Lightweight Examination of DLL Environments in Virtual Machines to Detect Malware

Xiongwei Xie
Department of SIS
UNC Charlotte
Charlotte, NC 28223
xxie2@uncc.edu

Weichao Wang
Department of SIS
UNC Charlotte
Charlotte, NC 28223
weichaowang@uncc.edu

ABSTRACT

Since it becomes increasingly difficult to trick end users to install and run executable files from unknown sources, attackers refer to stealthy ways such as manipulation of DLL (Dynamic Link Library) files to compromise user computers. In this paper, we propose to develop mechanisms that allow the hypervisor to conduct lightweight examination of DLL files and their running environment in guest virtual machines. Different from the approaches that focus on static analysis of the DLL API calling graphs, our mechanisms conduct continuous examination of their running states. In this way, malicious manipulations to DLL files that happen after they are loaded into memory can also be detected. In order to maintain non-intrusive monitoring and reduce the impacts on VM performance, we avoid examinations of the complete DLL file contents but focus on the parameters such as the relative virtual addresses (RVA) of the functions. We have implemented our approach in Xen and conducted experiments with more than 100 malware of different types. The experiment results show that our approach can effectively detect the malware with very low increases in overhead at guest VMs.

Keywords

Malware Detection through Hypervisor; Attacks on DLL Files; Lightweight Monitoring

1. INTRODUCTION

Computer systems face the threats from high spreading rate of computer malware (worms, Trojan horses, rootkits, botnets, etc.). Malware intrudes into computer systems and causes millions of dollars in damage. Host-based malware detection mechanisms have their limitations. On one side, since the anti-malware systems are installed and executed inside the hosts that they are monitoring, they can collect rich information from the local host. On the other side, since they are visible and tangible to advanced malware running in the system, effective attacks towards them become feasible. For example, some malware such as Agobot variant [1] can detect and remove more than 105 types of anti-malware programs in the victim machine.

SCC'16, May 30-June 03 2016, Xi'an, China
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4285-8/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2898445.2898456

Since it becomes increasingly difficult to trick end users to download and run executable files from unknown sources, attackers refer to more stealthy ways to avoid detection. Based on a report released by Kaspersky [13], about 60% of malware collected at KingSoft anti-malware lab are DLL files. From this point of view, protecting the authenticity and integrity of DLL files that are loaded into computer systems is essential for the safety of end users.

The emergence of cloud computing opens a new horizon for combating with the trends in malicious attacks on DLL files. Some researchers [7, 10, 23] proposed to place the intrusion detection mechanisms outside of the virtual machine being monitored. A well implemented hypervisor will enforce strong isolation among virtual machines and the programs running within them. Under this condition, even if a virtual machine is compromised by malware, it is difficult for the attacker to compromise the hypervisor. For example, VMwatcher [6] uses the general virtual machine introspection (VMI) methodology in a non-intrusive manner to inspect the low-level VM states. UCON (usage control model) [24] is an event-based logic model. It maintains the lowest level access to the system and ensures that such access cannot be compromised by internal processes of a VM.

Existing malware detection approaches often use information from DLL or other executable files in the following way. They will collect a large number of PE or DLL files and conduct static analysis of the API calling graphs in these applications. The learned knowledge will then be used as features to detect malware. Example solutions include [17, 18, 25]. These approaches are effective in detection of infected files when they are stored in hard-drive. However, if no continuous examination is conducted, they cannot capture infections to the DLL files that happen after their initial screening. Another thread of research uses signatures of different pieces of kernel code [12] or cross-compares code segments among multiple VMs [2] for code integrity checking purposes. However, it quickly becomes cumbersome and time consuming to maintain a database of all legitimate signatures. For example, ModChecker [2] will introduce a delay of 0.2 second when it tries to compare *http.sys* on two mostly-idle virtual machines.

In this paper, we propose a lightweight approach at the hypervisor level to continuously monitor the status of loaded DLL files in guest virtual machines to detect malware. Instead of using information that is extracted from different modules of a VM as individual components, our cross-verification schemes cover a wide range of properties of the DLL files including their loading path, loading order, and RVA (relative virtual address) of the functions. Our overall approach can be divided into three steps: collection, analysis, and monitoring. Through memory reconstruction technology of the virtual machines, we record the execution states of different applications in the VMs at the hypervisor level. Using freshly installed virtual machines, our collection procedures will extract and record information of DLL files in malware-free environments. After we collect enough information from the training data, we will start the analysis procedures. We will explore relationship between the active processes and the loaded DLL files. We will also generate fingerprint of the loading order and RVAs of the DLL files. We will then continuously monitor the DLL files running in the VMs and compare them to the extracted features. If attackers make any changes to the information under surveillance, we can detect the infection in real time.

The contributions of our research can be summarized as follows. First, instead of examining the DLL files for only once when they are loaded, our approach conducts continuous monitoring on the files. In this way, infections to the libraries can be detected on the fly. Second, our malware detection mechanism uses non-intrusive introspection of virtual machines. Since the detection mechanism is running at the hypervisor level, it is very difficult for malware running in a VM to detect, remove, or avoid our approach. Third, instead of examining the contents of the whole libraries, we focus on some high level yet essential information such as the RVA of the functions. In this way, even when we examine the VMs' memory at a relatively high frequency, the impacts on their performance are still very low. Last but not least, we conduct extensive experiments to evaluate the proposed approach. We use more than 100 malware of different types (Trojans, stealth backdoors, adware, and virus) to test our detection mechanism. Our solution detects almost all malware samples with very low false negatives.

The remainder of this paper is organized as follows. In Section 2 we discuss the related work. In Section 3 we present the details of the proposed approach. In Section 4 we describe the implementation of the malware detection mechanism and experiment results. Section 5 discusses a few issues in the detection procedure. Finally, Section 6 concludes the paper.

2. RELATED WORK

Existing approaches to manipulating DLL files in a computer system can be roughly classified into two groups. In the first group, malware will try to load its own DLL files into the system through DLL injection mechanisms [3]. An attacker can achieve the goal through either remote thread injection or registry DLL injection. For example, Conficker worm injects undesirable DLLs into legitimate software packages [19]. Another way of manipulating the DLL files is to change the files directly. For example, attackers can use API hooking [26] to redirect a benign function call to malicious code segments. In-line code overwriting can also be used to achieve the goal [22].

Our approach can effectively detect the attacks described above. During the collection phase, we have learned the relationship between the application software and the DLL files loaded by it. Therefore, if attackers try to link their malicious DLL with an application, we can detect the anomaly. Since our approach will examine the RVA of the function calls, API hooking can be detected as well.

Researchers have experimented with malware detection through verifying integrity of system files. For example, SBCFI (state-based control-flow integrity) [15] monitors kernel integrity of the OS to detect malicious changes. Copilot [14] implements a similar approach in coprocessor platforms. The disadvantage of these approaches is their relatively heavy overhead on the system. In our approach, we choose to examine only the high level information instead of the complete file contents to reduce impacts on the VMs. More discussion on this choice is presented in Section 5.

3. THE PROPOSED APPROACH

3.1 System Assumptions and Design Goals

In the investigated scenario, we assume that we can get access to malware-free virtual machines during the collection and learning phases of the approach. This can be achieved through using freshly installed systems. After the collection procedures, we do not restrict user behaviors on the VMs. They may be tricked by attackers and download or install some adware, virus, worm, or spyware into the system. We also assume that an attacker can acquire the administrator privilege of a compromised virtual machine after she/he intrudes into the system. Malware installed by the attacker may modify return results to anti-virus programs that are running in the local VM to hide the malicious process. The malware may also inject in some benign process, such as explorer.exe, and start several threads to conduct malicious activities. However, similar to the approaches in [27], we assume that the attacker cannot infect the hypervisor through the VM.

Our investigation has the following design goals. First, the malware detection mechanism should be transparent to end users. Moreover, it can extract and recover the virtual machine's execution states accurately. This goal is realized through the non-intrusive introspection technology. Since we examine the VM execution states from the hypervisor, it is very difficult for the malware to mislead the detection procedure. Second, the approach should be independent from specific hypervisors. The design should support VMM in both full virtualization mode (e.g., VMware[20] and KVM [9]) and paravirtualization (e.g., XEN [4]) modes. This property allows more users to benefit from the approach. Analysis in later parts will show that our malware detection mechanism does not depend on any specific hypervisors. Third, we also need to control the performance impacts of the proposed approach on virtual machines. This requirement is essential for future deployment and adoption of our malware detection mechanisms. Our experiments will show that examining only the execution states in VM memory, even periodically with a short time interval, will introduce a small increase in overhead.

3.2 Memory reconstruction

Before we can collect and analyze information from guest virtual machines, we must first reconstruct memory of the VMs at the hypervisor level. Since the hypervisor sees only the raw memory pages of a virtual machine, we need to rebuild its semantic view, so that we can extract high level semantic information.

To better explain the memory reconstruction procedure, below we use an example of a Windows guest OS to illustrate the information extraction operations. We can go through the process list from $PsActiveProcessHead$ (the head of the double linked list). In Windows XP, each process has an $\_EPROCESS$ object through which we can traverse the whole list. Each $\_EPROCESS$ object contains both $Flink$ (forward link) that points to the next $\_EPROCESS$ structure and $Blink$ (backward link) that points to the previous $\_EPROCESS$ structure. $PsActiveProcessHead$ is a member of the kernel debugger data block ($\_KDDEBUGGER\_DATA64$), which is used by the kernel debugger to find out the states of the operating system [11]. Furthermore, the Kernel Processor Control Region (KPCR) is a data structure used by the Windows kernel to store information about each process. It is located at virtual address $0xffdff000$ in Windows XP. In KPCR, the data structure $KdVersionBlock$ contains a linked list of $\_KDDEBUGGER\_DATA64$.

Through memory introspection technique, we can extract high level execution states of a virtual machine. The execution states that we can get include process list, network connections, opened files, dynamic loadable library, and relative virtual address of functions in DLLs. In the process list, we could get the full path of the files in execution. For each process, the order of the extracted DLL files is identical to the order in which the process loads them into memory. Moreover, we can get the relative virtual address (RVA) of functions in different DLLs. Access to the information provides a rich data set for us to conduct subsequent analysis and design of detection mechanisms.

3.3 Design of the Malware Detection Mechanisms

At the high level, we propose a malware detection mechanism that runs in hypervisor to detect infected DLL files in guest VMs. This is accomplished in three steps (Figure1): (1) the collection phase, in which a process collects information about different applications from malware-free virtual machines; (2) the analysis phase, in which we analyze the execution states of each benign process, and extract the characteristics of these benign applications; and (3) an online detection phase, in which the detection program is used to detect infected DLLs in a guest VM through comparing their execution states to the learned information. These three steps are described in more details below.

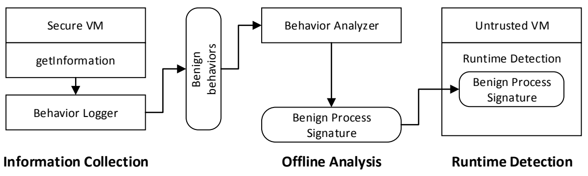

Figure 1: The proposed malware detection procedure.

First, we use a running example to illustrate the information collection procedure. In order to get a comprehensive view of the execution states and avoid impacts from a specific running environment, our information collection program will run many times in the hypervisor to extract behaviors of each process from different installations of malware-free VMs. For example, in a PE file, the export structure is called $Image\_Export\_Directory$ with eleven (11) members in it. $AddressOfFunctions$ is the head of the array that keeps RVAs to all functions in the module. $AddressOf-Names$ is the head of another array that keeps the names of functions in the module. Combining these two arrays, we can get export functions and RVAs in pairs.

Although the information that we gather through the collection phase contains a lot of data that can be used to differentiate malware from benign applications, two reasons lead us to apply some learning algorithms to filter out noises and generate behavior patterns of benign processes. The first reason is that different versions of the same software demonstrate different behaviors. We can use an example of the RVAs of the same function to illustrate this. In $ws2\_32.dll$ with version 5.1.2600.5512, the RVA of function $socket$ is $0x00004211$, while the RVA of the same function in version 6.3.9600.17415 is $0x00003BD0$. The second reason is that even when the same application software with the same version number is installed, the VMs may still demonstrate different behaviors in different environments. For example, under most conditions, $explorer.exe$ will not load $avcuf32.dll$ into memory. However, if we install $Bitdefender$ in our virtual machine, which is an anti-virus application, $explorer.exe$ will load $avcuf32.dll$ into memory after $KERNEL32.dll$. Similarly, $explorer.exe$ will load 7-$zip.dll$ into memory only if we install 7-zip. Because of the differences in behaviors, we need to experiment with the same application in different running environments to learn their behaviors.

The knowledge that we learn from the malware-free VMs covers a wide range of properties of the applications. For example, we will check the DLL names, their full loading paths, and RVAs of functions in them. In this way, if an attacker loads his own malicious DLLs into the system, we will be able to catch them. Some malware may impersonate a popular process name, such as $svchost.exe$, to fool the detection algorithm. The fake process, however, usually needs to load DLLs from a folder that is different from that of the real application. Therefore, we can distinguish between them based on these differences.

Another feature that we can use to detect malware is the relationship between the functionality of an application and the DLLs it loads. DLL files usually serve specific purposes. For example, $ws2\_32.dll$, $hnetcfg.dll$, $pstorec.dll$, and $crypt\_32.dll$ are used for networking, firewall maintenance, access to protected storage, and cryptography, respectively. An application should load only the DLLs that it needs to use. Through analyzing the functionality of different applications and the DLL files that they load, we can expose their relationship. If DLL files that deviate from the functionality of an application are loaded, we need to conduct further investigation. For example, if a malicious application calls functions from all four DLLs we mention above, it can read confidential information from the system, encrypt it with secret keys, and send it out to the attacker. This type of anomaly can be detected through cross comparison among the DLLs that are loaded by the applications with similar functionality.

We can also use dependency among DLL files to detect malware. Their dependency could impact the order in which they are loaded into the system. For example, $IEXPLORE.EXE$ is a frequent target of attackers. We analyze its behaviors and find out that there are 16 groups of consecutive loading orders of DLL files. The length of these consecutive segments ranges from 2 to 20. Some DLL injection attacks will break these consecutive segments. Therefore, we can use this change to detect the malware.

Last but not least, through checking the RVA addresses of the functions, our detection mechanism can catch several types of code injection and in-line code overwriting attacks upon DLLs. In order to increase the efficiency of our detection mechanism, we will examine the RVAs of the functions as one unit by calculating and checking their hash result.

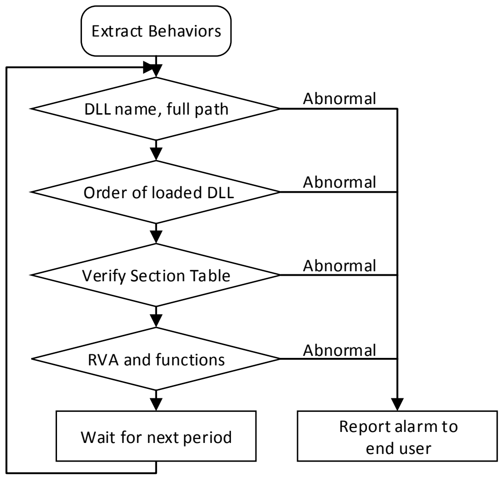

Figure 2: Runtime detection procedure.

After analyzing the extracted information from malware-free VMs, we will start an on-line monitoring and detection procedure in the hypervisor. The details of the detection procedure are shown in Figure 2. At the beginning of each round of detection, we need to take a snapshot of the memory pages of a VM that contain its execution states. This operation will take a very short period of time and will not impact the user experiences with the guest VM [10]. After the system data structures are reconstructed from the memory pages, we will compare the information to the knowledge that we have learned through Step 2. If any anomaly is detected, we will raise an alarm.

4. IMPLEMENTATION AND EXPERIMENTAL RESULTS

4.1 Experiment Setup

To evaluate the detection capabilities of the proposed approach and assess its impacts on the system performance, we implement the mechanisms in Xen and conduct two groups of experiments. In the first group, we investigate the effectiveness of our malware detection approach through a group of real-world malware. In the second group, we evaluate the impacts of our detection mechanism on the guest system performance. The experiment environment setup is as follows. The physical machine has a four core 3.30GHz Intel CPU, 10 GB RAM, and SATA hard drives. The hypervisor that we use is Xen version 4.1.2 with the libvir 0.9.8. The host operating system is Ubuntu Server 12.04 LTS (64bit). We test two virtual machines. One virtual machine is using Windows XP SP3 (32bit) as the operating system. The other virtual machine is using Windows 7 Professional (32bit) as the operating system. Each virtual machine occupies one CPU core and 20GB hard disk. We vary the allocated memory to VMs from 1GB to 3GB to evaluate the impacts of the memory size on our proposed approach.

4.2 Detection Capability of the Proposed Approach

To collect behavior patterns of the benign systems and software, we install popular benign applications in the guest VMs. We download 100 benign and freely available applications from a trustworthy and reputable web site. These benign applications cover freeware programs in a wide range of different domains (such as system utilities, office applications, media players, instant messaging, and browsers). We experiment with different combinations of the software in different environments so that the analysis phase can extract their special properties. After that, we download and install six groups of different malware to evaluate the detection capabilities of our program. We conduct malware detection experiments on two types of virtual machines. In the first group, the guest virtual machine runs Windows 7 Professional 32 bit. Our malware collection consist of 75 real world malware samples, including 10 stealth backdoors, 20 trojans, 15 adwares, 10 worms, 10 rootkits, and 10 virus. All of them are publicly available on the Internet (e.g., from web sites such as http://oc.gtisc.gatech.edu:8080/).

Category	Total	False Negative	DLL Path	Loading Order	RVA
Backdoor	10	0	8	6	8
Trojan	20	1	18	17	18
Adware	15	2	13	13	13
Worm	10	0	9	10	10
Rootkit	10	0	10	10	10
Virus	10	0	10	8	10

Figure 3: Summary of malware detection results on Windows 7 VM.

Figure 3 summarizes the detection results. Here 'False Negative' represents the number of malware that is missed by our detection mechanism. 'DLL Path' represents the number of malware that is detected based on anomaly in name/loading path of the DLL files. 'Loading Order' represents the number of malware that is detected based on the order in which the DLL files are loaded. 'RVA' represents the number of anomaly in hash results of the relative virtual addresses. For example, 17 out of 20 Trojan attacks are detected by our approach since they change the order in which DLL files are loaded into the system.

From Figure 3, we can see that our proposed mechanism is able to correctly identify most of the malware samples. There are three false negatives in our detection results. One of them is a Trojan that attempts to redirect our browser to another website. This Trojan is a JavaScript Trojan that does not have any DLL related behaviors. The other two are JavaScript adware. They attempt to display pop-up and pop-under advertisements when we are visiting some website in a JavaScript-enabled browser. The advertisements pop-up as separate windows to the active browser window so that they can bring additional profit to the designer. None of the missed malware demonstrates abnormal behaviors of a process or tries to infect DLL files. Therefore, they are not detected by our approach.

In the second group, the guest virtual machine runs Windows XP SP3 32 bit. From Figure 4, we can see that our proposed mechanism is able to correctly identify most of the malware samples. There are two false negatives in the detection results. One of them is a Trojan that changes a registry value. The computer is also showing an advertisement in the Yahoo Messenger chat window. Hence, its behavior resembles a benign application. If we click on that advertisement, it would download and execute a setup file that will run at every system boot-up. Our malware detection mechanism will catch it if this behavior shows up. The other false negative is an adware, which is a download manager. Every time a user wants to download a file from the internet, a window with advertisement will appear. However, no user data will be reused, stored, or shared. The reason that it is missed is because our malware detection mechanism can identify only the abnormal behaviors of a process, but not its intent or phishing. In real life, it is not difficult for a user to identify this type of advertisement.

Category	Total	False Negative	DLL Path	Loading Order	RVA
Backdoor	5	0	3	3	3
Trojan	10	1	9	8	9
Adware	5	1	4	4	4
Worm	5	0	4	5	5
Rootkit	3	0	3	3	3
Virus	4	0	4	3	4

Figure 4: Summary of malware detection results on Windows XP VM.

We have conducted a third group of experiments to assess the false positive mistakes of our approach. We download 60 benign and freely available applications that are different from our training set. These applications cover a wide range of different domains (such as browsers, audio players, video players, instant messaging, and security applications). From Figure 5, we can see that our malware detection mechanism causes no false positive mistakes.

	Total	False Positive	Browser	Audio	Video	Office	IM
Windows 7	15	0	3	3	3	3	3
Windows XP	15	0	3	3	3	3	3
	Total	**False Positive**	**Utilities**	**Graphic**	**Education**	**DVD/CD Tools**	**Security**
Windows 7	15	0	3	3	3	3	3
Windows XP	15	0	3	3	3	3	3

Figure 5: Tests for false positive mistakes of our approach.

4.3 Overhead and Performance Analysis

To protect a VM from malware infection, we need to run the proposed approach at the hypervisor level periodically. Since our detection mechanism needs to temporarily freeze a part of the memory in the VM, it will impact the operations of the guest VM. Therefore, we must study the relationship between the detection frequency and its impacts on the system performance. We conduct two sets of experiments to assess the impacts.

In the first group of experiments the guest VM is running CPU intensive applications. We choose two examples: (1) the *Fibonacci* benchmark that computes the fibonacci sequence; and (2) the *Prime* benchmark that generates prime numbers. Each of the software is running in parallel with the malware detection mechanism. When they are running in a virtual machine, the measured CPU usage is very close to 100%. The malware detection algorithm is running in the hypervisor. We measure changes in execution time of the software since this is the most intuitive parameter that end users adopt to evaluate the system performance.

In the second group of experiments the guest VMs are running CPU and memory intensive applications. We also choose two examples: (1) the *N-Queens* package that tries to generate all possible solutions to the *N-Queen* problem in chess; and (2) the *Combination* benchmark that computes all possible combinations of the input numbers and stores them in memory. We also measure their execution time when each of them is running in parallel with the proposed detection mechanism.

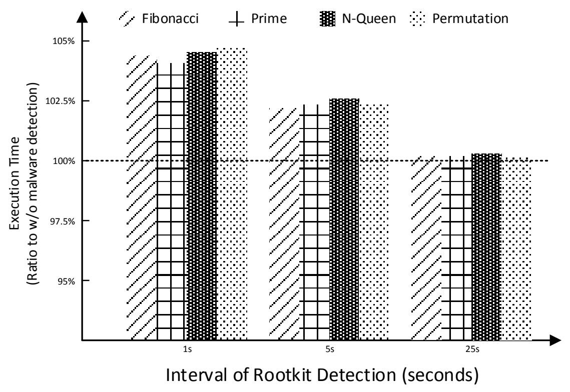

Figure 6: Relationship between malware detection frequency and its impacts on system performance.

From Figure 6, we find out that when we increase the interval between malware detections, the impacts on virtual machine performance are decreasing. When the interval is equal to 25 seconds, there is almost no measurable increases in execution time at guest VMs. Even when we execute the proposed approach every second, the increase in application execution time is less than 5%. At the same time, since our approach does not need a lot of memory from the VMs, the difference between the two groups of experiments is not large. Based on the results, we can see that our proposed approach has very low performance impacts on the VM.

We also study the relationship between the allocated memory to guest VMs and the impacts of our malware detection mechanism on system performance. In this experiment, we use CPU and memory intensive applications to assess the performance impacts. In each guest virtual machine with different memory allocation, we run the *N-Queens* package while we test the malware detection mechanism with different execution intervals. Here we allocate 1GB, 2GB, and 3GB RAM to the virtual machine. From Figure 7, we can see that the system performance stays almost the same as long as the intervals between malware detections do not change. In our melware detection mechanism, we extract only high level semantic information from the system data structures of the VMs. While malware detection is running at the hypervisor level, the differences in memory allocation size to guest VMs would not bring a huge difference to system performance since we do not conduct "brute force" memory scanning.

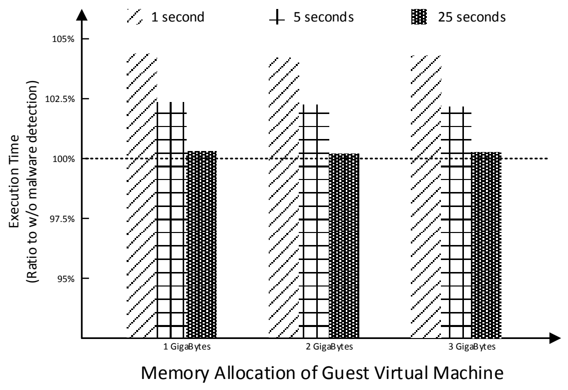

Figure 7: Relationship between memory allocation size and the impacts of our malware detection mechanism on VM performance.

To reduce performance impacts, in the proposed mechanism we examine only the RVA of the DLL functions. A more comprehensive detection mechanism would examine the contents of all read-only sections of DLL files. To compare the two schemes, we conduct a group of experiment and measure their execution time. The execution time measures only the hashing and comparison delay but not the loading time of the files. From Figure 8, we can see that the execution time of hashing all read-only sections of DLL files is about 3 times longer than that of examining only RVAs. When the total size of DLL files is about $350MB$, the execution time of RVA examination is about $250ms$, while it takes about $850ms$ to examine the whole read-only contents.

5. DISCUSSION

In this section, we will discuss a few potential extensions to our approach. We are especially interested in the tradeoff between detection capability and increases in overhead.

RVA only *vs* Read-only Content Examination

To reduce overhead of the proposed approach, in this paper we examine only the virtual addresses of the functions in the DLLs. The assumption is that if attackers inject malicious code segments into a DLL, the relative address will change. This assumption may not hold under some cases. For example, an attacker may apply compression algorithms to in-line overwriting contents so that the size of the malicious segments does not grow beyond the function boundaries [16].

To detect such attacks, we can use one of the following schemes. First, we could generate the hash result of the

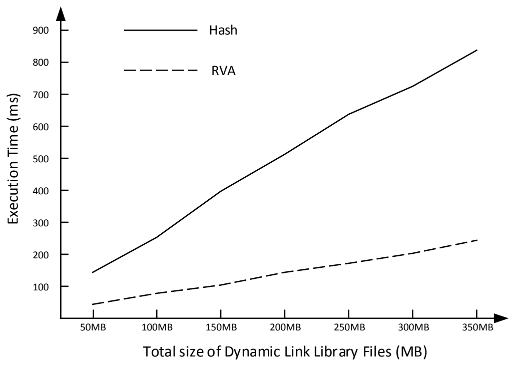

Figure 8: Comparison of hashing all read-only sections of DLL files with examining only RVAs.

read-only contents of DLL files during the learning phase so that any small changes could be caught. Our experiment results in Figure 8 show that we will have to pay for the increases in computation and memory access delay. Another mechanism is to scan the DLL files and try to locate the code segment for decompression [21]. This method has the same memory access overhead but avoids computation.

Diversity and Order of Loaded DLLs

Researchers have proposed several mechanisms to use function call graphs to detect malware [8, 5]. In this paper, we try to use information at a higher level of abstraction with the DLL names and their loading orders. One difficulty that we face is the diversity of the DLL functions. Very frequently the same operation can be accomplished by functions from different DLL files. Another challenge that we face is the usage of DLL files that are not directly related to the functionality of the application software. For example, an increasing number of applications will collect user information, encrypt it, and send it back to the company server so that user profiling can be conducted. Such operations also increase the difficulty of malware detection.

In this paper, we investigate the order in which the DLL files are loaded into the system and use it for malware detection. Here we do not differentiate tight dependency from loose dependency (several DLLs may switch their order of loading without impacting the software functionality). The lack of such knowledge may lead to false positive alarms. In the next step, we plan to investigate this problem and classify their dependency.

6. CONCLUSION

In this paper, we propose a lightweight malware detection mechanism for virtual machines. The hypervisor will collect, analyze, and monitor the execution states of virtual machines and detect compromised DLL files. In the experiments, we have evaluated more than 100 real world malware samples. We use both Windows XP and Windows 7 as test operation systems. Our experiment results show that the proposed approach is practical and effective. Furthermore, we conduct several groups of experiments to evaluate the increased overhead under different situations. The increases in overhead at virtual machines are very low since we access only a small portion of their memory pages through high level data structures.

Immediate extensions to our approach consist of the following aspects. First, we plan to experiment our approach with Linux virtual machines so that we can evaluate its practicability in other environments. Second, we plan to design innovative mechanisms to extract more high level information from virtual machines. The rich data set will allow us to better understand the difference between benign and malicious software. Finally, we plan to extend our approach to other hypervisors (such as KVM) so that more end users can benefit from our research.

7. REFERENCES

[1] Agobot. http://www.f-secure.com/v-descs/agobot.shtml, 2012.

[2] I. Ahmed, A. Zoranic, S. Javaid, and G. Richard. Modchecker: Kernel module integrity checking in the cloud environment. In *International Conference on Parallel Processing Workshops (ICPPW)*, pages 306–313, 2012.

[3] A. Alasiri, M. Alzaidi, D. Lindskog, P. Zavarsky, R. Ruhl, and S. Alassmi. Comparative analysis of operational malware dynamic link library (dll) injection live response vs. memory image. In *International Conference on Computing, Communication System and Informatics Management (ICCCSIM)*, 2012.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of ACM SOSP*, pages 164–177, 2003.

[5] A. A. E. Elhadi, M. A. Maarof, and A. H. Osman. Malware detection based on hybrid signature behavior application programming interface call graph. *American Journal of Applied Sciences*, 9(3):283–288, 2012.

[6] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *ACM CCS*, pages 128–138, 2007.

[7] A. Joshi, S. King, G. Dunplap, and P. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP*, 2005.

[8] J. Kinable and O. Kostakis. Malware classification based on call graph clustering. *Journal in Computer Virology*, 7(4):233–245, 2011.

[9] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. Kvm: the linux virtual machine monitor. In *Linux Symposium*, pages 225–230, 2007.

[10] P. F. Klemperer. *Efficient Hypervisor Based Malware Detection*. PhD thesis, CMU, May 2015.

[11] M. H. Ligh, A. Case, J. Levy, and A. Walters. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. Wiley, 2014.

[12] K. Makris and K. D. Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. *SIGOPS Oper. Syst. Rev.*, 41(3):327–340, 2007.

[13] M. Narouei. Mining modules dependencies for malware detection. www.kaspersky.com/downloads/pdf/masoud_narouei.pdf, 2012.

[14] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot: a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, 2004.

[15] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *ACM CCS*, pages 103–115, 2007.

[16] K. A. Roundy and B. P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Comput. Surv.*, 46(1):4:1–4:32, July 2013.

[17] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamze. Malware detection based on mining api calls. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 1020–1025, 2010.

[18] M. Shafiq, S. Tabish, F. Mirza, and M. Farooq. Pe-miner: Mining structural information to detect malicious executables in realtime. In E. Kirda, S. Jha, and D. Balzarotti, editors, *Recent Advances in Intrusion Detection*, volume 5758 of *Lecture Notes in Computer Science*, pages 121–141. Springer Berlin Heidelberg, 2009.

[19] A. Srivastava and J. Giffin. Automatic discovery of parasitic malware. In S. Jha, R. Sommer, and C. Kreibich, editors, *Recent Advances in Intrusion Detection*, volume 6307 of *Lecture Notes in Computer Science*, pages 97–117. 2010.

[20] B. Walters. Vmware virtual platform. *Linux J.*, (63es), 1999.

[21] T.-E. Wei, Z.-W. Chen, C.-W. Tien, J.-S. Wu, H.-M. Lee, and A. Jeng. Repef: A system for restoring packed executable file for malware analysis. In *International Conference on Machine Learning and Cybernetics (ICMLC)*, volume 2, pages 519–527, 2011.

[22] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5(2):32–39, March 2007.

[23] X. Xie and W. Wang. Rootkit detection on virtual machines through deep information extraction at hypervisor level. In *International Workshop on Security and Privacy in Cloud computing (SPCC), in conjunction with IEEE CNS*, 2013.

[24] M. Xu, X. Jiang, R. Sandhu, and X. Zhang. Towards a vmm-based usage control framework for os kernel integrity protection. In *SACMAT*, pages 71–80, 2007.

[25] Y. Ye, D. Wang, T. Li, D. Ye, and Q. Jiang. An intelligent pe-malware detection system based on association mining. *Journal in Computer Virology*, 4(4):323–334, 2008.

[26] H. Yin, Z. Liang, and D. Song. Hookfinder: Identifying and understanding malware hooking behaviors. In *NDSS*, 2008.

[27] A. Yu, Y. Qin, and D. Wang. Obtaining the integrity of your virtual machine in the cloud. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 213–222, 2011.