Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System

Tamas K. Lengyel
University of Connecticut
tklengyel@engr.uconn.edu

Steve Maresca
Zentific, LLC
steve@zentific.com

Bryan D. Payne
Nebula, Inc.
bdpayne@acm.org

George D. Webster
TU Munich
webstergd@sec.in.tum.de

Sebastian Vogl
TU Munich
vogls@sec.in.tum.de

Aggelos Kiayias
University of Athens
aggelos@di.uoa.gr

ABSTRACT

Malware is one of the biggest security threats on the Internet today and deploying effective defensive solutions requires the rapid analysis of a continuously increasing number of malware samples. With the proliferation of metamorphic malware the analysis is further complicated as the efficacy of signature-based static analysis systems is greatly reduced. While dynamic malware analysis is an effective alternative, the approach faces significant challenges as the ever increasing number of samples requiring analysis places a burden on hardware resources. At the same time modern malware can both detect the monitoring environment and hide in unmonitored corners of the system.

In this paper we present *DRAKVUF*, a novel dynamic malware analysis system designed to address these challenges by building on the latest hardware virtualization extensions and the Xen hypervisor. We present a technique for improving stealth by initiating the execution of malware samples without leaving any trace in the analysis machine. We also present novel techniques to eliminate blind-spots created by kernel-mode rootkits by extending the scope of monitoring to include kernel internal functions, and to monitor file-system accesses through the kernel's heap allocations. With extensive tests performed on recent malware samples we show that *DRAKVUF* achieves significant improvements in conserving hardware resources while providing a stealthy, in-depth view into the behavior of modern malware.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Invasive software

General Terms

Dynamic Malware Analysis, Virtual Machine Introspection

ACSAC '14, December 08 - 12 2014, New Orleans, LA, USA

ACM 978-1-4503-3005-3/14/12 $15.00.
http://dx.doi.org/10.1145/2664243.2664252.

1. INTRODUCTION

Over the last decades malware has become a central tool used by criminal organizations to conduct crime over the Internet. Based on the financial implications of the widespread occurrence of security breaches, an in-depth understanding of malware internals is critical for designing and deploying effective defensive solutions. However, the sheer number of malware samples poses a challenge that cannot be matched by manual reverse engineering efforts. The use of static analysis to create signatures for fast detection has been effectively undercut by the increasingly metamorphic nature of modern malware which enables the restructuring of the binary between infections [24]. In conjunction with anti-debugging techniques, the time it takes to formulate effective solutions is further exacerbated [4, 13].

Since the proliferation of metamorphic malware, dynamic malware analysis has become an effective approach to understand and categorize malware by observing the execution of the malware sample in a quarantined environment [11,37]. The interaction between the executing malware sample and the host OS allows dynamic malware analysis systems to collect behavioral characteristics that aid in formulating defensive steps. Consequently, dynamic malware analysis relies on the *breadth* and *fidelity* of the collected data.

As dynamic malware analysis systems have become widely deployed, malware has evolved to detect and evade such systems by either refusing to execute in a sandboxed environment, or by modifying its runtime behavior to lead the analysis system astray [1]. Consequently, it is critical for dynamic malware analysis systems to provide a *stealthy* environment to hide the presence of the data collection from the executing sample [9]. However, even recent analysis systems that meet the stealth requirements [8] have considerable resource requirements and require manual interaction with the malware samples, which constitutes a barrier to scalable, automated dynamic malware analysis.

In this paper we present the *DRAKVUF* ['drɔχvuf] dynamic malware analysis system designed to take advantage of the latest hardware virtualization extensions to provide a *transparent* and *scalable* environment and enable *in-depth* analysis of malware samples. By building on the effectiveness of virtualization extensions to hide the monitoring environment, *DRAKVUF* gains a deep insight into the execution of both user- and kernel-land malware. Furthermore, *DRAKVUF* is designed to minimize the resource require-

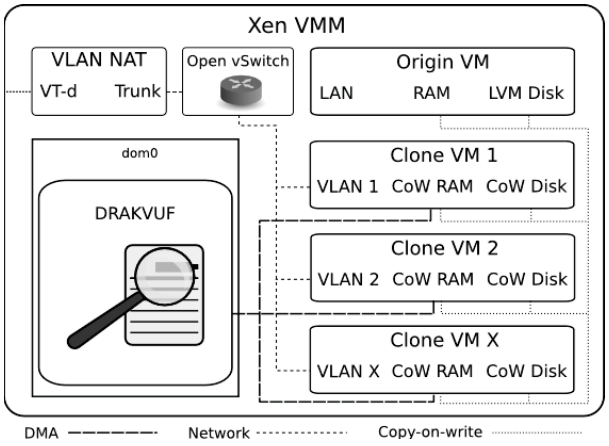

Figure 1: Overview of system components used with *DRAKVUF*.

ments and to enable the fast deployment of virtual machines for analysis by employing copy-on-write techniques for both memory and disk.

Our work makes the following contributions:

- We provide the implementation details of our stealthy and tamper resistant monitoring engine that provides deep insight into the execution of both user- and kernel-mode malware and rootkits. *As a further contribution to the information security community, the source-code is released under GPL at http://drakvuf.com.*
- We introduce a novel technique to directly monitor kernel heap allocations to detect hidden structures created by rootkits. With this novel technique *DRAKVUF* also gains the unique ability to observe filesystem accesses purely via memory introspection.
- We showcase the application of a novel technique to initiate the execution of malware samples without leaving a trace in the analysis environment with the use of *active* virtual machine introspection.
- We evaluate the system using shared hardware resources for large-scale malware analysis via copy-on-write memory and disk.
- We present detailed experiments on a set of in-the-wild malware samples that highlight the capabilities of the system.

The remainder of the paper is organized as follows. In Section 2 we establish our requirements and outline our system's design. In Section 3 we provide in-depth details about our implementation. Section 4 describes the experiments performed using recent, in-the-wild malware samples and also the performance evaluation of the system. Section 5 describes related work. Section 6 describes future work and finally Section 7 provides brief concluding remarks.

2. SYSTEM DESIGN

The goal of *DRAKVUF* is to provide a platform for automated dynamic malware analysis. The analysis has to be performed with transparency and efficiency such that the runtime behavior of the system components in the analysis virtual machines (VMs) can be observed in-depth. *DRAKVUF* is implemented with the use of virtualization technology but in this paper we consider attacks on the hypervisor (VMM) and on lower level system components to be out of scope. It is a standard assumption in virtualization based malware research to consider the VMM to be trustworthy [9], thus in *DRAKVUF* we also consider the limited interface exposed by the VMM to unprivileged guests to provide a reasonable barrier to uncontrolled malware propagation.

The requirements we set out for our system are defined as follows:

(R1) Scalability: The performance overhead for analyzing a sample should be kept to a minimum while the ability to concurrently analyse a large corpus of samples should be maximized.

(R2) Fidelity: The data collection has to capture a wide scope of runtime information while also providing resistance against tampering such that the state of the analysis system can be accurately examined.

(R3) Stealth: The monitored environment should not be able to detect the presence of *DRAKVUF*.

(R4) Isolation: *DRAKVUF* should be strongly isolated from the analysis VMs to protect against tampering.

DRAKVUF is implemented with the use of virtualization technology as virtualization provides several benefits that dynamic malware analysis can take advantage of. Creating and reverting analysis containers can be completed more rapidly as compared to imaging and reverting physical machines. Virtualization also provides an avenue to observe malware execution by providing external access to the state of the virtualized hardware components, commonly referred to as virtual machine introspection (VMI).

To utilize these benefits mentioned above, *DRAKVUF* is built on the open-source Xen VMM. The high-level organization of system components is illustrated in Figure 1. To allow fast deployment of analysis VMs, *DRAKVUF* creates full VM clones via Xen's native copy-on-write (CoW) memory interface and the Linux logical volume manager's (LVM) copy-on-write disk capability. The use of full VM clones via CoW enables the optimized use of the hardware as additional resources are only allocated when needed, matching our requirement set forth in **(R1)**. While the static components of the analysis VMs' memory and disk are shared, the use of copy-on-write prevents clone VMs from interacting with each other as they don't have access to exclusive resources allocated to other clones, hence **(R4)** is supported.

DRAKVUF runs in the control domain (dom0) to make use of direct memory access (DMA) through the LibVMI library [22]. Within the control domain *DRAKVUF* also has access to hypervisor features to control virtualization extensions provided by the CPU, such as the Extended Page Tables (EPT). In order to facilitate access to events associated with the execution of the analysis VMs, *DRAKVUF* uses a combination of techniques to trigger a transfer of control to the hypervisor (VMEXIT) when required. The core technique we employ is the use of *breakpoint injection* in which a #BP instruction (`INT3`, instruction opcode 0xCC) is written into the VM's memory at code locations deemed of

interest. The method by which these locations are automatically determined will be further described in Section 3.2. By configuring the CPU to issue a VMEXIT when breakpoints are executed and configuring Xen to forward such events to the control domain, *DRAKVUF* is capable of trapping the execution of any code within the analysis VM. The #BP injection technique has thus far only been used for stealthy debugging [8]. In *DRAKVUF* we are the first to apply the technique for automatic execution tracing of the entire OS and demonstrate how it is a key component in enabling *active* VMI. With this technique *DRAKVUF* gains deep insight into both kernel and user-land code execution, thus matching **(R2)**

In prior research [9] time skew introduced by trapping the guest into the hypervisor has been effectively countered by altering the guest's view of the time stamp counter register (TSC). While additional time sources could still be available, especially if network access is allowed, we consider this problem to be out of scope for this paper. In *DRAKVUF* we address the previously overlooked problem of starting the execution of a malware sample without leaving a trace: up to now the task had to be either performed manually, which hinders scalability [8, 9]; or with the use of in-guest agents that can be detected [5]. *DRAKVUF* addresses this shortcoming by enabling the automatic execution of a sample without the use of in-guest agents, as the presence of such agents could be used to detect the monitoring environment. Instead *DRAKVUF* uses *active* VMI via #BP injection to hijack an arbitrary process within the VM to initiate the start of the malware sample, a technique further described in Section 3.3. By using existing processes running within the VM, *DRAKVUF* does not introduce new code or artifacts into the analysis VM, thus greatly improving stealthiness, set forth in **(R3)**.

As malware is known to use external input and resources to function, providing network access to the analysis VMs is also required. In order to maintain isolation between *DRAKVUF* and the analysis VMs, as set forth in **(R4)**, network traffic passes through a domain running Open vSwitch and exits through a VLAN NAT domain containing a physical network card passed through using Intel VT-d. The clones are placed on separate VLANs, therefore the only network access they have is through the NAT engine which actively prevents the analysis VMs from discovering each other over the local network. This setup is aimed at minimizing the number of components within dom0 that expose an interface to the infected clones, as we consider emulated device backends to be a more likely attack surface as compared to the minimal interface exposed by the VMM.

To summarize and reflect upon our design requirements: **(R1)** is met by the combined use CoW memory and disk, and the use of #BP injection. **(R2)** is met by employing #BP injection at essential OS code locations which directly reveal the state of the runtime environment. By protecting the breakpoints with EPT page permissions and not having any in-guest agents *DRAKVUF* satisfies **(R3)**. Removing the network stack from the control domain and employing VLAN isolation on the clones provides improved isolation, as required by **(R4)**.

3. IMPLEMENTATION

In the following, we provide in-depth implementation details on the techniques used by *DRAKVUF*.

3.1 Scalability

The construction of defensive solutions requires the rapid analysis of malware samples, however, the sheer amount of data that needs to be processed presents a challenge. It is common that AV vendors need to process more than a 100,000 samples a day [14]. Our focus in *DRAKVUF* thus had been to maximize the *throughput* of the physical machine used for the analysis. While the performance overhead imposed on the execution of individual malware samples is also important and should be kept to a minimum, in our opinion such overhead is only of concern if it actively interferes with the analysis. In our system we take advantage of Xen's copy-on-write (CoW) memory feature to limit the memory requirements, and the LVM CoW disk capability to limit the harddrive space requirements on the host machine. With the combined use of these CoW techniques we are able to greatly increase the number of analysis sessions that can be performed concurrently on a given physical machine.

The CoW techniques employed by *DRAKVUF* require the presence of a static domain whose disk and memory can be used as a reference point, referred to as the *origin domain*. The origin domain can be configured as a regular domain before it is cloned, then once a clone is created it remains statically frozen. When cloning is initiated, first the LVM CoW disk is setup, followed by Xen creating the domain for the clone VM. The content of the origin domain's memory is piped into the newly created clone domain, immediately followed by memory de-duplication via memory sharing. After memory sharing, the only memory overhead is the memory allocated for QEMU to provide disk and network I/O to the clone, and the pages where breakpoints are injected. In our tests, the creation of a full VM clone with 2GB of CoW memory and CoW disk was performed in less than 10 seconds.

While such full VM clones have the advantage of requiring the minimal hardware resources for execution, providing concurrent access to the network poses a challenge, as the clones' IP and MAC addresses are identical. When multiple such clones are placed upon the same switch, MAC address collision occurs and the shared IP address prevents NATing of the upstream traffic. Reconfiguring the IP stack within a VM however has traditionally required an in-guest agent, an approach we specifically set forth to avoid with *DRAKVUF*. To overcome this problem, we use Open vSwitch to provide VLAN isolation to the clones by assigning a unique VLAN ID to each clone.

A VLAN NAT is built using the *pcap* library which reads the raw packets directly off the trunk interface and the uplink, gaining access to the Layer 2 network information, bypassing the Linux network stack. With this setup, the VLAN NAT engine gains access to the VLAN ID that uniquely identifies a clone analysis VM. By performing NAT with the extra unique ID, our system successfully bypasses the issue of MAC and IP collisions. Additionally, as we don't use the Linux networking stack, we also implemented ARP reply injection into the VLAN of each analysis VM when a request is issued for the preconfigured default gateway. By dynamically injecting ARP replies we avoid having to set a static ARP entry in the analysis VMs. This eliminates the possibility of the static ARP entry reveiling the analysis environment, while also opens the door for some novel analysis scenarios which we will discuss in Section 6.

3.2 Monitoring

As dynamic malware analysis relies on observing the live execution of malware, the *fidelity* of the collected data is essential. Furthermore, as rootkits employ a variety of techniques to hide their presence on a system, the broader the scope of data collection, the more likely the analysis will reveal useful features. In the following we discuss a set of data collection mechanisms implemented for 32-bit and 64-bit versions of Windows 7 SP1. In our current prototype system we focus on Windows but our system could easily be extended to monitor Linux and other operating systems as well, as the underlying monitoring techniques are OS agnostic.

3.2.1 Execution tracing

A key feature of existing dynamic malware analysis systems is the ability to trace the execution of processes by monitoring system calls. However, monitoring *only* system calls limits the execution trace to the interaction between user-space programs and the kernel, which does not include the execution of kernel-mode rootkits. To overcome this issue, in *DRAKVUF* we took an alternative approach by directly trapping *internal* kernel functions via #BP injection. With direct trapping *DRAKVUF* is able to monitor malicious drivers as well as rootkits, which was previously not possible with just system call interception.

The location of the kernel functions are determined by extracting information from the debug data provided for the kernel. The use of debugging information has been an established method in the forensics community and it is the most convenient avenue to gain insight into the state of the operating system. In *DRAKVUF* we make use of the Rekall forensics tool [28] to parse the debug data provided by Microsoft to establish a map of internal kernel functions.

At runtime, *DRAKVUF* locates the kernel automatically in memory without having to perform signature based scans, which improves resiliency as compared to existing forensics tools [25, 35]. To automatically locate the kernel in memory we make the observation that Windows 7 uses the FS and GS registers to store a kernel virtual address pointing to the `_KPCR` structure, which is always loaded into a fixed relative virtual address (RVA) within the kernel, identified by the `KiInitialPCR` symbol. As we have obtained the RVA for all kernel symbols, including `KiInitialPCR`, we only have to subtract the known RVA of the symbol from the address found in the vCPU register to obtain the kernel base address.

Once the kernel base address is established, *DRAKVUF* can trap all kernel functions via #BP injection. With internal kernel functions being trapped, the logs thus generated provide a full trace of the execution of the OS from the moment the malware sample is executed.

3.2.2 Tackling DKOM attacks

Rootkits notoriously modify internal kernel structures to hide their presence on a system, commonly referred to as direct kernel object manipulation (DKOM).Standard DKOM attacks are performed by unhooking structures from kernel linked-lists (like the running process' list), which effectively prevents tools that use these lists to enumerate the structures from discovering the additional elements. Forensics tools have long discovered that objects within the Windows kernel heap are created with an additional header attached (`_POOL_HEADER`). This header contains a four-character description of the structure that can be used to detect unhooked structures by simply performing a brute-force string search for these tags in physical memory.

As pooltag scanning became a standard approach in forensics, malware is known to attempt to overwrite the header to prevent scanning tools from later finding these structures [10]. Other rootkit techniques hide the structures by disconnecting the allocated object from its header by changing the requested object size to be greater than 4096 KB, as such allocation requests result in the object being placed into the *big page pool* where no such header is attached to the object. This technique effectively prevents basic pool tag scanning routines from fingerprinting the object.

However, having access to these internal kernel structures is critical in understanding the runtime state of the system, therefore in *DRAKVUF* we took a new approach to tackle DKOM attacks. As we see from the description of the attack methods, the root cause of the problem with DKOM attacks is that the location of the structures becomes unknown within the kernel's heap. If the location can be accurately determined, DKOM is effectively defeated [29]. With *DRAKVUF* we track the kernel heap allocations directly with #BP injection at internal Windows kernel functions responsible for allocating memory for structures used by Windows: `ExAllocatePoolWithTag` and `ObCreateObject`. *DRAKVUF* dynamically extracts the return address from the stack of the calling thread at function entry and traps it to catch the event when the allocation routine returns. Monitoring heap allocations allows us to detect the location of all kernel structures without malware being able to tamper with our view into the system.

In the current implementation *DRAKVUF* tracks the allocation of all objects on the kernel heap, and generates logs based on the associated tag of the structure. If the tag of the structure is one of the already known 2,254 tags, the log contains further details about the type of the object to aid the analyst in identifying allocations that may be of further interest. To detect for example a hidden process, an analyst can now apply a cross-view check to determine if the allocated structures are also accessible via standard lists [16]. *DRAKVUF* further traps the routines responsible for freeing these structures, thus providing a full-view into the life-cycle of the structures. In the next section we further illustrate how this approach enables us to track the active usage of `_FILE_OBJECT`s

3.2.3 Monitoring filesystem accesses with memory events

Monitoring filesystem accesses is one of the core feature of any dynamic malware analysis system, however, prior agentless VMI approaches have attempted to monitor filesystem accesses by modifying the disk emulator to intercept events [26]. While such an approach is effective, reconstructing high-level file-system accesses (like path and permissions) from the low-level disk-emulation perspective is in itself a form of the semantic gap problem and requires extensive knowledge of file-system internals. However, the internal kernel structures that *DRAKVUF* tracks reveal highly valuable information about the execution state of the system, such as the complete set of running processes, kernel modules, threads, and even objects allocated for filesystem accesses by the OS.

The process by which we catch filesystem accesses is shown

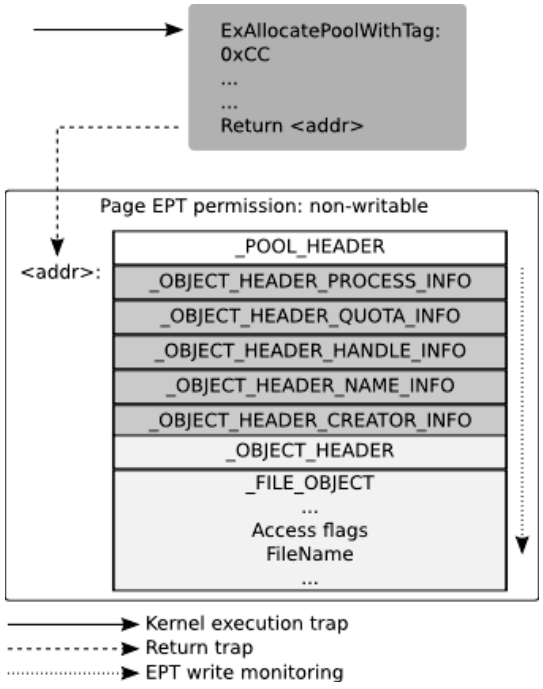

Figure 2: Tracking file accesses by monitoring the allocation of `_FILE_OBJECT`s in the Windows kernel heap.

in Figure 2. When a file is accessed, either by the OS or by a user-land process, a `_FILE_OBJECT` is allocated within the kernel heap with the accompanying tag, `"Fil\xe5"`. When the allocation address is caught, we mark the page on which the structure is allocated as non-writable in the EPT. As the `_FILE_OBJECT` is preceded by a set of optional object headers (shown with a gray background), we derive the exact location of the access flags and file name by subtracting the known size of the `_FILE_OBJECT` from the end of the heap allocation. This allows us to determine the full path of the file as well as the access privilege with which the file is accessed, such as read, write and/or delete permission, without the need to have any deeper understanding of the filesystem itself.

3.2.4 *Carving deleted files from memory*

A common feature of malware droppers is the rapid creation and deletion of temporary files used during the infection process [3]. These temporary files can potentially contain the unpacked malware binary before it is loaded into memory, or other forensically relevant information. However, malware authors are well aware of this fact and it is standard procedure to clean up the temporary files after the dropper finishes installing the malware.

Existing malware analysis systems implemented with the use of in-guest agents can simply retrieve these files when file-deletion is initiated. From a VMI perspective, retrieving these temporary files is complicated by the fact that Windows defaults to write caching being enabled on all harddrives. When files are created and destroyed rapidly, as is often the case when malware is being dropped on a system, the files are never written to disk. As a result simply mounting the analysis VM's disk in the control domain would not give access to these files and the only possible scenario is to carve the files directly from memory.

In *DRAKVUF*, the carving of deleted files is implemented by intercepting specific internal kernel calls that are responsible for file deletion, such as the `NtSetInformationFile` and `ZwSetInformationFile` routines. Once the functions are intercepted, the file is identified by examining the arguments passed, among them the file handle information. This handle does not point directly to the file object, it is only a reference number to an entry in the handle table of the owning process. By parsing the handle table of the process we can locate the corresponding `_FILE_OBJECT` and automatically carve it from memory with Volatility [35].

3.3 Stealth

The stealth of virtualization based analysis systems has been commonly considered in the context of the detection of the monitoring environment itself and not the detection of virtualization in general. The argument for this division is that the already wide-scale deployment of virtualization in commodity systems creates an economic incentive for malware not to exclude virtualized systems. With these assumptions in place *DRAKVUF* expands upon prior research that showed that the combined use of breakpoint injection with EPT protection is an effective technique to hide from even the most advanced anti-debugging techniques [8]. Time skew introduced by trapping the guest into the hypervisor has also been effectively countered in prior research by altering the guest's access to hardware time sources [9], although external time sources could still be available to the malware if network access is allowed.

In *DRAKVUF*, we have turned our attention to a critical step so far overlooked in automated dynamic malware analysis: we start the execution of the malware sample without leaving an identifiable trace of the monitoring environment. With systems where an in-guest component is used, the execution can be initiated by the monitoring agent itself, but the same in-guest component could be potentially used to detect monitoring, even if it is only an autostart script. On the other hand, when no in-guest agent is present, the sample has to be started manually. Therefore, in order to avoid creating any artifacts within the analysis VMs but to allow automated execution, we implemented an injection mechanism that hijacks an actively running but arbitrary process within the VM to initiate the start of the sample on our behalf.

On Windows, a new process can be created by any user-space application via the CreateProcessA function, which is part of the standard Windows API exposed by the kernel32.dll library provided by the operating system. While not every application on Windows has kernel32.dll loaded, generally only a view system processes are the exception, thus in practice *DRAKVUF* can hijack any normal application.

The injection mechanism relies on a set of events, shown in Figure 3, to successfully hijack a process without causing system instability or altering the state of the machine in a way that would reveal the monitoring environment. As the first step after the clone analysis VM is created, *DRAKVUF* traps write events that happen to the control register CR3 to catch when a process context switch occurs. When an event is caught, we examine what libraries are loaded in the address space of the now running process by walking the

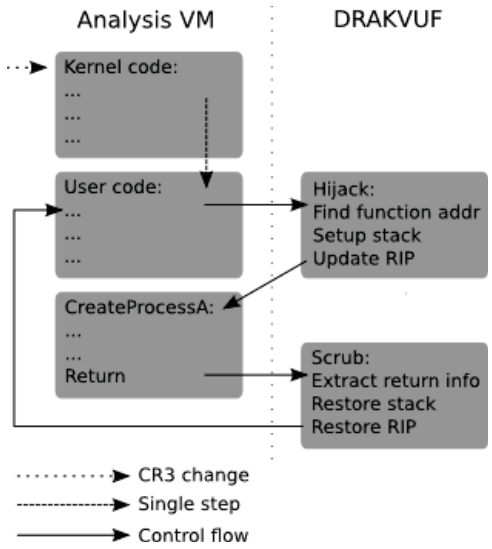

Figure 3: Process hijacking employed in *DRAKVUF* to externally start the execution of the malware sample.

list of loaded modules within the process. If the process has kernel32.dll loaded into its address space, the execution of the VM is switched into single-step mode until the process starts executing user-level code.

The hijack mechanism takes over the execution at the first instruction executed in user-level, and locates the CreateProcessA routine in kernel32.dll's export table. The stack of the process is updated to contain the input arguments required for calling the function, as well as the RCX, RDX, R8 and R9 registers on x86_64, while the original content of the RIP register is pushed as the return address on the stack. The return address is further injected with a breakpoint to notify us when the routine finished. The execution of the analysis VM is resumed after placing the address of CreateProcessA in the RIP register.

When the return breakpoint is hit, we can determine if the process has been successfully created by examining the RAX register, and if it was successful, we also obtain the PID and the handle information of the process that will be used by the executing malware sample. As a context switch could occur while CreateProcessA is executing, the return trap checks if the process at the return trap is the one that was hijacked. Before resuming the original execution of the hijacked process, the stack and vCPU registers are restored, thus seamlessly resuming the execution of the process.

In our implementation we use this mechanism to start malware samples in clean virtual machines. By using this hijacking routine, no artifacts are left on the system that could be detected as a fingerprint of the monitoring environment. This property may not hold if a malware sample is started in an already infected machine, a use-case that we considered out-of-scope for our implementation.

4. EXPERIMENTAL RESULTS

In the following, we discuss an extensive set of experiments performed to establish performance metrics and to evaluate the effectiveness and throughput of the systems. The experiments were performed on an Intel Second Generation i7-2600 CPU running at 3.4GHz.

Unless specified otherwise, the samples were executed on a Windows 7 SP1 x64 analysis platform with a runtime of 60 seconds. During these tests *DRAKVUF* was set to monitor the execution of each internal kernel function that starts with Nt or Zw. The functions starting with Nt are the functions available to user-space applications through regular system calls as listed in the SSDT and monitoring the Zw version of these functions reveals the execution of kernel-level code. We also trap two additional kernel internal-functions, `ExAllocatePoolWithTag` and `ObCreateObject`, which are responsible for kernel heap allocations, as described in Section 3.2.2. While monitoring all internal functions would have been possible during these tests, we reduced the scope of tracing as to reduce the verbosity of the collected data without hindering the insight into the execution of the system.

4.1 TDL4

The first sample we tested was TDL4[1]. This sample was chosen since an in-depth technical write-up has been already created by an antivirus company after reverse engineering the sample [15], which provides a contrast to our automatic analysis. The dropper itself had a 45/46 detection ratio on VirusTotal (VT) [33]. After executing the sample in a Windows 7 SP1 x64 analysis VM, we obtained two additional temporary files created by the dropper in the "C:\Windows\System32\sysrep" folder: cryptbase.dll[2] and syssetup.dll[3].

These temporary files were carved from memory as they were created by the dropper and never flushed to disk before deletion. After submitting the files to VT, the detection ratios were reported as 19/50 and 22/50 respectively. Further investigation into the nature of these temporary files, unmentioned in the original analysis report, revealed them as being part of a known method to elevate privileges by circumventing user access control (UAC) on Windows 7 and 8 [17]. After the exploit installed its payload, a system shutdown was initiated, at which point 1.1GB of memory out of the 2GB assigned to the VM remained shared.

4.2 Zeus

The next sample we analyzed was a recent sample of Zeus[4] with a VT detection ratio of 44/50. In our analysis no files were deleted. However, in our logs we see the sample interacting with files in a temporary folder, GoogleUpdate.exe having a VT detection ratio of 43/50, and FlashPlayer.exe which had no VT detection and was identified as a real Adobe installer. A DLL was also located in the temp folder, msimg32.dll[5], with a VT detection ratio 25/47 that has never been submitted before. The DLL was later revealed by our logs to be dropped into "\Windows\System32" by the executing flash player installer. The sample's installation behavior very closely follows that of ZeroAccess [39], also analyzed by HP [30]. After executing the sample for 60 seconds, the analysis VM had 1.4GB memory in shared state.

[1]md5sum a1b3e59ae17ba6f940afaf86485e5907
[2]md5sum d39d6893117cf1a80c77de1f7ff3d944
[3]md5sum 9b6e5b9c0deb825d5aed343beb090853
[4]md5sum ea039a854d20d7734c5add48f1a51c34
[5]md5sum e051308c2f0c1b280514c99aabd36e34

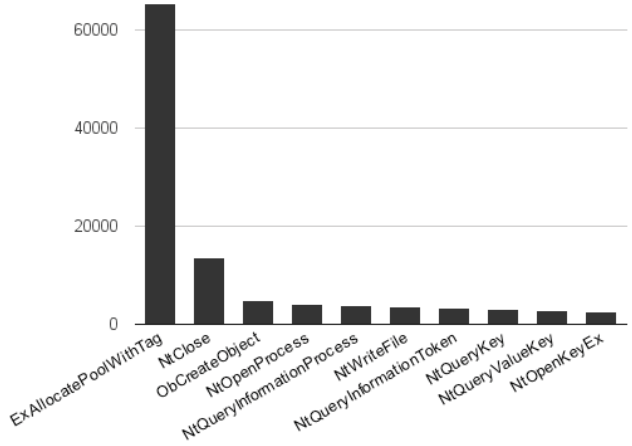

Figure 4: Top 10 monitored kernel functions in terms of average number of observed executions.

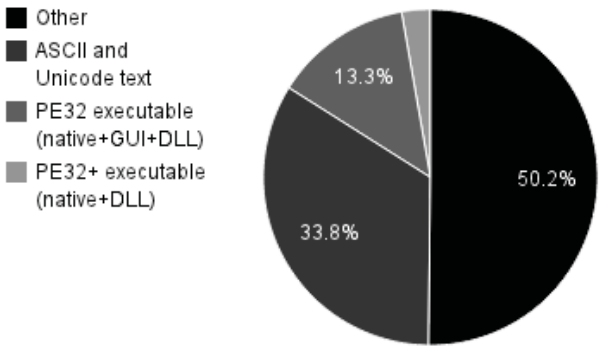

Figure 5: Breakdown of 8797 intercepted file deletion requests by type in recent malware samples.

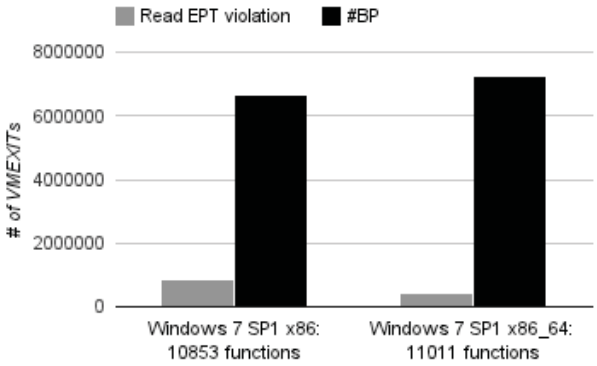

Figure 6: Number of Read EPT violations versus breakpoints hit within 60 seconds when trapping all internal kernel functions.

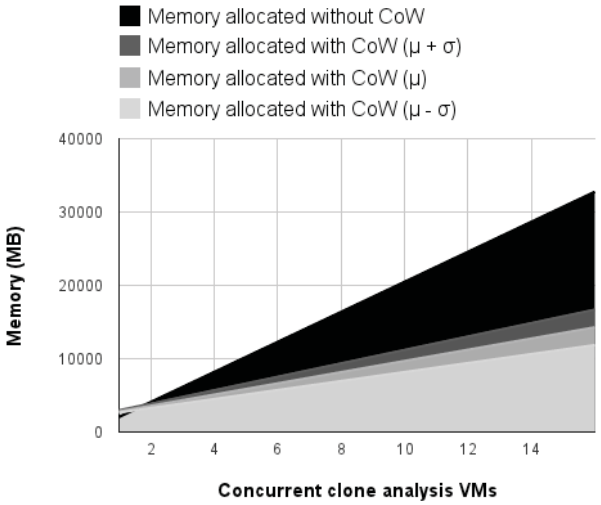

Figure 7: Projected copy-on-write memory allocations (μ = 764MB, σ = 151MB).

4.3 Samples from ShadowServer

In order to test *DRAKVUF* at scale, we obtained 1000 recent malware samples from ShadowServer [32]. The samples were selected with the AlienVault YARA signature that implies the use of anti-virtualization techniques.

Out of the 1000 samples, 241 failed to execute via CreateProcessA injection. These executions failed not because the malware shut itself down but because Windows failed to execute the samples. On average, 159,222 breakpoints had been hit in 60 seconds of execution, and on average 67,950 were pool allocation requests. Figure 4 illustrates the top 10 API calls that were hit across all samples with the internal functions used for heap allocations among the top three. From this figure we can see that some internal kernel functions are executed significantly more often, thus allowing the analyst to pick and choose those functions that are required for the analysis goal, as we did for these experiments.

A total of 8797 unique files were extracted from the analysis VMs by carving them from memory before deletion. To better understand the nature of the deleted files in the analysis VMs, we categorized the files by their type, shown in Figure 5. The 1,412 PE files were also submitted to VT. 561 were new submissions (39%), and nearly all of the files had at least one AV detection. On average, only 20.4% of the AVs categorized these files as malware.

4.4 Measuring overhead and throughput

In this section, we turn our attention to measuring the overhead of our system, a standard metric calculated for any monitoring engine, and to measuring the effective throughput we achieved.

To illustrate the source of the overhead in our system, we measured how many VMEXITs are triggered by our monitoring when all internal kernel functions are trapped. By using the debug symbols for the Windows 7 SP1 kernel all 10,853 internal functions for x86, and 11,011 functions

for x86_64 were trapped. The traps were further protected with EPT execute-only permissions, which on x86 resulted in trapping 727 pages, and 915 pages on x86_64, respectively. The VM was unpaused for 60 seconds and was running idle with only the default Windows system processes being active. As we can see in Figure 6, breakpoints were the main source of VMEXITs we triggered in the VM for both versions of Windows.

To measure the overhead in terms of CPU cycles by the VMEXITs caused by #BP injection, we performed a benchmark similar to that in SPIDER [8]. The test consisted of the monitored domain calling a function within a loop where the function increments a counter. The loop iterates from 10^4 to 10^6 with a step value of 10^4. Timing information was collected by reading the TSC register before and after the loop. The benchmark was performed with and without trapping at the function entry point to determine the overhead. The overhead thus measured was on average a factor of 10502, which is comparable to the overhead in SPIDER. The difference in the overhead likely arises from using different hypervisors, Xen in *DRAKVUF* and KVM in SPIDER respectively, as by design Xen requires an extra VMENTRY/VMEXIT for each trap that is forwarded to the control domain.

Both of these experiments showed that our approach imposes acceptable overhead on the analysis of a single sample. In our opinion, overhead on a per sample base is mainly a *stealth* concern for dynamic malware analsys. That is, overhead should not reveal the presence of the monitoring environment.

From a *scalability* perspective the throughput of the system is of particular importance, which we further evaluated based on the experiments described in Section 4.3. The main hardware constraint in our system was the amount of available RAM: 16 GB. Considering the standard 2 GB of RAM recommended for running Windows 7, the maximum number of concurrent sessions with existing open source tools [5] would be limited to 8 sessions (not counting memory allocated for the control domain). In our experiments, we achieved on average an effective memory saving of 62.4% by using copy-on-write memory, with a standard deviation of 7.3%. Projecting the memory savings as concurrent sessions, shown in Figure 7, we can see the immediate memory savings that can be achieved via CoW memory (the area shown in black). In our case, the number of concurrent analysis sessions has improved by a factor of two, a significant improvement in throughput.

5. RELATED WORK

CWSandbox [37] was one the of first dynamic malware analysis systems to utilize a sandbox environment for monitoring the interaction between the OS and the malware. CWSandbox operates by injecting a kernel driver into Windows that hooks all exported APIs to intercept the system call performed by user-space programs. Beside being vulnerable to detection and tampering, the system call monitoring interface provided by CWSandbox is insufficient for tracking kernel-mode rootkits. Cuckoo [5] is currently one of the most popular open-source dynamic malware analysis systems which uses the same approach as CWSandbox. Cuckoo supports a wide-range of hypervisors; however, neither Cuckoo nor CWSandbox takes advantage of the VMM beyond isolation, as they both rely on an in-guest agent to perform monitoring of the malware's execution. As no special protection is provided to the in-guest agent from the hypervisor, stealth or tamper resistance cannot be guaranteed, potentially leading to incorrect analysis results. In *DRAKVUF*, we directly addressed these shortcomings by observing kernel internal functions from an external point of view, thus providing a stealthy, in-depth view into the execution of the system.

Virtual machine clones were first deployed effectively in 2005 by Vrable et. al. [36] who implemented a highly scalable honeynet system named Potemkin, also based on Xen. Potemkin solved the scaling issue associated with running a large number of nearly identical VMs by introducing memory sharing that de-duplicates identical memory pages present across VMs. While Potemkin was limited to paravirtualized (PV) Linux systems, later works, such as SnowFlock [19], support fully virtualized (HVM) systems as well. As of the latest version of Xen, Potemkin-style VM cloning of fully virtualized systems is natively supported, although only in a "technology preview" stage. Our contribution to existing work in this area is the creation of the network stack via automatic VLAN tagging and NATing such that the clone domains' MAC and IP addresses don't require reconfiguration. In prior work, such reconfiguration has been performed via in-guest agents or by manual iptables configuration [21], a problem we have successfully solved in a transparent and automatic manner.

Ether [9] also made use of hardware virtualization extensions to perform virtual machine introspection. Ether, built on a modified version of Xen, exploited the fact that pagefaults can be configured to trigger VMEXITs on specific code locations. Ether used this to effectively trace system calls in the observed VM. While Ether has made significant efforts to hide these modifications, its effective stealth in practice has been called into question by Pek et. al. [27] who pointed out implementation issues that may still reveal the presence of Ether to the guest OS. In *DRAKVUF*, we also build on the virtualization extensions; however, the techniques employed in *DRAKVUF* to trace system execution allow for analyzing kernel-mode rootkits as well, which were invisible to Ether's system call tracing method. Furthermore, in *DRAKVUF*, we also addressed scalability of the analysis and stealth concerns regarding starting the malware sample without leaving a trace. As an added benefit, unlike Ether, *DRAKVUF* does not require any modification to be made to Xen.

CXPInspector [38] builds exclusively on the use of EPT for execution tracing. As EPT adds an additional layer of paging where translation between guest physical and machine physical address takes place, any access violation within this layer will trigger a VMEXIT, thus remaining transparent to the guest. By marking certain pages non-executable, CXPinspector is able to monitor the execution of the VM. However, the use of EPT for execution tracing has been known to still add significant overhead to the execution of the VM, mainly as a result of the granularity EPT can be set to trigger violations on. In *DRAKVUF*, we also make use of EPT page permissions, but we use it only to add protection to our injected traps, thus avoiding the performance overhead of multiple execute violations on the page.

Process implanting (PI) [12], kernel-module injection (XTIER) [34] and process hijacking (SYRINGE) [7] have been proposed in prior art as methods to perform introspection

into the state of virtual machines. For such introspection to be secure, extra protection needs to be provided by the hypervisor to deny the runtime modification of the stack and heap data of the implanted code. SYRINGE for example executes the injected function call atomically within the guest. While such methods can ensure the integrity of the injected code, they cannot ensure the integrity of the data that is provided to them by a potentially compromised OS. Therefore, if an attacker provides false information to the monitoring application, by performing for example DKOM attacks, these techniques would be unable to detect the tampering. In *DRAKVUF*, we implemented process hijacking for Windows for the first time and we use it to enable the stealthy execution of malware samples only, thus avoiding any issues that arise from relying on untrustworthy data within the observed OS.

SPIDER [8] evaluated the use of #BP injection to enable stealthy debugging of processes within virtual machines using the KVM hypervisor. However, starting a malware sample with SPIDER still requires either manual action or an in-guest agent, which either prohibits scalability or hinders the stealth capability of the system. Similarly, determining where to place breakpoints with SPIDER is a manual process, as would normally be done with debugging. Our contribution in *DRAKVUF* is applying the core technique of #BP injection to automatically trace the execution of the OS and to extract forensically relevant high-level state information without any manual action being required.

6. FUTURE WORK

While dynamic malware analysis systems can lift the burden of analyzing the ever-increasing number of samples, the inherent limitation of the approach is that the execution of the samples is indeterministic [2]. Whether a malware sample exhibits its true malicious nature (i.e., unpacks itself or just stalls) in a given period has generally been considered undecidable [31], although recent work on the topic raised hopes that the problem could one day be decidable if space and time constrains are taken into consideration [6]. Possible solutions that can be integrated into *DRAKVUF* in the future are multi-path exploration via aggressive snapshotting of the analysis VMs [23] as well as stall prevention via VMI based flow-control changes [18].

Having access to layer 2 network information in our VLAN NAT engine opens the door for new malware evaluation scenarios. While in our current setup the routing engine only injects an ARP reply if the request is for the default gateway, we could also inject an ARP reply to any local IP requested. By further integrating the routing engine with *DRAKVUF*, we could deploy a new analysis VM to construct dynamic LAN environments, providing an easy setup to test how malware would propagate on the LAN.

Memory sharing of the analysis VM is currently performed before the execution of the malware sample is initiated, therefore memory savings gradually decrease as the analysis VM is executing. However, more aggressive memory sharing can be performed by trapping updates made to the guest page tables and evaluating which physical pages can be safely deduplicated. This approach would also enable us to reboot the analysis VM, which in some cases is required to trigger the execution of an infection, without giving up on the advantages of memory sharing.

Our process injection mechanism focused on starting the execution of a malware sample by placing the samples on disk before the clone VMs are created. While this approach was practical in our prototype, in future work we plan to place the malware binary directly in memory by employing process hollowing techniques [20].

DRAKVUF currently follows a single path of the malware's execution. As it is possible that the malware is looking for certain conditions in order to execute or exhibit its malicious behavior, by monitoring the checks that are performed by the sample could be used to potentially explore multiple execution paths [23].

7. CONCLUSION

In this paper, we presented *DRAKVUF*, a novel dynamic malware analysis system. As part of *DRAKVUF*, we introduced several novel techniques to improve the *scalability*, *fidelity* and *stealth* of malware analysis by building on the latest hypervisor technology and hardware virtualization extensions. By monitoring kernel-internal functions *DRAKVUF* gains a complete view into the behaviour of both user- and kernel-land malware. Using active VMI to hijack an arbitrary processes within the analysis VM enables *DRAKVUF* to initiate the execution of malware samples without leaving a trace. *DRAKVUF* also showcases the unique ability to trace filesystem accesses by memory introspection and to extract temporary files never flushed to disk. In our experiments, we evaluated *DRAKVUF* using a wide scope of in-the-wild malware samples. These experiments showed that the system is both effective in analyzing modern malware and capable of improving hardware utilization by maximizing the number of concurrent analysis sessions running on shared physical hardware.

Acknowledgements

The authors would like to thank Peiter "Mudge" Zatko for the DARPA Cyber Fast Track program which made this project possible. Further thanks to Jonas Pfoh for the many productive discussions, to Borbala Mahr for naming the project and to the many anonymous reviewers for their insightful observations. The last author was partly supported by ERC project CODAMODA.

8. REFERENCES

[1] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna. Efficient detection of split personalities in malware. In *NDSS*, 2010.

[2] U. Bayer, E. Kirda, and C. Kruegel. Improving the efficiency of dynamic malware analysis. In *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM, 2010.

[3] B. Bencsáth, G. Pék, L. Buttyán, and M. Félegyházi. Duqu: Analysis, detection, and lessons learned. In *ACM European Workshop on System Security (EuroSec)*, volume 2012, 2012.

[4] R. R. Branco, G. N. Barbosa, and P. D. Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies, 2012.

[5] J. Bremer. Blackhat 2013 workshop: Cuckoo sandbox - open source automated malware analysis. `http://cuckoosandbox.org/2013-07-27-blackhat-las-vegas-2013.html`, 2013.

[6] D. Bueno, K. J. Compton, K. A. Sakallah, and M. Bailey. Detecting traditional packers, decisively. In *Research in Attacks, Intrusions, and Defenses*. Springer, 2013.

[7] M. Carbone, M. Conover, B. Montague, and W. Lee. Secure and robust monitoring of virtual machines through guest-assisted introspection. In *Research in Attacks, Intrusions, and Defenses*, volume 7462 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012.

[8] Z. Deng, X. Zhang, and D. Xu. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, New York, NY, USA, 2013. ACM.

[9] 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*. ACM, 2008.

[10] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009.

[11] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44, 2012.

[12] Z. Gu, Z. Deng, D. Xu, and X. Jiang. Process implanting: A new active introspection framework for virtualization. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*. IEEE, 2011.

[13] F. Guo, P. Ferrie, and T.-C. Chiueh. A study of the packer problem and its solutions. In *Recent Advances in Intrusion Detection*. Springer, 2008.

[14] Z. Hanif, T. Calhoun, and J. Trost. Binarypig: Scalable static binary analysis over hadoop, 2013.

[15] D. Harley. `http://www.welivesecurity.com/2012/02/02/tdl4-reloaded-purple-haze-all-in-my-brain/`, February 3 2014.

[16] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007.

[17] P. Kleissner. The art of bootkit development, 2011.

[18] C. Kolbitsch, E. Kirda, and C. Kruegel. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011.

[19] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. De Lara, M. Brudno, and M. Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009.

[20] J. Leitch. Process hollowing. `http://www.autosectools.com/process-hollowing.pdf`, November 4 2013.

[21] T. K. Lengyel, J. Neumann, S. Maresca, and A. Kiayias. Towards hybrid honeynets via virtual machine introspection and cloning. In *Network and System Security*. Springer, 2013.

[22] LibVMI. `https://code.google.com/p/vmitools`.

[23] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*. IEEE, 2007.

[24] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE, 2007.

[25] J. S. Okolica and G. L. Peterson. Extracting forensic artifacts from windows o/s memory. Technical report, DTIC Document, 2011.

[26] B. D. Payne, M. de Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE, 2007.

[27] 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*. ACM, 2011.

[28] Rekall. `https://github.com/google/rekall`.

[29] J. Rhee, R. Riley, D. Xu, and X. Jiang. Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory. In *Recent Advances in Intrusion Detection*. Springer, 2010.

[30] A. Roberts, R. McClatchey, S. Liaquat, N. Edwards, and M. Wray. Introducing pathogen: A real-time virtual machine introspection framework. Technical report, HP, 2013.

[31] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 2006.

[32] ShadowServer. The shadowserver foundation. `https://shadowserver.org`, February 4 2014.

[33] VirusTotal. Free online virus, malware and url scanner. `http://virustotal.com`, February 4 2014.

[34] S. Vogl, F. Kilic, C. Schneider, and C. Eckert. X-tier: Kernel module injection. In *Network and System Security*. Springer, 2013.

[35] Volatility. `https://github.com/volatilityfoundation/volatility`.

[36] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *ACM SIGOPS Operating Systems Review*, volume 39. ACM, 2005.

[37] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *Security & Privacy, IEEE*, 5(2), 2007.

[38] C. Willems, R. Hund, and T. Holz. Cxpinspector: Hypervisor-based, hardware-assisted system monitoring. Technical report, Ruhr-Universitat Bochum, 2013.

[39] J. Wyke. The zeroaccess rootkit. `http://sophosnews.files.wordpress.com/2012/04/zeroaccess2.pdf`, 2012.