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

Chonghua Wang1, Libo Yin1,∗, Jun Li1, Xuehong Chen1, Rongchao Yin1, Xiaochun Yun2, Yang Jiao3, Zhiyu Hao3,

1China Industrial Control System Cyber Emergency Response Team, Beijing, China, 100040
2National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing, China, 100029
3Institute for Information Engineering, Chinese Academy of Sciences, Beijing, China, 100093

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 with a minor performance overhead.

Received on 12 November 2018; accepted on 17 December 2018; published on 25 January 2019

Keywords: Provenance Tracing; System Logging; Kernel Malware; Forensic Investigation

Copyright © 2019 Chonghua Wang et al., licensed to EAI. This is an open access article distributed under the terms of the Creative Commons Attribution license (http://creativecommons.org/licenses/by/3.0/), which permits unlimited use, distribution and reproduction in any medium so long as the original work is properly cited.
doi:10.4108/eai.8-4-2019.157417

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 [1–3, 6, 7]. 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.

Provenance1 tracing [8, 12, 24, 29, 33, 34, 44] is an useful technique for security investigation that can provide a detailed record of the origin and evolution of events and entities in a system. Given a corrupted entity (e.g., a file, a data structure, a pointer, etc.), it could help to answer two questions:

1. What-provenance: What is the source/entry point of the corrupted entity? Which other entities in the system were derived from (and corrupted by) the entity?

2. How-provenance: Building causality dependencies to show the events/entities that led to the corruption of the entity and those that have been further corrupted by the entity.

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

∗Corresponding author. Email: yinlibo@cics-cert.org.cn

1The term provenance comes from the art world, where it refers to the ability to trace all activities related to an piece of art, in order to establish that it is genuine[11].
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 [8, 12, 24, 29, 33, 34, 44]. Specifically, Bates et al. [12], present Linux Provenance Modules (LPM) framework to capture whole-system provenance including a detailed record of processes, IPC mechanisms, network activities and the kernel itself. LPM takes for the kernel mechanisms, provenance recorder and storage back-ends as the Trusted Computing Base (TCB). There is no mechanism for protecting LPM from the rest of the kernel meaning that it trusts the kernel code. 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 use virtualization techniques to solve this problem. The kernel itself is excluded from our TCB and we only trust the hypervisor. The hypervisor in general has a smaller code base, and is more trustworthy [32]. In specific, we present a hypervisor level provenance tracing system, HProv2, to address the above problems and complement existing provenance systems. On one hand, HProv2 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, HProv2 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. HProv2 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. HProv2 achieves this by our provenance tap points uncovering technique.

In summary, we make the following contributions:

- We present HProv2, 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 HProv2 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. Background and Motivation

In this section, we give a brief introduction of kernel malware and describe the motivation of our approach.

2.1. Background

A kernel malware is typically used by loading a malicious kernel module into the kernel and then interacting with the kernel data to hide itself without being detected. As an example, an investigator may employ monitoring tools to find malicious files in directories, whereas a kernel-based malware may first detect such attempts and delete the malicious files before the kernel returns the identification of the files or return an empty result. To achieve an malicious goal, the kernel-mode components of malware typically employ hooking or DKOM (Direct Kernel Object Manipulation) strategies [4]. For hooking, the malware hijacks the key functionalities of the operating system such as the system call table, VFS (Virtual File System) functions, or IDT (Interrupt Descriptor Table) and then points to malicious functions. They are loaded in terms of LKM (Loadable Kernel Module) that have the same privilege of kernel. For DKOM, adversaries directly tamper with pointers fields or data values of sensitive kernel objects to hide or manipulate the OS semantics. DKOM adversaries are loaded through the kernel memory devices such as /dev/kmem. Such devices give access to the memory region occupied by the running

---

2A function call convention is a scheme to pass function arguments and a return value. We use the conventions for the x86 architecture and the gcc compiler.
Kernel malware is considered as one of the most stealthy threats in computer security field and becomes a major challenge for security research communities since it has the equal privilege as the kernel and often higher privileges than most security tools. Recently lots of work were proposed to tackle this attack: kernel rootkit detection [21, 22, 37, 43, 48], kernel rootkit prevention [26, 36, 38, 42] and kernel rootkit profiling [23, 27, 39, 45]. However, these works suffer from several drawbacks. Specifically, 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 [45]. However, profiling does not focus on obtaining 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?

Provenance tracing 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.

Limitations of the State-of-the-Art. Existing systems [12, 24, 25, 29, 34] make the assumption that the kernel is trusted, which is usually not the case. There are following concerns on these systems: Circumvention), the adversary may attempt to hide its behaviors by circumventing the provenance recorder. As an example, the malware may unlink itself from the module list provided by /proc/modules, which makes the malware behaviors stealthy; Deception), the adversary may trick the provenance recorder to collect inaccurate information. For example, the adversary may use a malicious system call handling function to send fake behavior to the system; Termination), the adversary may kill the provenance recorder process to make the system unable to track provenance.

Regarding these concerns, we have studied provenance systems like LPM [12], BEEP [24], LogGC [25], ProTracer [29], Hi-Fi [34] and analyzed their features in terms of system objects, provenance collector and provenance handler. We illustrate these features regarding provenance systems aiming at user space and kernel space malware respectively. System objects are critical for provenance systems to be recorded. They are mainly composed of files, processes, IPCs, sockets, etc in user space malware provenance, whereas kernel malware provenance is aimed at kernel APIs, kernel data structures, memory regions, instructions, etc. The granularity of recorded system objects determines whether the provenance information collected is complete or not. Provenance collector is responsible for observing and recording system objects and the related events. For user space malware provenance, the provenance collector mainly places hooks and analysis codes into the kernel or user space to capture a variety of events: file reads and writes, process communication, network communication, etc. For kernel malware provenance, the provenance collector should trace the entire kernel to capture kernel API calls, kernel objects changes, memory accesses, etc. Note that, user space malware provenance systems trust the kernel, whereas kernel space malware provenance systems exclude the kernel from the Trusted Computing Base. To achieve fidelity, the provenance collector can be deployed to hypervisors. Provenance handler is responsible for correlating the events and system objects to build the causality dependencies. For user space malware provenance systems, it is implemented in user space whereas kernel malware provenance systems implement provenance handler in hypervisor level. Many provenance applications can be deployed atop provenance handler, such as interpreting, processing and storing collected provenance data. User space malware provenance systems (e.g, BEEP [24], ProTracer [29], etc.) may aim to find out which process/thread (e.g, firefox, pine, etc.) or the specific link within a program brings in the malware source. Whereas kernel malware provenance systems concern on the entire impacts on the kernel.

Table 1 presents our analyzed results. Specifically, the second column shows the aimed system objects of these systems. The third and fourth columns present log information and implementation of the provenance collector respectively (e.g., ProTracer employs tracepoint, implemented in kernel space, to log selective syscalls that can induce causality with system objects
Table 1. Study of State-of-the-art Provenance Systems

<table>
<thead>
<tr>
<th>Systems</th>
<th>System Objects</th>
<th>Provenance Collector</th>
<th>Provenance Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Log</td>
<td>Implementation</td>
</tr>
<tr>
<td>BEEP</td>
<td>Sockets, processes, files, IPC, etc</td>
<td>All syscalls</td>
<td>Linux audit /user</td>
</tr>
<tr>
<td>LogGC</td>
<td>Sockets, processes, files, IPC, etc</td>
<td>All syscalls</td>
<td>Linux audit /user</td>
</tr>
<tr>
<td>ProTracer</td>
<td>Sockets, processes, files, IPC, etc</td>
<td>Selective syscalls</td>
<td>Tracepoint /kernel</td>
</tr>
<tr>
<td>Hi-Fi</td>
<td>Sockets, processes, files, IPC, etc</td>
<td>Syscalls, vfs hooks</td>
<td>Linux security module /kernel</td>
</tr>
<tr>
<td>LPM</td>
<td>Users, groups, processes, inodes, sockets, memory, IPC</td>
<td>Kernel function hooks</td>
<td>Linux security module/kernel</td>
</tr>
</tbody>
</table>

![Figure 1](image-url)  
**Figure 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.

Motivating Scenario. Suppose a user wants to install a kernel driver and downloads a 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 Figure 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.

Fig. 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.

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. According to what we have discussed in Section 2.1, kernel malware may issue malicious activities in different ways, but the essence of them 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 [47], our method can perform provenance tracing from there. And we did not consider kernel ROP or other advanced kernel malware in this paper.

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 provenance component is responsible for retrieving provenance information by analyzing the standard function call conventions and building complete causality dependencies regarding impacts from kernel malware to a victim system. HProve supports off-the-shelf Linux OSes of different guest kernel versions.

Since kernel malware could manipulate the entries of the system call table via system call table hijacking, HProve keeps track of the changes of these entries. Then it obtains the allocated memory region of the system call table in memory and records memory access of the memory region. 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. Writes to these entries are considered as suspicious and recorded. Note that the writes to system call table entries make the relative system call service routine points to the malicious function in kernel malware. The above process is achieved by our technique: Memory Access Tracing. To associate the memory access to these sensitive entries with the impacts on the kernel, HProve captures the program counter that initiates the access points and backtracks the kernel starting from the kernel API that calls the program counter. Backtracking makes it possible to trace back to the original point at which malware has been loaded into the kernel. This is achieved by our technique: Provenance Tap Point Uncovering.

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 [15] in the kernel at which we wish to capture a set of function callers. It is defined as a four-tuple:

$\text{(call\_site, func\_entry, func\_arg, func\_ret\_val)}$,

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. Before identifying the provenance tap points, we initially identify instruction level tap points, which we call raw tap points. Each raw tap point is defined formally as a pair:

$\text{(call\_site, program\_counter)}$,

where the $call\_site$ is the caller of the kernel function and the program counter uniquely represents the...
address of the instruction. To determine the call site, we use the return address of the call to the \texttt{func_entry}. In the instruction stream, the return address is the address of the instruction after the call instruction. Once a raw tap point is discovered, data-flow analysis and memory introspection \cite{19} are needed to correlate the identified instruction with a certain argument of the kernel function. Hence, we can eventually retrieve the function level tap point: \textit{provenance tap points}.

\textbf{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:

\[ m = (\text{addr}, \text{data}, \text{type}, \text{program_counter}). \]

\texttt{Addr} is the address of memory being accessed. \texttt{Data} is the amount of data written or read. \texttt{Type} is the type of the memory access (either a read or a write). \texttt{Program_counter} is the address of the instruction invoking the access.

4.2. Design Goals

\texttt{HProve} employs kernel event replay to track the provenance of kernel malware attacks. We have the following goals for designing \texttt{HProve}.

- \textbf{G1: Fidelity}. The provenance information collected should be secure and trustworthy for obtaining true causality dependences.

- \textbf{G2: Flexibility}. It should be flexible to add custom instrumentation code into malicious code execution so that it can conduct various provenance analyses depending on needs.

- \textbf{G3: Efficiency}. The efficiency for kernel malware attack provenance tracing is considered in two-fold: 1) It should be efficient to collect abundant information to build causality dependencies; 2) The performance overhead for replay should be acceptable.

The architecture of \texttt{HProve} is depicted in Fig. 2. The record and replay modules are implemented in the virtualization layer using QEMU to achieve fidelity (G1). The instrumentation process is completed during the replaying phase rather than the recording phase which offers the provenance analysis with different requirements determined in the off-line stage, ensuring that the information to be collected is flexible to choose (G2). Execution traces recorded do not include the state of the emulated hardware devices to make the recorder lightweight. \texttt{HProve} focuses on the analysis of a process for provenance tracing rather than the replay of a generic VM. \texttt{HProve} is able to replay and analyze a trace without the access to the VM image that is used for recording. Meanwhile to reduce runtime overhead, the instrumentation code is inserted into the translated code only when necessary during replay (G3).

4.3. Recording Non-deterministic Events

\texttt{HProve} leverages Panda \cite{14}, built atop on QEMU, to record the non-deterministic events (e.g., \texttt{IN}, the data entering the CPU on port input; \texttt{INT}, a hardware interrupt and its parameters; \texttt{DMA}, the data written to RAM during a direct memory access operation from a peripheral device). \texttt{Panda} extends the original recording process of the QEMU emulator 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. However, this feature satisfies our requirements. Eliminating the execution traces of device code helps to reduce the logging overhead significantly.

4.4. Instrumentation during Replay

Before discussing the instrumentation details during replay, we introduce the QEMU Translation Block first.

\textbf{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. Fig. 3 shows how the guest code is transformed into translation blocks.

\textbf{Instrumentation before/after Execution}. \texttt{HProve} instruments analysis code during replay to obtain the \textit{Provenance Tap Point} and \textit{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, \texttt{HProve} works as follows.
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`, HPprove 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, HPprove uses a shadow stack to store these values. Specifically, HPprove 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, HPprove checks whether it matches a return address on the shadow stack; If so, we know that the current function has returned, thus HPprove 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 HPprove reads the value from the registers and memory addresses using the introspection technique [19]. The obtained values of provenance tap points will be stored in the form of (call_site, func_entry, func_arg, func_ret_val) as described in Section 4.1.

**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. Fortunately, there are only a few hundreds of system call entries in Linux as discussed in Section 3.2. HPprove keeps track of these addresses to check whether there is a write operation executed on them with low overhead, if so it 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) as described in Section 4.1.

**4.5. Causality Dependencies**

To build causality dependencies, HPprove uncovers the connections among the events occur in the Top-Half and Bottom-Half. When the allocation function allocates memory for LKM malware, HPprove acquires the address range that is being allocated by interpreting the `func_arg`. Then HPprove gets a address range that is being allocated for the LKM malware. Once the PC is captured during Memory Access Tracing, HPprove checks whether

---

**Figure 4. Building Causality Dependencies among Kernel Malware Behaviors and Impacts on the Victim System.** PTP denotes Provenance Tap Point.
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.

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

Before verifying the effectiveness of HProve, we show that kernel malware could bypass Linux audit utilized by state-of-the-art provenance systems like BEEP [24], LogGC [25], and ProTracer [29]. These systems employ the audit system to log system calls for analysis. We execute our collected kernel malware samples one by one, then start the audit system and set some rules [5] to record system calls triggered by the malware. Since all LKMs loaded into the kernel can be exported through /proc/modules directory, if everything goes well, the audit system can log LKM lists by monitoring /proc/modules. Take Kbeast as an example, it manipulates the system call entry NR_delete_module to cheat kernel, thus the Kbeast would not be listed through /proc/modules. As a result, the audit system fails to log Kbeast, leading to inaccurate provenance results by BEEP, LogGC and ProTracer. Other kernel malware that employ similar hooking mechanisms would bypass the audit system as well. We did not test LPM and Hi-Fi that employ the Linux Security Module for logging. However, since both of these systems intercept system calls as provenance information and thus they could be bypassed by kernel malware that manipulate system call entries. 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. As discussed in Section 4.4, LKMs are inserted into the kernel by the insmod or modprobe utility in Linux. These utilities encapsulate a sys_init_module system call 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:

\[
\text{layout_and_allocate} \rightarrow \text{move_module} \\
\rightarrow \text{module_alloc_update_bounds} \\
\rightarrow \text{module_alloc} \rightarrow \text{__vmalloc_node_range}.
\]

After initialization, allocation and relocation are finished, and the LKM can execute as expected. Fig. 5 shows the detailed caller-callee relationship chain after LKMs are inserted into the kernel.
With this prior knowledge, HPprove treats these functions shown in Fig. 5 as the function_entry of one of the provenance tap points. Take _mmalloc_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 [19].

Specifically, once we have inferred module_alloc_update_bounds, HPprove acquires the allocation information of LKMs including the address range from the provenance tap point. The address range is critical for HPprove to link the causality dependency between Top-Half and Bottom-Half as discussed in Section 4.5. In our experiments, HPprove uncovers provenance tap points for all kernel malware samples. The address range allocated for each malware sample is shown in Table 2. Since DKOM type malware are loaded into the memory regions, then the two events are correlated. A table for the Memory Access Trace tuples is constructed for each kernel malware sample.

Table 3 shows one of the results obtained by HPprove. In the second row, _NR_open entry is located at 0xc1541234 and has been written by PC 0xf867445f. HPprove refers to the result of Table 2 and determines that this PC and other PCs in Table 3 belong to the memory region allocated for Kbeast.

After correlating memory access traces with provenance tap points, HPprove is able to identify which malware manipulates which kind of kernel objects. Table 4 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 HPprove. In the first experiment setup, we insert all the LKM samples, including the malicious and benign ones into the guest kernel and start HPprove. Once the kernel begins to load these samples, HPprove 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
Table 4. Manipulated System Call Entries. ‘√’ denotes that the entry has been manipulated.

<table>
<thead>
<tr>
<th>System call entry</th>
<th>Kbeast</th>
<th>Xing-yiquan</th>
<th>Suterusu</th>
<th>Knark</th>
<th>Enyelkm</th>
<th>Synapsys</th>
<th>Rial</th>
<th>Kis</th>
<th>Kbdv3</th>
<th>Adore-0.42</th>
</tr>
</thead>
<tbody>
<tr>
<td>_NR_open</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_read</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_write</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_rmdir</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_mkdir</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_unlink</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_chdir</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_kill</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_fork</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_ioctl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_close</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_clone</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_exit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_execve</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_rename</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_utime</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_unlinkat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_socketcall</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_getdents</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_getdents64</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_getuid</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_gettimeofday</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_quiry_module</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_init_module</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_delete_module</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_stat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_NR_lstat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 5. The second column of Table 5 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 30MB 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 17KB. The duration of replaying Memory Access Tracing for all LKM samples is 113 minutes and the average duration of replaying Memory Access Tracing for each malware sample is 32.2 minutes. Replaying for uncovering Provenance Tap Points took 62 minutes for all LKM samples and 11.8 minutes for each malware sample in average.

6. Discussion

In this section, we discuss the limitations of HProve and our future work.

HProve employs Panda [14] 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. Recording under hardware virtualization of KVM and replaying using CPU emulation would be a significant speedup. We plan to migrate Panda to leverage such techniques to get better performance.

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) as discussed in Section 2.1. The system will fail if an object that are not
Table 5. Evaluation for space and time for provenance

<table>
<thead>
<tr>
<th>Sample</th>
<th>Recording time</th>
<th>Record size</th>
<th>Memory traces size</th>
<th>Replaying time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Provenance</td>
<td>Memory</td>
<td>Access tracing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tap points</td>
<td>size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kbeast</td>
<td>1.2 mins</td>
<td>26 MB</td>
<td>11 KB</td>
<td>13 mins</td>
</tr>
<tr>
<td>xingyiquan</td>
<td>0.8 mins</td>
<td>17 MB</td>
<td>7 KB</td>
<td>12 mins</td>
</tr>
<tr>
<td>suterusu</td>
<td>0.2 mins</td>
<td>4 MB</td>
<td>2 KB</td>
<td>10 mins</td>
</tr>
<tr>
<td>knark</td>
<td>1.1 mins</td>
<td>24 MB</td>
<td>10 KB</td>
<td>13 mins</td>
</tr>
<tr>
<td>enyelkm</td>
<td>0.3 mins</td>
<td>6 MB</td>
<td>3 KB</td>
<td>10 mins</td>
</tr>
<tr>
<td>synapsys</td>
<td>1.1 mins</td>
<td>25 MB</td>
<td>12 KB</td>
<td>14 mins</td>
</tr>
<tr>
<td>rial</td>
<td>0.4 mins</td>
<td>9 MB</td>
<td>3 KB</td>
<td>11 mins</td>
</tr>
<tr>
<td>kis</td>
<td>1.5 mins</td>
<td>30 MB</td>
<td>17 KB</td>
<td>14 mins</td>
</tr>
<tr>
<td>kbdv3</td>
<td>0.3 mins</td>
<td>5 MB</td>
<td>2 KB</td>
<td>10 mins</td>
</tr>
<tr>
<td>adore-0.42</td>
<td>0.6 mins</td>
<td>14 MB</td>
<td>5 KB</td>
<td>11 mins</td>
</tr>
<tr>
<td>All LKMs</td>
<td>11 mins</td>
<td>148 MB</td>
<td>80 KB</td>
<td>62 mins</td>
</tr>
</tbody>
</table>

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 [9, 47], 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. Note that if the kernel malware disguises itself as a benign kernel module such as a device driver, HProve still works.

Since HProve locates the address of an instruction executing a malicious memory operation into the code region of the kernel malware, it cannot handle kernel ROP or other advanced kernel malware. We consider to extend our system to adapt to more categories of kernel malware in our future work.

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 [27] 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 [43] 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 [23] 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 [39] 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 [45] 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 deliver it to its present state. Hi-Fi [34] 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 [24] instruments an application binary at the instructions and use the Linux audit system to capture the system calls triggered by the application. The log collected from the audit system can be analyzed to investigate which application brings the malware into the system for provenance. LogGC [25] 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 [29] proposes to combine both logging and unit level tainting techniques, aiming at reducing log volume to achieve cost-effective provenance tracing. Bate et al. [12] 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.

Deterministic Replay: Deterministic replay creates an execution that is logically equivalent to an original execution of interest. It records the not-deterministic events (e.g., hardware interrupts, I/O inputs, DMA events) and replays the system at a checkpoint deterministically [14–18, 30, 31, 35, 46]. Deterministic replay is helpful to roll back a victim system after an attack for forensic analysis. Our system utilizes the record and replay technique of Panda to obtain the execution traces of the whole system. HProve instruments the provenance code in the replay phase to obtain causality dependencies among behaviors of kernel malware and impacts on the victim system.

Kernel Monitoring: Kernel monitoring helps to understand the exact execution of the whole system. DRIP [20] is a framework for purifying trojaned kernel drivers. It records all kernel API invocations from the driver to the kernel, aim at eliminating malicious effects from the driver. Gateway [41] isolates all drivers from the kernel code by creating a separate address space for drivers to monitor the interaction of drivers with the core kernel. It records kernel APIs invocation by drivers to monitor the untrusted kernel-mode execution. Starting from the interface of system calls, the exported kernel APIs, and the data structure definitions for kernel driver developers, AutoTap automatically tracks kernel objects, resolves their kernel execution context, and associates the accessed context with the objects [47]. Note that AutoTap does not build connections among these objects or the causality dependencies among the objects and the subjects that access them. HProve is capable of monitoring some kernel functions to backtrack causality dependencies among kernel malware behaviors and impacts on the victim system.

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, especially for large enterprises. Due to the limitations of HProve discussed in Sec. 6, more efficient designs for kernel malware provenance are still highly needed.

References


Wang, Z., Jiang, X., Cui, W., and Wang, X. Countering persistent kernel rootkits through systematic hook


