BareBox: Efficient Malware Analysis on Bare-Metal

Dhilung Kirat
University of California, Santa Barbara
dhilung@cs.ucsb.edu

Giovanni Vigna
University of California, Santa Barbara
vigna@cs.ucsb.edu

Christopher Kruegel
University of California, Santa Barbara
chris@cs.ucsb.edu

ABSTRACT

Present-day malware analysis techniques use both virtualized and emulated environments to analyze malware. The reason is that such environments provide isolation and system restoring capabilities, which facilitate automated analysis of malware samples. However, there exists a class of malware, called VM-aware malware, which is capable of detecting such environments and then hide its malicious behavior to foil the analysis. Because of the artifacts introduced by virtualization or emulation layers, it has always been and will always be possible for malware to detect virtual environments.

The definitive way to observe the actual behavior of VM-aware malware is to execute them in a system running on real hardware, which is called a "bare-metal" system. However, after each analysis, the system must be restored back to the previous clean state. This is because running a malware program can leave the system in an instable/insecure state and/or interfere with the results of a subsequent analysis run. Most of the available state-of-the-art system restore solutions are based on disk restoring and require a system reboot. This results in a significant downtime between each analysis. Because of this limitation, efficient automation of malware analysis in bare-metal systems has been a challenge.

This paper presents the design, implementation, and evaluation of a malware analysis framework for bare-metal systems that is based on a fast and rebootless system restore technique. Live system restore is accomplished by restoring the entire physical memory of the analysis operating system from another, small operating system that runs outside of the target OS. By using this technique, we were able to perform a rebootless restore of a live Windows system, running on commodity hardware, within four seconds. We also analyzed 42 malware samples from seven different malware families, that are known to be "silent" in a virtualized or emulated environments, and all of them showed their true malicious behavior within our bare-metal analysis environment.

ACSAC '11 Dec. 5-9, 2011, Orlando, Florida USA

Keywords: bare metal, dynamic malware analysis, system restore, VM-aware

1. INTRODUCTION

The use of virtualized environments is ubiquitous in malware analysis. The ability to isolate and quickly restore the system to a known configuration after an analysis run are two key features of virtualized environments that facilitate malware analysis. Furthermore, during the first quarter of 2011 alone, McAfee Labs have identified more than six million unique malware samples [1]. This ever-increasing flood of new malware demands high performance analysis frameworks that can inspect as many malware samples as possible within a given period of time. Virtualization-based solutions have been an obvious choice. Unfortunately, malware authors have developed a new class of malware, called VM-aware malware, which can detect virtualized or emulated environments and subsequently change its behavior to foil the analysis.

The environment detection techniques used by virtual machine (VM) and emulation-aware malware are well-known and well-described in the literature [2, 3]. These techniques exploit some of the software or hardware artifacts introduced by the virtualization or emulation layer between the running operating system and the hardware. For example, some of the detection techniques identify in-guest, system-specific configurations or specific I/O ports, while others rely on inaccurate system emulation or sophisticated time-based detection attacks [2,4]. For instance, a system running inside a VMware guest OS can simply inspect the names of the available virtual devices, read the magic I/O port (`0x5658, 'VX'`), or simply read the value of LDTR or IDTR registers (using unprivileged SIDT or SLDT instructions), which will be different from the known values found on a bare-metal system [5]. All of these detection techniques can be executed in *usermode*.

More transparent, hardware-assisted, virtualization-based malware analyzers, such as Ether [6] and Azure [7] have been proposed. Although these systems reside outside of the guest OS, they share the underlying hardware CPU with the virtualized guest OSes. As a result, all privileged instructions must be properly intercepted and virtualized by the virtual machine monitor (VMMs). This consumes more CPU cycles per each instruction, allowing the detection of the analysis system. For example, certain detection techniques can measure the difference of execution times of privileged instructions between real and virtualized machines [3,4] or the ratio of the execution times between privileged and unprivileged

instructions in a single virtualized machine [2]. It was proposed that timing attacks can be thwarted by intercepting timing instructions and providing disguised timing information. However, Garfinkel et al. [8] have argued that this is not always possible, and indeed nEther [4] has demonstrated in-guest detection of the Ether platform using local timing attack.

Another approach to transparent malware analysis is based on the whole-system emulators, such as QEMU [9]. In theory, one should be able to correct all CPU emulation bugs in the emulators to avoid detection based on inaccurate system emulation. For instance, Anubis [10], which is based on an instrumented QEMU, randomizes hardware IDs and keeps patching known CPU emulation bugs. However, it is always possible that new bugs will be found, resulting in an arms race. In fact, Paleari et al. [11] have demonstrated an automated technique to produce thousands of such attacks to detect CPU emulation inconsistencies.

There has been a considerable amount of research focused on the detection of environment-sensitive malware by identifying differences in their execution behavior within a virtualized system and a *reference system* [12–14]. The reference system is ideally an OS running on real hardware, inside which VM-aware malware samples show their true malicious behavior. However, all previous systems have used either virtualized or emulated systems as the reference systems to evaluate their techniques. A sophisticated malware can detect all such incomplete realizations of the reference systems and hide its malicious behavior, showing no difference in their execution behavior [12].

The definitive way to observe the actual behavior of any unknown malware is to execute the sample in a real-hardware-based system, also called a bare-metal system. This approach entirely eliminates the possibility of modified behavior based on the detection of a virtual environment. However, this approach poses several practical challenges. One of the critical challenges is the ability to *efficiently* restore the analysis environment into the initial *clean state* after each malware infection. Although there exist both hardware and software-based solutions that restore the hard disk state, these solutions require a system reboot [15, 16]. System reboots are inherently slow as they are heavily I/O bound. Starting from the initial execution of the Power-On Self-Test (POST), several devices need to be initialized by the BIOS, and the corresponding drivers initialized by the operating system. Also, other OS components, including the malware analysis parts, have to be reinitialized after every reboot. In our experience, a typical system on a commodity hardware takes more than 15 seconds to reboot. We believe that the downtime required for a reboot is a major impediment towards the deployment of automated malware analysis frameworks on bare-metal.

In this paper, we propose a bare-metal analysis framework based on a novel technique that allows reboot-less system restoring. This framework, called BareBox, executes and monitors malware run in a native environment. After an analysis run has been completed, the running OS is restored back to the previously-saved clean state on-the-fly, *within a few seconds*. This restoration process recovers not only the underlying disk state but also the volatile state of the entire operating system (running processes, memory cache, filesystem cache, etc). Our restoration system consists of a overlay-based volatile mirror disk, a symmetric physical memory partition, and a small custom-written operating system, which we call *Meta-OS*. *Meta-OS* restores the entire physical memory of the running operating system once a malware analysis run completes. Because of the circular dependencies between the physical memory and the CPU context, it is practically impossible to perform a complete OS restore from within the OS that is being restored. To resolve this problem, *Meta-OS* provides *out-of-OS* execution control. The operation of other devices and peripherals also closely depend upon the state of the main physical memory. Careful handling of the device states is required to safely restore the state of the OS.

As the majority of malware targets Windows systems, we have chosen an x86-based Windows XP system as the target for implementing our prototype. However, the concept can be easily extended to other operating systems.
This work makes following main contributions:

- BareBox can efficiently profile the true behavior of malware by actually executing it in a native hardware environment.
- BareBox can restore a running operating system on commodity hardware back to its previously-saved state faster than any existing solution (within four seconds on average). The reason is that no reboot is required.
- Restoring the volatile state of the operating system (running programs, memory cache, filesystem cache, etc) makes it possible to quickly reproduce identical system states for repeated security experiments in a bare-metal environment. This is trivial for a virtualized system, but required the development of a small, custom OS to perform this on bare metal.

2. GOALS AND CHALLENGES

In this section, we describe the main goals and challenges of the proposed system.

2.1 Goals

No virtualization.

Our primary goal is to develop a bare-metal framework for analyzing malware that does not rely on virtualization or emulation techniques, such that all of the VM detection techniques used by malware are rendered ineffective. This excludes the possibility of utilizing hardware assisted virtualization such as the Intel VT and AMD-V technologies.

Efficient environment restore.

The environment has to be restored after each malware analysis run. Speed and efficiency of the restoration technique largely affects the effectiveness and throughput of the entire analysis framework. We want to restore the environment as fast as possible to the point where the next analysis can be initiated.

Malware profiling.

The system should be able to monitor a malware execution in the bare-metal environment. In particular, monitoring should include the interactions between the malware program and the operating system.

2.2 Challenges

In this section we describe some of the major challenges when trying to analyze malware on a bare-metal system.

Isolation.

Although the whole purpose of the bare-metal analysis system is to create an environment that closely resembles a "real" system, there are unavoidable risks when executing unknown malicious code on real hardware. With direct access, there is the possibility for serious system-level corruption, for instance, by malicious updates to the BIOS. Complete isolation of BIOS and peripheral devices, such as the hard disk, is not as trivial as it is in a virtualized system.

Stealthiness.

Bare-metal based analysis systems are definitely stealthier than any VM/emulation-based analysis systems with respect to VM-aware malware. However, the analysis component must reside inside or next to the operating system running on bare metal to extract analysis information. Theoretically, complete transparent analysis is impossible if the malware runs at the same privilege level as the analyzer [6]. One practical approach is to be undetectable from user-mode malware by implementing stealth rootkit techniques or by employing security by obscurity (for instance, randomizing the signature and/or the location of the analysis components). Another approach can be a *black box* approach where a malware sample is executed on bare metal without the presence of any in-guest analysis component. In this case, the analysis focuses on inspecting network traffic from outside and later checks for changes in the system state and configuration. Although this approach is stealthy, a lot of the contextual and dynamic information about the behavior of a malware sample is lost.

System Restore.

Without virtualization, options available for fast and efficient system restore solutions are limited. Most of the available solutions only restore the operating system's persistent state by restoring the hard disk, which then requires a system reboot to complete the restoration process. System reboot is heavily I/O bound, which causes long downtimes between subsequent analysis runs. Restoration of BIOS and other device firmwares are yet another challenge.

3. SYSTEM OVERVIEW

3.1 Threat Model

Without any in-guest presence, it is very difficult to perform detailed dynamic malware analysis in a non-virtualized, bare-metal environment (unless some specialized external hardware instrumentation is devised to directly monitor the memory and CPU). As our goal is to analyze malware in a bare-metal system on commodity hardware, we consider this inevitable trade-off of in-guest-presence as acceptable.

Malware running at ring0 can always detect the presence of BareBox analysis component using a memory scan. By forcefully mapping the reserved extended area of the physical memory, such malware can even corrupt the Meta-Os or the saved snapshot. However, mapping of reserved physical memory area is unusual and can be considered suspicious. To this end, the BareBox framework focuses only on the analysis of user-mode malware by disabling new kernel module loading, and preventing usermode access to kernel memory. The framework hides its presence from the user-mode malware using rootkit-like techniques. As effective restrictions for loading arbitrary kernel modules became operative in recent operating systems, such as Windows 7, an overwhelming portion of the present-day malware is user mode malware.

Attacks using return-oriented techniques [17] or kernel-exploits can potentially bypass the restrictions. In such cases, the most effective recovery would be an off-line disk-restore based recovery. As of now, an internal timeout for each analysis run is enforced, whose expiration forces a trap into the *Meta-OS* to perform a physical memory clean up.

3.2 General approach

Our approach to quick system restore is based on the symmetric partition of mutable resources such as disk and memory. In this scheme, one of the partitions is used to preserve a *snapshot* of the system's state. A *snapshot-save* operation creates a restorable state of the running operating system. This state can be restored back to a previous snapshot by a *snapshot-restore* operation. The target OS is installed on the main disk, and a disk identical to the main disk is attached to operate as mirror. Access to the disk is controlled by a software-based overlay mechanism that selectively redirects sector read and write operations to either the main disk or the mirror disk.

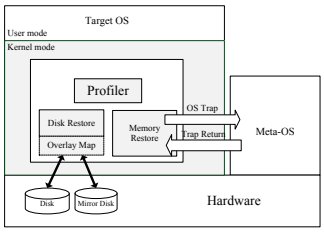

Figure 1: Architecture Overview

The physical memory is partitioned in such a way that one of the partitions can be used as a snapshot area. The other partition stores the target OS, where the actual malware is executed. During a *snapshot-save* operation, the entire physical memory corresponding to the target OS is copied over to the snapshot area. Live state of the running operating system is restored by restoring the physical memory of the operating system from the previously-saved snapshot. With this approach, only one snapshot point can be saved at a time, but the system can be restored back to this snapshot as many times as needed.

Complete restore of the running operating system involves the overwriting of critical OS structures residing in physical memory (e.g., the page table, interrupt handlers, etc.) and the restoring of the CPU context, including system-critical registers (e.g., IDTR, GDTR and other control registers). These registers are highly coupled with the content of the physical memory. We explain in detail in Section 4 why this restoration process can only be done from an *out-of-OS* execution. We implemented a small operating system, called *Meta-OS*, that resides outside the physical memory of the target operating system. *Meta-OS* creates the physical memory snapshot of the target OS and later restores it.

4. IMPLEMENTATION DETAILS

4.1 Disk Restore

A disk restore simply requires canceling all changes made to the disk by write operations. When we cancel only those changes that are made after a certain point in time, called the snapshot point, we effectively restore the state of the disk to that particular snapshot point. We achieve this by proper redirections of read and write operations to the main and the mirror disks. A *snapshot-save* operation simply implies that all further write operations to the main disk are redirected to the mirror disk. With this redirection in place, we effectively freeze the contents of the main disk. However, all read operations are still forwarded to the main disk, except those read operations to particular sectors that were previously redirected and written to the mirror disk. Such reads to "dirty sectors" are served from the mirror disk to reflect the changes made to the main disk after a *snapshot-save* operation.
We implemented two methods for the storage of the mirror disk; a RAM disk and a physical hard disk.

RAM disk.

In this technique, the mirror disk is implemented as a RAM disk so that no external hardware is required. The maximum amount of changes to the main disk is limited by the size of the RAM disk, which in turn, depends on the size of the available system memory. Although not suitable for lengthy analysis sessions that generate large amounts of disk write operations, this technique is very effective for analyzing malware that is executed for short periods of time.

Hard disk.

For this implementation, two identical disks are required to be installed in the machine, similar to a RAID1 configuration. One disk contains the OS, while the other disk (the mirror disk) is used as a temporary storage. As the disks are identical, they can have a sector-level one-to-one mapping, allowing us to redirect any number of write operations from the main disk to the mirror disk.

An overlay map for the book-keeping of the disk write operations is required to selectively redirect read operations for dirty sectors on the main disk to the mirror disk. This overlay map needs to be in the memory for efficient disk I/O. However, the size of the map gets quite large even if we use a single bit per sector mapping (e.g., a 256 GB drive with 512-byte sector size, requires 64 MB overlay map). To reduce the size of the map, we aggregate consecutive sectors into chunks. The granularity of the dirty-sector-write is a chunk, instead of individual sectors. We found that the average number of sectors accessed at once during a normal usage is 80 sectors or 40 KB. We chose a chunk size of 32KB for efficient bit-shift calculations of chunk locations. For a 256 GB drive with a 32KB chunk size, our mechanism only requires 1MB for the overlay map.

To keep the consistency of the underlying filesystem, the OS is forced to flush its file-system-cache before taking a snapshot. We perform this by sending appropriate device control codes to all logical volume devices. Disk operation redirection is implemented by intercepting the low-level disk driver major functions. Every access to the disk from higher-level drivers, such as the file system driver, volume and partition manager, page-fault memory manager, etc., are directed through this disk driver.

4.2 Memory Restore

The disk restore component can restore the disk image on-the-fly by simply resetting the overlay map. Unfortunately, this is not enough. The modified state of the disk is still cached within several data structures of the operating system in memory, file system structures being the most prominent. Also, there can be several open file handles or memory-mapped files related to new files created after a snapshot point. Therefore, an on-the-fly restore of the underlying disk will induce serious inconsistencies between the in-memory filesystem structure and the actual in-disk structures, resulting in data corruption. This means that one cannot simply restore the disk on-the-fly. Instead, a system reboot is required for a clean restore. As a result, volatile state of the running processes is lost after the reboot. However, our approach to physical memory restoration makes the rebootless restore of a bare-metal system possible.

4.2.1 Physical memory partitions

The available physical memory is partitioned into three parts. The operating system is loaded into the first part, which starts from the absolute hardware address zero. The second part of the physical memory is used to take a snapshot of the first part. Finally, the small operating system *Meta-OS* resides in the third part of the physical memory. We implemented this component as a kernel module that is loaded into the target OS.

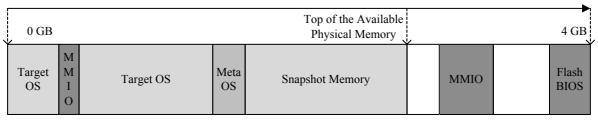

Figure 2: Physical memory allocation

In the x86 architecture, the usable physical memory address space is not entirely contiguous. Two types of devices are mapped into the processor address space - actual physical memory devices (RAM) and other memory-mapped I/O (MMIO) devices such as AGP and PCI cards. For 32-bit compatibility, the BIOS has to map such devices within the 4GB addressable range of the processor address space. In a machine with 4GB of RAM memory, the MMIO device mapping creates an unusable region of the physical memory usually known as the "PCI hole". *Meta-OS* needs to resolve such resource conflicts by carefully relocating and resizing the memory partition. This leads to effectively four types of memory sections: The target OS memory, the *Meta-OS* memory, the snapshot memory, and the hardware-reserved memory section (shown in Figure 2).

4.2.2 Need of out-of-OS execution control

When the operating system is fully initialized, the CPU is in protected mode with paging enabled. Memory restoration needs to be performed when the CPU is in this state. Before we explain the details about the memory restore mechanism, we briefly review the basics of the protected mode CPU execution.

When the processor is in protected mode with paging enabled, every single memory reference made by the executing instruction is treated as a virtual address, which must be

translated into a physical address using the Global Descriptor Table (GDT) and the page table. These tables reside in memory, and the processor knows their locations from two registers, the GDTR and CR3, respectively. For example, if the memory reference is made for the code segment to execute an instruction, the CPU needs to first find the offset of that segment. Unlike in real mode, during protected mode execution, finding the segment offset is a two-step process. The CS segment register only provides an index into the GDT table, and the actual segment offset is retrieved from the corresponding entry in the GDT. Once the effective virtual address is computed by adding the segment offset, it is translated into the physical address using the page table. The appropriate page table is retrieved from the page directory, whose physical address location is stored in the CR3 register. Note that the location of the GDT stored in the GDTR resister is actually a virtual address, which must first be translated into a physical address to access contents of the GDT table for the initial segment offset calculation.

To copy and restore the physical memory, it must be first mapped into virtual memory using page tables, as this is the only way of accessing memory while the CPU is in protected mode. In the x86 architecture, restoring the physical memory becomes challenging because it involves overwriting both the GDT table and the page table, which, in turn, are used to translate virtual addresses into physical memory locations. This includes the current location (virtual address) of the code (EIP) that performs the memory restoring. That is, the memory restore code would change memory mappings that interfere with its own execution. Because of these circular dependencies, it is impossible to restore physical memory of a live operating system from within the same operating system (with arbitrary physical memory content). For this reason, we have implemented a small operating system, called *Meta-OS*, as a memory restore component that resides outside of the physical memory of the target OS.

4.2.3 Meta-OS

Meta-OS is loaded into the extended area of the physical memory. This area of the memory is marked as unavailable (reserved) to the target OS during the boot time. Thus, the *Meta-OS* loader has to directly modify the page tables to forcefully map this area of the physical memory. A specific software interrupt or a device I/O control request triggers the *snapshot-save* and *snapshot-restore* operations. These operations involve operating system context switches from the target OS to the Meta-OS and back to the target OS. Interrupts are disabled during this process, and non-maskable-interrupts (NMI) are simply passed while the CPU is executing in the context of the *Meta-OS*. During any other time, *Meta-OS* remains dormant.

4.2.4 OS context switch

An OS context switch involves switching the Segment Descriptor Table (SDT), the Interrupt Descriptor Table (IDT), the page table, and other CPU register contexts. The final execution control transfer is implemented using the x86 interrupt handling mechanism. As both operating systems are running in the same privilege level (ring0) with separate Interrupt Dispatch Tables (IDTs), once the execution control is transfer from one OS to the another, the previous OS will have no way to force the control transfer back to itself. Thus, all OS context switches are voluntary.

SDT and IDT pointers are stored in the GDTR and IDTR registers, which can be modified simply using LGDT (Load GDT) and LIDT (Load IDT) instructions, respectively. A page table switch is achieved by modifying the CR3 control register, which holds the physical address of the page directory [18]. The page table switch operation forces the CPU to flush the TLB and the instruction pipeline cache. At this point, the new page table must contain a correct mapping for the virtual addresses of at least the GDT and the instruction pointer (EIP). A similar situation occurs when switching the CPU from physical to virtual address mode and during process context switches. Physical to virtual address mode-switch is typically handled by a one-to-one mapping between physical to virtual addresses. In case of the process context switches, OS maintains an identical kernel page map for each process. We follow the strategy of process context switch and copy the current page table to the *Meta-OS* page table before switching to it.

Switching back from *Meta-OS* is trivial after a *snapshot-save* operation because both page tables are identical. However, this is not the case while switching back after a *snapshot-restore* operation. Since page tables are not identical, simply switching back to previous table will cause the CPU to reference an invalid GDT structure, resulting in a hard reset of the CPU. Meta-OS handles this situation using a combination of the following techniques.

GDT relocation.

Before switching to the target OS page table from the *Meta-OS* page table (after a *snapshot-restore* operation), we have to make sure that the page table transition will be successful. Since the page table of the target OS is the restored version of the previous snapshot, we are not allowed to modify them in a way such that current GDTR will eventually point to a valid structure. Instead, one has to relocate the current GDT of *Meta-OS* to the previously-saved GDTR address of the target OS. We do this by first mapping the physical address of the *Meta-OS* GDT to the previously-saved target OS GDTR address (a virtual address), and then switching the GDTR to that address. By doing this, we guarantee that even after switching to a previously-saved page table with arbitrary mappings, the GDTR will still point to the proper descriptor table.

Segmented Address Space.

Although, GDT relocation technique maintains the proper descriptor table after the page table switch, the instruction pointer will still hold the address of the *Meta-OS* address space. In this case, the next instruction execution depends on where the restored page table of the target OS map this address to. Since there is no atomic instruction to simultaneously update both CR3 and EIP registers, this transition has to be handled incrementally. Note that the restored descriptor table and the page tables are in fact the same tables that were copied to *Meta-OS* page tables during the previous *snapshot-save* operation. However, the *Meta-OS* code segment might be mapped to different virtual address at each OS context switch. *Meta-OS* uses segmented memory address model for its code segment, such that the instruction pointer becomes an offset address, rather than the actual virtual address. By this approach, even after switching to an old page table having different base address for *Meta-OS*, the instruction pointer (EIP) address is still translated into

correct *Meta-OS* instructions. These instructions complete the proper execution control transfer to the restored target OS.

Since we disable interrupts during snapshot operations, we cannot use the interrupt-based DMA controller for copying physical memory. Instead, we achieve fast memory copy by enabling the cache attributes in the hardware page tables that are used to map the physical memory into a linear virtual address. Even if the TLB miss rate is high as *Meta-OS* swipes across the address space, this type of cache-enabled page mapping helps to fully utilize the L1 and L2 CPU caches during memory copy operations. In our case, this technique improved the performance by more than ten times. We achieve additional performance improvement by deploying the pre-unrolled execution of the memory copy instructions [19]. The main idea of this technique is trying to fill the CPU instruction pipeline with multiple memory move instructions.

4.3 Device Restore

Beside disk and memory restore, a complete restore of a live running system requires that all its devices are restored to the previous snapshot state as well. This is also necessary because the state of some devices depends on the contents of the physical memory. This requirement is difficult to satisfy, as the exact definition of "device state" is device-specific. To deal with this problem, we manipulated the device power state, which in turn forces a reset to the internal device state. A device is set to *Sleep Power State* (or *Off Power State* if the device does not support the sleep state) before *snapshot-save* operation is started. Its power state is restored back after the snapshot process completes. Similar power state manipulations are performed before and after the *snapshot-restore* operation. This way, instead of saving and restoring the internal states of the devices, which is in fact challenging, we forcefully reset/restart those devices.

Manipulation of the device power state is a time-consuming process, while one of our goals is to achieve the fastest possible system restore. By assuming that certain devices do not change their state during the analysis, we could exclude such devices from power state manipulation and considerably speed up the restore process. However, this assumption does not hold for all PCI devices, such as network adapters. In our selective device restore experiments, we could safely exclude several PCI devices, such as the hard disk controller, PCI bridge, memory controller, SMBus controller, graphics controller etc. Although we skipped the power state manipulation of the graphics controller, each *snapshot-restore* operation forces a video memory refresh operation, if the video-memory is mapped to the physical memory. However, to this end, we do not restore the internal memory of the GPU.

4.4 Execution Monitoring

We also implemented a prototype malware monitoring component. This component hooks the System Service Descriptor Table (SSDT) to record the interactions of malware with the operating system by tracing system calls. System call arguments of important calls are automatically extracted along with their semantic information.

We disabled new driver loading during the analysis to protect the integrity of the monitoring component, even though driver load could be reverted by the physical memory restore. As malware can potentially scan the physical memory using `\device\PhysicalMemory` or using the undocumented API `NtSystemDebugControl` to detect kernel level hooks, we disabled both scanning techniques by instrumenting related system calls.

The monitoring component of BareBox receives new malware samples from an external controller through the network. The recorded analysis data is streamed back to the controller, where it is recorded in a database.

5. EVALUATION

In this section, we present the evaluation of our system. All experiments were conducted on a freshly-installed Windows XP (Service Pack 3), running on bare-metal commodity hardware (3.60GHz Intel Pentium 4 processor with 1GB Memory). Two 256GB hard drives were installed as a main and a mirror disk.

5.1 Restore Verification

We first verify that the disk operation redirections consistently reflect the possible changes made to the system. This is important, as we need to allow the malware to execute properly during analysis. In the second part of the experiment, we verified that all such changes are successfully restored by our system.

5.1.1 Disk restore verification

To verify disk restoration, we calculated the MD5 hash of all files in the *Windows\System32* directory in the clean state of the system. A snapshot of the system was taken using the *snapshot-save* operation. After that, we made a copy of the *Windows\System32* directory and again calculated the MD5 hash of all files. We compared the file hashes and found them to be identical. Therefore, we verified that the redirected read/write operations work properly. To verify that new/changed files can be properly mapped as memory mapped files, we executed some of the applications from the copied version of the *Windows\System32* directory. The applications were executed without any issue. Note that, while executing an *.exe* file, the Windows subsystem first maps the executable as a memory mapped file before executing it.

In the next step, we resized the existing partition of the disk and created a new partition to effectively modify the Master Boot Record (MBR) of the disk. We also deleted all existing system files from the system root C:\directory (except locked paging and hibernation files). This action will cause the system to fail to boot after a restart.

We executed the *snapshot-restore* operation, and the disk was restored to the previous clean state with a single partition that contains all the deleted system files intact but without the copy of the *Windows\System32* directory. To ensure that the restore is permanent, we performed a cold reboot of the system. The disk was verified to be in the previous, clean state.

5.1.2 Volatile state restore verification

In this experiment, we wanted to verify that the volatile state of the running system, including running processes, open file handles, etc., are properly restored.

Before executing a *snapshot-save* operation, we started multiple applications from the *Windows\System32* directory and then saved the list of running process and loaded drivers. After taking the snapshot we installed a DKOM-

based FU.rootkit malware driver. We logged off from the current user session, effectively terminating all the running applications, and logged in as a different user. We initiated the *snapshot-restore* operation from that user session, and within seconds, the system was restored back to the previous user's active session. All the running processes were intact but without the rootkit driver loaded in the system.

5.2 Performance

We compared the performance of our system with different system restore options available for bare-metal and virtual machines systems. We did not test a hardware-based disk restore option (such as Juzt-reboot [15]), but since all of them require a system reboot, we simply measure the system reboot time of the bare-metal system and use this as a lower bound. To measure a fast reboot with commodity hardware, we installed a Windows XP system on an Intel quad-core CPU with 4GB RAM and a 48GB solid-state drive (SSD) with average throughput of 450MB/s. We ignored the initial BIOS initialization of the machine, as it is largely depended on its hardware configuration (e.g., for the above mentioned configuration, the BIOS initialization time was more than 7 seconds because of the custom configuration prompt of the SSD drive). We only measured the time of the soft-reboot of the operating system. That is, the time duration from the operating system boot menu to a completely logged on session. This will typically underestimate the actual time that a system requires to perform a soft-boot.

By profiling the time required for different stages of BareBox' restore process, we found that a significant amount of time was required for device power state manipulation. Thus, we performed experiments by manipulating the power state of only a subset of attached hardware devices. In particular, we performed three sets of experiments:

- All Devices: All devices that are suspended by the system during a suspend-to-RAM operation are included.
- Minimum Devices: Only those devices that are required for the malware analysis framework were included.
- Memory Only: No device related operations were performed. The purpose of this experiment was to measure the memory copy performance because not all devices supported this type of memory restore.

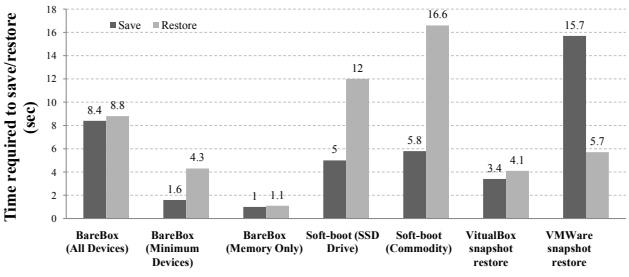

Figure 3: Evaluation of different system restore techniques.

Figure 3 shows the times that it takes to save and restore snapshots for different configurations. This includes BareBox, performing a soft-boot on bare metal, and virtual-machine-based system restore solution. Looking at the results, we can see that the performance of BareBox is comparable to fast, virtual-machine-based solutions. For example, in the required "minimum device" configuration, BareBox is as fast as VirtualBox. The difference, of course, is that BareBox restores the system on bare metal. Alternative mechanisms to restore the system on bare metal take significantly longer (at least three to four times as long).

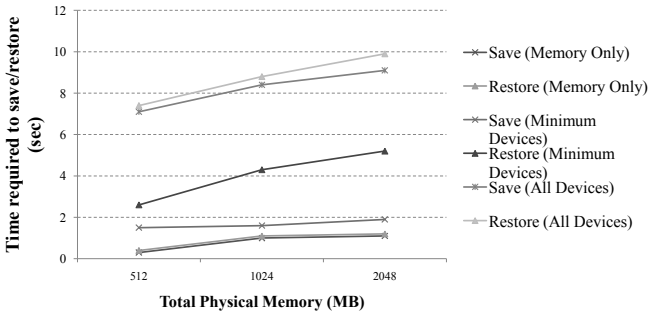

Figure 4: BareBox restore compared with different device restore options.

We also measured the system restore performance of BareBox with different physical memory sizes. As shown in the Figure 4, with a minimum device configuration, the time required for a restore is about four seconds on average. This is about three times faster than booting the system from an SSD-based hard disk.

5.3 Dynamic Malware Analysis

A malware analysis system can be evaluated based on three main factors - quality, efficiency, and stealthiness. Quality refers to the richness and completeness of the analysis results produced. Efficiency measures the number of samples that can be processed per unit time. Stealthiness refers to the undetectability of the presence of the analysis environment.

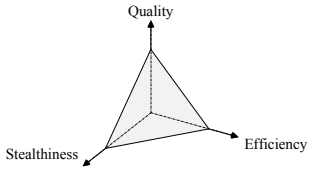

Figure 5: Malware analysis framework evaluation

There is a constant tension between these factors, which is represented as a triangular surface in Figure 5. That is, while trying to achieve better results for one factor, the analysis technique has to make compromises in remain two other factors. For example, an analysis system with an *in-guest* agent tends to produce high quality analysis results but it is less stealthy and prone to subversion. VMM and emulator-based *out-of-OS* analysis systems are stealthier against in-guest agent detection, but because of the *semantic gap*, some level of introspection is required, using the domain knowledge of the target OS [10, 20]. This makes analysis less efficient and limits the quality of the produced results. Qual-

ity and efficiency trade-offs between the fine-grained and the coarse-grained analyses are well described in the literature. BareBox aims to achieve high efficiency while producing results of good quality, which, however, is against the constraints discussed above. A particular focus is on being stealthy when facing VM-aware malware.

5.3.1 Stealthiness

Dinaburg et al. [6] have proposed five theoretical requirements, for building dynamic, transparent malware analysis system, with absolute stealthiness. By executing malware on bare-metal, BareBox immediately satisfies three out of five of these requirements, namely *Identical Basic Instruction Execution Semantics*, *Transparent Exception Handling*, and *Identical Measurement of Time*. We maintain the requirement of *Higher Privilege* by disabling the loading of new kernel modules. As an in-guest analysis system, it is hard to satisfy the requirement of *No non-privileged side effects*. We hide the presence of our analysis component from user mode applications by instrumenting file and memory-related system calls.

For the evaluation of the BareBox with VM/emulation-aware malware, we collected 200 malware samples from the Anubis [10] database that are known to detect virtualization and emulation environments. To maintain the diversity of the sample set, we included six samples per malware family in our actual experiments. Seven different malware families were included, as identified by several major anti-virus vendors such as Kaspersky, McAfee, and ESET. For each family, we tried to include different versions of the malware, if available. We executed them inside a virtualized environment (VMware), an emulated environment (QEMU), and in our bare-metal system and compared their system call traces. The same system call monitoring component was used for all analysis environments.

Malware execution is bound to be slower within virtualized or emulated environment, which might result in smaller number of observed system calls per unit time. However, we assume that one minute, the execution time allowed per malware sample, is enough for the above-selected malware families to show malicious behavior through network interaction and/or new-process creation, regardless of the slowed execution. In fact, we executed some benign programs on these analysis environments, and we observed almost exactly the same number of network and process-creation related system calls within the one minute of execution.

Table 1: Interactions with the Network

Malware Family	BareBox	VMware	QEMU
Rebhip	346	10	55
Telock	205	107	34
Conficker	24	20	16
Zlob/Zeus	408	406	176
Sdbot	152	45	30
Agobot	50	48	3
Armadillo-4	172	82	58

We first wanted to see how much more network activity can be observed in BareBox compared to virtual environments. There are no network-specific system calls for Windows XP. Instead, the network communication is handled by IOCTL calls to the network-endpoint-related file objects. Thus, we monitored calls to these endpoints to assess the network activity of our malware samples. The results in Table 1 clearly show the increased number of network related system calls in our BareBox environment. In addition, we checked how many new processes were created by the malware samples, depending on their environment. As can be seen in Table 2, BareBox was able to elicit more process-creation activities across the board.

Table 2: Number of new process creation

Malware Family	BareBox	VMware	QEMU
Rebhip	9	0	3
Telock	2	1	1
Conficker	0	0	0
Zlob/Zeus	10	10	4
Sdbot	4	1	1
Agobot	50	3	1
Armadillo-4	1	0	0

5.3.2 Efficiency

For a given hardware configuration, usually a program is executed most efficiently when run on bare metal, which is true for the execution of malware programs as well. In addition, while using available disk-restoration-based solutions, the system has to not only reboot but it also requires to load the analysis framework after each reboot. Our system does not have this limitation as the restore operation also restores the state of the analysis component that is ready to analyze the next malware.

To compare the overall efficiency, we measured the overall throughput of each analysis system. We allowed each sample to load and execute for an arbitrary time, 15 seconds, and then the system was automatically restored back for the next run. Virtualized and emulated environments were restored using corresponding snapshot restore mechanisms. To compare with the disk-restore based system, we used the disk mirror component of the BareBox for preserving the disk-state. The system was rebooted after each analysis run to restored the system. We performed an automated analysis of 100 samples in each system to assess the average throughput.

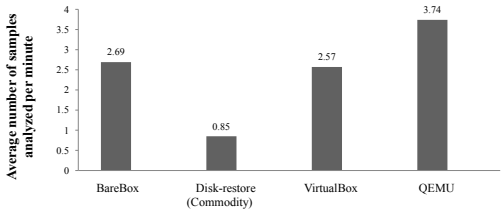

Figure 6: Malware analysis throughput

Figure 6 presents the average number of malware samples analyzed per minute. Compared to the disk-restore based system on commodity hardware, BareBox improved the throughput threefold. Slightly better performance of

the BareBox compared to the VirtualBox-based system can be attributed to the expedited initialization of the network device. We observed a slightly longer delay in network device initialization in case of VirtualBox, after a snapshot restoration. Fully working network interface is essential to initiate a next run of the malware analysis.

5.3.3 Quality

BareBox analysis is limited to system calls only. However, without executing direct or indirect system calls, it is almost impossible to comprehend malicious activities. Since malware is executed on bare-metal, we expect the behavior observed by BareBox to be more close to the actual behavior of the malware exhibited in the wild. Being an in-guest analyzer, our monitoring component can access operating system internals and potentially extract all semantic information of the malware execution. For example, it can initiate additional system calls in the context of executing malware to extract more contextual information. BareBox does not provide fine-grained analysis. Although, it is possible to implement capabilities like user-mode API monitoring and debugger-like fine-grained analysis to improve the quality of the analysis, it compromises BareBox's efficiency and stealthiness.

6. LIMITATIONS

The efficiency of BareBox's rebootless restore largely depends on the actual hardware configuration of the system. For instance, a system with many hardware devices will take longer to restore because BareBox has to force a change in the power state of all of these devices. Currently, BareBox does not restore the entire internal state of attached devices (such as base address registers (BARs)). Instead, it assumes that they are not changed during the short analysis period. This assumption is reasonable, as such states are rarely changed once adjusted by theOS during the initial boot process. For obvious reasons, the restore system cannot restore the changes made to the external resources such as network communication.

Chen et al. have developed a detailed taxonomy of evasion techniques used by malware against dynamic analysis systems [21]. Based on their taxonomy, although BareBox is not detectable through *Hardware*, *Application*, and *Behavior* (timing) based evasion attacks, not all *Environment*-based attacks are defeated. For example, attackers could perform network resource fingerprinting using an external agent (connecting to remote server to identify the host network). Moreover, to prevent the detection of in-guest memory presence and to maintain privilege isolation from the executing malware, BareBox instruments related system calls and disables loading of new kernel modules. Such restriction itself, although not an unconditional indication of BareBox, could be used as a non-privileged side effect for detection. When automating dynamic malware analysis, even a fully transparent analysis system typically requires some form of in-guest agent to receive and initiate execution of a new malware sample. This problem can be mitigated by completely relying on OS-provided RPC mechanisms or implementing a self-destructing agent that removes itself from the system after launching the malware payload. BareBox currently does not implement either of the mechanisms.

Our system must allocate a finite analysis time for each inspection. Malware can potentially exploit this limitation using a *delayed infection* technique. For example, during the initial execution, a malware can choose to stay dormant for long enough so that it passes the allocated, finite analysis time without revealing any malicious activities. Time-triggered or condition-triggered infections (logic bombs) are likely to bypass our dynamic analysis.

7. RELATED WORK

System restore.

AVMM [22] has implemented a physical memory partition technique to improve the efficiency of virtualized clients. Some of the related work on fast reboot and restore are primarily based on preserving system cache. Otherworld [23] uses microreboot techniques to quickly recover applications in case of a kernel crash. This is done by initiating another kernel, called crash kernel, and restoring the state. *Warm-cache-reboot* [24] leverages virtual machine monitor (VMM) technology to preserve the page cache and to help restore the operating system after a reboot. RootHammer [25] reuses the previous VM image from memory, and Recovery Box [26] uses non-volatile memory for fast system restore. However, these systems are focused on fault tolerance and are not designed for restoring the OS to some previous snapshot after a number of modifications to the system's volatile and persistent states, such as hard disk contents. Also, most of these quick restore solutions rely on VMM technology, which is not compatible with our threat model.

Transparent analysis.

For more transparent analysis of malware, researchers have implemented many *out-of-OS* analysis systems that eliminate the *in-guest* presence of components. Some systems are based on whole-system emulation (e.g., Anubis [10], Norman Sandbox [27]), while other systems leverage hardware-assisted virtualization technology to achieve transparency (e.g., Ether [6], VMwatcher [20]). Although there is no *in-guest* presence with these analysis system, nEther was able detect Ether [4], while Paleari et al. were able to automatically produce hundreds of detection codes for emulation based systems [11].

Evasion detection.

Researchers have explored different techniques to detect evasive malware behaviors. Lau and Svajcer have employed a dynamic-static tracing system based on an instrumented Bochs virtual machine to identify VM detection techniques used inside packers [14]. Kang et al. [13] uses a trace matching algorithm to locate the point of execution diversion between an emulated and real environment, and dynamically patching the program to make it behave as observed in a reference system. Balzarotti et al. [12] proposed a system for detection of split personality malware based on the deterministic replay of system call traces generated in a reference system. All of these evasion detection techniques require an ideal *reference system*. BareBox can serve as such a reference system and, hence, complements previous work.

8. CONCLUSION

In this paper, we presented BareBox, a framework for dynamic malware analysis in a bare-metal environment. To facilitate efficient analysis, we introduce a novel technique for

reboot-less system restore. Since the system executes malware on real hardware, it is not vulnerable to any type of VM/emulation-based detection attacks. We evaluated the effectiveness of the system by successfully monitoring the true malicious behavior of VM/emulation-aware malware samples that did not show malicious behavior in emulated or virtualized environments. After each such analysis, our system was able to efficiently restore the bare-metal system so that the next analysis could be initiated.

Acknowledgments

This work was supported by the ONR under grant N000140911042, the ARO under grant W911NF-09-1-0553, National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537, and Secure Business Austria.

9. REFERENCES

[1] M. Labs, "Mcafee threats report: First quarter 2011," McAfee, Tech. Rep., 2011. [Online]. Available: https://secure.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2011.pdf

[2] T. Raffetseder, C. Kruegel, and E. Kirda, "Detecting system emulators."

[3] P. Ferrie, "Attacks on virtual machine emulators," Symantec Corporation, Tech. Rep., 2007.

[4] G. Pék, B. Bencsáth, and L. Buttyán, "nether: in-guest detection of out-of-the-guest malware analyzers," in *Proceedings of the Fourth European Workshop on System Security*, ser. EUROSEC '11. New York, NY, USA: ACM, 2011, pp. 3:1–3:6.

[5] J. Rutkowska, "Red pill... or how to detect vmm using (almost) one cpu instruction," 2004. [Online]. Available: http://invisiblethings.org/papers/redpill.html

[6] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM conference on Computer and communications security*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 51–62.

[7] P. Royal, "Alternative medicine: The malware analyst's blue pill," Aug 2008.

[8] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, "Compatibility is Not Transparency: VMM Detection Myths and Realities," in *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI)*, May 2007.

[9] "Qwmu open source processor emulator." [Online]. Available: http://wiki.qemu.org/

[10] "Anubis: Analyzing unknown binaries." [Online]. Available: http://anubis.iseclab.org/

[11] R. Paleari, L. Martignoni, G. Fresi Roglia, and D. Bruschi, "A fistful of red-pills: How to automatically generate procedures to detect CPU emulators," in *Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT)*. Montreal, Canada: ACM.

[12] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna, "Efficient Detection of Split Personalities in Malware," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2010.

[13] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song, "Emulating emulation-resistant malware," EECS Department, University of California, Berkeley, Tech. Rep., May 2009.

[14] B. Lau and V. Svajcer, "Measuring virtual machine detection in malware using dsd tracer," *Journal in Computer Virology*, vol. 6, pp. 181–195, 2010, 10.1007/s11416-008-0096-y.

[15] "Juzt-reboot." [Online]. Available: http://www.juzt-reboot.com/

[16] "Partimage." [Online]. Available: http://www.partimage.org/

[17] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: bypassing kernel code integrity protection mechanisms," in *Proceedings of the 18th conference on USENIX security symposium*, ser. SSYM'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 383–398.

[18] "Intel®64 and ia-32 architectures software developer's manual." [Online]. Available: http://www.intel.com/Assets/PDF/manual/325384.pdf

[19] "Fast memory copy." [Online]. Available: http://now.cs.berkeley.edu/Td/bcopy.html

[20] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection and monitoring through vmm-based "out-of-the-box" semantic view reconstruction," *ACM Trans. Inf. Syst. Secur.*, vol. 13, pp. 12:1–12:28, March 2010.

[21] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, "Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware," in *Proceedings of the 38th Annual IEEE International Conference on Dependable Systems and Networks (DSN '08)*, Anchorage, Alaska, USA, June 2008, pp. 177–186.

[22] N. Xiong, Y. Zhou, H. Liu, and Y. Zhang, "Avmm: Virtualize client with a bare-metal and asymmetric partitioning approach," Submitted, ICC 2011, Tech. Rep., 2011.

[23] A. Depoutovitch and M. Stumm, "Otherworld: giving applications a chance to survive os kernel crashes," in *EuroSys*, 2010, pp. 181–194.

[24] K. Kourai, "Cachemind: Fast performance recovery using a virtual machine monitor," in *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on*, 28 2010-july 1 2010, pp. 86–92.

[25] ——, "Fast and correct performance recovery of operating systems using a virtual machine monitor," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '11. New York, NY, USA: ACM, 2011, pp. 99–110.

[26] M. Baker and M. Sullivan, "The recovery box: Using fast recovery to provide high availability in the unix environment," in *In Proceedings USENIX Summer Conference*, 1992, pp. 31–43.

[27] "Norman sandbox analyzer." [Online]. Available: http://www.norman.com/products/sandbox_analyzer/en