several strategies but ignore the path coverage rate.

![Control-Flow flattening](image3)

**Figure 3. Control-Flow flattening**

C. Challenges

Some malicious code has external resources. External resources can be downloaded from the Internet. If external resources are unavailable, malicious code will crash. So external resources should be downloaded, then dynamic analysis can execute normally. Dynamic analysis only executes one path of malicious code each time. So the coverage rate of dynamic analysis may be very low.

This paper, on the basis of them, improve the path coverage rate by analyzing the path conditions and constraint solving to get path condition for various execution paths. Besides, the method in this paper also monitor system I/O API to locate external resources since tracing them can be helpful to detect separated malicious code.

III. DYNAMIC ANALYSIS FOR OBfuscAted MAlicious Code DETECTION

To solve problems mentioned above, the method in this paper modifies ISR to trace software and do dynamic analysis. Setting TF flag in EFLAGS register to 1 can trigger INT 1 interruptions, which are handled by ISR 1. And modifying the process of ISR 1 is a convenient and efficient way to do dynamic analysis. Malicious code that can detect TF flags can be bypassed by adding instruction judgment in the routine.

The framework of the system is displayed in Figure 2. We can create a dynamic analysis debugger by modifying ISR 1 of Windows XP and then obtain execution logs by tracing targets with the debugger. This method will conduct constraint solving for the path conditions in the software execution flow, get the required input for executing other paths using Concolic testing [15, 16] and re-analyze malicious code. During the re-analysis, the inputs obtained from Concolic testing for new paths will drive the target to follow those paths. Thus, the mining of malicious code will have a better coverage. Besides, this
method will monitor system I/O APIs for functions like file reading and network downloading to detect malicious behavior in external resources. Finally, the analyzer will judge whether the target is malicious code according to its system call sequence, behavior signatures and other factors.

A. ISR Debugger

Interrupt Service Routines is interrupt handler routines provided by the Windows system. Interrupt Descriptor Table stores the interrupt number and the corresponding exception handling routine address. When the software or hardware trigger these interrupts, the system automatically lookup IDT and find the address of ISR, then call the routine to handle the interrupt. Operating system by reading the interrupt descriptor table register locates the interrupt descriptor table in memory location and size, shown in Figure 3.

In the architecture of Intel 80386, the length of IDT is 48 bits. The first 32 stored the memory address of IDT, and the size of IDT saved in after 16 size. There are 256 entries in IDT, each corresponding to a 64-bits gate. There are three categories of gates: Task Gate, Trap Gate and Interrupt Gate. The data of different gate corresponds different meaning, as shown in Figure 4.

Interrupt gate and Trap gate are used widely in IDT. The 32-bits offset points the memory address of interrupt service routine. We can trace the CPU instructions by modifying the interrupt service routine. When the TF flag in the EFLAGS register is set to 1, an interrupt INT 1 is triggered, INT 1 interrupt service routine is the ISR 1. We can easily and efficiently implement dynamic tracing by modifying ISR 1.

B. Detection Driven by Path Condition Analysis

Based on dynamic tracing, the key to malicious code detection is to accurately spot suspicious behavior. However, malicious attackers usually will cover their traces via conditional jump. This paper can also deal with such situations. Based on analysis and recording of path conditions, the method described here can do constraint solving and calculate new path conditions. Then, the interaction between analyzer and debugger will drive the target to follow the new paths, and eventually detect the hidden malicious code. When the current instruction is a
conditional jump instruction, the system will combine Log and debugger to accomplish path-condition-driven detection. The procedure is:
1. Judge whether both branches for current instruction have been executed. If yes, then look for the next branch condition.
2. Select the un-executed branch, and calculate the condition for this path.
3. Communicate with debugger to execute this branch, log the system call sequence and behavior signature for it, and compare them with the signature library.
4. Combine the system calling sequence and behavior signature of several branch conditions, compare them with signature library.
5. Continue with execution. Go to step 1 if encounter another condition; otherwise, exit.

With this method, we can mine the behavior signature of malicious code in suspicious code with better accuracy, and use the result as feedback for the signature library which will be used in future detection.

B. Locating and Analysis of External Resources

The space available for programs that exploit vulnerability is far from ample, and thus limit the volume of malicious code, that's why malicious attackers will load external resources to implement complicated operations.

So, the external resources should be an input for the analysis of detection target. The method in this paper use SSDT hook to locate various system I/O APIs for features such as file, network and register, analyzes their parameters to locate and save external resources accurately, as shown in Figure 5.

Those resources will be included in the detection for further analysis. For various external resources, such as file, network data flow, code in memory, different loading method will be used.

**PE file**: The resources in form of PE file can be executed with API like Winexec, CreatProcess. The detection system will compare with the parameter with external resources' paths and detect the execution of them. The Log of original resource file will be combined with the execution path for further checking.

**Network data flow and memory data**: such resources can be executed with instructions like call and jump in malicious code. The detection system will match the target address range, identify the memory space where the external resource resides, monitor the code execution in memory, and analyze it together with detection result for the original resource file.

The workflow is described below:
1. Locate and identify external resources by monitoring key system APIs.
2. Execute the malicious code, monitor APIs related to process execution and special instructions like call and jump.
3. If malicious code executes the external resources or jump to it, trace and get the log of external resources.

4. Combine the Log of original malicious code and the Log of external resources to conduct comprehensive detection.

As shown in Figure 6, through the detection of external resources, the system can discover separated malicious code with higher accuracy and extract a more thorough profile for such code.

C. Signatures of Malicious Code

The key problem of malicious code detection is accurately analysis the behaviors of target code based on dynamic tracing. Researchers have done a lot of fruitful research in area of malicious code detection. To avoid malicious code infect the host system if progress of analysis, Fred Cohen proposed a method which detects malicious code in the sandbox [17]. The developers of SAFE use static analysis to detect malicious code, and even obfuscate malicious code with "NOP" instruction, SAFE is still able to effectively detect [18]. Based on SAFE, Christopher Kruegel et al improved static disassembly techniques after in-depth study. They proposed a novel recursive disassembly algorithm and use statistical methods to achieve the detection of obfuscated malicious binary code [19]. In these methods, some of the researchers use some character strings as the signature of malicious code. Others use N-Gram sequences of system calls as the signature of malicious code.

Malicious code is designed to obtain system information, and steal a user’s critical data, or destroy the system and propagate itself through the network. These behaviors need to call the system I/O API, so our malicious code detection tool traces system I/O API calls and other behavioral characteristics and use these as signatures. This method can more effectively classify malicious code.

Analyzing system API such as file operation, register operation, winsock etc. In accordance with the sequence of calls to get vectors $k$. Which $c_i, i = 1, 2, \ldots, n$ represents the i-th called API, its value is the frequency of the called API.

Using the vector as malicious signature, this new method not only can effectively detect known viruses, but also achieve unknown malicious code detection and classification. Through the behavior and characteristics of the object to the library vector chi-squared distribution analysis to achieve. Chi-squared test is a statistical test used to compare two sets of statistical data whether there are different. It is one of statistical hypothesis testing. Assuming system call sequence vector of malicious code is $P = \{p_1, p_2, \ldots, p_n\}$, and the vector of the suspicious program is $C = \{c_1, c_2, \ldots, c_n\}$. According to the chi-square test, two vectors difference value is calculated as:

$$\chi^2 = \sum_{i=1}^{n} \frac{(c_i - p_i)^2}{p_i}$$

Compare the calculated value with the standard chi-squared distribution $\chi^2_{0.05}(1) = 3.84$. When $\chi^2 > \chi^2_{0.05}(1) = 3.84$, the suspicious program does not belong to the class of malicious code, otherwise the suspicious programs is one of the malicious code classes.

IV. EXPERIMENT AND ANALYSIS

A. Path Feasibility Analysis and Statistical of Threshold

Malicious code analysis methods based on path conditions need to deal with a large number of infeasible paths, which are used by malicious attackers to avoid anti-virus software. These infeasible paths usually combined with opaque predicates results the problem that it is difficult to identify static infeasible paths. To cope this problem, we propose the method of dynamic segmental execution to analyze infeasible paths dynamically. In our scenario, obtain suspected-segments of infeasible paths at first, and then segmented execute the program by using different input repeatedly. When the number of program executions is above a certain threshold, the path is infeasible.

In order to determine the threshold of dynamically segmented execution times, this section provides statistics of running times $k$, path coverage ratio $p$, branch-uncover ratio $v$, and verifies their relationship. The statistic method is to select abundance samples that include opaque predicates and malicious codes, test these samples $k$ times, record path coverage ratio $p$ and branch-uncover ratio $v$ during every testing. When coverage ratio $p$ is above 95%, the corresponding $k$ is the threshold.

<table>
<thead>
<tr>
<th>Path coverage ratio $p$ (%)</th>
<th>Branch-uncover ratio $v$</th>
<th>Running times $k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.9771</td>
<td>26782</td>
</tr>
<tr>
<td>20</td>
<td>0.9684</td>
<td>38661</td>
</tr>
<tr>
<td>30</td>
<td>0.9529</td>
<td>52521</td>
</tr>
<tr>
<td>40</td>
<td>0.8546</td>
<td>76949</td>
</tr>
<tr>
<td>50</td>
<td>0.806</td>
<td>107855</td>
</tr>
<tr>
<td>60</td>
<td>0.7767</td>
<td>151871</td>
</tr>
<tr>
<td>70</td>
<td>0.765</td>
<td>220835</td>
</tr>
<tr>
<td>80</td>
<td>0.7465</td>
<td>361733</td>
</tr>
<tr>
<td>90</td>
<td>0.6838</td>
<td>856365</td>
</tr>
</tbody>
</table>

Figure 9. W32/Online games: path coverage ratio by segmented testing
TABLE II. VIRUS BULLETIN SAMPLE: W32/AUTOIT

<table>
<thead>
<tr>
<th>Path coverage ratio (p (%))</th>
<th>Branch-uncover ratio</th>
<th>Running times</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.9831</td>
<td>5861</td>
</tr>
<tr>
<td>20</td>
<td>0.9126</td>
<td>9033</td>
</tr>
<tr>
<td>30</td>
<td>0.8354</td>
<td>13193</td>
</tr>
<tr>
<td>40</td>
<td>0.7735</td>
<td>18719</td>
</tr>
<tr>
<td>50</td>
<td>0.6993</td>
<td>27371</td>
</tr>
<tr>
<td>60</td>
<td>0.6361</td>
<td>39586</td>
</tr>
<tr>
<td>70</td>
<td>0.6402</td>
<td>58103</td>
</tr>
<tr>
<td>80</td>
<td>0.5816</td>
<td>102230</td>
</tr>
<tr>
<td>90</td>
<td>0.5995</td>
<td>247153</td>
</tr>
</tbody>
</table>

In both samples, path coverage ratio increased rapidly with the growth in the number of tests in the early stage, which is due to the beginning of testing, all the branches are not traversed and Concolic testing can cover a path in each execution. When the number of segmental tests became large enough to cover most paths, there were still infeasible paths and some paths that influenced by black-box functions were difficult to cover. Hence, the growth of path coverage ratio slowed down, and path coverage ratio cannot reach 100% because of infeasible paths. To determine the threshold $K$, we used the aforementioned segmental Concolic testing to analyze abundance samples of malicious code, and the relationship between path coverage ratio and running times is shown in Figure 11.

![Figure 10. W32/Autoit: path coverage ratio by segmented testing](image)

![Figure 11. The relationship between path coverage ratio and running times](image)

In Figure 11, when the number of running times is small, path coverage ratio with segmental Concolic testing increases rapidly. As the number of running times increasing, the speed of growth of path coverage ratio slows down. In the early stage, branches coverage is zero, therefore, segmental Concolic testing is easy to traverse a new path. After running a couple of times, more branches traversed while branches uncover ratio $v$ became smaller, it is difficult for segmental Concolic testing to traverse new paths. Overall, we can conclude the calculation formula of threshold value is:

$$K = \frac{i}{v} \times \left(1 - \frac{1}{\log p}\right) \quad 0 < p < 1$$

In the segmental Concolic testing, $i$ is the number of branches is the branch-stack; $v$ is the branch-uncover ratio; $p$ is the path coverage ratio. The threshold $K$ can be calculated by this formula with these three parameters.

Code segments including black-box functions were performed multiple times by segmental Concolic testing, after which we analyzed the program and calculated the actual executions. If the actual executions were above the threshold $K$, we determined the path was infeasible.

B. Detection Results of Obfuscated Malicious Code

The prototype system for this method is developed on Windows XP, and the experiment is carried out in a computer with AMD Athlon II X2 5000+ and 4GB RAM. We collect 300 malicious code samples from the WildList Organization International [20]. We created polymorphic versions of these samples, and compete the detection efficiency of our prototype system and other security software. The detection result for polymorphic versions is listed in Table 1. It shows the results of detection before obfuscation and after obfuscation.

<table>
<thead>
<tr>
<th>Anti-virus Software Name</th>
<th>Detection Rate Before</th>
<th>Detection Rate After</th>
</tr>
</thead>
<tbody>
<tr>
<td>AntiVir</td>
<td>100%</td>
<td>156%</td>
</tr>
<tr>
<td>Avast</td>
<td>100%</td>
<td>178%</td>
</tr>
<tr>
<td>AVG</td>
<td>100%</td>
<td>134%</td>
</tr>
<tr>
<td>BitDefender</td>
<td>100%</td>
<td>145%</td>
</tr>
<tr>
<td>Comodo</td>
<td>99.33%</td>
<td>167%</td>
</tr>
<tr>
<td>DrWeb</td>
<td>99.33%</td>
<td>142%</td>
</tr>
<tr>
<td>NOD32</td>
<td>100%</td>
<td>151%</td>
</tr>
<tr>
<td>Kaspersky</td>
<td>100%</td>
<td>189%</td>
</tr>
<tr>
<td>Kingsoft</td>
<td>100%</td>
<td>129%</td>
</tr>
<tr>
<td>McAfee</td>
<td>100%</td>
<td>133%</td>
</tr>
<tr>
<td>Rising</td>
<td>100%</td>
<td>139%</td>
</tr>
<tr>
<td>Symantec</td>
<td>100%</td>
<td>168%</td>
</tr>
<tr>
<td>Prototype System</td>
<td>99.33%</td>
<td>241%</td>
</tr>
</tbody>
</table>

Before obfuscation, almost all anti-virus software were able to fully detect these malicious code samples, our prototype system also achieved a good result. However, the detection rate after obfuscation has changed significantly. Most anti-virus software’s detection rate has dropped to less then 50%, only a few remains at between 50% - 65% detection rate. Meanwhile, Prototype system detection rate reached 80.33%. This result shows the prototype system can detect obfuscated malicious code more effectively than other anti-virus software.

V. CONCLUSIONS

A new malicious code detection method is presented in this paper. This method effectively detects obfuscated malicious code by using path condition analysis. At the same time, External resources are located and traced to improve the accuracy of detection. The experiment result of prototype system proves the method is effective.
Compare with traditional methods, this method can be more accurate in identifying characteristics of malicious code.

REFERENCES