MwSandbox: On Improving the Efficiency of Automated Coarse-grained Dynamic Malware Analysis

Chengyu Song[1,2], Chao Qin[3], Jianwei Zhuge[1,2], Zhiyin Liang[1,2], Zhiyuan Ye[1,2]

1. Key Laboratory of Network and Software Security Assurance(Peking University), Ministry of Education, Beijing 100871, China

2. Institute of Computer Science and Technology, Peking University, Beijing 100871, China

3. First Research Institute of Ministry of Public Security of China, Beijing 100048, China

1. {songchengyu,zhugejianwei,liangzhiyin,yezhiyuan}@icst.pku.edu.cn

3. qinc@fri.com.cn

Abstract: Malware is a major threat to the cyber world and the number of unique malware samples captured by antivirus software venders is making an explosive growth in recent years. To improve the malware analysis efficiency, researchers have developed several automated coarse-grained dynamic malware analysis systems, including Norman Sandbox, CWSandbox and TTAnalyze, etc. However, these systems' analysis capabilities still cannot compare with the growth of malware, because they rely on heavy virtual machines to build malware execution environments. To further improve the efficiency, this paper analyzes the bottlenecks in these systems and proposes two mechanisms, *flash revert* and *VM fork* to reduce the found overheads. Experiments on an OS-level VMM based prototype implementation (MwSandbox) show that the efficiency is improved at least an order of magnitude without losing analysis quality.

Keywords: sandbox; dynamic malware analysis; virtual machine; API hooking

1 Introduction

Malware, e.g. Trojan horses, bots, worms, viruses and so on, is one of the major threats to the whole cyber world. Trojan horses are used to steal users' sensitive information like personal identity, online bank and online game accounts, while botnets are used as generic attacking platforms to spread spams, launch distributed deny-of-service (DDoS) attack, break passwords. Driven by the economic benefits brought by these malicious activities, many experienced programmers are joining this underground industry every day, and devoting their time to write more and more powerful malicious codes or even code generators. This in turn, makes an explosive growth of the malware pieces every year. According to the latest Symantec Internet Security Threats Report [1], in the last half year of 2007 the malware they detected was 2.36 times as was detected in the first half of 2007 and 6.71 times of last half 2006. The situation is even worse now as Rising has reported a nearly 12 times growth of malware samples they collected in 2008 than in 2007[2], and Trend Micro has reported more than a twenty-fold (2000 percent) increase in web threats between 2005 and 2008[3]. To combat with this growing threat, it is important that these antivirus (AV) software venders be able to 1) analyze the collected samples, 2) generate signatures for malicious samples and 3) dispatch the generated signature to their consumers as fast as possible. Among this process, analyzing collected samples is the most time consuming step.

Generally, computer program analysis can be roughly divided into two categories: static analysis and dynamic analysis. While static analysis is fast and complete - all execution paths can be covered - it is extremely difficult to analyze malware samples statically because almost every piece of malware captured nowadays is armored by one or more anti-analysis mechanisms [4-7], such as self-check, self-modify, encryption and packer. Dynamic analysis hereby becomes a hot topic among research groups and a feasible way for industry companies.

Dynamic analysis can be further divided into coarse-grained and fine-grained. Coarse-grained analysis usually provides information at operating system (OS) level (e.g. file creation, registry modification). Although it is possible to determine whether the analyzed sample is malicious or not only based on these information, it lacks certain information like how those malicious activities is

863 High-Tech Research and Development Program of China, under Grant No. 2006AA01Z445.
Research Fund for the Doctoral Program of Higher Education of China, under Grant No. 200800011019.
National Fundamental Science of First Research Institute of Ministry of Public Security of China, under Grant No A07313.

achieved which is necessary to develop the remove/recovery routine. Fine-grained analysis, on the other hand, is able to provide this kind of information. Ideally, every sample should be analyzed fain-grained so as to extract as much information as possible, but this requires a lot of time that is not affordable. Luckily enough, in reality, most collected samples are mutated variant of known malware and classifying analyzed samples to known variant based on coarse-grained analysis is possible[8, 9], which means, only a few novel samples are necessary to analyzed fine-grained, and the rest can be analyzed coarse-grained.

To overcome the drawbacks of manual analysis, several automated systems have been developed to handle coarse-grained task [10-13] and fine-grained task [6, 14-19] in the past few years. While these systems have greatly improved the analysis efficiency (CWSandbox is able to analyze more than 500 samples per day per instance), to win the war with exponentially growing malware (about 24,000 samples per day as indicated in [2, 3] and is still growing), it is necessary to further improve their efficiency, especially the efficiency of coarse-grained systems, because all samples have to be analyzed coarse-grained.

Roughly speaking, the procedure of automated coarse-grained analysis systems is: each instance maintains an execution environment, automatically takes a malware sample, executes the sample in the execution environment, records the sample's run-time behaviors and then generates a report. Since a potential malicious sample may compromise the underlying execution environment during the analysis, the execution environment can neither be reused nor be shared, otherwise either false negatives or false positives are likely to be introduced. Although most of these systems leverage the virtual machine technologies to provide and recover isolated execution environments far better than physical machine, since the virtual machine monitors (VMM, such as VMware[20], QEMU[21], Xen[22]) used in these systems are not designed for malware analysis, they are always too heavy and import several overheads that reduce these systems' efficiency.

Firstly the virtual machines (VM) they use are highly resource consuming thus cannot support many samples analyzed in parallel on one physical machine. Secondly, these dynamic analysis systems frequently need to restore the execution environment to an initial state. However, the recovery time is relatively long when compared with the whole analysis time. Thirdly, the maintenance and updating of the execution environment would cause a long off-line time because every execution environment is so isolated that they need to be handled separately, and there is likely to be tens, or even hundreds VMs when such systems are used practically to support daily malware analysis tasks.

Based on these facts, to further improve the efficiency of the coarse-grained analysis, we proposed two mechanisms: *flash revert* and *VM fork*. Flash revert reduces the recovery time by keeping a copy of the 'clean' volatile states in memory instead of only saving them on disk. VM fork reduces the memory consumption and maintenance overhead by letting the large amount identical volatile and non-volatile states be shared among execution environments.

We have implemented these two mechanisms on the basis of an OS-level virtual machine, Feather-weight Virtual Machine (FVM) [23]. By integrating the traditional dynamic behavior capture and analysis mechanisms into the improved VMM, we create a light-weight sandbox for dynamic malware analysis, MwSandbox. The evaluation result of this prototype implementation shows that these two mechanisms can improve the efficiency by about an order of magnitude without losing analysis quality.

In summary, this paper targets at improving the efficiency of automated coarse-grained dynamic malware analysis so the newly arrived samples can be analysis as soon as possible in a good enough manner. And it makes following contributions:

- We systemically analyzed the bottlenecks in current dynamic malware analysis systems that prevent them from scaling and how these bottlenecks can be eliminated.
- We proposed two mechanisms, *flash revert* and *VM fork* to reduce the overheads in the malware execution environment.
- We implemented a new automated dynamic malware analyzer MwSandbox based on Feath-

978-1-935068-01-3 © 2009 SciRes.

er-weight Virtual Machine which increases the number of parallel analysis on a single physical machine by about 10 times and decreases the sandbox recovery time by about 10 times.

2 Related Work

2.1 Dynamic Malware Analysis

Because most malware today leverage obfuscation technology to avoid AV detection and/or resist static analysis, dynamic malware analysis has made a huge progress in recent years. The first trend is to automate coarse-grained analysis. CWSandbox [12], the new emerged competitor to Norman Sandbox [10, 11], is a coarse-grained malware analysis tool based on user space API-Hooking and uses VMware Server to build its execution environment. TTAnalyze [13] is another coarse-grained malware analyzer, and is the first tool that realizes out-of-box system call monitoring by using a modified version of QEMU. The efficiency of these tools, though much better than manual analysis, is still constrained by the underlying execution environment (CPU emulator or VMware Server). Our work is to reduce these overheads and improve the overall performance.

The second trend is to automate the fine-grained analysis. Cobra [6] is a fine-grained malware analysis framework facilitated by stealth localized-execution and supports automated code tracing. Panorama [14] is a tool uses dynamic taint technology [24] to automatically detect and analyzing private information leaking. Hookfinder [15] is another tool powered with this technology to detect and analyze malware which has hooking behaviors. K-Tracer [16] and its preceding [17] uses a different approach to analyze kernel level malware. BitScope [18] and Moser et al's tool [19] are able to detect trigger-based behaviors in malware and perform multiple execution path analysis. These tools, while can provide better understand of the malware behavior, are much less efficiency than coarse-grained tools. Our system differs from these tool in we are only going to provide an analysis good enough to determine whether the sample is malicious or not and classify the sample. Some potential malicious behaviors (e.g. load kernel drivers) are not permitted in our system, but we will record these behaviors hence samples having these behaviors can be analyzed fine-grained latter. And though performance is not a primary requirement of these works, we believe their efficiency can be improved by our flash revert and VM fork technology.

2.2 Light-weight Virtual Machines

Feather-weight Virtual Machine [23], as indicated by its name, is a light-weighted OS-level VMM build upon Microsoft Windows, and has been used in area like vulnerability assessment [25], scalable web site testing; shared binary service for application deployment and distributed Display-Only File Server (DOFS) [26]. Due to several reasons (shown in Section 5), it is chosen as the base VMM of our sandbox.

Vrable et al. [27] proposed two techniques flash cloning and delta virtualization which are very similar to our solution. However, Potemkin is a honeyfarm, its VMM only emulate the execution behavior of dedicated honeypot hosts for a short periods of time. It requires modification to the guest OS and supports Linux only. More important, it lacks reliable clone and protection of disk device, which, however, is critical for malware analysis.

3 Problem Analysis

In this section, we systematically analyze the process of coarse-grained dynamic malware analysis to figure out the bottlenecks in current systems. Then we discuss possible ways to resolve these bottlenecks.

3.1 Coarse-grained Analysis

Given a malware sample, coarse-grained analysis means, executing the binary in an execution environment for a specific time (usually several minutes) and record its behavior during the execution. An execution environment, from the perspective of a running program (i.e. the process), is a memory space to store its code and data, and an application binary interface (ABI) to execute its code. For malware analysis, besides providing the target ABI, the execution environment must satisfy more requirements:

First, the environment should never let recorded behaviors be polluted with behaviors that do not inhabit in the sample binary. Otherwise a sample would be miss classified as malicious which in turn, will result in a false

positive in the detection system, and false positives are highly unacceptable in practice. It is also possible that a compromised execution environment prevents some malicious behaviors of a sample from appearing or being monitored, therefore causes false negatives. Since most ABI of the operating systems widely-used now is not strong enough to stop a malware from impacting other processes on the OS, to eliminate false positives and false negatives requires: (R1) an execution environment should never be reused before it is recovered to a clean state; (R2) no more than one sample can be analyzed in one execution environment.

Second, the environment should provide some interaction conditions to trigger certain behaviors of a sample. For example, a virus would require some certain types of file to trigger its infection behavior. This requirement implies: (R3) the environment should be updated from time to time so as to provide the interaction conditions required by the new emerged malware.

To satisfy these requirements, it is unavoidable that the efficiency of the analysis system suffers. However, some of the costs in present dynamic malware analysis systems are not necessary and the performance of such systems will be improved when these overheads are reduced.

3.2 System Recovery

To satisfy R1, besides the specific execution time, the whole analysis time of a sample must plus a recovery time. Since the execution time cannot be reduced, otherwise some malicious behavior may not be observed, one way to improve the analysis efficiency is to reduce the recovery time.

By replacing physical machine which requires dd[28] or Norton Ghost[29] to recover the disk with VM and leveraging the snapshot function provided by VMM, the recovery time can be reduced from several minutes to several tens of seconds (see Section 6.1 for detail). However, since the specific execution time is only 2-3 minutes, this recovery time still takes a relatively big portion of the whole analysis process, and this portion grows with the memory preserved for the VM and the load of the host machine.

The reason why reverting a VM is much faster is because the very slow disk operation is heavily reduced by protecting the disk image under an accumulation mode (i.e. old disk image is not overwritten after a snapshot is created). Therefore, recovering the disk image only requires deleting the new image. In fact, this disk recovery time is so short that most of the reverting time of a VM is used to recover the volatile states (memory, register, etc.) from disk. So if we could avoid loading the volatile state image from disk, we can further shorten the reverting time and make analysis system more efficiency.

3.3 Parallel Analysis

Although it is hard to reduce the whole analysis of a single sample, the average analysis time can be reduced by analyzing samples in parallel. And to satisfy R2, isolated execution environment is required. Since modern hardware is too powerful for single analysis, most analysis systems use virtual machines to improve the analysis capability. However, facing the exponentially growing malware, is it possible to run more VMs on one physical machine?

Intuitively, the answer is yes. In a dynamic malware analysis system, every VM is almost identical. Therefore, by decreasing the granularity VMM used to protect memory resources, the number of concurrently running VM can be increased.

3.4 Environment Update

Another limitation we found in present dynamic malware analysis systems is that, when the execution environment needs an update (R3), it will cause a long off-line time of the system. Because there could be tens or even hundreds of VMs in the system, and these VMs need to be updated separately because they are so isolated from each other.

But, similar to memory, most of the disk image content of different VMs, not only before the update but also after, is identical. Therefore, if only one disk image needs to be updated, the off-line time of the analysis system is reduced.

4 Solution

To overcome the bottlenecks discussed, we propose two mechanisms: *flash revert* and *VM fork*. In this section we describe these two mechanisms.

978-1-935068-01-3 © 2009 SciRes.

4.1 Flash Revert

To reduce the system recovery overhead, we propose a new snapshot and recovery mechanism called flash revert. Under this mechanism, besides the non-volatile states, the volatile states of the execution environment are also protected under an accumulation mode: 1) when a snapshot is made, the volatile states are saved both in memory (old image) and on disk (for cold recovery); 2) a new image is allocated and links the old image; 3) further changes to the volatile states are stored in the new image; 4) when read operation is issued, the VMM first tries to read from the new image, if fails, it then reads the content from the old image; and 5) after the analysis, the new image is discarded (freed) and the old image becomes the default image.

4.2 VM Fork

To reduce the memory resources consumption overhead caused by large granularity memory isolation mechanism, we propose a new isolation mechanism called VM fork, which is inspired by the memory isolation mechanism in modern operating systems. Under this isolation mechanism, the identical volatile states will be share between different VMs: 1) on a host machine, only one VM is booted normally from the disk image; 2) every other VM is forked from this VM and shares all the memory content with it in the beginning; 3) the farther VM and the forked VMs can read the shared memory, but if any of them tries to write the memory, the VMM performs a COW operation, and marks the new allocated space as private, the original memory content is still shared between the rest VMs; 4) no VM can read or write other VM's private space.

To reduce the system maintenance overhead, we extend the VM fork mechanism to not only protect the volatile but also protect the non-volatile states. To satisfy the prerequisite to share non-volatile states, the VMM stores the per-VM configuration in its configuration file and configures the VM properly when it is forked.

5 MwSandbox

In this section we describe the design and implementation of our automated dynamic malware analyzer, MwSandbox, which implements flash revert and VM fork.

5.1 System Overview

Our parallel dynamic malware analysis system is composed of three main components that collaborated with each other to accomplish the goals (Figure 1):

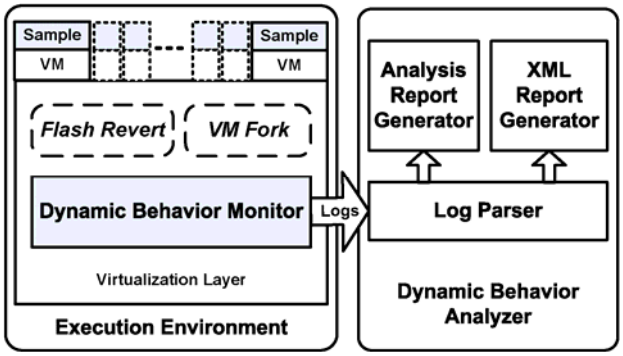

Figure 1 System Overview

Execution Environment. A light-weight execution environment for analyzing Windows based malware samples that implements the flash revert and VM fork.

Dynamic B ehavior Monitor. For capturing the system calls samples made during their execution and recording them into log files.

Dynamic B ehavior Analyzer. For extracting samples' system behaviors from the recorded system call logs and generating analysis reports.

5.2 Execution Environment

The execution environment of our sandbox is built upon the OS-level VMM, FVM [23]. The basic idea behind FVM is to use namespace isolation to isolate both volatile and non-volatile resources between host OS and guest VMs, and to use the COW strategy to share resources so as to make the VM feather weight. We choose it as the foundation of our sandbox based on three reasons:

Firstly, FVM is implemented as a kernel driver of the Microsoft Windows OS, which is also the target ABI of our execution environment.

Secondly, FVM has partially implemented the VM fork mechanism. Under FVM, the non-volatile states (file system and registry system) are shared and protected under the COW strategy between host OS and guest VMs.

But it lacks protection between guest VMs. For the volatile states, FVM satisfies most policies required by VM fork (listed in Table 1), except the protection of private spaces (both user and kernel) between VMs and the protection of kernel space memory.

Table 1 Memory Protection Policies Required by VM fork

User Space Memory	Shares all the content with the host OS when created; When is about to be modified, private physical memory page is allocated for this operation; All user space private pages cannot be accessed by processes in other virtual machines.
Kernel Space Memory	Shares all the content with the host OS when created; When is about to be modified, this operation is done inside the VM's own namespace; No process in virtual machines is able to modify kernel space memory or read private kernel space memory.

Thirdly, FVM shares a large common base with the Dynamic Behavior Monitor. Both of them use the API hooking technology and most of the hooked functions are identical. So it is easier to integrate the Monitor into the FVM VMM than into other popular system-level VMMs, such as VMware and QEMU. And integrating the behavior monitor component into the VMM layer has several advantages: it is harder to be detected, tampered or bypassed [30].

Therefore, to use FVM to build the required execution environment, we need to: 1) complete the implementation of VM fork mechanism; 2) implement the flash revert mechanism; and 3) strengthen the self-protection mechanisms in FVM.

5.2.1 Complete VM fork

To completely implement the VM fork mechanism, we need to protect the private resources (both non-volatile and volatile) from being accessed by processes in other guest VMs.

To protect files and registry keys in one VM's private namespace, we make the root of each VM's file and registry namespace the child of a certain directory, for example C:\Sandbox and HKLM\SOFTWARE\Sandbox. This directory is then hidden from processes in the VMs, and any attempt to access either the directory itself or items rooted from it would fail.

For memory resources, they are protected by four more policies listed in Table 2:

Table 2 Memory Protection Policies to Complete VM fork

Process Enumeration	Processes in other virtual machines are removed from the result of process enumeration
User Space Memory	Process cannot manipulate virtual memory of another process besides processes in the same virtual machine
Kernel Space Memory	Private namespace for kernel object are rooted at the same directory and hidden from access; Processes in virtual machines are not allowed to load kernel module.

5.2.2 Implement Flash Revert

In our sandbox, the flash revert is done by 1) forcing all the running processes in that VM to terminate, which will then makes the resources management functions in Windows to free all the user space memory allocated for these processes and all the kernel objects in the VM private; 2) deleting all the files and registry keys in the VM's private namespace.

4.2.3 Strengthen Security

FVM has some self-protection mechanisms, but they are not strong enough for malware analysis. So we add more mechanisms to strengthen the security of the virtualization layer.

- We hook system shutdown function to handle the system shutdown and restart requirement needs properly.
- We hook the system time related functions to prevent the system wide time from being modified maliciously. And the adjustment will be stored and applied when processes in that VM query system time afterward.
- We hook the token privilege adjust functions to avoid processes in the VM getting some critical privileges.
- We hook the kernel driver unload function NtUnloadDriver to avoid the VMM module being unloaded.

5.3 Dynamic Behavior Monitor

We use API hooking technology to accomplish the goal of gathering sample's coarse-grained dynamic behaviors (i.e. system activities). The Dynamic Behavior Monitor intercepts important system calls of Windows, and captures the related information when these functions are called within the sandbox. The captured infor-

978-1-935068-01-3 © 2009 SciRes.

mation is written to log _les, one for each sandbox. After the dynamic execution finishes, the corresponding log file is uploaded to the Behavior Analyzer to extract the behavior information. We decouple the logging and analysis so the Analyzer can be updated without modifying the sandbox, and different off-line analysis can be performed separately.

5.4 Dynamic Behavior Analyzer

The Dynamic Behavior Analyzer is in charge of extracting dynamic behaviors from the log _le. In our system, it has three components: one low level parser to recover semantic information of logged system call and provides a more convenient interface for high level analyzer; two high level analyzer, one generates a human friendly analysis report to help malware analysis experts classify the sample, and the other one generates XML based reports for other machine procedures.

5.5 System Implementation

The execution environment with the integrated Dynamic Behavior Monitor is implemented as two parts: one kernel driver which consists of about 22,000 lines of C code for the kernel level virtualization layer and kernel level of the Dynamic Behavior Monitor, and one DLL consisted of about 1,000 lines of Delphi (with the help of madCodeHook[31]) for the user level virtualization layer and the user level of Dynamic Behavior Monitor. (Although the FVM VMM is open-sourced earlier this year, we did not have the access to that code when we began the development of our system, so the entire sandbox is developed from scratch all by ourselves.) The three components of Dynamic Behavior Analyzer are implemented in about 1,000 lines of Python code. A sandbox controller is implemented in about 3,000 lines of Delphi codes.

6 Evaluation

In this section we present the evaluation result of our MwSandbox. The first part is the performance evaluation of our light-weighted sandbox; the second part is an effectiveness evaluation of sandbox; in part three, we demonstrate that our sandbox is able to evade some anti-VM detection; in the last part we present an overview of the result of a large-scale analysis we conducted on our prototype system.

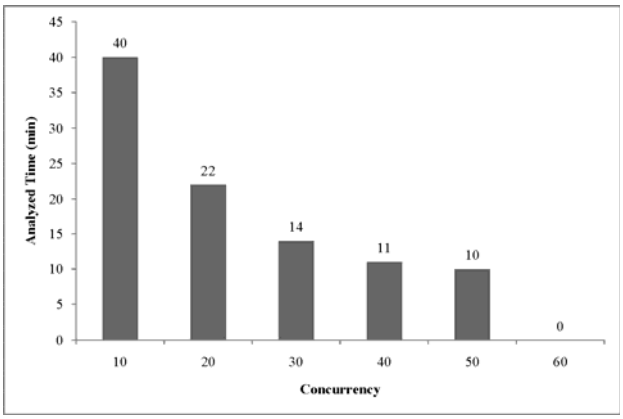

Figure 2 Time consumption for analyzing 200 samples concurrently (timed out when concurrency reaches 60).

6.1 Performance

Since our primary goal is to reduce the overheads in present automatic dynamic malware analysis systems thus improve the overall performance, we first conduct a performance evaluation of our light-weighted sandbox. The test-bed we use is a Core Duo 2 E6750 2.66G Dell Optiplex 755 PC with 2GB memory running Windows XP SP3.

6.1.1 Concurrent Analysis

We design this experiment as described below to find out how many sandboxes can run concurrently in our prototype system:

- We select 200 samples captured by our distributed honeynet[32] in December 2008 as the sample pool;
- To ensure every sample gains enough CPU cycles, we define a rough constraint: every operation (start, fetch sample, stop) must finish in 60 seconds;
- From 10, we increase the concurrent running analysis tasks once by 10 and record the overall time used to analyze the 200 samples. At every concurrency level, we perform the analysis three times then use the mean time (rounded to minutes).

The analyzing time is shown in Figure 2. From it we can imply that 50 (timed out when concurrency reaches 60) is the most suitable number of concurrent running tasks, which is about one order of magnitude higher compared with the number of VMs that VMware and QEMU can start without serious performance delegation.

This also implies that on our test bed we are able to analyze about 28,800 samples per day.

Theoretically, the memory requirement of a sandbox only consists of the memory used by processes in the sandbox and an additional 2MB used by the VMM[23], in our test bed, we should be able to start hundreds of sandboxes concurrently. However, in practice, since most of the malware samples are CPU-bounded programs, if we analyze too many samples concurrently, some sample may not get enough CPU cycles to finish its execution path. As a result, the quality of the analysis report will delegate. Besides, the synchronization and mutex overhead grows as the concurrent degree increases, this will decrease the overall throughput of the system. So we believe 50 is a reasonable concurrency number for our test-bed.

6.1.2 System Revert Time

In this experiment we test how fast one sandbox can revert to a clean state under different situation, and compare this result with the average revert time of VMware Workstation (v6.5) and QEMU (v0.9.1 without kqemu). The result is shown in Figure 3. For VMware Workstation and QEMU, the evaluation is done under Ubuntu Linux 8.04 32-bit version, each VM with 224MB memory and 8GB disk. All there systems run Windows XP. The reverting time of one sandbox of our system is the time elapses between the STOP command is send and the OK result is received. The reverting time of one VM of VMware Workstation is the time elapses between the stop command is issued and the start command is finished. The reverting time of one VM of QEMU is the time elapses between the start command is issued and Windows is logged in.

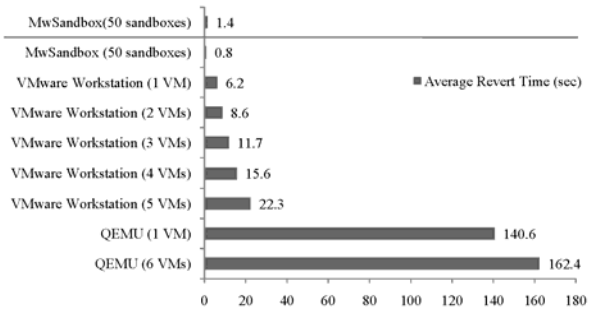

Figure 3 Average revert time of MwSandbox compares to VMware Workstation and QEMU.

This evaluation shows 1) the revert time grows with the concurrency number (the average revert time of VMware Workstation grows from 6.2 seconds to 22.3 seconds); 2) revert time of VMM that support non-volatile states snapshot (VMware Workstation, MwSandbox) is much faster than that does not (QEMU); 3) our system has greatly shorten the system revert time, even when 50 sandboxes are analyzing samples in parallel, the average revert time is only about 1/5 of reverting one VM of VMware Workstation, which does not analyze any sample.

6.2 Effectiveness

In this part we measure the effectiveness of our system, that is, whether the quality of the analysis report is damaged as a result of the performance improvement. The measurement is done by comparing the analysis reports generated by our system with the analysis reports we get from Norman Sandbox [33] and CWSandbox [34], two most famous dynamic malware analysis systems in the world, for the same sample. These samples are chosen randomly from our sample pool. The result is listed in Table 3, = means the report generated by our system is similar to that system, i.e. besides some random parts like file name, the analysis report from our system is identical to the report from the comparison system; + means our report is better, i.e. our system captures more dynamic behaviors than the comparison system, take Trojan.Win32.VB.heo as an example, our system captures two temporary file creation, two registry auto-start extensibility points (ASEP) creation and one HTTP connection, all of them are not captured by Norman Sandbox (no behavior is reported) nor CWSandbox; - means our report is not as good as that system, i.e. our system captures less dynamic behaviors than the comparison system; `No Report' means the report from that system is blank.

Table 3 Analysis reports compares to Norman Sandbox and CWSandbox

Sample Name	Norman Sandbox	CWSandbox
Backdoor.Win32.Nepoe.ej	=	No Report
Trojan.Win32.VB.heo	+	+
Net-Worm.Win32.Allaple.e	=	=
Virus.Win32.Virut.n	=	=
Backdoor.Win32.Rbot.aus	=	No Report
Virus.Win32.Virut.av	=	=
Net-Worm.Win32.Allaple.b	=	No Report
Virus.Win32.Virut.n	=	No Report
Backdoor.Win32.Rbot.gen	=	=
Virus.Win32.Parite.b	+	=
Backdoor.Win32.VanBot.ax	+	=

978-1-935068-01-3 © 2009 SciRes.

As the result indicated, the performance improvement is not a detriment to the effectiveness. Moreover, in some cases, the quality of our system is even better than those two systems, the reasons might be: a)the malware sample does not detects our system as an analysis environment; b)the Behavior Monitor is integrated into the virtualization layer (kernel space in our case), therefore it is harder to be bypassed or tampered.

6.3 Anti VM Detection

As most dynamic malware analysis are now performed in VM, more and more malware writers have added VM detection code in their malware or packers [35-39] to detect widely used VMM like VMware and QEMU. In this experiment, we present two samples captured in the wild by our distributed honeynet that detect VMware. We are curious about whether our system is able to pass these detection mechanisms.

One sample is a simple VMware detection program (we do not find any malicious behavior). If it is executed inside VMware Workstation, the result is shown in Figure 4 but if it is executed in our sandbox, it does not report such detection information, which means our sandbox can bypass this kind of detection.

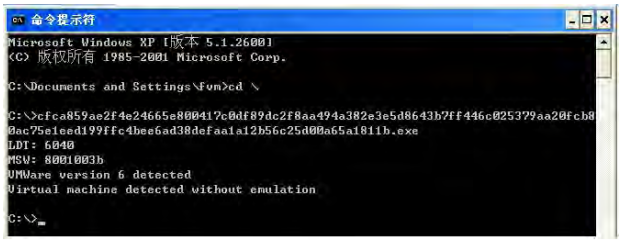

Figure 4 Sample captured in wild that det ects VMware

Another sample (Virus.Win32.Virut.a) detects the presenting of virtual machine environment or debugger. If runs inside VMware Workstation, it only create a executable file under C:\WINDOWS\system32. But if runs inside our sandbox, besides this file creation, it also creates two auto-start values under HKCU\Software\Microsoft\Windows\CurrentVersion\Run and HKLM\Software\Microsoft\Windows\CurrentVersion\Ru\Services, deletes the original sample and opens a backdoor at port 113.

6.4 Large-scale Long-term Analysis

We conducted a large-scale long-term test to further evaluate our system's efficiency and report quality. This test begins at April, 2008, every day we import samples captured by Matrix distributed honeynet[32] (each node is consisted of both low-interaction honeypot Nepenthese [40] and high-interaction honeypot HoneyBow [41] to capture self-propagation malwares) into our sample pool and let our prototype system analyze them. During the last nine months, we have analyzed about 44,000 malware samples.

7 Conclusion

We analyzed the performance bottleneck of current automated coarse-grained dynamic malware analysis systems, found out three main overheads imported by the underlying virtual execution environment. We proposed two mechanisms flash revert and VM fork, to reduce these overheads. We developed a new parallel dynamic malware analysis system MwSandbox to demonstrate our approach. The evaluation with malware samples we captured in the wild proves our system has improved the analysis throughput about 10 times than present systems without losing analysis quality, and has better ability to avoid current VM detection strategies used by the in-the-wild malware.

Acknowledgments

We acknowledge the VMware Academic Program for providing the VMware Workstation license.

References

[1] Turner, D., Fossi, M., Johnson, E., Mack, T., Blackbird, J., Entwisle, S., Low, M.K., McKinney, D., Wueest, C.: Symantec global internet security threat report. http://www.symantec.com/business/theme.jsp?themeid=threatreport (April 2008).

[2] Beijing Rising International Software Co., L.: Computer virus epidemic situation and internet security report for china mainland. http://it.rising.com.cn/new2008/News/NewsInfo/2008-11-18/1226970618d50435.shtml (November 2008).

[3] Inc., T.M.: Trend micro 2008 annual threat roundup and 2009 forecast. http://us.trendmicro.com/imperia/md/content/us/pdf/threats/securesecursecur/ (2009).

[4] Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: CCS '03: Proceedings of the 10th ACM conference on Computer and communications security, New York, NY, USA, ACM (2003) 290-299.

[5] Vasudevan, A., Yerraballi, R.: Spike: engineering malware anal-

ysis tools using unobtrusive binary-instrumentation. In: ACSC '06: Proceedings of the 29th Australasian Computer Science Conference, Darlinghurst, Australia, Australia, Australian Computer Society, Inc. (2006) 311-320.
[6] Vasudevan, A., Yerraballi, R.: Cobra: Fine-grained malware analysis using stealth localized-executions. In: SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy, Washington, DC, USA, IEEE Computer Society (2006) 264-279.
[7] Alsagoff, S.: Malware self protection mechanism. Information Technology, 2008. ITSim 2008. International Symposium on 3 (Aug. 2008) 1-8.
[8] Rieck, K., Holz, T., Willems, C., Dessel, P., Laskov, P.: Learning and classiffcation of malware behavior. In: Detection of Intrusions and Malware, and Vulnerability Assessment. Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2008) 108-125.
[9] Bayer, U., Comparetti, P.M., Hlauschek, C., Kirda, E., Kruegel, C.: Scalable, behavior-based malware clustering. In: NDSS '09: Proceedings of the 16th Annual Network and Distributed System Security Symposium. (February 2009).
[10] Natvig, K.: Sandbox technology inside av scanners. In: In Proceedings of the 2001 Virus Bulletin Conference. (September 2001) 475C487.
[11] Natvig, K.: Sandbox ii: Internet. In: In Proceedings of the 2002 Virus Bulletin Conference. (2002) 1-18
[12] Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using cwsandbox. IEEE Security and Privacy (2007) 32-39.
[13] Bayer, U., Moser, A., Kruegel, C., Kirda, E.: Dynamic analysis of malicious code. Journal in Computer Virology (August 2006).
[14] Yin, H., Song, D., Manuel, E., Kruegel, C., Kirda, E.: Panorama: Capturing system-wide information
ow for malware detection and analysis. In: CCS'07: Proceedings of the 14th ACM Conferences on Computer and Communication Security. (October 2007).
[15] Yin, H., Liang, Z., Song, D.: HookFinder: Identifying and understanding malware hooking behaviors. In: NDSS '08: Proceedings of the 15th Annual Network and Distributed System Security Symposium. (February 2008).
[16] Lanzi, A., Sharif, M., Lee, W.: K-tracer: A system for extracting kernel malware behavior. In: NDSS '09: Proceedings of the 16th Annual Network and Distributed System Security Symposium. (February 2009).
[17] Wang, Z., Jiang, X., Cui, W., Wang, X.: Countering persistent kernel rootkits through systematic hook discovery. In: Recent Advances in Intrusion Detection. Volume 5230 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2008) 21-38.
[18] Brumley, D., Hartwig, C., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Song, D., Yin, H.: Bitscope: Automatically dissecting malicious binaries. In: Technical Report CMU-CS-07-133. (2007)
[19] Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy (2007) 231-245.
[20] VMware, I.: Vmware: Virtualization via hypervisor, virtual machine & server consolidation. http://www.vmware.com/.
[21] Bellard, F.: Qemu, a fast and portable dynamic translator. In: ATEC '05: Proceedings of the Annual Conference on USENIX Annual Technical Conference, Berkeley, CA, USA, USENIX Association (2005) 41-41.
[22] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. SIGOPS Oper. Syst. Rev. (2003) 164-177.
[23] Yu, Y., Guo, F., Nanda, S., chung Lam, L., cker Chiueh, T.: A feather-weight virtual machine for windows applications. In: VEE '06: Proceedings of the 2nd International Conference on Virtual Execution Environments, New York, NY, USA, ACM (2006) 24-34.
[24] Newsome, J., Song, D.X.: Dynamic taint analysis for automatic detection, analysis and signature generation of exploits on commodity software. In: NDSS '05: Proceedings of the 12th Annual Network and Distributed System Security Symposium, The Internet Society (2005).
[25] Guo, F., Yu, Y., Chiueh, T.c.: Automated and safe vulnerability assessment. In: ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference, Washington, DC, USA, IEEE Computer Society (2005) 150-159.
[26] Yu, Y., Kolam, H., Lam, L.C., Chiueh, T.c.: Applications of a feather-weight virtual machine. In: VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, New York, NY, USA, ACM (2008) 171-180.
[27] Vrable, M., Ma, J., Chen, J., Moore, D., Vandekieft, E., Snoeren, A.C., Voelker, G.M., Savage, S.: Scalability, fidelity, and containment in the potemkin virtual honeyfarm. SIGOPS Oper. Syst. Rev. 39(5) (2005) 148-162.
[28] Unix: dd - convert and copy a file. http://www.opengroup.org/onlinepubs/009695399/utilities/dd.html.
[29] Symantec: Norton ghost. http://www.symantec.com/norton/ghost
[30] Jiang, X.,Wang, X.: "out-of-the-box" monitoring of VM-Based high-interaction honeypots. In: Recent Advances in Intrusion Detection. Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2007) 198-218.
[31] Rauen, M.: madcodehook. http://www.madshi.net/madCodeHookDescription.htm
[32] Zhou, Y., Zhuge, J., Xu, N., et al.: Matrix, a distributed honeynet and its applications. In: FIRST08: Proceedings of the 20th Annual FIRST Conference, British Columbia, Canada (2008).
[33] Inc, N.D.D.S.: Norman sandbox malware analyzer. http://www.norman.com/microsites/nsic/Submit/en
[34] Lehrstuhl f ur Praktische Informatik, U.o.M.: Cwsandbox - automated malware analysis. http://www.cwsandbox.org/?page=submit
[35] Liston, T., Skoudis, E.: On the cutting edge: Thwarting virtual machine detection. http://handlers.sans.org/tliston/ThwartingVMDetection Liston Skoudis.pdf (July 2006).
[36] Raffetseder, T., Kruegel, C., Kirda, E.: Detecting system emulators. In: Information Security. Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2007) 1-18.
[37] Zeltser, L.: Virtual machine detection in malware via commercial tools. http://isc.sans.org/diary.html?storyid=1871 (November 2006).
[38] Holz, T., Raynal, F.: Detecting honeypots and other suspicious environments. Information Assurance Workshop, 2005. IAW '05. Proceedings from the Sixth Annual IEEE SMC (June 2005) 29-36.
[39] Chen, X., Andersen, J., Mao, Z.M., Bailey, M., Nazario, J.: Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In: DSN '08: The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. (June 2008).
[40] Baecher, P., Koetter, M., Holz, T., Dornseif, M., Freiling, F.: The nepenthes platform: An efficient approach to collect malware. In: Recent Advances in Intrusion Detection. Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2006) 165-184.
[41] Zhuge, J., Holz, T., Han, X., Song, C., Zou, W.: Collecting autonomous spreading malware using high-interaction honeypots. In: Information and Communications Security. Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2008) 438-451.

978-1-935068-01-3 © 2009 SciRes.