IntroLib: Efficient and Transparent Library Call Introspection for Malware Forensics

By

Zhui Deng, Dongyan Xu, Xiangyu Zhang and Xuxian Jiang

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2012 USA

Washington, DC (Aug 6th - 8th)

Digital Investigation 9 (2012) S13–S23

Contents lists available at SciVerse ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

IntroLib: Efficient and transparent library call introspection for malware forensics

Zhui Deng [a,*], Dongyan Xu [a], Xiangyu Zhang [a], Xuxiang Jiang [b]

[a] Department of Computer Science, Purdue University, West Lafayette, IN 47907-2107, USA
[b] Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206, USA

Keywords:
Malware forensics
Library call introspection
Virtualization
Dynamic analysis
Performance

ABSTRACT

Dynamic malware analysis aims at revealing malware's runtime behavior. To evade analysis, advanced malware is able to detect the underlying analysis tool (e.g. one based on emulation.) On the other hand, existing malware-transparent analysis tools incur significant performance overhead, making them unsuitable for live malware monitoring and forensics. In this paper, we present IntroLib, a practical tool that traces user-level library calls made by malware with low overhead and high transparency. IntroLib is based on hardware virtualization and resides outside of the guest virtual machine where the malware runs. Our evaluation of an IntroLib prototype with 93 real-world malware samples shows that IntroLib is immune to emulation and API hooking detection by malware, uncovers more semantic information about malware behavior than system call tracing, and incurs low overhead (<15% in all-but-one test case) in performance benchmark testing.

1. Introduction

Malware analysis, which aims at revealing the goal and detailed behavior of malware, is important to malware defense. To complement the static malware analysis techniques (Moser et al., 2007), many current malware analysis tools (Willems et al., 2007; Bayer et al., 2006; Norman, 2012; International Secure Systems Lab, 2012) adopt the dynamic analysis approach, where malware samples are run in a controlled environment to capture and analyze their runtime behavior. To make sure the analysis tools are not tampered with and have higher privilege over the execution of malware, such environments are often emulated or virtualized with emulators (e.g. QEMU) or hypervisors (e.g. Xen, KVM).

As a counter-measure, malware authors have devised anti-analysis methods to thwart dynamic analysis. Before executing malicious functionality, a malware program may first check if it is running in an analysis environment. If the malware finds itself being analyzed, it will withhold its malicious behavior and appear to be benign, exhibit random behavior or even simply terminate. The kind of checks performed by malware depends on the dynamic analysis approaches it tries to detect. For example, traditional dynamic analysis approaches like software breakpoints and instrumentation could easily be detected by malware using checksum verification since they need to modify the code of malware. Full system emulators such as QEMU naturally offer higher degree of transparency because they enable instruction-level analysis without the use of any in-guest components that could be observed by malware. However, detection (Ferrie, 2006, 2007; Raffetseder et al., 2007) is still possible by exploiting any discrepancy of instruction semantics between the emulator and real hardware, since perfect emulation of every instruction is not feasible. In fact, Dinaburg et al. (2008) has proved that determining whether an emulator achieves perfect emulation is undecidable.

[*] Corresponding author. Tel.: +1 202 687 5874.
E-mail addresses: deng14@purdue.edu (Z. Deng), dxu@cs.purdue.edu (D. Xu), xyzhang@cs.purdue.edu (X. Zhang), jiang@cs.ncsu.edu (X. Jiang).

1742-2876/$ – see front matter © 2012 Z. Deng, X. Hu, X. Zhang & X. Jiang. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2012.05.013

Such kind of analysis-dodging malware widely exists in the wild—a recent study (Chen et al., 2008) showed that more than 40% of the real-world malware samples they captured were equipped with techniques such as anti-debugging and anti-virtualization to detect presence of analysis environment. In response to this trend, Ether (Dinaburg et al., 2008) is the first to leverage hardware virtualization to build more transparent dynamic analysis systems upon hypervisors. A later work MAVMM (Nguyen et al., 2009) features similar design. Hardware virtualization brings two benefits: first, moving malware analysis systems to hypervisor gives them higher privilege than the malware being analyzed; second, in hardware virtualization, guest instructions run natively on CPU, hence the problem of instruction semantic discrepancies in emulators is avoided. Both Ether and MAVMM succeed in remaining transparent to subject malware. However, both systems suffer from significant performance overhead when performing fine-grained live malware analysis. They both use single-stepping to trigger a transition between guest and hypervisor for each guest instruction to enable instruction-level analysis. Although detailed performance numbers are not reported, the slowdown can be inferred by considering the fact that common instructions such as MOV take a few CPU cycles, whereas a transition between guest and hypervisor usually take hundreds or thousands of CPU cycles. The authors of Ether acknowledge in their paper that their fine-grained analysis “is not meant to be used for real-time analysis” and “induces a significant performance penalty.”

Another category of tools aim to execute malware transparently in an emulator for more efficient fine-grained analysis. Kang et al. (2009) proposed to guide the execution of malware in emulators using transparent reference systems, such as Ether. Execution traces of a malware sample are captured in both environments and compared using trace alignment algorithm to find out divergence, which indicates that the malware has detected the emulator and chose to execute another path. To fix the divergence, a runtime patch will be generated to force the next run of emulation following the path in the reference system. The idea is appealing, however, the tool incurs high performance penalty when obtaining execution traces. Although it only needs to run Ether in fine-grained tracing mode once to get the execution trace, the hundred-fold slowdown during that step is not desirable for live malware analysis. A more recent work V2E (Yan et al., 2012) aims to solve the transparency issue of emulator yet remain efficient by selectively emulating instructions: For those instructions that are fully emulated, V2E let the emulator translate and execute them; for the remaining instructions, instead of translating them, V2E records the state changes caused by them in a reference system and replays the state changes in the emulator. This could significantly boost performance since the reference system only needs to capture state changes caused by instructions that are not fully emulated. However, how to enumerate *all* such instructions remains a challenge.

As our effort toward practical malware forensics with (1) high efficiency for live monitoring and analysis and (2) improved transparency compared with emulation-based tools, we present *IntroLib*, a tool that performs library call introspection on malware from outside the virtual machine (VM) where the malware is executing. IntroLib externally tracks and logs the sequence of user-level library calls made by the malware without significant slowdown. Compared with the traditional system call based introspection techniques (e.g. VMscope (Jiang and Wang, 2007)), IntroLib is more informative and provides more insights into malware attack goals and behaviors. Compared with techniques that rely on instruction-level dynamic analysis, IntroLib is more lightweight and suitable for live malware forensics. Compared with emulation-based tools, IntroLib is more immune to malware's emulation detection logic by adopting hardware virtualization. IntroLib covers all kinds of user-mode library calls, such as Windows API library functions and C library functions.

To address the challenge of intercepting user-level library calls which, unlike system calls, cannot be easily trapped into the hypervisor under hardware virtualization, we propose a page table-based mechanism that creates a “barrier” in memory between the malware binary and the library binaries that it calls. Any control-flow transition crossing the barrier will be intercepted by IntroLib. Similar mechanisms have been used in previous works to capture kernel-level control-flow transitions (Srivastava and Giffin, 2011). We have developed a prototype of IntroLib using the KVM hypervisor. Our IntroLib prototype supports malware analysis on both Windows XP and Ubuntu Linux 11.04 (guest) operating systems. We evaluate IntroLib with more than 90 real-world Windows-based malware programs. The analysis results from IntroLib uncover interesting behaviors of the malware that cannot be revealed by (lower-level) system call tracing logs. We also compare the anti-detection capability of IntroLib with state-of-the-art malware analysis systems (e.g. Anubis (International Secure Systems Lab, 2012) and CWSandbox (Willems et al., 2007)) using both synthetic and real-world analysis-detecting malware samples; and our results find samples that detect Anubis and/or CWSandbox but not IntroLib. Finally, our performance evaluation using the PCMark05 (Futuremark, 2012) benckmarks shows that IntroLib incurs reasonably low overhead (<15% in all but one test) compared with running the same workload on vanilla KVM.

2. Goal and assumptions

The goal of IntroLib is to reveal malware's intent and behavior by tracing its user-level library calls. As such, IntroLib helps analyzes user-land (instead of kernel-level) malware. The design requirements of IntroLib are as follows:

R1. Trustworthiness: The tracing result should cover all user-mode library calls made by the malware, and should not be tampered with.

R2. High transparency to malware: The presence of IntroLib should be difficult to detect by advanced anti-analysis malware. However, we point out that, because of the VM-based nature of IntroLib, it is a *no-goal* for us to thwart the detection of VM by a malware program. Instead, we aim at avoiding the malware's other anti-analysis logic such as that for the detection of emulation or library API hooking. R3. Efficiency: The performance overhead incurred by IntroLib should be reasonably low for live forensic analysis.

To meet the above requirements, we have to make a number of assumptions. First, we assume that our trusted computing base consists of the hardware, the hypervisor (and its host OS), and the guest OS. In particular, the integrity of the guest OS kernel can be protected by existing techniques as SecVisor (Seshadri et al., 2007), NICKEL (Riley et al., 2008), and HookSafe (Wang et al., 2009). Second, we only trace calls to functions in the *dynamically linked* libraries. For statically-linked libraries, their functions called by the malware will be embedded into the malware's binary, making it impossible to distinguish the library functions from the malware code. Our study of both Windows and Linux-based malware shows that dynamic library linking is more often used by malware for the sake of footprint minimization and efficient propagation.

3. Design and implementation

IntroLib is based on a hypervisor that utilizes hardware virtualization. We present key aspects of its design in the following subsections.

3.1. Intercept control-flow transitions

To intercept control-flow transitions between malware and library functions, we utilize the shadow page table (SPT) to set a transparent barrier in memory. Shadow paging is a memory virtualization technique widely used in hypervisors. The guest OS still manages its own conventional page tables. Such a guest page table (GPT) maps guest virtual address (GVA) to guest physical address (GPA). However, since the physical address space of the guest is virtualized by the hypervisor, GPA cannot be directly used by the CPU to access memory. Instead, the CPU translates GVA to the final physical address for memory access using SPT, which maps GVA to host physical address (HPA). SPT is managed by the hypervisor according to GPT and kept invisible to the guest OS by shadowing the CR3 register: When the guest OS tries to read the CR3 register, it gets the value that points to GPT instead of SPT. That is, from the view of the guest OS, SPT is transparent.

Normally, the hypervisor only maintains one SPT for each guest. In IntroLib, we maintain *two mutually exclusive SPTs for each guest* to trigger events that could be captured by the hypervisor when control-flow transitions between malware and library functions happen. The two SPTs have exactly the same virtual-to-physical address mappings; they are mutually exclusive in the sense that no user-mode page is set as executable in both of these SPTs. In one of the two SPTs, which we call malware SPT (MSPT), we set the pages containing the malware code to executable, and pages containing library code to non-executable; in the other SPT, which we call library SPT (LSPT), we do the opposite. Other user-mode pages are set to non-executable in both SPTs. At any time during the execution of malware, only one of the two SPTs is active, namely, being used as the current SPT: when executing the malware code, MSPT is active; when executing the library code, LSPT is active.

To better illustrate the technique, let us take a closer look at what happens when the malware calls a library function. As shown in Fig. 1, since the page containing code of the target library function is set as non-executable in the active SPT—MSPT, a page fault will occur. We set the hypervisor to intercept all guest page faults, so a VMExit will be triggered due to the page fault and switch execution from the guest to the hypervisor. The hypervisor will find out that the reason for the page fault is that the malware attempted to make a library call; it will then record detailed information about this call, such as the function name, the caller address and the parameters. Before resuming the guest, the hypervisor must switch the active SPT from MSPT to LSPT, as the guest will execute the library code after resume. The guest is completely unaware of the page fault as the hypervisor handled it transparently. Later on, when the library function finishes and returns to the malware caller, similar steps will be taken, and this time the

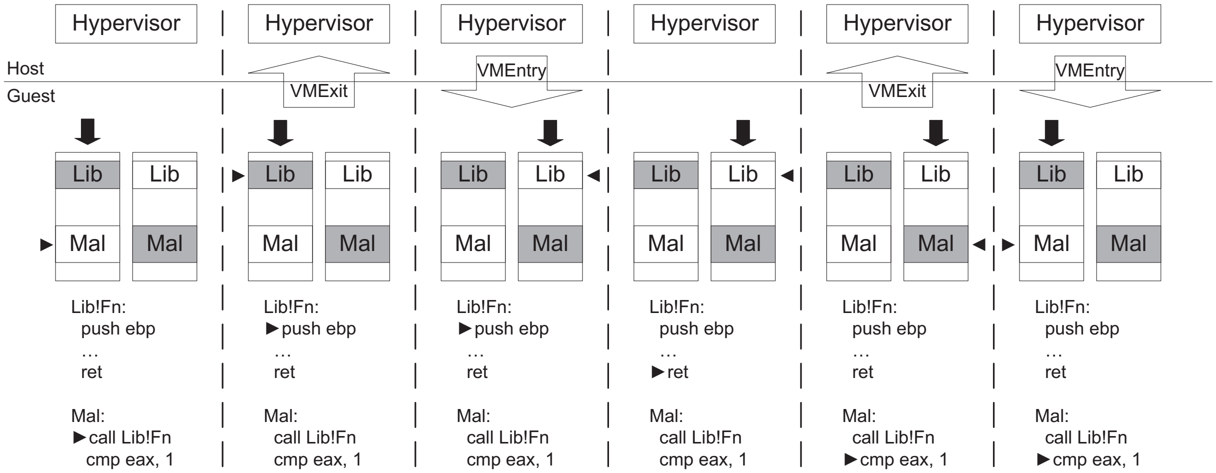


Fig. 1. Execution snapshots of a library call.

hypervisor will record the return value of the call. We note that the two mutually exclusive SPTs are only maintained for a process being traced. For other processes, the hypervisor still use one SPT for their execution.

Special care must be taken when context switches happen in the guest. It is possible that the guest context switches from an execution context running the malware code to another context running the library code; or vice versa. Such a context switch could happen between different malware processes, or even between different threads of the same multi-threaded malware process. Consequently, we must switch the active SPT to ensure that it is consistent with the new execution context. Since it's infeasible to *transparently* trap context switches into hypervisor under hardware virtualization, we choose to switch SPT in a passive way: When the active SPT is inconsistent with the current execution context, a page fault will be triggered due to instruction-fetch from a non-executable page. The hypervisor will then catch that page fault and switch the active SPT to the consistent state.

3.2. Identify memory layout

To create the "barrier" between malware and library code, we must first identify the memory layout of the malware process. We categorize user-mode pages into three types: malware code, library code and data pages. Since a malware program might generate new code (e.g. binary packers) and load additional libraries at runtime, data pages could become either malware code or library code pages; our identification mechanism must be able to capture such changes of memory layout and update SPTs accordingly.

We introduce a mechanism called *lazy identification* to solve the problem. With lazy identification, an area of code pages is not identified until there is an instruction-fetch from one of its pages. Hence, when the area is being mapped, all of its pages are assumed as data pages and set as non-executable in the two mutually exclusive SPTs (Section 3.1). When the first instruction-fetch from one of its pages happens, a page fault will be triggered and captured by the hypervisor.

Given the address of the faulting page, the hypervisor uses VM introspection (Garfinkel and Rosenblum, 2003) to gather information from the guest to infer the position and size of the area containing the page, and determine whether the area contains malware code or library code. In both Windows and Linux, a binary executable and its dependent libraries are loaded into memory for execution through file mapping, a mechanism that associates an area of virtual memory space with the content of a binary file. For each process, the kernel maintains data objects to record information of the files it mapped in its virtual memory space, including the path of the associated file and the starting and ending addresses of the mapped area. By traversing these data objects of the current process, the hypervisor can locate the mapped area containing the faulting page. For Windows guests, the hypervisor will further examine the path of the corresponding associated file to determine the type of code stored in the mapped area: If the associated file is the binary executable itself, then the mapped area contains code of the malware binary; otherwise the mapped area contains library code. The path of the binary executable could be obtained by following the link in another kernel object: *EPROCESS → SectionObject → Segment → ControlArea → FilePointer.* For Linux guests, the hypervisor determines the type of the mapped area by checking if the faulting page falls between the starting and ending addresses of the area containing code of the malware binary, which could be read from *start_code* and *end_code* fields in the *task_struct.mm* kernel object respectively.

If the faulting page does not fall into any file-mapped area, it means that the code being executed in that page is not loaded into memory through file mapping, which is usually the standard mechanism used by the kernel to load binaries. Note that even if the malware loads additional libraries at runtime through APIs provided by the OS (e.g. LoadLibrary function in Windows), such libraries are still loaded using file mapping. Hence, the faulting page which is not associated with a binary file indicates that code in the page is generated by the malware itself at runtime, so it is categorized as a malware code page.

After the hypervisor knows the position and size of the area, the hypervisor will set the two exclusive SPTs accordingly, switch the active SPT to match the execution context if needed, and resume execution of the guest.

3.3. Logging

When IntroLib intercepts a control-flow transition between malware code and library code, it will read the source and destination addresses of the transition from the Last Branch Record (LBR) stack. LBR stack is a circular stack that stores information about the recent branches taken by the processor. It could be enabled by setting the lowest bit of the *IA32_DEBUGCTL* Model Specific Register (MSR). Another MSR holds the LBR top-of-stack (TOS) pointer, indicating which entry in the LBR stack stores source and destination addresses of the most recent branch. The most recent branch must be a control-flow transition, as the hypervisor captures a transition as soon as it happens.

IntroLib then tries to locate the code areas which contain the source and destination addresses by looking up in the memory layout it has obtained. If the area is associated with a library, IntroLib will get all functions exported by the library by parsing the library file. In a DLL file, the addresses and names of the exported functions could be read in the PE export table section (Microsoft, 2010); In a shared object file, such information could be read in the ELF symbol table (TIS Committee, 1995). We use a copy-on-write disk to run malware and always let IntroLib read the original disk to make sure the library file parsed by IntroLib is not tampered. IntroLib will further translate the source and destination addresses into the format of function name plus an offset.

If the destination address is the entry address of a library function, it means the control-flow transition is a library call. IntroLib will read the function arguments from the stack according to the function prototype we prepared in a database. We extracted function prototypes of C library functions, most Windows APIs and Linux APIs by extending cproto (Huang, 2012) to parse header files in

Windows SDK and Linux. The caller address, the called library name, function name and values of arguments will all be logged. For each call it logs, IntroLib will read and record its return address from stack. If the destination address of a control-flow transition matches one of the return addresses recorded by IntroLib, it means the transition is a return from library call. IntroLib will read the function return value from the *EAX* register and match it with the corresponding library call in the log.

3.4. Improving transparency to malware

3.4.1. Avoid timing attack

In KVM, given a period of time *T*, the execution on a CPU consists of three parts:

(1) execution in host mode, which includes hypervisor, host OS and applications, which takes time T_h;

(2) execution in-guest mode, which includes guest OS and its applications, which takes time T_g;

(3) switch between host and guest, which takes time T_s.

Without special handling, the time seen by the guest is the total time *T*. Assuming the CPU could execute *N* instructions per time unit, and the guest is only executing malware code. After that period of time, a total number of $(N \cdot T_g)$ instructions are executed in guest. From the malware's view, the CPU only executed $(N \cdot T_g/T)$ instructions per time unit, which is less than expected. Such discrepancy could be exploited by malware to detect the existence of IntroLib.

To avoid such timing attack, we must subtract T_h and T_s from the time seen by the guest, so the guest only see the time it actually runs for. We use the unit of the Time Stamp Counter (TSC) as our time unit. T_h could be obtained in this way: execute a *RDTSC* instruction right after the guest exits to the host to get a read of the TSC T_1, and execute it again right before the host resumes the guest to get another read T_2. Then $T_h = T_2 - T_1$. We could not measure T_s at runtime, but an empirical value could be instrumented beforehand by executing a loop of *RDTSC* in the guest. Since T_s may vary on different systems, the empirical T_s value for a specific system should be instrumented before running IntroLib on it.

We set the TSC offset field in the virtual machine control data structure (VMCS) to the value of $-(T_h + T_s)$, so that when the guest queries the TSC, the offset is automatically applied to the result. Guest accesses to other internal time resources such as the 8254 programmable interval timer (PIT) and the advanced programmable interrupt controller (APIC) timer, are all controlled by the hypervisor thus can also be adjusted using the offset.

3.4.2. Shadow LBR stack

IntroLib must enable the LBR feature all the time to ensure precise logging. However, if the malware knows that IntroLib must use LBR, it may try to turn off LBR and see if the LBR stack is still recording the recent branches. If the malware found that LBR could not be turned off, or if LBR still functions even if it appears to have already been turned off, the existence of IntroLib will be revealed by such discrepancy.

˜To hide from malware, IntroLib must conceal its use of LBR. Fortunately, LBR are accessed through MSRs using *RDMSR* and *WRMSR* instructions, which could be intercepted by the hypervisor. First, the *IA32_DEBUGCTL* MSR needs to be shadowed. That is, the hypervisor maintains a shadow of this MSR, and all guest read and write accesses to this MSR go to the shadow. Modification to the MSR, except the lowest bit of it, should be synchronized to the real MSR.

We may also need to shadow the LBR stack and the LBR TOS pointer depending on whether the guest currently enables LBR. When the guest tries to turn off LBR, we copy the current LBR stack and the TOS pointer to their shadows. Any further accesses to the LBR stack and the TOS pointer when LBR is disabled in guest go to the shadows. When the guest turns on LBR, we copy the values stored in the shadows to the real LBR stack and the TOS pointer. The guest will access the real LBR stack and the real TOS pointer when LBR is enabled. Note this won't affect the trustworthiness of our tracing result as our hypervisor will always read the most recent branch right after the branch happens, before the guest has a chance to modify it.

4. Evaluation

In this section, we present the evaluation of IntroLib in three aspects: functionality, transparency (to malware), and performance. Our experiments are done on a Dell Inspiron 15R laptop with Intel(R) Core(TM) i5-2410M 2.30 GHz CPU and 4 GB memory. We use Ubuntu Linux 11.04 64bit with kernel version 2.6.38 as the host OS. We allocate a 10 GB raw image file as the hard disk and 1 GB memory for the guest VM. The guest VM runs Windows XP with no service pack.

4.1. Functionality

We have evaluated IntroLib with a pool of 93 real-world, Windows-based malware samples obtained from an online malware repository (http://www.offensivecomputing.net/) and security researchers. IntroLib is able to disclose more semantic information of malware execution compared with the traditional system call introspection techniques (Jiang and Wang, 2007). We present two case studies to demonstrate such benefits.

4.1.1. Case study I

In the first case study, we traced a malware sample of Win32/FakeRean,[1] which is a malware disguised as a rogue anti-virus tool. The malware pretends to scan disks to find virus and spyware; in fact, it just displays the names of the files it iterates through without reading them. It will randomly choose some files and display them as "malware." The malware will also close running programs in order to prevent the user using security tools to terminate it after knowing he/she has been lied to.

[1] MD5 hash: 9e7e65802546c273b6157d4bbe4e02fc.

Part of the library call trace of this malware is shown in Fig. 2.[2] When the malware was executed, it first called a function *ldap_count_references*. The two pointer arguments were assigned value 1, which was apparently not a valid pointer address. Observe that the call is made in an exported function named *antiemu33*. We disassembled this function and verified that it was an anti-emulation trick. In this function, the malware tried to trigger an SEH exception by calling *ldap_count_references* with invalid pointer arguments, which should then be caught by the SEH handler and the function would return 1. If the exception was not triggered, or if an emulator failed to handle SEH exceptions, this function would return 0 and lead to termination. We searched on the Internet about this anti-emulation technique and found an online article (Kaspersky Lab, 2012) that confirms our analysis.

After determining that it is not being emulated, the malware called *HeapCreate* to allocate a heap area of 0x83b5 bytes, starting at address 0x370000. The heap area was then identified as a malware code area when the malware began to execute code from it. This code area actually stored the unpacking routine of the packer used by the malware, which was identified from the following library calls it made: (1) It called *RtlDecompressBuffer* to decompress the packed binary; (2) It made a lot of calls to the functions *LoadLibrary* and *GetProcAddress* to load the libraries required by the packed binary and fill the imported function addresses; (3) It first called *VirtualProtect* to set the entire region of the original binary as writable to write the unpacked binary there, then called *VirtualProtect* again to set the access control for each section of the unpacked binary after it finished unpacking. We later used PEiD (Jibz et al., 2012) to check the packer used in this malware; although PEiD could not precisely identify which packer is used, its entropy scan confirmed this malware was packed. Note that a system call based technique would not capture the calls to *RtlDecompressBuffer* and *GetProcAddress* which are important hints to reveal the unpacking behavior.

The malware then copied its binary executable to a file in the temporary path, mapped the file into memory and called *CheckSumMappedFile* to verify its checksum. After that, it tried to use several attack strategies similar to the TDL4 rootkit to avoid being detected by host-based intrusion prevention systems (HIPS). It first hooked an undocumented function *NtConnectPort* in *ntdll.dll*. From the log, we can see it first called *GetProcAddress* to locate the address of *ZwConnectPort* function (an alias of *NtConnectPort*), then copied the first 5 bytes of the function to a code buffer at 0x404808 using *memcpy*, and called *WriteProcessMemory* to overwrite these bytes with a relative jump instruction. This technique is referred to as "inline hooking" and is widely used by malware to detour the control flow of a function. The malware then called *AddPrintProviderA* to load itself as a print provider into the spool service process (spoolsv.exe), which is a trusted system process. This is the reason why it hooked *NtConnectPort*: *AddPrintProviderA* calls *NtConnectPort* to connect to the LPC port of the spool service process, and this might be intercepted by HIPS. More specifically, HIPS intercepts all calls to *NtConnectPort* and checks if the name of the LPC port being connected to is *|RPC Control|spoolss*, which is the LPC port name of the spool service process. In order to avoid detection by HIPS, the malware called *RtlPrefixUnicodeString* in the hooked *NtConnectPort* function to determine if the LPC port name starts with *|RPC Control*. We can see in the following call to the original *NtConnectPort*, the LPC port name had been modified to *|LPC Control|spoolss*. This is slightly different from the TDL4 rootkit: in TDL4, the LPC port name is prepended with *|??|GLOBALROOT* (Rodionov and Matrosov, 2012), so *AddPrintProviderA* could still connect to the same port. In this malware, the modified LPC port name was invalid, so the connection would fail. The calls to *RtlGetLastWin32Error* returned 0x6be, confirming that *AddPrintProviderA* had failed. A system call tracer could not capture the *CheckSumMappedFile* call, so it would be infeasible to know the reason of mapping the executable; regarding the attack, it could only capture the event of *WriteProcessMemory* and an attempt to connect to LPC port *|LPC Control|spoolss*, which is insufficient to precisely understand the attack.

The malware then extracted another malicious binary executable it carried. It allocated an area in the heap, then called *RtlDecompressBuffer* to decompress the binary executable into that area. After that, it created a file named "*privacy*" in the *C:|Documents and Settings|All Users|Application Data* directory and wrote the binary executable from memory into the file. The malware then renamed the file to *privacy.exe* and called *CreateProcessA* to execute it. The reason why it did not directly create and write to the file *privacy.exe* is that most HIPS consider writing to files with name extensions of binary executable (e.g. ".exe", ".dll") as malicious behavior.

The newly spawned malware process of *privacy.exe* was also packed and had the same unpacking behavior as the primary process. The last behavior of the primary process was to call functions in *wininet.dll* to connect to a broken URL "*http://getmilfs.com/logo/go.php?id=96*"; then it was killed by the new process. The new process did the following in an infinite loop: it called *CreateToolhelp32Snapshot* to get a list of all running processes, then used *Process32First* and *Process32Next* to iterate through the list and selected some processes to terminate. The way it determines whether a process should be terminated could not be directly inferred from the log, so we dumped the instructions near these library call sites and disassembled them for further investigation. Our investigation suggested that the malware selected those processes whose names were not in a predefined list of system processes (e.g. "explorer.exe", "csrss.exe") to terminate. To verify our analysis, we renamed an executable file to "explorer.exe" and executed it; the executable ran successfully and was not terminated by the malware.

4.1.2. Case study II

In the second case study we investigated a malware sample of Win32/Dorkbot.AJ,[3] which is a variant of worm Win32/Dorkbot.A. This variant is packed with an advanced packer Armadillo (Silicon Realms, 2012), which features multiple levels of packing and encryption/decryption to

[2] In Figs. 2 and 3, the comments on top of each block (e.g. [ANTI-EMULATION]) are not part of the original trace. They are added manually for better readability.

[3] MD5 hash: f27dd9ba200266d7c9df2d345b8f882f.

```
[ANTI-EMULATION]
sample.exe(120): sample.exe:0x101a => wldap32.dll:ldap_count_references([0x1] = , [0x1] = , ) = 0x0(0)

[UNPACKING]
sample.exe(120): sample.exe:0x118c => kernel32.dll:HeapCreate(262145(0x40001), 0x6c1e, 0x83b5, ) = 0x370000(3604480)
00370000-00379000 [unknown](00000000)
sample.exe(120): [unknown]:0x8ca => kernel32.dll:GetProcAddress(0x77e60000, [0x371149] = "VirtualAlloc", ) = 0x77e7980a(2011666442)
sample.exe(120): [unknown]:0xdcd => ntdll.dll:RtlDecompressBuffer = 0x0(0))
sample.exe(120): [unknown]:0xa5f => kernel32.dll:VirtualProtect([0x400000], 0x1d6000, 64(0x40), [0x12fde4] = 1244784(0x12fe70), ) = 0x1(1)
sample.exe(120): [unknown]:0x9f5 => kernel32.dll:LoadLibraryA([0x403552] = "ntdll.dll", ) = 0x77f50000(2012545024)
...
sample.exe(120): [unknown]:0xb08 => kernel32.dll:VirtualProtect([0x400000], 0x1000, 2(0x2), [0x12fde4] = 1244784(0x12fe70), ) = 0x1(1)
sample.exe(120): [unknown]:0xb5f => kernel32.dll:VirtualProtect([0x401000], 0x1194, 32(0x20), [0x12fde4] = 64(0x40), ) = 0x1(1)
....

[VERIFY CHECKSUM]
sample.exe(120): sample.exe:0x2003 => kernel32.dll:GetTempPathA(259(0x103), [0x12fb8c] = "", ) = 0x21(33)
sample.exe(120): sample.exe:0x201b => kernel32.dll:GetTempFileNameA([0x12fb8c] = "C:\DOCUME~1\Tyrael\LOCALS~1\Temp\", [0x0] = "", 0(0x0), [0x12fc90] = "", ) = 0x1(1)
sample.exe(120): sample.exe:0x202d => kernel32.dll:CopyFileA([0x12feb8] = "C:\samples\sample.exe", [0x12fc90] = "C:\DOCUME~1\Tyrael\LOCALS~1\Temp\1.tmp", 0(0x0), ) = 0x1(1)
sample.exe(120): sample.exe:0x16db => kernel32.dll:CreateFileA([0x12fc90] = "C:\DOCUME~1\Tyrael\LOCALS~1\Temp\1.tmp", 2032127(0x1f01ff), 1(0x1), [0x0], 3(0x3), 2147483648(0x80000000), 0x0, )
= 0x54(84)
sample.exe(120): sample.exe:0x16f0 => kernel32.dll:CreateFileMappingA(0x54, [0x0], 4(0x4), 0(0x0), 0(0x0), [0x0] = "", ) = 0x58(88)
sample.exe(120): sample.exe:0x1706 => kernel32.dll:MapViewOfFile(0x58, 983071(0xf001f), 0(0x0), 0(0x0), 0x0, ) = 0xbc0000(12320768)
...
sample.exe(120): sample.exe:0x204b => ntdll.dll:RtlImageNtHeader = 0xbc00f8(12321016)
sample.exe(120): sample.exe:0x206c => imagehlp.dll:CheckSumMappedFile = 0xbc00f8(12321016)

[EXPLOIT SPOOLSV.EXE]
sample.exe(120): sample.exe:0x1fd8 => kernel32.dll:GetModuleHandleA([0x40329c] = "ntdll.dll", ) = 0x77f50000(2012545024)
sample.exe(120): sample.exe:0x1fdf => kernel32.dll:GetProcAddress(0x77f50000, [0x40328c] = "ZwConnectPort", ) = 0x77f7e5a3(2012734883)
sample.exe(120): sample.exe:0x150f => kernel32.dll:VirtualProtect([0x404808], 0x14, 64(0x40), [0x12fb40] = 4207244(0x40328c), ) = 0x1(1)
sample.exe(120): sample.exe:0x2188 => ntdll.dll:memcpy([0x404808], [0x77f7e5a3], 5(0x5), ) = 0x404808(4212744)
sample.exe(120): sample.exe:0x15bf => kernel32.dll:WriteProcessMemory(0xffffffff, [0x77f7e5a3], [0x12fb38] = 3941235001222121(0xe0088483963e9), 0x5, [0x12fb40], ) = 0x1(1)
...
sample.exe(120): sample.exe:0x10ed => winspool.drv:AddPrintProvidorA([0x12fb5c] = "10edd", 1(0x1), 0x12fb6c, ) = 0x12f558(1242456)
sample.exe(120): sample.exe:0x1f30 => ntdll.dll:RtlPrefixUnicodeString([0x12f4a8] = "\RPC Control", [0x12f558] = "\RPC Control\spoolss", 1(0x1), ) = 0x7ffd2001(2147295233)
sample.exe(120): sample.exe:0x480d => ntdll.dll:NtConnectPort+0x5(1329416(0x144908), [0x12f558] = "\LPC Control\spoolss", ) = 0x0(0)
sample.exe(120): sample.exe:0x10f3 => ntdll.dll:RtlGetLastWin32Error = 0x6be(1726)

[EXTRACT PRIVACY.EXE]
sample.exe(260): sample.exe:0x1635 => ntdll.dll:RtlAllocateHeap = 0xc90020(13172768)
sample.exe(260): sample.exe:0x1693 => ntdll.dll:RtlDecompressBuffer = 0x0(0)
...
sample.exe(260): sample.exe:0x198c => kernel32.dll:CreateFileA([0x12f660] = "C:\Documents and Settings\All Users\Application Data\privacy", 1073741824(0x40000000), 1(0x1), [0x0], 2(0x2), 0
(0x0), 0x0, ) = 0x6c(108)
...
sample.exe(260): sample.exe:0x19b3 => kernel32.dll:WriteFile(0x6c, [0xc90434], 817152(0xc7800), [0x12f62c] = 1(0x1), [0x0], ) = 0x1(1)
...
sample.exe(260): sample.exe:0x1a32 => kernel32.dll:MoveFileA([0x12f21c] = "C:\Documents and Settings\All Users\Application Data\privacy", [0x12f660] = "C:\Documents and Settings\All Users
\Application Data\privacy.exe", ) = 0x1(1)
...
sample.exe(260): sample.exe:0x1c1f => kernel32.dll:CreateProcessA([0x0] = "", [0x12f660] = "C:\Documents and Settings\All Users\Application Data\privacy.exe", [0x0], [0x0], 0(0x0), 0(0x0),
[0x0], [0x0] = "", [0x12f5e0], [0x12f628], )

[CONNECT TO URL]
sample.exe(260): sample.exe:0x1174 => wininet.dll:InternetOpenA([0x403198] = "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)", 0(0x0), [0x0] = "", [0x0] = "", 0(0x0), )
sample.exe(260): sample.exe:0x11d6 => wininet.dll:InternetCrackUrlA([0x12f970] = "http://getmilfs.com/logo/go.php?id=96", 0(0x0), 2147483648(0x80000000), [0x12f5b8], ) = 0x1(1)
sample.exe(260): sample.exe:0x11ed => wininet.dll:InternetConnectA(13369348(0xcc0004), [0x12f4f8] = "getmilfs.com", 80(0x50), [0x0] = "", [0x0] = "", 3(0x3), 0(0x0), [0x0] = , ) = 0xcc0008
(13369352)
sample.exe(260): sample.exe:0x122e => wininet.dll:HttpOpenRequestA(13369352(0xcc0008), [0x403200] = "GET", [0x12ece0] = "/logo/go.php?id=96", [0x0] = "", [0x0] = "", [0x0], 2215117056
(0x84080100), [0x0] = , ) = 0xcc000c(13369356)
sample.exe(260): sample.exe:0x1251 => wininet.dll:InternetSetOptionA(13369356(0xcc000c), 31(0x1f), [0x12f5f4], 4(0x4), ) = 0x1(1)
sample.exe(260): sample.exe:0x12a5 => wininet.dll:HttpSendRequestA(13369356(0xcc000c), [0x12f0f8] = "", 4294967295(0xffffffff), [0x0], 0(0x0), )

[KILL PROGRAMS]
privacy.exe(320): privacy.exe:0x15d5d9 => kernel32.dll:CreateToolhelp32Snapshot = 0x20c(524)
privacy.exe(320): privacy.exe:0x15d5f9 => kernel32.dll:Process32First = 0x1(1)
privacy.exe(320): privacy.exe:0x15d619 => kernel32.dll:Process32Next = 0x1(1)
...
privacy.exe(320): privacy.exe:0x7eb4 => kernel32.dll:OpenProcess(1(0x1), 0(0x0), 260(0x104), ) = 0x200(512)
privacy.exe(320): privacy.exe:0x7f34 => kernel32.dll:TerminateProcess(0x200, 0(0x0), ) = 0x1(1)
privacy.exe(320): privacy.exe:0x15d619 => kernel32.dll:Process32Next = 0x1(1)
privacy.exe(320): privacy.exe:0x15d619 => kernel32.dll:Process32Next = 0x0(0)
privacy.exe(320): privacy.exe:0x7c64 => kernel32.dll:CloseHandle(0x20c, ) = 0x1(1)
```

Fig. 2. Library call log of Win32/FakeRean.

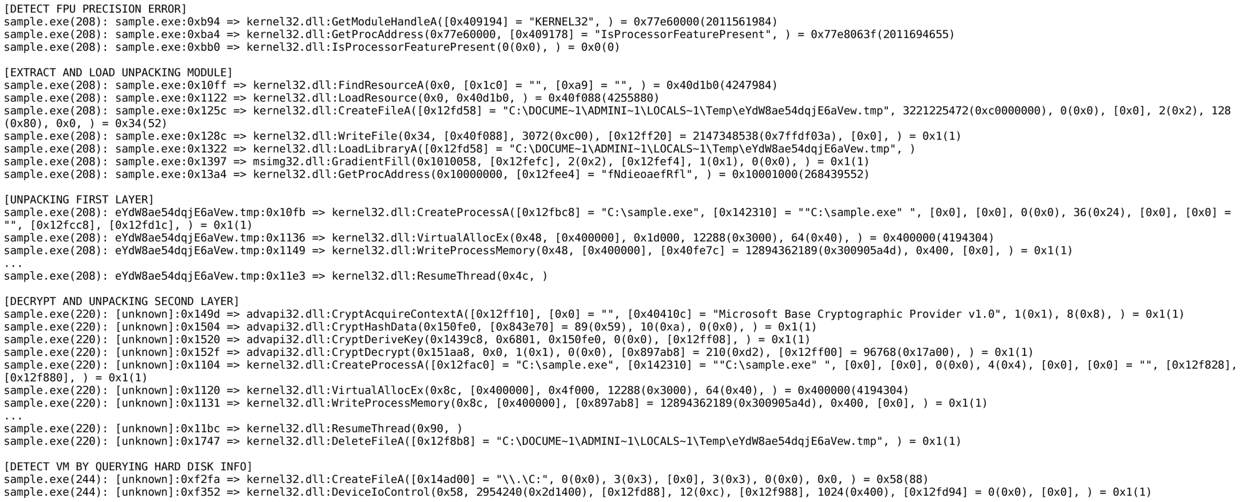
prevent static analysis. The malware sample also used several anti-debugging and anti-VM tricks to thwart dynamic analysis.

As shown in the library call log in Fig. 3, the malware first called *GetModuleHandleA* and *GetProcAddress* to find the address of the function *IsProcessorFeaturePresent* after it started. This function is used to determine whether a specified feature is supported by the CPU. The malware called this function with argument of value *0* to check if the floating point precision error exists. This error is also called the “FDIV bug” (Wikipedia, 2012), which is an FPU bug in old Pentium processors; modern processors should not have this bug so the call should return *0*. However, some emulators do not handle this check well and return *1*, which could be exploited by the malware to detect their existence. Our system returns *0* so the malware continues to run.

The malware then began to unpack itself. It first called *FindResourceA* and *LoadResource* to locate and extract the unpacking module from the resource section. It created a file named “*eYdW8ae54dqjE6aVew.tmp*” in the temporary directory and wrote the content of the unpacking module into that file. To observe the behavior of this module, we manually added it as part of the malware code. It called *LoadLibraryA* to load the unpacking module, then called *GradientFill* to draw a blue rectangle on the top left corner of the screen. After that, it located and called a function named “*fNdieoaefRfl*” in the unpacking module to do the actual work.

The unpacking method used in this malware is different from our first malware sample. Instead of unpacking into the existing process, the unpacking module first created a new process as a placeholder and then wrote the unpacked binary into the newly created process for execution. The new process was created with the *CREATE_SUSPENDED* flag so it would not execute before the unpacking finished. Then the unpacking module called *VirtualAllocEx* to allocate an area of memory in the address space of the new process, and called *WriteProcessMemory* to write the unpacked binary into that area. After that, it called *ResumeThread* to resume the execution of the new process.

The unpacking module only unpacked the first layer of the malware. The unpacking of the second layer was done by the newly created process, using a similar unpacking method. The difference here is that the unpacked binary was decrypted before injected into a new process. From the log we can see the malware called *CryptAcquireContextA* to request cryptography service of “*Microsoft Base Cryptographic Provider v1.0*”. It first called *CryptCreateHash* and *CryptHashData* to hash the 10 bytes starting from *0x843e70*.



```
[DETECT FPU PRECISION ERROR]
sample.exe(208): sample.exe:0xb94 => kernel32.dll:GetModuleHandleA([0x409194] = "KERNEL32", ) = 0x77e60000(2011561984)
sample.exe(208): sample.exe:0xba4 => kernel32.dll:GetProcAddress(0x77e60000, [0x409178] = "IsProcessorFeaturePresent", ) = 0x77e8063f(2011694655)
sample.exe(208): sample.exe:0xbb0 => kernel32.dll:IsProcessorFeaturePresent(0(0x0), ) = 0x0(0)

[EXTRACT AND LOAD UNPACKING MODULE]
sample.exe(208): sample.exe:0x10ff => kernel32.dll:FindResourceA(0x0, [0x1c0] = "", [0xa9] = "", ) = 0x40d1b0(4247984)
sample.exe(208): sample.exe:0x1122 => kernel32.dll:LoadResource(0x0, 0x40d1b0, ) = 0x40f088(4255880)
sample.exe(208): sample.exe:0x125c => kernel32.dll:CreateFileA([0x12fd58] = "C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\eYdW8ae54dqjE6aVew.tmp", 3221225472(0xc0000000), 0(0x0), [0x0], 2(0x2), 128
(0x80), 0x0, ) = 0x34(52)
sample.exe(208): sample.exe:0x128c => kernel32.dll:WriteFile(0x34, [0x40f088], 3072(0xc00), [0x12ff20] = 2147348538(0x7ffdf03a), [0x0], ) = 0x1(1)
sample.exe(208): sample.exe:0x1322 => kernel32.dll:LoadLibraryA([0x12fd58] = "C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\eYdW8ae54dqjE6aVew.tmp", )
sample.exe(208): sample.exe:0x1397 => msimg32.dll:GradientFill(0x1010058, [0x12fefc], 2(0x2), [0x12fef4], 1(0x1), 0(0x0), ) = 0x1(1)
sample.exe(208): sample.exe:0x13a4 => kernel32.dll:GetProcAddress(0x10000000, [0x12fee4] = "fNdieoaefRfl", ) = 0x10001000(268439552)

[UNPACKING FIRST LAYER]
sample.exe(208): eYdW8ae54dqjE6aVew.tmp:0x10fb => kernel32.dll:CreateProcessA([0x12fbc8] = "C:\sample.exe", [0x142310] = ""C:\sample.exe" ", [0x0], [0x0], 0(0x0), 36(0x24), [0x0], [0x0] =
"", [0x12fcc8], [0x12fd1c], ) = 0x1(1)
sample.exe(208): eYdW8ae54dqjE6aVew.tmp:0x1136 => kernel32.dll:VirtualAllocEx(0x48, [0x400000], 0x1d000, 12288(0x3000), 64(0x40), ) = 0x400000(4194304)
sample.exe(208): eYdW8ae54dqjE6aVew.tmp:0x1149 => kernel32.dll:WriteProcessMemory(0x48, [0x400000], [0x40fe7c] = 12894362189(0x300905a4d), 0x400, [0x0], ) = 0x1(1)
...
sample.exe(208): eYdW8ae54dqjE6aVew.tmp:0x11e3 => kernel32.dll:ResumeThread(0x4c, )

[DECRYPT AND UNPACKING SECOND LAYER]
sample.exe(220): [unknown]:0x149d => advapi32.dll:CryptAcquireContextA([0x12ff10], [0x0] = "", [0x40410c] = "Microsoft Base Cryptographic Provider v1.0", 1(0x1), 8(0x8), ) = 0x1(1)
sample.exe(220): [unknown]:0x1504 => advapi32.dll:CryptHashData(0x150fe0, [0x843e70] = 89(0x59), 10(0xa), 0(0x0), ) = 0x1(1)
sample.exe(220): [unknown]:0x1520 => advapi32.dll:CryptDeriveKey(0x1439c8, 0x6801, 0x150fe0, 0(0x0), [0x12ff08], ) = 0x1(1)
sample.exe(220): [unknown]:0x152f => advapi32.dll:CryptDecrypt(0x151aa8, 0x0, 1(0x1), 0(0x0), [0x897ab8] = 210(0xd2), [0x12ff00] = 96768(0x17a00), ) = 0x1(1)
sample.exe(220): [unknown]:0x1104 => kernel32.dll:CreateProcessA([0x12fac0] = "C:\sample.exe", [0x142310] = ""C:\sample.exe" ", [0x0], [0x0], 0(0x0), 4(0x4), [0x0], [0x0] = "", [0x12f828],
[0x12f880], ) = 0x1(1)
sample.exe(220): [unknown]:0x1120 => kernel32.dll:VirtualAllocEx(0x8c, [0x400000], 0x4f000, 12288(0x3000), 64(0x40), ) = 0x400000(4194304)
sample.exe(220): [unknown]:0x1131 => kernel32.dll:WriteProcessMemory(0x8c, [0x400000], [0x897ab8] = 12894362189(0x300905a4d), 0x400, [0x0], ) = 0x1(1)
...
sample.exe(220): [unknown]:0x11bc => kernel32.dll:ResumeThread(0x90, )
sample.exe(220): [unknown]:0x1747 => kernel32.dll:DeleteFileA([0x12f8b8] = "C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\eYdW8ae54dqjE6aVew.tmp", ) = 0x1(1)

[DETECT VM BY QUERYING HARD DISK INFO]
sample.exe(244): [unknown]:0xf2fa => kernel32.dll:CreateFileA([0x14ad00] = "\\.\C:", 0(0x0), 3(0x3), [0x0], 3(0x3), 0(0x0), 0x0, ) = 0x58(88)
sample.exe(244): [unknown]:0xf352 => kernel32.dll:DeviceIoControl(0x58, 2954240(0x2d1400), [0x12fd88], 12(0xc), [0x12f988], 1024(0x400), [0x12fd94] = 0(0x0), [0x0], ) = 0x1(1)
```

Fig. 3. Library call log of Win32/Dorkbot.AJ.

The value of the second argument of *CryptCreateHash* was *0x8003*, which indicates MD5 was used for hashing here. It then called *CryptDeriveKey* to compute a key from the hash. Again, we could infer that the RC4 algorithm was used for decryption as the value of the second argument was *0x6801*. After that, the malware called *CryptDecrypt* to decrypt the unpacked binary using the derived key and inject the decrypted binary into the last process for execution. Finally, it deleted the file of the unpacking module used in the unpacking of the first layer and then exited. Note that a system call tracing tool would not observe the decryption behavior as no system call was involved here.

Surprisingly, in our experiment, we did not observe any malicious behavior of the last process, which was assumed to execute the actual payload of the malware. We suspected our system was detected, so we submitted this sample to CWSandbox for reference. The result confirmed that the last process did have malicious behavior such as spawning remote threads in other processes. We found out the reason when we looked at the library call log again: the malware called *DeviceIoControl* with the control code *IOCTL_STORAGE_QUERY_PROPERTY* to detect us by querying the description of hard disk. Since KVM use QEMU to emulate hardware devices, the disk model is "*QEMU HARDDISK*" and the disk vendor is "*QEMU*", which is apparently not a real disk drive. Note that it's the hypervisor we choose to implement IntroLib but *not* the design of IntroLib that leads to the detection. If we implement IntroLib on another hypervisor that allows the guest to directly access real hardware devices, such as BitVisor (Shinagawa et al., 2009), then such method of detecting emulated devices would fail.

4.1.3. Summary

It is impossible to get such detailed understanding of the malware samples as above without the fine-grained library call tracing logs. Some malicious behavior would not be revealed with system call tracing, either because the behavior does not even involve a system call, or because the system call is too low-level to enable reconstruction of the high-level semantics of malware behavior. Even if sometimes the behavior could not be directly observed in the log, the library call sites in the log could help to pinpoint the function or narrow the range of the code that has to be manually inspected by analysts.

4.2. Transparency to malware

To evaluate the transparency of IntroLib to malware, we first crafted synthetic anti-emulation samples using detection code in previous work (Ferrie, 2006, 2007; Raffetseder et al., 2007). We chose the detection code targeting QEMU, as QEMU is most commonly used by researchers to build dynamic analysis systems. Each of these samples executes some instructions that are not perfectly emulated (e.g. the emulator fails to generate an exception.) We traced these samples using IntroLib to see if any detection happened. As we expected, none of these samples were able to detect our system. IntroLib is built on hardware virtualization which executes instructions natively on the CPU, so anti-emulation attacks that rely on detecting imprecise instruction emulation would not work.

We then verified transparency of IntroLib against real-world malware. Among the 93 real-world malware samples we analyzed, 3 of them crashed before showing any behavior. We submitted these 3 samples to Anubis and CWSandbox and found out these samples also crashed in their analysis environments. They still crashed when being executed in an unmodified KVM, which means the crashes were not due to detection of IntroLib.

We successfully use IntroLib to obtain library call logs of all the remaining 90 samples. By manually inspecting the call logs, we were able to identify malicious behavior of each sample. We then submitted these 90 samples to Anubis and CWSandbox for comparison.

Table 1
Performance overhead of IntroLib measured by PCMark05 benchmarks.

Test case	Vanilla KVM	IntroLib(Standby)	IntroLib(Tracing)	Overhead(Standby)	Overhead(Tracing)
HDD-XP Startup	7.15 MB/s	7.13 MB/s	5.10 MB/s	0.26%	40.17%
Html Render	3.96 Pages/s	3.53 Pages/s	3.57 Pages/s	12.26%	11.21%
File Decryption	80.67 MB/s	74.93 MB/s	74.78 MB/s	7.67%	7.88%
Text Edit	102.60 Pages/s	95.4 Pages/s	90.67 Pages/s	7.54%	13.16%
Image Decompression	15.90 MPixel/s	14.10 MPixel/s	14.18 MPixel/s	12.71%	12.11%
File Compression	3.49 MB/s	3.26 MB/s	3.23 MB/s	6.91%	7.99%
File Encryption	20.26 MB/s	19.04 MB/s	18.88 MB/s	6.39%	7.29%
Total Time	552 s	572 s	633 s	3.62%	14.67%

Anubis failed to analyze 6 of these 90 samples. From the screenshot captured by Anubis, we observed three samples[4] showed a message box with text "*OK CRC error! File content has been modified. If you run a system debugger, clear all breakpoints before running this program!*" and then exited, which clearly indicated that Anubis had been detected. Further investigation showed that these 3 samples are packed with tElock, which is known to be able to detect Anubis. In contrast, from the library call logs generated by IntroLib, we were able to identify that the goal of these 3 samples is to steal user accounts of online games by hijacking their executable file paths in registry. The other 3 samples[5] are variants of worm Win32/Dorkbot.A. All of them are packed using Armadillo. It is known that Anubis could be detected by Armadillo, so it's no surprise that these samples showed no behavior in Anubis. In IntroLib, all of them exhibited malicious behavior. One of these 3 samples, the Win32/Dorkbot.AJ, has been thoroughly analyzed in our second case study.

CWSandbox successfully analyzed all these 90 samples. However, this does not mean it is transparent; in fact CWSandbox could be easily detected even by user-mode malware as it relies on user-mode API hooking. To prove this, we wrote a small user-mode sample using the detection code in Royal (2008). The detection code checks if *CreateFileA* is hooked by reading the first two bytes of the function. If the value of the first two bytes is *0xff25*, which is the opcode of a jump instruction, it means the function is hooked by CWSandbox; the program will then create a file named "*in_cwsandbox*" to indicate detection. We submitted our sample to CWSandbox and its log showed the file "*in_cwsandbox*" was created, which meant it was detected. We ran the same sample in IntroLib and the sample exited without creating the file. IntroLib could maintain transparency to all similar attacks which detect the presence of analysis system in memory, as it neither resides in nor introduces any modification to guest memory.

4.3. Performance overhead

We measured the performance of IntroLib using PCMark05, an industry standard benchmark for testing the comprehensive performance of a system. The system test suite of PCMark05 includes both computation intensive test cases such as image decompression, and library call intensive ones such as emulating Windows XP startup. We ran the benchmark in three environments: unmodified KVM, IntroLib in standby mode and IntroLib in tracing mode. In standby mode, IntroLib was not tracing any process, but it was ready to identify and trace specified processes as soon as they appeared; in tracing mode, we let IntroLib trace the benchmarking process to see how much IntroLib affected the performance of the program being traced. In this part of evaluation, we turned off the logic of handling timing attack in IntroLib to make sure that PCMark05 in the guest VM measured the real elapsed time instead of the time adjusted by IntroLib.

The result of the performance test is shown in Table 1. We can see that in computation intensive test cases such as file encryption, decryption and compression, the performance degradation introduced by IntroLib is about 6%–12%. As we expected, in these cases, IntroLib did not introduce much more overhead in tracing mode than in standby mode. On the other hand, for library call intensive ones such as emulating Windows XP startup, the overhead is much larger when IntroLib is performing library call tracing. The size of the log we obtained during the test in tracing mode is 86.3 MB, which confirmed a lot of library calls had been made. We point out that, unlike Windows XP startup, most malware programs are *not* as library call intensive and hence will incur much less tracing overhead. We also measured the total time used for execution in each testing environment, respectively. When IntroLib is standby, the total performance overhead is about 3.62%. Even when IntroLib is fully working at library call tracing, the performance overhead is only 14.67%. This is acceptable for deployment in live malware forensics system.

5. Discussion

We have verified IntroLib's transparency property based on the formal requirements defined in Dinaburg et al. (2008). The formal proof is elided due to lack of space. Still, IntroLib is not completely undetectable. For example, when IntroLib intercepts a control-flow transition and switches the SPT, the TLB will be flushed and attackers can use method as described in Ptacek (2012) to detect the change. We may use tagged TLB to associate TLB entries with the SPT and avoid the TLB flush, but the problem still exists: as the TLB entries become tagged, when we switch SPT, the TLB entries associated with the previous SPT will be inactive in the context of the new SPT.

[4] MD5 hash: eb26dbddcd3913ef93eb9e212baabd39, c61121bae21bb01a34b7f4f00d378139 and eceea0402dbb1bcc2c2b802c0a28350d.

[5] MD5 hash: d7f673210ccd361f69b4676e9c8752ce, eafab52ac69dee4de4e02da1f5ccd9c4 and f27dd9ba200266d7c9df2d345b8f882f.

Such change could still be detected by carefully crafted malware. However, TLB-based detection incurs high false positive rate as the TLB changes are not solely caused by IntroLib. For example, context switches in the guest also cause TLB flushes; and some Intel CPUs with hardware virtualization support also flush the TLB on every VMExit.

It's worth noting that IntroLib tries to conceal itself, but not the underlying virtualization platform, from malware. Detection of virtualization is still possible. However, as virtualization becomes widely used in production systems, malware that identifies all virtualized systems as analysis environments will lose a considerable and increasing number of targets.

Our control-flow transition interception mechanism relies on page level protection, so *intra-page* transitions could not be intercepted. Attackers could move library code to the same page of the malware code to avoid being intercepted. However, as the code section of many library binaries are larger than the minimum page size in x86 (4 KB), it is difficult (if at all possible) to co-locate the needed library code with the malware code in one single page.

IntroLib relies on some in-guest data when performing memory layout identification and library call logging, for example, kernel objects that describe file mapping and metadata of library binaries. Although we leverage the copy-on-write disk to make sure that IntroLib only reads un-tampered metadata, attackers might still thwart tracing by not presenting the required data. Malware could obfuscate its memory layout, for example, load library without using file mapping, or move the library code to another area after loading. To frustrate library function identification, the malware could copy an existing library to a new file and load it; since the new file does not exists on the original read-only disk, IntroLib will fail to read it.

To address this problem, one possible solution is content-based identification of library code area and functions. We can first build a database which consists library function binaries of all existing libraries on the disk. When a control-flow transition is detected, we could identify the target function by matching its binary (content) with those in the database (and skipping those relocatable bytes). To thwart such a solution, malware authors would have to polymorph in-memory library functions, which is known to be hard to implement. We leave the development of this solution as our future work.

IntroLib does not capture system calls directly made by malware. In such case, we could utilize existing system call tracing work (Jiang and Wang, 2007) as a complement.

6. Conclusion

We have presented IntroLib, a tool that performs efficient and transparent library calls tracing for malware forensics. IntroLib utilizes hardware virtualization to elevate its transparency to malware, and use page table-based mechanism to efficiently intercept user-level library calls at hypervisor level. We evaluate our KVM-based IntroLib prototype using 93 real-world Windows malware samples. The result shows that IntroLib uncovers richer information about malware intent and behavior than system call tracing-based approaches, yet it remains transparent to malware with emulation detection logic. Our performance test shows that IntroLib incurs low overhead and can therefore be applied to live malware analysis and forensics.

Acknowledgment

We thank the anonymous reviewers for their insightful comments. This research was supported, in part, by DARPA under Contract 12011593 and NSF under grant 0855141. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of DARPA or NSF.

References

Bayer U, Kruegel C, Kirda E. Ttanalyze: a tool for analyzing malware. In: 15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference; 2006.

Chen X, Andersen J, Mao Z, Bailey M, Nazario J. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In: DSN'08. IEEE; 2008. p. 177–86.

Dinaburg A, Royal P, Sharif M, Lee W. Ether: malware analysis via hardware virtualization extensions. In: CCS'08. ACM; 2008. p. 51–62.

Ferrie P. Attacks on virtual machine emulators. Symantec Advanced Threat Research; 2006.

Ferrie P. Attacks on more virtual machine emulators. Symantec Technology Exchange; 2007.

Futuremark. Pcmark05. URL: http://www.futuremark.com/products/pcmark05/; 2012.

Garfinkel T, Rosenblum M. A virtual machine introspection based architecture for intrusion detection. In: NDSS'03; 2003.

Huang C. cproto. URL: http://sourceforge.net/projects/cproto/; 2012.

International Secure Systems Lab. Anubis: analyzing unknown binaries. URL: http://anubis.iseclab.org/; 2012.

Jiang X, Wang X. Out-of-the-box monitoring of vm-based high-interaction honeypots. In: RAID'07. Springer-Verlag; 2007. p. 198–218.

Jibz, Qwerton, snaker, xineohP. Peid. URL: http://www.peid.info/; 2012.

Kang M, Yin H, Hanna S, McCamant S, Song D. Emulating emulation-resistant malware. In: Proceedings of the 1st ACM workshop on virtual machine security. ACM; 2009. p. 11–22.

Kaspersky Lab. Av vs fakeav. URL, http://habrahabr.ru/company/kaspersky/blog/133621/; 2012.

Microsoft. Microsoft portable executable and common object file format specification; 2010.

Moser A, Kruegel C, Kirda E. Limits of static analysis for malware detection. In: ACSAC'07. IEEE; 2007. p. 421–30.

Nguyen A, Schear N, Jung H, Godiyal A, King S, Nguyen H. Mavmm: lightweight and purpose built vmm for malware analysis. In: ACSAC'09. IEEE; 2009. p. 441–50.

Norman. Norman sandbox. URL: http://www.norman.com/about_norman/technology/norman_sandbox/; 2012.

Ptacek T. Side-channel detection attacks against unauthorized hypervisors. URL: http://www.securityfocus.com/blogs/253; 2012.

Raffetseder T, Kruegel C, Kirda E. Detecting system emulators. Information Security; 2007:1–18.

Riley R, Jiang X, Xu D. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In: RAID'08. Springer; 2008. p. 1–20.

Rodionov E, Matrosov A. The evolution of tdl: conquering x64. URL: http://go.eset.com/us/resources/white-papers/The_Evolution_of_TDL.pdf; 2012.

Royal P. Alternative medicine: the malware analysts blue pill. USA: Black Hat; 2008.

Seshadri A, Luk M, Qu N, Perrig A. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In: ACM SIGOPS Operating Systems Review, vol. 41. ACM; 2007. p. 335–50.

Shinagawa T, Eiraku H, Tanimoto K, Omote K, Hasegawa S, Horie T, et al. Bitvisor: a thin hypervisor for enforcing i/o device security. In: VEE'09. ACM; 2009. p. 121–30.

Silicon Realms. Armadillo. URL: http://www.siliconrealms.com/armadillo.php; 2012.

Srivastava A, Giffin J. Efficient monitoring of untrusted kernel-mode execution. In: NDSS'11; 2011.

TIS Committee. Tool interface standard (tis) executable and linking format (elf) specification version 1.2; 1995.

Wang Z, Jiang X, Cui W, Ning P. Countering kernel rootkits with lightweight hook protection. In: CCS'09. ACM; 2009. p. 545–54.

Wikipedia. Pentium fdiv bug. URL: http://en.wikipedia.org/wiki/Pentium_FDIV_bug; 2012.

Willems C, Holz T, Freiling F. Toward automated dynamic malware analysis using cwsandbox. IEEE Security & Privacy; 2007: 32–9.

Yan L, Jayachandra M, Zhang M, Yin H. V2e: combing hardware virtualization and software emulation for transparent and extensible malware analysis. In: VEE'12. ACM; 2012. to appear.