A Hypervisor Level Provenance System to Reconstruct Attack Story Caused by Kernel Malware

Chonghua Wang[1,4], Shiqing Ma[2], Xiangyu Zhang[2], Junghwan Rhee[3], Xiaochun Yun[1], and Zhiyu Hao[1(✉)]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
haozhiyu@iie.ac.cn
[2] Purdue University, West Lafayette, USA
[3] NEC Laboratories America, Princeton, USA
[4] School of Cyber Security, Chinese Academy of Sciences, Beijing, China

Abstract. Provenance of system subjects (e.g., processes) and objects (e.g., files) are very useful for many forensics tasks. In our analysis and comparison of existing Linux provenance tracing systems, we found that most systems assume the Linux kernel to be in the trust base, making these systems vulnerable to kernel level malware. To address this problem, we present HProve, a hypervisor level provenance tracing system to reconstruct kernel malware attack story. It monitors the execution of kernel functions and sensitive objects, and correlates the system subjects and objects to form the causality dependencies for the attacks. We evaluated our prototype on 12 real world kernel malware samples, and the results show that it can correctly identify the provenance behaviors of the kernel malware.

Keywords: Provenance tracing · Kernel malware
Forensic investigation

1 Introduction

Nowadays, enterprises are suffering from rapidly increasing serious attack threats, especially Advanced Persistent Threat (APT). Compared to traditional attacks, APT attacks are stealthier and more sophisticated by employing multi-step intrusive attacks. This kind of attacks would impose disastrous impacts on the systems if the associated attack vector aims at kernel [1]. Detecting such attacks is an urgent matter in enterprise environments, but is far from enough. In addition to detecting the existence of the attacks, deep investigation should be performed to find out where the attacks are, how the attacks are derived, and when they are introduced. For instance, a kernel mode attack can modify kernel objects or entities, which is potentially more dangerous. Acquiring such details about how the kernel objects and entities are manipulated is crucial to understand the attack for forensic investigations.

X. Lin et al. (Eds.): SecureComm 2017, LNICST 238, pp. 778–792, 2018.
https://doi.org/10.1007/978-3-319-78813-5_42

Provenance tracing [4,12,16–18,25] is an efficient approach to address these challenges since it can associate these events together to find the causality dependencies among them. The provenance records provide the holistic view of the whole system, thus can be well suited to system forensics. Even though the system is subverted by malware, provenance points out the possibility to restore the victim system to a good state in confidence. For a provenance system, the provenance information should be complete and faithful to provide the holistic view of the events occurred in the system for forensic applications. If the investigator fails to foresee the need for a particular kind of provenance information to be captured, then it would be difficult to rebuild the complete causality dependencies. Whereas an untrusted kind of provenance information could infer an innocent source.

State-of-the-Art: Lots of existing works employ audit logging to record events (e.g., memory reads and writes, process reading a file, messages being sent or received, etc.) during system execution and then correlate these events for building the causality dependencies during investigation [4,12,16–18,25]. These systems assume the Linux kernel to be in the trusted computing base (TCB), making these systems vulnerable to kernel malware. If an intruder employs a kernel malware to compromise the kernel, it is trivial to cheat or even undermine the audit logging, thus leading to inaccurate provenance results. However this assumption does not hold in practical settings in the examples of kernel malware.

Our Approach: The key to solve the above problem is to backtrack an untrusted kernel using an external monitor. Thus, we choose to employ virtualization techniques to exclude the kernel from our TCB to keep the provenance information secure and complete. In specific, we present a hypervisor level provenance tracing system, HProve, to address the above problems and complement existing provenance systems. On one hand, HProve ports the logging module to the hypervisor to keep the log recorded trustworthy, especially for kernel malware. On the other hand, in order to obtain complete provenance information, HProve employs lightweight record and replay techniques to record the whole execution of system and replay the system meanwhile instrumenting hypervisor for provenance. For efficiency, execution traces recorded do not include the state of emulated hardware devices focusing on the provenance tracing process rather than replaying a generic VM. HProve is able to replay and analyze a trace without having access to the VM image that was used for recording. Meanwhile to reduce runtime overhead, the instrumentation code is inserted into the hypervisor only when necessary during replay. After obtaining the execution traces, the backtracking technique is applied to the kernel APIs to find out the caller-callee chain using *function call convention*. HProve achieves this by our *provenance tap points uncovering* technique. In summary, we make the following contributions:

- We present HProve, a hypervisor level provenance tracing system that can replay kernel level malware attack to acquire accurate provenance details.
- To provide valuable insights about how kernel malware impacts on the kernel internals, we devise a novel approach to backtrack the kernel for acquiring

caller-callee chain of kernel functions reversely and correlate malware behaviors with tampered kernel objects to explore the causality dependencies.
– We have built a proof-of-concept prototype of HProve to demonstrate the feasibility of our approach. We have conducted extensive experiments with a variety of representative malware samples collected in the wild, and demonstrated that our system could correctly build the causality dependencies within the victim system.

2 Motivation

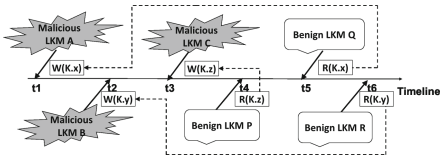

Fig. 1. An abstract diagram to illustrate a scenario that needs kernel malware attack provenance. W denotes *write* operation, R denotes *read* operation and K.x denotes kernel object x. The end that the dash line points to is the source of the data read by benign LKMs.

Kernel malware is considered as one of the most stealthy threats in computer security field and becomes a major challenge for security research communities [3,5,23] since it has the equal privilege as the kernel and often higher privileges than most security tools. We collect a variety of kernel malware samples and manually analyzed them. In summary, there are several categories that kernel malware falls into: system service hijacking (e.g., hooking *system call table* entries and replacing *system call table*), dynamic kernel object hooking (KOH, e.g., VFS hooking) and DKOM [20,23]. Recently lots of work were proposed to tackle this attack: kernel rootkit detection [10,19,24], kernel rootkit prevention [14,20,21] and kernel rootkit profiling [11,15,22,26]. However, detection is done after the victim system has been attacked, but the malware behaviors may have been missed. Prevention is adapted to detection systems, which is mainly to enforce kernel integrity, whereas it lacks the understanding of what had happened in the past. Profiling is capable of producing malware traces, such as hooking behavior, target kernel objects, user-level impact and injected code [26], whereas it fails to obtain the connections among these traces. These systems do not meet the goal of comprehensively revealing the causality dependencies among kernel malware behaviors and impacts on the victim system. For this goal, we need to solve three key challenges: *(1)* What kernel functions, kernel APIs and system calls have been called by malware?, *(2)* What kind of kernel objects (e.g., pointer fields and data values, etc.) have been accessed or damaged by malware?, *(3)* How to connect kernel malware behaviors and impacts on the victim system?

Scenario. Suppose a user wants to install a kernel driver and downloads a loadable kernel module (LKM) without being aware that it is malicious. The malicious LKM subverts important kernel objects (e.g., *K.x*, *K.y* and *K.z* as shown in Fig. 1) to hide itself and transfers confidential information. The system investigator inspects the victim system and starts scanning and monitoring work as usual. But nothing has been detected for some days which may raise questions

to the administrator. Also the user may download more than one malicious LKM which manipulates multiple kinds of kernel objects. What the system investigator needs to know is which LKM tampered with what kind of kernel objects. He has to design some investigation techniques to detect dependences among LKMs, files, kernel objects and memory accesses or even instructions and build causality dependencies through causal analysis of the historical events. Figure 1 shows that three different kernel malware issue malicious activities (e.g., hide processes, hide files and directories, etc.) by tampering with kernel objects (e.g., x, y, z, etc.) at different time t1, t2 and t3 respectively. At time t4, t5 and t6, the benign LKMs begin to read the tampered objects as usual. How the investigator knows where the kernel objects read by the benign LKMs come from? Have they been modified by the malicious LKM A or B or C? All these questions can be answered by kernel malware provenance (Fig. 2).

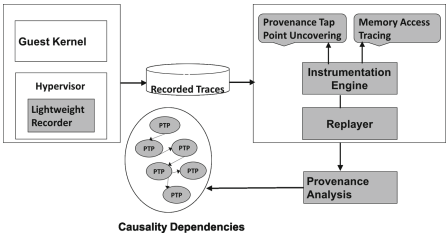

Fig. 2. System overview of HProve. PTP in the causality dependences denotes *provenance tap points* defined in next section.

3 System Overview

3.1 Scope, Assumptions and Threat Model

In this paper, we do not differentiate the terms of kernel malware and kernel rootkit. Both of them represent the kernel-mode components of malicious behaviors. They may issue malicious activities in different ways, but the essence is the same: they need to tamper with kernel objects. Regarding the scope of different categories of kernel malware and to focus on the provenance problem itself for kernel malware, system call hooking is our initial implementation decision for a prototype and our approach can be extended with other approaches which handle DKOM and VFS hijacking. Once the detection of DKOM and VFS hijacking is included [27], our method can perform provenance tracing from there.

We assume we can acquire the knowledge of kernel APIs, e.g., the kernel object allocation functions (e.g., *kmalloc/kfree*, *vmalloc/vfree*, *kmem_cache_alloc/kmem_cache_free*, etc.) so that we can instrument and track the creations and deletions of the kernel objects, and the kernel APIs as well as the function arguments. In addition, we assume that we can get knowledge of the *system call table* and the corresponding entries so that we can locate them in memory and reveal each access on them. Meanwhile, we assume the *function call conventions* is not variable so that we can infer the caller of kernel APIs accurately. As HProve is implemented on Linux, these assumptions are reasonable and practical.

We define a threat against HProve as any way of compromising the fidelity or completeness of the provenance information collected. HProve guarantees that even though the kernel is compromised by the adversaries, we can track the

tampered objects and further conduct provenance tracing. The hypervisor level attack is out of scope of HProve, and we can employ hypervisor integrity checking techniques such as [21] to ensure the intactness of the hypervisor before conducting provenance tracing.

3.2 Overview

HProve is designed to comprehensively reveal the causality dependences among kernel malware behaviors and impacts on the victim system. It is capable of obtaining a deep insight on what kind of behaviors kernel malware may conduct. HProve ports the logging module to the hypervisor to keep the log recorded trustworthy, especially for kernel malware. In order to obtain complete provenance information, HProve employs lightweight record and replay techniques to record the whole execution of system and replay the system meanwhile instrumenting hypervisor for provenance. In particular, the kernel functions being tracked include those being executed by the kernel from loading the kernel malware to allocating memory for them. With the captured execution traces, the backtracking technique is applied to the kernel functions to find out the caller-callee chain using *function call convention* in runtime. Meanwhile, HProve records memory accesses to sensitive kernel objects (e.g, *system call table*, etc.) that kernel malware may tamper with. HProve correlates these events happened within the kernel to reconstruct the attack story. For efficiency, execution traces recorded do not include the state of emulated hardware devices focusing on the provenance tracing process rather than replaying a generic VM. HProve is able to replay and analyze a trace without having access to the VM image. Meanwhile to reduce runtime overhead, the instrumentation code is inserted into the hypervisor only when necessary during replay.

4 Design and Implementation

In this section, we first present several definitions used in our approach. Then we describe the design and implementation of HProve in details.

4.1 Definitions

Provenance Tap Points. We define a *provenance tap point*, an execution point [7] in the kernel at which we wish to capture a set of function callers. It is defined as a four-tuple: (*call_site, func_entry, func_arg, func_ret_val*), where *func_entry* is the kernel function whose caller to be tracked, *func_arg* refers to the argument of the function, *func_ret_val* is the return value of the function and *call_site* denotes the caller of the function_entry.

Memory Access Trace. *Memory Access Trace* is used to connect the kernel events and function calls within the kernel, where each access m is formatted as a four-tuple: $m = (addr, data, type, program\_counter)$. *Addr* is the address of memory being accessed. *Data* is the amount of data written or read. *Type* is the type of the memory access (either a read or a write). *Program_counter* is the address of the instruction invoking the access.

4.2 Recording Non-deterministic Events

HProve leverages Panda [6], built atop on QEMU to record the non-deterministic events. Panda extends the original recording process of the QEMU and the recorded information can be replayed deterministically for the entire execution at any later time. Since the execution traces recorded do not include the state of emulated hardware devices, it does not support the execution of device code during replay. Fortunately, this feature satisfies our requirements. Eliminating the execution traces of device code helps to reduce the logging overhead significantly.

4.3 Instrumentation During Replay

QEMU Translation Block. The guest code is split into "translation blocks" (corresponds to a list of instructions terminated by a branch instruction). QEMU then translates them into an intermediate language using TCG (Tiny Code Generator), which provides the APIs to insert additional code. This intermediate translated block is converted into a corresponding basic block of binary code that can be directly executed on the host. Figure 3 shows how the guest code is transformed into translation blocks.

Instrumentation Before/After Execution. HProve instruments analysis code during replay to obtain the *Provenance Tap Point* and *Memory Access Trace*. As seen in the dashed translation block shown in Fig. 3, analysis code can be instrumented before or after the execution of each translation block by the instrumentation engine. We take LKM kernel malware as an example for describing our techniques. At the conceptual level, HProve works as follows.

First, it conducts off-line analysis of the typical execution route of kernel malware and reveals the common characteristics of them. We found that before loading a LKM malware, it is inserted into the kernel using utilities such as *insmod* or *modprobe*. Then the kernel initializes the LKM through system calls, calls *load_module* function to load the LKM, and allocates memory space for it. We set the *insmod* or *modprobe* operation as the start point and the allocating memory operation as the end point of the work done by kernel for all the LKMs. We define the timeline between the start point and the end point as *Top-Half*, and the timeline after the end point is defined as *Bottom-Half*. The analysis of the events ocurrs during *Top-Half* and *Bottom-Half* is completed by *Provenance Tap Point Uncovering* and *Memory Access Tracing* respectively.

Uncovering Provenance Tap Points. No matter what kind of objects will the kernel malware manipulate, its execution file should be allocated into the memory. Since HProve records whole execution of the running kernel, it instruments analysis code into the recorded traces to track the kernel allocation/deallocation related functions (e.g., *kmalloc/kfree*, *vmalloc/vfree*). Whenever these kinds of allocation/deallocation events occur at runtime, HProve replays the execution for capturing the allocated address range and location of the code that calls the memory allocation function. HProve determines the *call_site*, *func_entry*, *func_arg*, *func_ ret_val* for *Provenance Tap Point* in the replay phase. HProve instruments provenance code before (after) the execution of each basic block during replay as depicted in Fig. 3. Take an allocation function (e.g.,*vmalloc*) as a *func_entry*, the address of objects being allocated can be determined by the *func_arg*, and the size of object can be determined by *func_ret_val*.

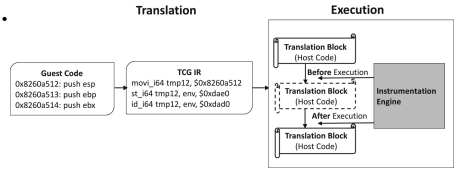

Fig. 3. Illustration on how our instrumentation engine works during replay

Take a deallocation function (e.g., *vfree*) as a *fuc_entry*, the address of objects being deallocated can be determined by the *func_arg*. *Call_site* determines which function calls the *func_entry*. Each item of the *Provenance Tap Point* can be captured by analyzing *function call conventions* within the hypervisor. To capture the *call_site*, HProve uses the return address of the call to *func_entry*. In the instruction stream, the return address is the address of the instruction after the *CALL* instruction. *Func_arg* and *func_ret_val* can be captured through the stack or registers. Integers up to 32-bits as well as 32-bit pointers are delivered via the *EAX* register. *Func_arg* is delivered through the *EBP* with corresponding offsets. *Func_arg* and *func_ret_val* are only available when *func_entry* returns to the call_site. In order to capture *func_arg* and *func_ret_val* at the correct time, HProve uses a shadow stack to store these values. Specifically, HProve checks if it ends with a *CALL* instruction after each basic block executes during replay. If so, the return address is pushed into a shadow stack. Correspondingly, before execution of each basic block, HProve checks whether it matches a return address on the shadow stack; If so, we know that the current function has returned, thus HProve pops it from the shadow stack and captures the return value from the *EAX* register as well as the function arguments from *EBP* with corresponding offsets. Then HProve reads the value from the registers and memory addresses using the introspection technique [8]. The obtained values of *provenance tap points* will be stored in the form of (*calle_site, func_entry func_arg,func_ret_val*).

Memory Access Tracing. After malware being allocated into the memory, it is able to start carrying out malicious activities. These events occur in the phase of *Bottom-Half*. Typically, LKM malware would try some tricks (e.g., bypass *CR0* protection and search for *System.map* file) to get the entry address of

system call table, and manipulate the relative *system call entries* for different purposes. SYSTEM keeps track of the changes of these entries, obtains the allocated memory region of the *system call table* and records memory access of the memory region. Fortunately, there are a few hundreds of entries in the *system call table* (e.g., 350 and 312 entries in Linux 3.2 kernel for 32-bit and 64-bit respectively), thus only a few hundreds of memory addresses are to be tracked by HProve. Note that the writes to system call table entries make the relative system call service routine points to the malicious function in kernel malware, which are considered as suspicious. Specifically, if there is a write, HProve records the *PC* that initiates the *write* operation. The retrieved values of *memory access traces* will be stored in the form of *m = (addr, data, type, program_counter)*.

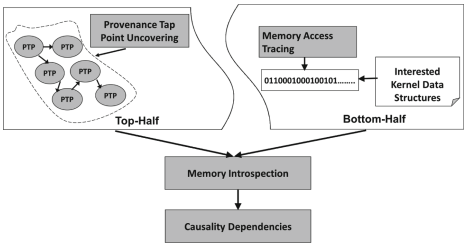

Fig. 4. Building causality dependencies among kernel malware behaviors and impacts on the victim system. PTP denotes *provenance tap point*

4.4 Causality Dependencies

To build causality dependencies, HProve uncovers the connections among the events occurr in the *Top-Half* and *Bottom-Half*. When the allocation function allocates memory for LKM malware, HProve acquires the address range that is being allocated by interpreting the *func_arg*. Then HProve gets a address range that is being allocated for the LKM malware. Once the *PC* is captured during *Memory Access Tracing*, HProve checks whether the *pc* locates within one of the address range that has been allocated for malware. If so, HProve correlates the writes on *system call entries* with the *func_entry* that execute the allocation. Then HProve determines the *call_site* of the *func_entry* that executes the allocation by the *Provenance Tap Point Uncovering* technique. Through backtracking successively, HProve acquires the complete *call_site* to determine the original malware source that initials the write operation on *system call entries* (Fig. 4).

5 Evaluation

In this section we present the effectiveness of using HProve to build causality dependencies among kernel malware behaviors and impacts on the system. Then we evaluate HProve's efficiency to show that our approach does not incur significant overheads. In our experiments, the host machine is an Intel Core i5 desktop running Ubuntu 12.04. We use Linux kernels as the guest VM. To validate our experiments results with the ground truth, we have collected 12 kernel malware samples that contain a mix of malicious capabilities found in the wild, including 10 system services hijacking malware (e.g., *kbeast*, *xinqyiquan*, etc.), 1 DOH malware (e.g., adore-ng-.0.56), and 1 DKOM malware (e.g., hp rootkit).

5.1 Effectiveness

To evaluate the effectiveness of our system, we should obtain *provenance tap points* and *memory access traces* of the targeted kernel objects accurately with HProve. In the experiment setup, HProve loads 12 kernel malware samples and 6 benign LKMs into the guest kernel. Once all of these modules are loaded into the kernel, HProve starts recording whole execution of the guest kernel with the lightweight recorder. Then the recorded traces are instrumented with provenance code during its replay to obtain *provenance tap points*, and *memory access traces*. After that provenance information is retrieved to build the causal dependencies.

Provenance Tap Points. The utilities that insert LKMs encapsulate *sys_init_module* which performs initialization and calls the *load_module* function. This function is responsible for loading the LKM from the user space to the kernel space. First, it calls the *copy_and_check* function which calls the *vmalloc* function to allocate temporary memory for copying the LKM file into the memory region. Second, the *load_module* function calls *layout_and_allocate* to allocate the final memory for a specific section of the LKM (e.g., *core space*, *.init.text*, etc). The remaining caller-callee relationship chain is shown as below:

$$layout\_and\_allocate \longrightarrow move\_module \longrightarrow module\_alloc\_update\_bounds \longrightarrow module\_alloc \longrightarrow \_\_vmalloc\_node\_range.$$

After initialization, allocation and relocation are finished, and the LKM can execute as expected. With this prior knowledge, HProve treats these functions as the *function_entry* of one of the *provenance tap points*. Take *__vmalloc_node_range* as an example, it is used for allocating specific pages in physical memory for LKMs. We can infer other items of *provenance tap points* (e.g., *call_site*, *function_argument*, *function_return_value*) with *provenance tap point uncovering* and memory introspection techniques [8]. Specifically, once we have inferred *module_alloc_update_bounds*, HProve acquires the allocation information of LKMs including the address range from the *provenance tap point*. The address range is critical for HProve to link the causality dependency between *Top-Half* and *Bottom-Half* as discussed in Sect. 4.4. In our experiments, HProve uncovers *provenance tap points* for all kernel malware samples. The address range allocated for each malware sample is shown in Table 1. Since DKOM type malware are loaded into kernel in terms of */dev/kmem*, we do not list it in the table.

Memory Access Traces. Before building the complete causality dependencies, the memory region which the LKMs belong to needs to be identified. HProve achieves this by recording the memory access to the system call table for the running malware. We then build the *Memory Access Trace* tuple for each system call entry manipulated by each kernel malware. In the tuple, *PC* is critical field to determine which LKM is manipulating the relative system call entry. As discussed above, HProve acquires various memory regions that are allocated for the LKMs loaded into the kernel. If *PC* follows in one of the memory

Table 1. Allocated start address range for each kernel malware

Address range	Kbeast	Xingyiquan	Suterusu	Knark	Enyelkm	Synapsys	Rial	Kis	Kbdv3	Adore-0.42	Adore-ng0.56
Start address	0xf86-73000	0xf86-82000	0xf86-85000	0xf86-83000	0xf86-75000	0xf86-77000	0xf86-71000	0xf86-89000	0xf86-68000	0xf86-79000	0xf86-64000
Size/ bytes	215	308	276	413	356	218	196	525	298	418	382

regions, then the two events are correlated. A table for the *Memory Access Trace* tuples is constructed for each kernel malware sample. Table 2 shows one of the results obtained by HProve. As we can see, in the second row, *_NR_open* entry is located at *0xc1541234* and has been written by *PC 0xf867445f*. HProve refers to the result of Table 1 and determines that this *PC* and other *PCs* in Table 2 belong to the memory region allocated for *Kbeast*.

After correlating *memory access traces* with *provenance tap points*, HProve is able to identify which malware manipulates which kind of kernel objects. Table 3 shows the system call entries that are manipulated by kernel malware samples of system services hijacking we collect. For instance, *Kbeast* tampered with *_NR_open*, *_NR_read*, *_NR_write*, *_NR_rmdir*, *_NR_unlink*, etc. We also analyze the source code of all the malware samples for the validation purposes, and it turned out that the entries discovered by our provenance tracing method correctly matched the malware behaviors in the source code.

5.2 Efficiency

We conduct several experiments to evaluate the efficiency of HProve. In the first experiment setup, we insert all the LKM samples, including the malicious and benign ones into the guest kernel and start HProve. Once the kernel begins to load these samples, HProve records the execution once, and then replays it multiple times for different provenance requirements. In the following experiments, we insert one malware sample into the kernel at a time and repeat 10 times. For each case, we report the recording time, the size of a record, the size of a memory trace, and the replay time in Table 4. The second column of Table 4 presents the recording time of the sample's execution. The third column shows the size of impact traces that are recorded by the lightweight recorder of HProve. The forth column lists the size of memory access traces of the system call entries. The fifth and sixth columns present the replay time for *Provenance Tap Points Uncovering* and *Memory Access Tracing* respectively.

As we can see, a record size in the table is at most 30 MB for the evaluated LKM samples, which is acceptable for these samples executing millions of instructions. Since there are only a few hundreds of memory addresses to be tracked, the size of memory traces is at most 17 KB. The duration of replaying *Memory Access Tracing* for all LKM samples is 113 min and the average duration of replaying *Memory Access Tracing* for each malware sample is 32.2 min. Replaying for uncovering *Provenance Tap Points* took 62 min for all LKM samples and 11.8 min for each malware sample in average.

Table 2. One of *memory access trace* table obtained by HProve.

Data	Addr	Type	PC
_NR_open	0xc1541234	W	0xf867445f
_NR_read	0xc154122c	W	0xf86743b4
_NR_write	0xc1541230	W	0xf86743c9
_NR_rmdir	0xc15412c0	W	0xf867411
_NR_unlink	0xc1541248	W	0xf86743f9
_NR_rename	0xc15412b8	W	0xf8674447
_NR_kill	0xc15412b4	W	0xf8674477
_NR_getdents64	0xc1541590	W	0xf86743e1
_NR_unlinkat	0xc15416d4	W	0xf867442c
_NR_delete_module	0xc1541424	W	0xf86743d4

Table 3. Manipulated system call entries. '√' denotes that the entry has been manipulated.

System call entry	Kbeast	Xingyiquan	Suterusu	Knark	Enyelkm	Synapsys	Rial	Kis	Kbdv3	Adore-0.42
_NR_open	√	√				√	√	√		√
_NR_read	√		√	√			√			
_NR_write	√		√			√				√
_NR_rmdir	√	√						√		
_NR_mkdir								√		
_NR_unlink	√	√						√		
_NR_chdir		√						√		
_NR_kill	√	√		√	√	√				√
_NR_fork				√		√		√		√
_NR_ioctl				√						
_NR_close										√
_NR_clone				√		√	√	√		√
_NR_exit								√		
_NR_execve				√						
_NR_rename	√	√						√		
_NR_utime									√	
_NR_unlinkat	√									
_NR_socketcall								√		
_NR_getdents				√		√	√	√		
_NR_gentdents64	√			√	√					
_NR_getuid						√				
_NR_getuid32									√	
_NR_gettimeofday										
_NR_quiry_module						√	√			
_NR_init_module								√		
_NR_delete_module	√									
_NR_stat								√		
_NR_lstat								√		

6 Discussion

HProve employs Panda [6] to record the whole execution of system, it shares the overhead with Panda for keeping track of instructions and the program counter at the instruction level. On average, for every 1 min of recorded execution, the replay takes 30 min It so far is not easy to port it to real systems even though the replay phase could be done off-line. We consider to use introspection technique with hardware virtualization instead of record-and-replay (e.g., PANDA) to keep track of a series of kernel functions (e.g., *kmalloc*, *vmalloc*, *load_module*, etc.). However, Jain et al. [9] had shown that there are non-trivial challenges associated with introspection because of the *strong semantic gap problem* without trusting the kernel. Regarding the scope of different categories of kernel malware and to focus on the provenance problem itself for kernel malware, system call hooking is our initial implementation decision for a prototype. HProve can not deal with all the types of kernel malware (e.g., DKOM and VFS hijacking). The system will fail if an object that are not being tracked is modified (e.g., the malware creates new kernel objects with altered semantics). We have tested a type of DKOM and VFS hijacking malware (e.g., *hp rootkit*, *adore-ng-0.56*) that can elude our system. But our approach can be easily extended with other approaches which handle DKOM and VFS hijacking. Once the detection of DKOM and VFS hijacking is included [2,27], our method can perform provenance tracing from there. Other than *system call table*, we can keep track of other sensitive kernel objects that DKOM or VFS hijacking malware may manipulate. We leave the above limitations of HProve to our future work.

Table 4. Evaluation for space and time for provenance

Sample	Recording time	Record size	Memory traces size	Replaying time	
				Provenance tap points	Memory access tracing
Kbeast	1.2 min	26 MB	11 KB	13 min	50 min
Xingyiquan	0.8 min	17 MB	7 KB	12 min	33 min
Suterusu	0.2 min	4 MB	2 KB	10 min	10 min
Knark	1.1 min	24 MB	10 KB	13 min	45 min
Enyelkm	0.3 min	6 MB	3 KB	10 min	12 min
Synapsys	1.1 min	25 MB	12 KB	14 min	51 min
Rial	0.4 min	9 MB	3 KB	11 min	13 min
Kis	1.5 min	30 MB	17 KB	14 min	78 min
Kbdv3	0.3 min	5 MB	2 KB	10 min	9 min
Adore-0.42	0.6 min	14 MB	5 KB	11 min	21 min
All LKMs	11 min	148 MB	80 KB	62 min	113 min

7 Related Work

Kernel Malware: Many researchers have studied the behaviors of kernel malware and proposed lots of effective approaches to detect their existence. HookFinder [15] identifies all the impacts made by the malicious code and keeps track of the impacts flowing across the system to identify the hooking behavior of a rootkit in the kernel execution. HookMap [24] employs a more elaborate method to identify all potential hook in the execution path of kernel code that could be utilized by the kernel level malware. K-Tracer [11] discovers information about rootkit capabilities through its data manipulation behavior to help defend against rootkit as well as user-level malware that gets help from them. PoKeR [22] is a kernel rootkit profiler that generates multi-aspect kernel rootkit profiles (e.g.,hooking behavior, targeted kernel objects, user-level impacts and injected code) during rootkit execution. Rkprofiler [26] is also a kernel malware profiler that can track both pointer-based and function-based object propagation, while PoKeR only tracks the pointer-based object propagation. To complement these work, our work analyzes the behavior of kernel malware reversely (from bottom to top and from impact to cause) which is orthogonal to theirs.

Provenance Tracing: Provenance tracing provides the ability to describe the history of a data object, including the conditions that led to its creation and the actions that delivere it to its present state. Hi-Fi [18] leverages Linux Security Module to collect a complete provenance record from early kernel initialization through system shutdown. It maintains the fidelity of provenance collection under any user space compromise. BEEP [12] instruments an application binary at the instructions and use the Linux audit system to capture the system calls triggered by the application for investigating which application brings the malware into the system for provenance. LogGC [13] employs the garbage collection method to prune some system objects such as temporary files that have a short life-span and have little impact on the dependency analysis to save space. ProTracer [16] proposes to combine both logging and unit level tainting techniques, aiming at reducing log volume to achieve cost-effective provenance tracing. Bates et al. [4] proposes Linux Provenance Module, a generalized framework for the development of automated, whole-system provenance collection on the Linux. However, these systems rely on the safety of provenance collector (e.g., Linux audit system, Linux Security Module). In the events of kernel malware, the adversary is able to compromise the provenance collector or even the kernel, which makes the provenance results untrusted. Our contribution is to complement these techniques by porting the provenance collector as well as the analysis module into the hypervisor for the resistance to kernel level malware.

8 Conclusion

We develop HProve, a hypervisor level provenance tracing system that can backtrack the causality dependencies among impacts on a victim system and kernel malware behaviors. It is capable of understanding the kernel APIs triggered

and the objects manipulated by kernel malware. HProve is a new system that provides the capability of replaying kernel malware attack story for provenance tracing. Such hypervisor level technique is needed in current cloud computing environment. Due to the limitations of HProve discussed in Sect. 4, more efficient designs for kernel malware provenance are still highly needed.

Acknowledgement. We would like to thank the anonymous reviewers for their insightful comments that greatly helped improve this paper. This work is a part of the project supported by Beijing Municipal Science Technology Commission (Z161100002616032), Beijing Natural Science Foundation (4172069) and a joint Ph.D program funded by Chinese Academy of Sciences.

References

1. Unmasking kernel exploits. https://www.lastline.com/labsblog/unmasking-kernel-exploits/
2. Aristide, F., Andrea, L., Davide, B., Engin, K.: Hypervisor-based malware protection with AccessMiner. Comput. Secur. **52**, 33–50 (2015)
3. Bahram, S., Jiang, X., Wang, Z., Grace, M., Li, J., Srinivasan, D., Rhee, J., Xu, D.: DKSM: subverting virtual machine introspection for fun and profit. In: SRDS, pp. 82–91 (2010)
4. Bates, A., Tian, D., Butler, K., Moyer, T.: Trustworthy whole-system provenance for the Linux kernel. In: USENIX Security, pp. 319–334 (2015)
5. Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., Jiang, X.: Mapping kernel objects to enable systematic integrity checking. In: CCS, pp. 555–565 (2009)
6. Dolan-Gavitt, B., Hodosh, J., Hulin, P., Leek, T., Whelan, R.: Repeatable reverse engineering with panda. In: Proceedings of 5th Program Protection and Reverse Engineering Workshop, pp. 4:1–4:11 (2015)
7. Dolan-Gavitt, B., Leek, T., Hodosh, J., Lee, W.: Tappan zee (north) bridge: mining memory accesses for introspection. In: CCS, pp. 839–850 (2013)
8. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: NDSS, pp. 191–206 (2003)
9. Jain, B., Baig, M.B., Zhang, D., Porter, D.E., Sion, R.: SoK: introspections on trust and the semantic gap. In: Proceedings of 35th IEEE S&P, pp. 605–620 (2014)
10. Jiang, X., Wang, X., Xu, D.: Stealthy malware detection through VMM-based out-of-the-box semantic view reconstruction. In: CCS, pp. 128–138 (2007)
11. Lanzi, A., Sharif, M., Lee, W.: K-tracer: a system for extracting kernel malware behavior. In: NDSS (2009)
12. Lee, K., Zhang, X., Xu, D.: High accuracy attack provenance via binary-based execution partition. In: NDSS (2013)
13. Lee, K., Zhang, X., Xu, D.: LogGC: garbage collecting audit log. In: CCS, pp. 1005–1016 (2013)
14. Li, J., Wang, Z., Jiang, X., Grace, M., Bahram, S.: Defeating return-oriented rootkits with "return-less" kernels. In: EuroSys, pp. 195–208 (2010)
15. Liangnd, Z., Yin, H., Song, D.: HookFinder: identifying and understanding malware hooking behaviors. In: NDSS, pp. 41–57 (2008)
16. Ma, S., Zhang, X., Xu, D.: ProTracer: towards practical provenance tracing by alternating between logging and tainting. In: NDSS (2016)

17. Pei, K., Gu, Z., Saltaformaggio, B., Ma, S., Wang, F., Zhang, Z., Si, L., Zhang, X., Xu, D.: HERCULE: attack story reconstruction via community discovery on correlated log graph. In: ACSAC, pp. 583–595 (2016)
18. Pohly, D., McLaughlin, S., McDaniel, P., Butler, K.: Hi-Fi: collecting high-fidelity whole-system provenance. In: ACSAC, pp. 259–268 (2012)
19. Rhee, J., Xu, D., Riley, R., Jiang, X.: Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory. In: RAID, pp. 178–197 (2010)
20. Rhee, J., Riley, R., Xu, D., Jiang, X.: Defeating dynamic data kernel rootkit attacks via VMM-based guest-transparent monitoring. In: 2009 International Conference on Availability, Reliability and Security, pp. 74–81 (2009)
21. Riley, R., Jiang, X., Xu, D.: Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In: RAID, pp. 1–20 (2008)
22. Riley, R., Jiang, X., Xu, D.: Multi-aspect profiling of kernel rootkit behavior. In: EuroSys, pp. 47–60 (2009)
23. Rudd, E., Rozsa, A., Gunther, M., Boult, T.: A survey of stealth malware: attacks, mitigation measures, and steps toward autonomous open world solutions. IEEE Commun. Surv. Tutor. **PP**(99), 1–28 (2016)
24. Wang, Z., Jiang, X., Cui, W., Wang, X.: Countering persistent kernel rootkits through systematic hook discovery. In: RAID, pp. 21–38 (2008)
25. Xu, Z., Wu, Z., Li, Z., Jee, K., Rhee, J., Xiao, X., Xu, F., Wang, H., Jiang, G.: High fidelity data reduction for big data security dependency analyses. In: CCS, pp. 504–516 (2016)
26. Xuan, C., Copeland, J., Beyah, R.: Toward revealing kernel malware behavior in virtual execution environments. In: RAID, pp. 304–325 (2009)
27. Zeng, J., Fu, Y., Lin, Z.: Automatic uncovering of tap points from kernel executions. In: RAID, pp. 49–70 (2016)