HyperLink: Virtual Machine Introspection and Memory Forensic Analysis without Kernel Source Code

Jidong Xiao*, Lei Lu†, Haining Wang‡, Xiaoyun Zhu§

*Boise State University, Boise, Idaho, USA
†VMware Inc., Palo Alto, California, USA
‡University of Delaware, Newark, Delaware, USA
§Futurewei Technologies, Santa Clara, California, USA

Abstract— **Virtual Machine Introspection (VMI) is an approach to inspecting and analyzing the software running inside a virtual machine from the hypervisor. Similarly, memory forensics analyzes the memory snapshots or dumps to understand the runtime state of a physical or virtual machine. The existing VMI and memory forensic tools rely on up-to-date kernel information of the target operating system (OS) to work properly, which often requires the availability of the kernel source code. This requirement prevents these tools from being widely deployed in real cloud environments. In this paper, we present a VMI tool called HyperLink that *partially* retrieves running process information from a guest virtual machine without its source code. While current introspection and memory forensic solutions support only one or a limited number of kernel versions of the target OS, HyperLink is a one-for-many introspection and forensic tool, i.e., it supports most, if not all, popular OSes regardless of their versions. We implement both online and offline versions of HyperLink. We validate the efficacy of HyperLink under different versions of Linux, Windows, FreeBSD, and Mac OS X. For all the OSes we tested, HyperLink can successfully retrieve the process information in one minute or several seconds. Through online and offline analyses, we demonstrate that HyperLink can help users detect real-world kernel rootkits and play an important role in intrusion detection. Due to its version-agnostic property, HyperLink could become the first introspection and forensic tool that works well in autonomic cloud computing environments.**

I. Introduction

Nowadays cloud computing is commonly used to provide IT services over the Internet. Typically, cloud systems are large scale and complex by nature, and their complexity is still growing at a steadily increasing rate. To handle the complexity inside a cloud, autonomic computing solutions have been adopted for easing administrative tasks. As firstly defined by IBM researchers, the major goal of autonomic computing is to enable self-management in a distributed system and hide intrinsic complexity from human administrators and users [1], [2]. A successful autonomic computing system should have the property of self-protection, self-healing, self-configuration, and self-optimization. To achieve these properties, the capability of a computing system to monitor and detect security threats in an accurate and timely manner is needed.

Over the years, in order to automatically detect security threats, protect system from attacks, and recover from the effects of attacks, virtualization has been widely employed for various defensive purposes, including intrusion detection [3], malware detection and analysis [4], [5], honeypots [6], [7], kernel rootkit detection [8], [9], kernel integrity protection [10], and detection of covertly executing binaries [11]. Being the main enabling technology for cloud computing, virtualization allows us allocating finite hardware resources among a large number of software systems and programs. As the key component of virtualization, a hypervisor runs directly on the hardware or a host operating system (OS) to create and manage the guest OSes. From a security perspective, the hypervisor, working at a higher privilege level than guest OSes, is an ideal place to deploy security infrastructures. Among all autonomic security techniques, a concept called *Virtual Machine Introspection* (VMI) [3] was proposed and has progressed substantially over the last decade. In essence, VMI refers to inspecting the states of a guest OS from the hypervisor level and its use in virtualized environments is absolutely crucial to effect risk mitigation in the filed of cloud computing. Practical applications, such as list running processes, scan for hidden processes, show open files, and show kernel loaded modules, form the basis for many autonomic security solutions at large scale.

However, it is known that VMI has a serious challenge to overcome, commonly referred to as the *semantic gap problem* [4], [12]. This problem is due to the fact that a hypervisor can only see the hardware level information of the guest OS, such as registers and memory pages, while system administrators need to know the internal high level information of the guest OS, such as processes, files, and modules. To this end, some research [4], [13], [14] has been undertaken to bridge the semantic gap, and therefore reconstruct those key information of the guest OS. As a representative piece of work, VMwatcher [4] attempts to reconstruct the process list from the guest OS memory. The key idea of VMwatcher is to retrieve the type structure of a process descriptor from kernel source code. With this layout information, it can reconstruct every process descriptor instance from the guest OS memory and print out all the processes.

A closely-related research area is the memory forensic analysis, in which researchers attempt to analyze memory snapshots or dumps. Based on such analysis, researchers extract some key information of the target system to investigate system attacks. Similar to the VMI techniques, modern memory forensic tools also require kernel source code to obtain the memory layout of a process descriptor or other components.

However, as kernel source code evolves, the layout of those key data structures changes regularly. It is not uncommon that the VMI or memory forensic tools developed for one specific kernel version do not work for other versions. It would be even more complicated when coping with OSes that are closed source, since knowing the layout of their key data structures requires great non-trivial reverse engineering efforts. Because of these, existing VMI or memory forensic tools can only support a limited number of kernel versions, which prevents them from being deployed in a cloud computing environment. Considering a cloud environment in which there may be tens of different guest OSes with different distributions or kernel versions, to extend existing tools to support a heterogeneous environment at such a scale requires significant human efforts, which is contradicting with the major objective of autonomic computing—producing computing systems that require less human efforts.

In this paper, we explore the possibility of developing a one-for-many VMI or memory forensic tool. More specifically, we propose the concept of *partial retrieval*. While existing tools extract every field of the key data structures from memory, we observe that some fields are more critical than other fields. Partial retrieval assumes that only retrieving these critical fields would help for VMI and forensic analysis. Based on the partial retrieval concept, we develop a tool called HyperLink to support most, if not all, popular OSes regardless of their versions. Moreover, while existing tools resort to kernel source code to obtain the knowledge of the key data structures in an OS, we observe that some user-perceivable invariants can help us identify the data structures from memory. Thus, HyperLink employs these user-perceivable invariants for information reconstruction and completely avoids the dependence on kernel source code.

We implement two versions of HyperLink prototype: an online and an offline version. We evaluate the effectiveness of HyperLink under different OSes with different versions. The results show that HyperLink can accurately retrieve the running process list of an OS in a timely fashion. Through online and offline analyses, we demonstrate that HyperLink can help to detect real-world kernel rootkits. We further perform a comparison between HyperLink and other state-of-the-art tools. While different tools have different advantages and features, HyperLink prevails against existing tools in two aspects – easy usability and low maintenance cost, which make HyperLink an improvement over current tools. The major research contributions of this work are summarized as follows:

- We are the first to make an attempt for developing one-for-many VMI and/or memory forensic tools to retrieve the list of processes from the guest OS, when certain assumptions are satisfied. We present the design, implementation, and evaluation of the HyperLink tool. As far as we know, HyperLink is the first tool that supports both online and offline analyses for different OSes, including Windows, Linux, FreeBSD, and Mac OS X systems, regardless of their versions.
- HyperLink consists of two technical novelties: (1) introducing the concept of partial retrieval and (2) leveraging user-perceivable invariants for information reconstruction, instead of relying on kernel source code. These two innovative approaches enable HyperLink to achieve the one-for-many goal and work well in a cloud environment.

The remainder of the paper is organized as follows. Section II describes the background of our work. Section III presents the design of HyperLink. Section IV details the implementation of HyperLink. Section V presents the experimental results on our testbed. Section VI discusses some potential extensions to the HyperLink tool. Section VII surveys related work, and finally, Section VIII concludes the paper.

II. Background

A. Challenges in Modern Introspection and Forensics Tools

Existing introspection and memory forensic tools all face one challenge, which is to re-construct kernel data structures, one needs to know the definition of each data structure. However, the definitions change constantly. We use one of Linux kernel's key data structures, the process descriptor `task_struct`, as an example. In a Linux system, the `task_struct` is defined as a struct in the kernel, with each instance of the structure representing a process. The `task_struct` is complicated and in a recent version of the Linux kernel (version 3.19), it contains 210 members. However, some members in one kernel version do not necessarily exist in other kernel versions. For example, several years ago, when Linux kernel 3.0 was released, the `task_struct` only contained 175 members. The definition of the `task_struct` constantly varies with the evolution of Linux kernel versions. Since Linux is an open source OS, developers can obtain the definition of the `task_struct` from the kernel source, and compute the offset of each member inside the `task_struct` accordingly. However, because of this, very often the tools developed can only work for one or limited versions of kernels.

For those closed source OSes, it is more challenging to compute the layout of their key data structures. Normally some sophisticated reverse engineering efforts are needed. Again, the efforts spent on one version of OS may not be sufficient for a different version. In summary, existing introspection and memory forensic tools require a very high development and maintenance cost. To address this limitation, we explore the possibility of developing a one-for-many introspection/forensic tool, which is able to work for most modern OSes regardless of their versions.

B. Key Observations

We build the HyperLink tool based on the following key observations:

- Modern OSes usually organize their key information using linked lists. The key information consists of processes, kernel modules, open files, and network connections etc. A linked list is a data structure including a group of nodes with each node having the same layout or definition.
- Since the layout of a node changes regularly across different kernel versions, researchers commonly believe that up-to-date kernel data structure information is needed for introspection and forensic analysis. Take process for an example, while Linux uses `task_struct` to represent a process, Windows uses EPROCESS to represent a process; the layouts of the `task_struct` and EPROCESS are surely different from each other, and they also change across different kernel versions. However, in these data

TABLE I: Process Invariants

OS	PID - Name	PID - Name	PID - Name
Linux	0 - swapper	1 - init	not used
Windows	0 - Idle	4 - System	not used
FreeBSD	0 - kernel	1 - audit	2 - init
Mac OS X	0 - kernel_task	1 - launchd	2 - kextd

structures, some members are more crucial and more fundamental than the others. More specifically, all of these data structures include a process id, a process name, and a next pointer that points to the next node in the linked list. This property has never changed and it is also valid for most of the other modern OSes.

- Researchers commonly believe that the entire layouts of these key data structures have to be known, before one can re-construct the linked lists from memory. However, we observe that this is not necessarily true. Still take process for an example, the data structure that represents a process usually consists of a process id, a process name, and a next pointer. We believe that with only these pieces of information, one can re-construct the process list that is sufficient for intrusion detection and memory forensic analysis.

C. Assumptions

As we know, the offsets of the process id, the process name, and the next pointer in the data structure vary across different kernel versions. However, we assume that there exist some invariants for each OS. More specifically, we assume that a small number of process related information does not change often. For example, for the Linux systems that we tested the first process is always named "swapper" with a process id 0, and the second process is always named "init" with a process id 1. For Windows system, its first process is named "Idle" whose process id is 0, and the second process is named "System" whose process id is 4. These observations hold for the operating systems we studied spanning nearly 12 years (Table III and Table **??**). We summarize our assumptions in Table I for this work.

The other OSes have similar properties. These properties are persistent across different kernel versions, and more importantly, are user perceivable. This implies that we can easily obtain and update the assumption information by running process or task list command without the availability of kernel source code. Based on this user perceivable information and the basic structure of the linked list, we are able to compute the offsets of these key members and then reconstruct the task linked list.

III. Design

In this section, we first outline the design rationale behind HyperLink. Next, we introduce the linked list used in modern OSes, given that the main task of HyperLink is to reconstruct the process linked list. Finally, we explain how we extract the linked list from memory.

```
#define TASK_COMM_LEN 16
struct task_struct {
     ...
     struct list_head tasks;
     ...
     pid_t pid;
     pid_t tgid;
     ...
     char comm[TASK_COMM_LEN];
     ...
};
```

Fig. 1: Definition of task_struct in Linux 3.19

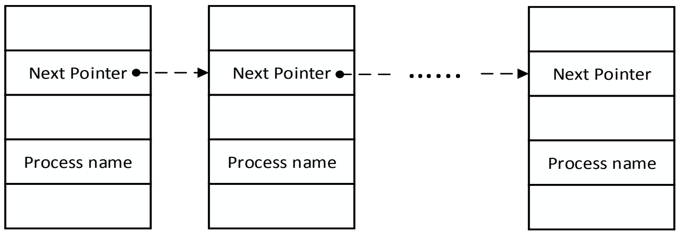


Fig. 2: Process linked list in Linux and Windows

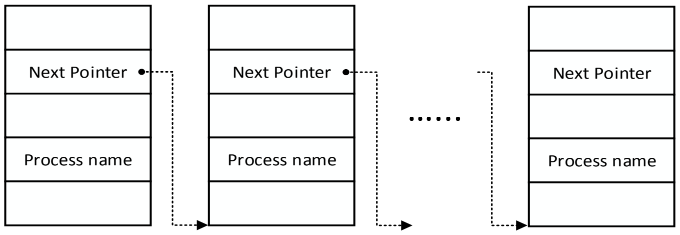


Fig. 3: Process linked list in FreeBSD and Mac OS X

A. Design Rationale Behind HyperLink

On one hand, HyperLink shares the same goal with many existing introspection/forensic tools, i.e., deriving high level system information from memory. On the other hand, HyperLink is different from existing tools in information reconstruction. For each node in the linked list, existing tools treat every field equally. Developers believe every field is equally important, and they attempt to reconstruct every field of the node. That is the reason why they need the source code or the memory layout of the whole data structure. We coin the term *full retrieval* to represent this approach. In contrast, we believe that not every field is critical for VMI or memory forensic analysis. In fact, some fields are more critical than many other fields, such as process name, process id, and next pointer. Therefore, we decide to only reconstruct the these fundamental fields. Figure 1 gives an example of `task_struct` definition in Linux 3.19. The 'tasks' field uses a list to store the records of all processes. The 'pid' field stores the PID of a process and the 'comm' field stores a fixed size array (16 bytes) of characters to contain the executable name of a process. We will show in Section V that, extracting the process linked list with only these pieces of information would help for forensic analysis, including hidden process or rootkit detection. We coin the term *partial retrieval* to represent our approach.

B. Linked List in Modern Operating Systems

We observe that there are two types of linked lists used by modern OSes to organize their processes, as shown in Figures 2 and 3. While an OS usually defines hundreds of members for the process structure, in order to reconstruct the process linked list, what we care most are the process name and the next pointer. It can be seen from Figures 2 and 3 that, the difference between these two types of linked lists is: in the first type (Type 1), the next pointer points to the next pointer of the next node, but in the second type (Type 2), the next pointer points to the start of the next node. In fact, both Linux and Windows use Type 1 linked lists to organize their processes, while both FreeBSD and Mac OS X use Type 2 linked lists to organize their processes.

C. Extract the Linked List from Memory

For both types of linked lists, though we do not know the offset of the next pointer or the process name inside the `task_struct`, one fact is that for each instance, the gap between its next pointer and its name is a constant. We define α_1 to be the offset of the next pointer of process 1 in `task_struct`, α_2 to be the offset of the next pointer of process 2, β_1 to be the offset of the name of process 1, and β_2 to be the offset of the name of process 2. We have the following equation holds:

$$(\alpha_1 - \beta_1) = (\alpha_2 - \beta_2). \tag{1}$$

Note that for the first couple of processes in the linked list, normally their id and their name do not change across different kernel versions. Therefore, based on the equation above, we can retrieve the whole linked list out of memory by conducting the following three steps.

- We first search the process name in kernel memory space. For example, for Linux systems, we know that "swapper" is the name of the first process and "init" is the name of the second process. Therefore, we can search these two strings in memory and there could be many matches. Note that we only search the process names in the kernel space to reduce the number of false positives. Then, for each pair of the results, we search their surrounding memory addresses[1], and check if it is a pointer. If so, we test whether it is the "next pointer" based on the above equation: if it satisfies the equation, we think it is the "next pointer"; otherwise, we ignore it.
- Once we identify the name and the next pointer of the first process, we can compute the offset from the name to the next pointer in bytes. This computed offset is an invariant for every process. Starting from the first process, based on this invariant, we are able to traverse the memory space and identify every node in the process linked list.
- When the nodes in memory are identified, combining with the knowledge of the process ids of the first couple of processes, we can also compute the offset of a process id. Thus, we can retrieve the whole linked list, with process id, name, and next pointer of each node, which we think are sufficient for performing forensic analysis.

[1]Based on our experience, the next pointer is usually located within 4096 bytes distance of the name, and this observation holds for every OS we have tested.

IV. Implementation

We implement two versions of the HyperLink tool – online and offline version. Both versions are implemented in C language. The offline HyperLink requires a memory dump file as its input, while the online HyperLink can access the guest OS memory while it is running. We present the implementation details of both versions as follows.

A. Online Version

We incorporate the online HyperLink within the Qemu/KVM hypervisor source code. Qemu provides a monitor, a command line console through which users can type commands to interact with Qemu. For instance, users can type "info registers" to view the virtual registers for the running guest OS. Qemu has implemented a standard interface through which we can define our own command. Therefore, we define a "ps" command and implement the corresponding functions. The main purpose of these functions is to print the processes list of the guest OS.

The major challenge for the online HyperLink is to access guest memory. However, modern hypervisors have already provided such facilities. For example, in Qemu, one can invoke the function *cpu_memory_rw_debug()* to access the virtual memory address of the guest OS and can call *cpu_physical_memory_rw()* to access the physical memory address of the guest OS.

B. Offline Version

While our online HyperLink is coupled with the hypervisor code, the offline HyperLink is an independent tool (compiled and ran alone). The offline HyperLink takes guest OS memory dump files as its input. It has options to take one or two dump files dependent on the usage. Normally, when users provide one dump file, HyperLink analyzes the dump file, retrieves field offsets from the dump file, and uses these offsets to reconstruct the linked list. When two dump files are provided, the first file is used as a clean dump and the second file is the target dump. HyperLink first retrieves offsets from the clean dump file, uses the offsets information to analyze the target dump, and extracts the linked list from the target dump file. This is a special use case and is very likely in a virtualized or cloud environment, in which there could be many virtual machines running the same OS. The assumption is that the clean dump file can always be trusted, but the target OS is likely compromised.

C. Data Structure in Multiple Linked Lists

One problem we might have is that, one data structure instance could be contained in multiple linked lists in the memory. Take the process linked list as an example, one `task_struct` instance can be contained in several linked lists: the task list, which is what we are looking for, the children list, which represents a list of children processes, and the sibling list, which represents the list of sibling processes.

For any given `task_struct` nodes in memory, they may be contained in both list tasks and list children. To avoid false positives, we have to identify the right linked list. In cases like this, we can increase the search space to locate three

TABLE II: Testbed Setup

Component	Specification
Host CPU	Intel i5-2410M 2.30GHz, Quad-Core
Host Memory	4GB
Host OS	Fedora 20 x86_64
Host Kernel	3.11.10-301.fc20.x86_64
Qemu	2.2.0
Guest Memory	1GB

TABLE III: List of OSes that HyperLink can support

	Version	Guest Kernel	Release
Linux	Red Hat Linux 8	2.4.18-14	2002.09
	Cent OS Linux 4.4	2.6.9-42	2006.08
	Cent OS Linux 6.3	2.6.32-279	2012.09
	Ubuntu Linux 14.04.1	3.13.0-32.57	2014.07
Windows	Windows 2000 Professional	NT 5.0	1999.12
	Windows 2003 Server R2	NT 5.2	2005.12
	Windows Vista Home SP2	NT 6.0	2009.04
	Windows 7 Ultimate SP1	NT 6.1	2009.07
	Windows 8 Pro	NT 6.2	2012.10
FreeBSD	FreeBSD 6.1	6.1-RLS	2006.05
	FreeBSD 7.2	7.2-RLS	2009.05
	FreeBSD 8.4	8.4-RLS	2013.06
	FreeBSD 9.2	9.2-RLS	2013.09

```
[ system1 ]./bin/main -o mac dump.mac dump.mac
-------
OS Type: Mac OS X
-------
       PID        COMMAND
        0        kernel_task
        1        launchd
       21        kextd
       22        notifyd
       23        DirectoryService
       24        syslogd
       25        configd
       26        diskarbitrationd
       27        distnoted
       28        mDNSResponder
       32        securityd
       36        slapd
       37        ntpd
       40        named
       41        Python
       42        httpd
       43        nmbd
       44        krb5kdc
       45        kadmind
       46        Python
       50        usbmuxd
       51        SystemStarter
       54        servermgrd
       56        RFBRegisterMDNS
       57        PasswordService
-------
Total number of processes: 93
```

Fig. 4: Running HyperLink offline analysis against a Mac OS X 10.6 memory dump

consecutive process names. Theoretically it is possible that three nodes can be linked consecutively in multiple linked lists, our experimental results show that this scenario does not happen for the first three nodes in Windows, Linux, FreeBSD, and Mac OS. During our evaluation, we manually compared the output results of HyperLink and the ground truth (which can be obtained from inside the guest OS), and we do not see any false positives. Based on the experimental results, we can see that for Windows and Linux, searching the first two nodes would suffice, while for FreeBSD and Mac OS, the first three nodes are needed.

V. Experimental Evaluation

Our evaluation consists of six parts: (1) we validate that HyperLink has the property of one-for-many, i.e., it supports many popular OSes and is version agnostic; (2) we illustrate our results for Mac OS X and explain why Mac OS X is special; (3) we demonstrate that, by using HyperLink, one can detect a hidden process, and thereby, detect kernel rootkits or intrusion; (4) we perform a comparison between HyperLink and various state of the art tools; (5) in order to further demonstrate the advantage of HyperLink, we perform a more detailed comparison between HyperLink and the most mature product: Volatility; (6) we study whether a process list is sufficient for intrusion detection. The system configuration of our testbed is shown in Table II.

A. *OS Generality*

We first validate that HyperLink supports multiple different OSes. We list all the OSes we have tested in Tables III. More specifically, we have tested HyperLink with a variety of operating systems including: four versions of Linux systems, four versions of Windows systems, and four versions FreeBSD systems. The results show that both the online and the offline version of HyperLink work effectively in retrieving the process information from the guest OSes. For most of the cases, HyperLink can accurately retrieve the process information in less than one minute. It is evident that these OSes span a long time, in terms of the their release dates. Take Linux and Windows operating systems for example, the first and the last studied versions span nearly 12 years, from personal computer OS to server computer OS. This demonstrates the efficacy of HyperLink for most mainstream operating systems.

B. *Mac OS X*

Mac OS X is another mainstream operating system that HyperLink can naturally support. However, at this moment, it is not easy for us to fully test HyperLink against Mac OS X with different versions. There are two major observations.

First, running Mac OS X as a guest OS has always been an issue in modern hypervisors, such as Xen and Qemu. This is mainly because Apple, as a hardware company, does not allow people to run their OSes on non-Apple hardware. Thus, running Mac OS X as a guest on Intel/AMD based hardware is a violation of Apple's end user license agreements (EULA) [15]. However, a large number of developers are using Intel/AMD based hardware and such a legal limitation to some extent prevents many developers from actively extending the hypervisor to support Mac OS X. Therefore, the progress of supporting Mac OS X in a virtual machine is quite slow. As far as we know, only until 2014, Qemu/KVM has not provided support for Mac OS X. In contrast, it has been almost 10 years since Qemu/KVM supported Windows/Linux as guest OSes. According to a Qemu/KVM developer's guide [16], to install a Mac OS X on top of Qemu/KVM hypervisors, it requires

Qemu to be newer than version 2.1.0 (released in July, 2014) and the underlying Linux kernel to be newer than version 3.15 (released in June, 2014.

Second, based on this guide [16], we are able to install Mac OS X 10.6 (also named Snow Leopard) and run HyperLink against Mac OS X 10.6. Our experiments show that HyperLink works well for Mac OS X 10.6 for both online and offline analysis, HyperLink can quickly print all the processes in the Mac OS X. However, we cannot install any old version of Mac OS X (Such as Leopard or Cheetah) as a Qemu/KVM guest. Our speculation is that the support for Mac OS X is still immature in Qemu/KVM. To demonstrate HyperLink works for Mac OS X 10.6, the screen shot for our offline analysis is shown in Figure 4. Due to limited space, the figure only illustrates the first 25 processes while there are totally 93 processes. Given the fact that commodity hypervisors have recently started to support for Mac OS X, we envision that installing Mac OS X on top of modern hypervisors as a guest OS would not be a problem in the near future. We will attempt to test HyperLink against more versions of Mac OS X in our future work.

C. Kernel Rootkit Detection Case Study

Next, we demonstrate how we use HyperLink to detect kernel rootkits by conducting both online and offline analyses. To this end, we design a two-step experiment.

The first step is, for a clean Linux system, we try to inspect its processes internally and externally. For internal inspection, we mean inspecting its processes from inside the guest OS, which can be done by running `ps` command. For external inspection, we mean inspecting its processes from outside of the guest OS, i.e., by using HyperLink either via online analysis or via offline analysis in hypervisor. Since the guest OS is clean, the internal view and external view should be the same. Figure 5 shows the internal view of the clean system, due to limited space, we do not show the external view in the paper, but we observe that the internal view is indeed the same as the external view.

In the next step, we create a compromised Linux system. When a system is compromised, very often attackers install kernel rootkits. Typically the rootkit creates a malicious process, and by intercepting system calls, the rootkit can hide that malicious process and it is invisible from inside the OS. However, the malicious process is still in the process linked list. By reconstructing the process linked list from outside, we should be able to detect the existence of the malicious process.

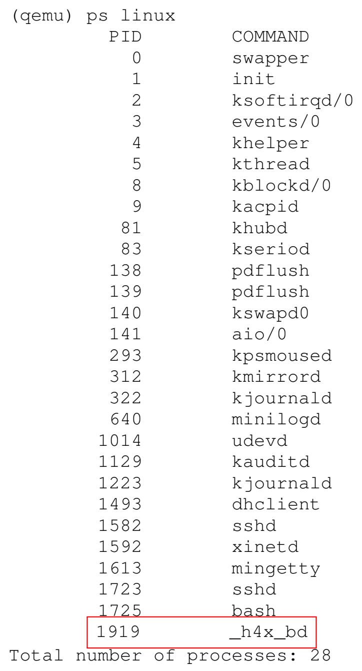
In our experiment, we employ several real world kernel rootkits for validation of our approach, including KBeast [17], Adore-ng [18], DR [19], [20], and Suterusu [21]. These rootkits have different implementation schemes, but one commonality shared between these rootkits is that they all achieve the malicious goal via a hidden process. Take the rootkit KBeast for instance, when it is installed, and we run the malicious binary _h4x_bd (provided by KBeast), it creates a malicious process named _h4x_bd. However, when we run the `ps` command inside the infected OS, the malicious process is not shown to the user. What we see from the internal view is still the same as we see in a clean OS (Figure 5). Then, we inspect the process list from outside. As Figures 6 and 7 show, by using HyperLink with either offline or online analysis, we can identify the hidden malicious process and HyperLink successfully help users to detect kernel rootkits process. Similarly, using this view-comparison approach, we verify that, the other kernel rootkits, including Adore-ng, DR, and Suterusu can also be detected.

```
[root@localhost ~]# ps -eo pid,comm
        PID     COMMAND
        1       init
        2       ksoftirqd/0
        3       events/0
        4       khelper
        5       kthread
        8       kblockd/0
        9       kacpid
        81      khubd
        83      kseriod
        138     pdflush
        139     pdflush
        140     kswapd0
        141     aio/0
        293     kpsmoused
        312     kmirrord
        322     kjournald
        640     minilogd
        1014    udevd
        1129    kauditd
        1223    kjournald
        1493    dhclient
        1582    sshd
        1592    xinetd
        1613    mingetty
        1723    sshd
        1725    bash
        1758    ps
```

Fig. 5: Internal view of a **clean** Cent OS 4.4 Linux system

```
[ system1 ]./bin/main -o linux  dump.cent44.2 dump.cent44.2
-------
OS Type: Linux
-------
         PID         COMMAND
           0         swapper
           1         init
           2         ksoftirqd/0
           3         events/0
           4         khelper
           5         kthread
           8         kblockd/0
           9         kacpid
          81         khubd
          83         kseriod
         138         pdflush
         139         pdflush
         140         kswapd0
         141         aio/0
         293         kpsmoused
         312         kmirrord
         322         kjournald
         640         minilogd
        1014         udevd
        1129         kauditd
        1223         kjournald
        1493         dhclient
        1582         sshd
        1592         xinetd
        1613         mingetty
        1723         sshd
        1725         bash
        1919         _h4x_bd
Total number of processes: 28
```

Fig. 6: Running HyperLink *offline* analysis against a **compromised** Cent OS 4.4 Linux memory dump



```
(qemu) ps linux
        PID        COMMAND
          0        swapper
          1        init
          2        ksoftirqd/0
          3        events/0
          4        khelper
          5        kthread
          8        kblockd/0
          9        kacpid
         81        khubd
         83        kseriod
        138        pdflush
        139        pdflush
        140        kswapd0
        141        aio/0
        293        kpsmoused
        312        kmirrord
        322        kjournald
        640        minilogd
       1014        udevd
       1129        kauditd
       1223        kjournald
       1493        dhclient
       1582        sshd
       1592        xinetd
       1613        mingetty
       1723        sshd
       1725        bash
       1919        _h4x_bd
Total number of processes: 28
```

Fig. 7: Running HyperLink *online* analysis from outside of a **compromised** Cent OS 4.4 Linux system

D. A Comparison between HyperLink and state-of-the-art Tools

We also perform a detailed comparison between HyperLink and the other state-of-the-art tools. First, we briefly introduce those state-of-the-art tools:

- VMwatcher: VMwatcher [4] is one of the pioneer works in the VMI area. Its key idea is that: assume kernel source code is available and the data structures of processes or file systems are known facts, these data structures can be used as a signature; by searching these signatures in memory, one can identify the process list, similarly, by searching these signatures in disk, one can identify various files.
- Virtuoso: Virtuoso [13] is another related work which aims to narrow the semantic gap in VMI. What makes Virtuoso different from previous VMI tools lies in that, it does not rely on kernel source, instead it introduces another OS. In other words, in its threat model, there are two virtual machines: one is the untrusted virtual machine and the other one is trusted or secured virtual machine. Running system utilities in the trusted guest OS multiple times, Virtuoso records all the instructions which they call instruction traces, and then it analyzes and merges these instruction traces for converting memory references from the trusted guest OS to the untrusted guest OS. In this way, it enables the system utility to run in the trusted guest OS and gather information from the untrusted guest OS.
- VMST: VMST [14] can be seen as a large step forward compared to Virtuoso. Again, it requires another secure virtual machine. The idea is to redirect system calls from the untrusted virtual machine to the secure virtual machine. In this way, it leverages the code from the secure guest OS and analyzes data from the untrusted guest OS and hence producing introspection results for the untrusted guest OS.
- Volatility: While the above tools work in the context of VMI, Volatility [22] is a pure memory forensic tool. Compared to the tools above, Volatility is the most mature product. It has been widely used for commercial and research purposes. One of its assumptions is the same as VMwatcher, i.e., either the source code of the target OS has to be known or one need to do some pre-processing steps to gain the knowledge of those key data structures Take Linux as an example, one has to write a kernel module, install the kernel module in the target OS, and this kernel module will compute the offsets. Thereafter, the layout of those key data structures will be captured. With such information, Volatility can then analyze memory dumps and extract security related information, such as process list, open connection, and kernel module list, from the memory dump.
- Volafox: Volafox [23] is similar to Volatility, but it is mainly designed for FreeBSD, Mac OS X, and Solaris, while Volatility only supports Windows and Linux (although now it also supports Mac OS X). Similarly, users either have to access kernel source code, or have access to the target OS. With this information, Volafox can then analyze memory dumps and extract security related information from the memory dump file, just as Volatility can do.

Here we can compare HyperLink with the above tools, and Table IV shows the comparisons. We can see from Table IV that HyperLink prevails against existing tools in the following aspects:

- VMwatcher, Volatility, and Volafox require source code or access to the target OS, but HyperLink does not.
- Virtuoso and VMST require an additional VM, i.e., the trusted OS, however, HyperLink does not require.
- While VMwatcher, Virtuoso, and VMST only enable online analysis, Volatility and Volafox only support offline analysis, HyperLink supports both online and offline.
- Virtuoso, Volatility, and Volafox require users to do certain preparation work, but HyperLink can run immediately. For example, Virtuoso requires a training stage and analysis stage before it can generate a tool for runtime use. And Volatility and Volafox both require users to install tools, compile modules, and run commands in the target system to gain kernel data structures and kernel symbols. In contrast, HyperLink does not need to perform these actions.

In summary, compared with those state of the arts tools, HyperLink has advantages for the following two reasons:

- From users' perspective, easy usability. Since HyperLink does not require any setup effort and an additional OS, users can run one simple command to perform the analysis.
- From developers' perspective, low maintenance cost. Since HyperLink is based on those invariants that do not change across different kernel versions, we do not need to frequently update HyperLink to support new kernel versions.

Because of these, we believe HyperLink can be better utilized in a cloud environment, in which there might be

TABLE IV: A Comparison Between HyperLink and State-of-the-Art Tools

Tool	Support OS	Online/Offline	Source Code	A Trusted OS	Setup
HyperLink	Windows, Linux, FreeBSD (Every Version)	Both Online and Offline	No	No	No
VMwatcher	Windows XP, Linux (Specific Versions)	Online Only	Yes	No	No
Virtuoso	Windows, Linux, Haiku OS	Online Only	No	Yes	Yes
VMST	Linux Only	Online Only	No	Yes	No
Volatility	Windows, Linux, Mac OS X	Offline Only	Yes	No	Yes
Volafox	Mac OS X, BSD, Solaris	Offline Only	Yes	No	Yes

thousands of guest OSes running different distributions and different kernel versions. Note that none of existing tools can support many different OSes/versions, and it is unlikely the cloud administrators/vendors would spend significant development efforts to support every individual guest OS. In this scenario, HyperLink will play an important role in VMI/memory forensic analysis. Note that we do not claim that HyperLink will replace these promising tools, but we expect it to be complementary to existing tools and offers system administrators one more option.

E. *Case Study: A More Detailed Comparison Between HyperLink and Volatility*

For a more detailed comparison with existing tools, we conduct an additional experiment. In this experiment, we consider that when a new release of an OS is out, how much efforts the developers as well as end users need to make, before they can do the introspection/forensic analysis on this new OS. Among the above state-of-the-art tools, VMwatcher, Virtuoso, and VMST are not publicly available. Furthermore, considering that Volatility is the most mature and widely used product, we choose it as our target for comparison.

After we had tested HyperLink against the OSes listed in Tables III, we realized that Microsoft was about to release its Windows 10 [24], [25] and it was in the public beta testing stage when the paper was written. Therefore, we downloaded Windows 10 Technical Preview Evaluation version from Microsoft website [26], installed it on our Qemu/KVM virtual machine, and then we tested HyperLink against Windows 10. Our experimental results showed that, without any changes to HyperLink source code, both the online and offline version of HyperLink could print the process list of the Windows 10.

We noticed that Volatility developers also announced that they had included the support for Windows 10 in their release, i.e., Volatility 2.4. By inspecting its source code, we observed that two python files were added: win10.py and win10_tp_x64_vtypes.py. The former has 48 lines of python code, and the latter has 11,616 lines of python code. Further investigation revealed that win10_tp_x64_vtypes.py was called the profile for Windows 10. More specifically, for each version of Windows, Volatility included a profile file in its source code, and the profile file described the key data structures of the Windows kernel. The profile file was generated as follows: although Windows is a closed source OS, Microsoft releases a debug package for each of its OS [27]. The debug package included the symbol information of the Windows kernel. By parsing the debug package, one can know the data structure information of the Windows kernel. In fact, Volatility developers use a third-party tool, called pdbparse [28], to analyze the debug package and then generate the profile file.

Therefore, when Windows 10 is rolled out, Volatility developers need to do: (1) add 48 lines of python code in win10.py; (2) download Windows 10 debug package, use the tool pdbparse to parse the debug package, and generate the profile file, i.e., win10_tp_x64_vtypes.py. In contrast, HyperLink developers need no changes to support Windows 10. *This manifests the low maintenance cost of HyperLink.*

Compared with Windows, Linux has more diverse kernels. Linux kernel is released more frequently than Windows and end users can also customize their own Linux kernels. It is not practical for Volatility developers to include a profile file for each Linux kernel version. Thus, Volatility developers provide a kernel module in its source code, and this kernel module is shipped with Volatility. According to Volatility's official document [29], when end users need to analyze a new kernel, they need to do: (1) compile and install the kernel module in the target system (which might be compromised already) and (2) run commands to create the profile file. Once the profile file is generated, the end users can then run Volatility to do the forensic analysis. In contrast, HyperLink end users does not require any work and support any versions of Linux kernel, regardless of its distribution or version. *This manifests the easy usability of HyperLink.*

VI. Discussion

In this section, we discuss some potential enhancements and limitations to HyperLink.

A. *Hypervisor Independency*

While our offline analysis version is an independent tool, our online analysis version is based on Qemu/KVM, in other words, we modify the Qemu source code for developing HyperLink. However, since HyperLink is implemented as a command, it is not closely coupled with any other parts of Qemu source code. Therefore, we believe that it is straightforward to port HyperLink to other hypervisors, such as Xen. Although we are not able to modify closed source hypervisor, such as VMware ESX server and Hyper-V, we believe that these hypervisor vendors can adopt the same approach.

B. *Direct Kernel Object Manipulation*

Using HyperLink, we can detect kernel rootkits that hijack system calls and filter the malicious process. However, there is a more sophisticated technique used by attackers called DKOM (Direct Kernel Object Manipulation). With such a technique, attackers, instead of filtering the malicious process, can directly remove the corresponding node from the linked list. Such an attack is built on the knowledge that kernel usually maintains two process linked lists: one is used by user level utilities, like ps command, and the other is used by the CPU scheduler. Therefore, removing its node from the former linked list does not prevent the malicious process from being scheduled. To address this problem, we can extend HyperLink to extract both linked lists from memory, any inconsistency between these two linked lists would suggest anomaly.

C. Limitations

On one hand, the process related information is essential to VMI and memory forensics as shown in previous sections. On the other hand, there are some other information that might also be helpful for VMI and memory forensics, such as open connections and loaded modules. Compared with those matured forensic tools like Volatility, HyperLink currently does not provide many features. However, since these kinds of information are also organized using a linked list, we believe that the same approach of HyperLink can be applied to extract these kinds of information from the memory, which will be our future work.

VII. Related Work

The related work can be broadly divided into two categories: virtual machine introspection (VMI) and memory forensics.

A. Virtual Machine Introspection (VMI)

VMI was first proposed by Garfinkel et al. [3] in 2003. Since then, researchers have made significant progress towards solving the semantic gap problem in VMI. Payne et al. [30] presented XenAccess, a monitoring library that leverages existing functionalities included in Xen, enables developers to access virtual memory, and provides virtual disk monitoring capabilities. Later, based on XenAccess, the same group of researchers then released LibVMI [31], [32], aiming to be extensible to other hypervisors, such as KVM. LibVMI itself is not a VMI tool, but provides APIs to simplify VMI development. They also integrated LibVMI into existing memory forensic tools like Volatility. While LibVMI's goal is to support multiple hypervisors, HyperLink's goal is to support multiple operating systems.

Dolan-Gavitt et al. [13] presented Virtuoso, a technique that can automatically extract introspection related kernel information by first monitoring the execution of an in-guest tool and then mimicking the same execution sequences. Fu et al. [14] proposed VMST tools that can automatically identify the introspection related data and redirect these data accesses to the in-guest kernel memory. Gu et al. [33] presented a process implanting technique with the idea that, instead of inspecting the guest OS from the outside, it can implant a host process inside the guest OS and protect that process from the hypervisor. In this way, the implanted process can monitor the guest OS internally (as opposed to externally). Hale et al. [34], [35] argued that it is useful to deploy certain services in the guest OS and place corresponding safeguard services in the hypervisor.

As Jain et al. [12] surveyed and summarized, compared to the original prototype [36], immense progress has been made in the last decade to improve the robustness and efficiency of VMI. However, as we illustrated in Section 5.4, current VMI tools are still inadequate for large scale, real-world environments, such as the cloud computing environments. We believe that HyperLink has its advantage of requiring significant smaller setup and maintenance efforts than the existing solutions.

B. Memory Forensics

Memory forensics is the analysis of a computer system's memory dump. Prior to 2004, this term is mainly used for debugging purposes. In 2004, it was introduced by Ford [37] to the field of security investigation by defining the concept of forensic analysis in the context of a computer crime: "Forensic analysis is the investigation of an event that involves looking for evidence and interpreting that evidence. In the case of a computer crime in which a system was compromised, the investigator needs to find out who, what, where, when, how, and why." Since then, a suite of memory forensic tools have been created, including commercial tools such as Memoryze [38], MoonSols Windows Memory Toolkit [39], and open source tools such as Volatility and Volafox. Nowadays, these tools are widely used for investigating computer attacks, in particular, those stealthy attacks that avoid leaving data on the system's hard drives. Among them, Volatility, because of its free and open source property, has arguably become the most popular memory forensic framework. In fact, it has been used, studied, and enhanced by many researchers [40], [41], [42]. Another memory forensic tool close to Volatility and HyperLink is Volatilitux [43]. Volatilitux is essentially the equivalent of the Volatility framework and shares some key heuristics with HyperLink; however, it is mainly used for investigating physical memory dumps of Linux systems and is not related to virtualization.

In addition, previous research has studied the problem of digging specific data structures from memory, such as [44], [45], and different approaches have been proposed. From the security's perspective, those specific data structures can be used as a signature for intrusion detection. The researchers have demonstrated that they can detect botnets and malware by identifying those data structures in memory.

VIII. Conclusion

In this paper, we have investigated the possibility of developing a one-for-many VMI and/or memory forensic tool. In particular, we have presented the design, implementation, and evaluation of the proposed tool called HyperLink. As the first step towards one-for-all, HyperLink has been tested against different OSes, including different versions of Linux, Windows, and FreeBSD. Our evaluation results show that HyperLink is able to accurately retrieve the process information of a target OS in a timely manner. HyperLink can be used for both online and offline analyses. By conducting online and offline analyses, we have demonstrated that HyperLink can detect real world kernel rootkits. We have also performed a detailed comparison between HyperLink and the state-of-the-art tools. Overall, HyperLink outperforms the existing tools in terms of easy usability and low maintenance cost. Therefore, HyperLink will help cloud vendors to fulfill the major objective of autonomic computing, system self-management, inside clouds.

IX. Acknowledgment

We would like to thank our shepherd Artur Andrzejak and the anonymous reviewers for their insightful comments and valuable feedback. This work was partially supported by ONR grant N00014-15-1-2136.

References

[1] D. M. Chess, C. C. Palmer, and S. R. White, "Security in an autonomic computing environment," *IBM Systems Journal*, vol. 42, no. 1, p. 107, 2003.

[2] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[3] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proceedings of the 10th Annual Symposium on Network and Distributed Systems Security (NDSS)*, 2003, pp. 191–206.

[4] 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 (CCS)*, 2007, pp. 128–138.

[5] 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 (CCS)*. ACM, 2008, pp. 51–62.

[6] 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," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, pp. 148–162, 2005.

[7] X. Jiang and D. Xu, "Collapsar: A vm-based architecture for network attack detention center." in *USENIX Security Symposium*, 2004, pp. 15–28.

[8] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *Recent Advances in Intrusion Detection (RAID)*. Springer, 2008, pp. 1–20.

[9] N. L. Petroni Jr and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, 2007, pp. 103–115.

[10] X. Xiong, D. Tian, and P. Liu, "Practical protection of kernel integrity for commodity os from untrusted extensions," in *Proceedings of the 18th Annual Symposium on Network and Distributed System Security (NDSS)*, 2011.

[11] L. Litty, H. A. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries." in *USENIX Security Symposium*, 2008, pp. 243–258.

[12] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, "Sok: Introspections on trust and the semantic gap," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2014, pp. 605–620.

[13] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2011, pp. 297–312.

[14] Y. Fu and Z. Lin, "Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2012, pp. 586–600.

[15] "Licensed application end user license agreement," http://www.apple.com/legal/macapps/dev/stdeula/.

[16] "Running mac os x as a qemu/kvm guest," http://www.contrib.andrew.cmu.edu/~somlo/OSXKVM/.

[17] IPSECS, "The kbeast rootkit," http://core.ipsecs.com/rootkit/kernel-rootkit/kbeast-v1/.

[18] stealth, "Announcing full functional adore-ng rootkit for 2.6 kernel," http://lwn.net/Articles/75991/.

[19] B. Alberts, "Dr linux 2.6 rootkit released," http://lwn.net/Articles/296952/.

[20] halfdead, "Mistifying the debugger, ultimate stealthness," http://phrack.org/issues/65/8.html.

[21] M. Coppola, "Suterusu rootkit: Inline kernel function hooking on x86 and arm," http://poppopret.org/2013/01/07/suterusu-rootkit-inline-kernel-function-hooking-on-x86-and-arm/#hide.

[22] "Volatility," http://www.volatilityfoundation.org/.

[23] "Volafox," http://code.google.com/p/volafox/.

[24] "The next generation of windows: Windows 10," http://blogs.windows.com/bloggingwindows/2015/01/21/the-next-generation-of-windows-windows-10/.

[25] "Windows 10's new features: Cortana, a 'spartan' browser, xbox streaming, and more," http://www.pcworld.com/article/2873219//windows-10s-new-features-cortana-on-the-pc-continuum-and-more.html.

[26] "Download windows 10 technical preview iso," http://windows.microsoft.com/en-us/windows/preview-iso.

[27] "Download windows symbol packages," https://msdn.microsoft.com/en-us/windows/hardware/gg463028.aspx.

[28] "pdbparse: Open-source parser for microsoft debug symbols (pdb files)," https://code.google.com/p/pdbparse/.

[29] "Instructions on how access and use the linux support," https://code.google.com/p/volatility/wiki/LinuxMemoryForensics.

[30] B. D. Payne, M. De Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2007, pp. 385–397.

[31] B. D. Payne, "Simplifying virtual machine introspection using libvmi," *Sandia Report*, 2012.

[32] "Libvmi," http://libvmi.com/docs/gcode-intro.html.

[33] Z. Gu, Z. Deng, D. Xu, and X. Jiang, "Process implanting: A new active introspection framework for virtualization," in *30th IEEE Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2011, pp. 147–156.

[34] K. C. Hale, L. Xia, and P. A. Dinda, "Shifting gears to enable guest-context virtual services," in *Proceedings of the 9th international conference on Autonomic computing*. ACM, 2012, pp. 23–32.

[35] K. C. Hale and P. A. Dinda, "Guarded modules: Adaptively extending the vmm's privilege into the guest," in *11th International Conference on Autonomic Computing (ICAC 14)*, 2014, pp. 85–96.

[36] T. Garfinkel and M. Rosenblum, "A virtual machine introspection-based architecture for intrusion detection," *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, pp. 191–206, 2003.

[37] "Linux memory forensics," http://www.drdobbs.com/linux-memory-forensics/199101801.

[38] "Memoryze," https://www.mandiant.com/resources/download/memoryze.

[39] "Moonsols windows memory toolkit," http://www.moonsols.com/windows-memory-toolkit/.

[40] S. Suneja, C. Isci, V. Bala, E. de Lara, and T. Mummert, "Non-intrusive, out-of-band and out-of-the-box systems monitoring in the cloud," in *The 2014 ACM international conference on Measurement and modeling of computer systems (SIGMETRICS)*. ACM, 2014, pp. 249–261.

[41] T. Haruyama and H. Suzuki, "One-byte modification for breaking memory forensic analysis," *Black Hat Europe*, 2012.

[42] C. Grobler, C. Louwrens, and S. H. von Solms, "A multi-component view of digital forensics," in *International Conference on Availability, Reliability, and Security (ARES)*. IEEE, 2010, pp. 647–652.

[43] "Volatilitux : Physical memory analysis of linux systems," http://www.segmentationfault.fr/projets/volatilitux-physical-memory-analysis-linux-systems/.

[44] A. Cozzie, F. Stratton, H. Xue, and S. T. King, "Digging for data structures," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2008, pp. 255–266.

[45] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, "Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures." in *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, 2011.