V2E: Combining Hardware Virtualization and Software Emulation for Transparent and Extensible Malware Analysis

Lok-Kwong Yan[†‡] Manjukumar Jayachandra[†] Mu Zhang[†] Heng Yin[†]

[†] Department of Electrical Engineering & Computer Science
Syracuse University, Syracuse, New York, USA

[‡]Air Force Research Laboratory
Rome, New York, USA

{loyan, mjayacha, muzhang, heyin}@syr.edu

Abstract

A transparent and extensible malware analysis platform is essential for defeating malware. This platform should be transparent so malware cannot easily detect and bypass it. It should also be extensible to provide strong support for heavyweight instrumentation and analysis efficiency. However, no existing platform can meet both requirements. Leveraging hardware virtualization technology, analysis platforms like Ether can achieve good transparency, but its instrumentation support and analysis efficiency are weak. In contrast, software emulation provides strong support for code instrumentation and good analysis efficiency by using dynamic binary translation. However, analysis platforms based on software emulation can be easily detected by malware and thus is poor in transparency. To achieve both transparency and extensibility, we propose a new analysis platform that combines hardware virtualization and software emulation. The essence is *precise heterogeneous replay*: the malware execution is recorded via hardware virtualization and then replayed in software. Our design ensures the execution replay to be precise. Moreover, with page-level recording granularity, the platform can easily adjust to analyze various forms of malware (a process, a kernel module, or a shared library). We implemented a prototype called *V2E* and demonstrated its capability and efficiency by conducting an extensive evaluation with both synthetic samples and 14 realworld emulation-resistant malware samples.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection—Invasive software

General Terms Security, Performance

1. Introduction

Malware analysis is an important step to defend against malware. Given a piece of unknown malware, the objective of malware analysis is to reverse engineer it and quickly reveal its inner workings. As malware is often heavily obfuscated to thwart static binary analysis, dynamic binary analysis becomes increasingly prevalent. In this approach the analyst runs a malware sample of interest in an emulated execution environment (e.g., QEMU [30]) and then monitors and analyzes its malicious behavior in a fine-grained manner (at the instruction level). A great deal of research efforts take this approach [4, 8, 9, 15, 21, 24, 27, 32, 34, 40, 41].

The main advantages of this analysis approach are flexibility and efficiency for code instrumentation. Since the emulator is a regular user-level program, it is relatively simple for security analysts to add instrumentation code (such as taint analysis and symbolic execution). Due to dynamic binary translation, the performance overhead for such heavy instrumentation is often acceptable. However, its prominent drawback is lack of transparency. Because it is extremely difficult (if not completely infeasible) to emulate every aspect of real hardware, malware can take advantage of these discrepancies to detect the emulated environment and stay dormant to avoid analysis. Researchers have investigated this problem extensively and identified a large number of different detection methods [16, 26, 31]. Our measurement study in Section 5.2 shows that emulation-resistant malware has become a prevalent threat in the wild.

To address this transparency issue, Dinaburg et al. proposed to leverage hardware virtualization and developed a system called Ether [13]. Since the malicious code is executed on bare metal hardware and the in-guest changes caused by analysis can be intercepted and hidden by the hypervisor, this approach can achieve ideal transparency. However, Ether is not the ultimate solution. It incurs a prohibitive performance penalty to conduct instruction-level analysis by simply enabling single-step mode. Our experiment shows an approximately 3000 times slowdown by enabling single-step, not to mention the extra heavy instrumentation needed by in-depth malware analysis. Moreover, it is far more challenging to implement the code instrumentation tools within a hypervisor (i.e. Ring -1) than an emulator (i.e. Ring 3).

In this paper, we aim to bring the best of these two worlds. We aim to develop a malware analysis platform that is both transparent and extensible to facilitate custom fine-grained malware analysis. The essence of this platform is *precise heterogeneous replay*. That is, we record malware execution using hardware virtualization for transparency, and then replay and analyze the malware's execution using dynamic binary translation for flexibility and efficiency of in-depth analysis. The idea of heterogeneous replay was first proposed and implemented in Aftersight [10], which records the virtual machine execution from VMware and replays it in QEMU, for heavyweight analyses (such as bug detection) on production workload. In contrast to Aftersight, our platform needs to work under the malicious context: the emulator should exactly replay the execution recorded from the hardware virtualization platform in spite of the fact that malware is trying to detect every possible heterogeneous property in these two systems.

One challenge of precise heterogeneous replay is how to strike a balance between the recorder and the replayer. On one hand, if the recorder does not record enough events and states, the replayer

VEE'12, March 3–4, 2012, London, England, UK.

cannot precisely reconstruct the execution. On the other hand, if the recorder gathers complete information for every single instruction or event and leave an easy task to the replayer, the recording performance would become unacceptable. We carefully classify various operations and instructions into several categories and handle them properly to ensure precise replay.

We implemented a prototype system, called *V2E*. The recorder has been implemented in KVM [23], and TEMU (a dynamic binary analysis platform [37]) has been modified to precisely replay the execution. With minimum changes, the existing analysis plugins (such as taint analysis, unpacker, and tracing) work properly, achieving the advantages of transparency and greater analysis efficiency.

We have evaluated V2E using both synthetic and realworld emulation-resistant malware samples. These same samples were successfully recorded and replayed on our modified TEMU revealing the behavior hidden from the original TEMU platform.

In summary, this paper makes the following contributions:

- We propose a new technique that combines hardware virtualization and dynamic binary translation to achieve both transparency and efficiency for fine-grained analysis.
- We designed and implemented a prototype called V2E. While the recording component was implemented in KVM to record malware execution in a transparent fashion, the replay component was built in TEMU, largely by modifying its dynamic binary translation logic. Consequently, existing TEMU plugins have gained transparency and higher analysis efficiency after minor changes.
- We conducted extensive experiments and analysis, using both synthetic and realworld emulation-resistant malware samples. These samples utilize a large variety of methods for detecting emulated environments, so we believe our evaluation is well rounded. All emulation-resistant malware samples were successfully analyzed completely.

The rest of the paper is organized as follows. The next section lists the design goals that are essential for in-depth malware analysis and gives an overview of our approach. Section 3 and Section 4 describe the design and implementation of the recording component and replay component, respectively. Section 5 presents our experimental results. Section 6 discusses the limitations of the current implementation. Section 7 surveys the related work. Finally, the paper concludes with Section 8.

2. Design Goals & Approach

We first list the design goals for in-depth malware analysis, and then explain how our approach is able to address these design goals.

2.1 Design Goals

The following design goals are essential for in-depth malware analysis:

- **G1: Transparency.** The presence of the analysis environment should remain invisible to malware, voiding its intent to escape investigation.
- **G2: Instrumentation Support.** It should be relatively simple to add custom instrumentation on malicious code execution. In many cases, this instrumentation can be heavyweight, such as dynamic taint analysis and instruction-level tracing.
- **G3: Efficiency.** The efficiency for malware analysis is twofold: 1) it should be efficient enough to monitor computation intensive and highly interactive malware; and 2) performance overhead for heavy code instrumentation should be acceptable.

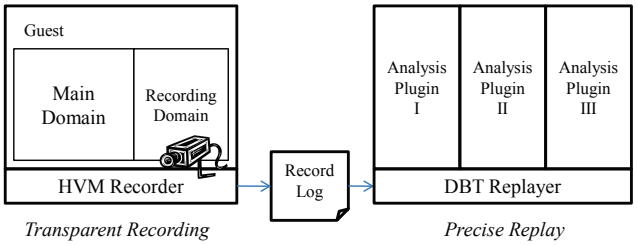

Figure 1. Architecture Overview

- **G4: Adjustable View.** Malware can present itself in various forms, such as user process, shared library, dynamically injected code, kernel module, etc. It should be easy to adjust our analysis focus to concentrate on malware's behavior instead of the execution of the rest of the system.

2.2 Architecture

The overall architecture is depicted in Figure 1. The malware sample under investigation is loaded into the guest system using hardware virtualization, to achieve transparency (**G1**). Although hardware virtualization may still be detected under certain circumstances [31] and a remote time source can be used to measure the real timing difference [12], as hardware virtualization has been widely deployed on production systems, detecting hardware virtualization environments becomes increasingly irrelevant. Ether demonstrated that hardware virtualization can achieve transparency in a practical sense [13].

The guest system is partitioned into two realms. The malware resides in the *recording realm*, and the rest of the guest system (such as the guest OS and the other applications) remains in the *main realm*. Depending on where the malware is present, we can put a user process, a shared library, a kernel module, or any combination of them into the recording realm to have a closer look at the malware's behavior. Such a separation fulfills the design goal of adjustable view (**G4**). It also partially addresses the efficiency issue (**G3**), because the irrelevant system execution is excluded from the recording realm. Some analysis techniques (such as Panorama [40] and HookFinder [41]) do need to observe the entire system execution. In this case, the recording realm includes the entire guest system, falling back to the whole-system recording.

The record log obtained from the recorder is then fed into the replayer. Using the dynamic binary translation technique, the replayer is able to offer acceptable performance for fine-grained code instrumentation, and thus achieves analysis efficiency (the second part of **G3**). The replayer facilitates any existing malware analysis platforms that are based on dynamic binary translation. Therefore, the existing analysis plugins on these analysis platforms can continue to work with minimum changes. It addresses the instrumentation support (**G2**).

2.3 Precise Heterogeneous Replay

Using hardware virtualization, we can record the malware execution in a transparent and efficient manner. To support various in-depth malware analysis techniques, we need to *precisely* replay the execution using dynamic binary translation. That is, at every single execution time, the program state in replay is exactly the same as in recording in spite of the fact that the malware execution is trying to detect various discrepancies between the real hardware and the emulated system.

Formal Definition. At each time i, S_i represents the program state (including CPU registers and memory) and I_i specifies the input. I_i may be null if no input occurs at time i. Then a transition

function f characterizes the real hardware: $S_i = f(S_{i-1}, I_{i-1})$. Similarly, there is a transition function f' for the emulated hardware. Suppose that $f' = f$, given S_0 and I we can replay the whole execution. However, according to automata theory, determining if $f' = f$ is equivalent to the problem of determining whether two Turing machines are equal, which is known to be undecidable [35]. In practice, $f' \neq f$, because it is nearly impossible to correctly emulate some aspects of hardware. That is, for some i, $S'_i = f'(S_{i-1}, I_{i-1})$ and $S'_i \neq S_i$. Therefore, in addition to recording S_0 and I, for any moment j when $S'_j \neq S_j$, we need to record the state change $\Delta_j = S_j - S_{j-1}$. Then the new transition function f'_r is defined as below:

$$S'_i = f'_r(S'_{i-1}, I_{i-1}, \Delta_i) = \begin{cases} S'_{i-1} + \Delta_i & \text{if } \Delta_i \neq null \\ f'(S_{i-1}, I_{i-1}) & \text{Otherwise} \end{cases}$$

In other words, during replay, whenever a state change Δ_i has been recorded for time i, we will directly apply the recorded state change such that $S'_i = S'_{i-1} + \Delta_i$. Using simple induction, we can prove that with (S_0, I, Δ) and f'_r, $S'_i = S_i$ always holds true for all $i \in [0, n]$.

From Theory to Practice. From the formal discussion, we can see that the key to successful replay is to determine when to use f' and when to apply Δ. In other words, if we are confident that certain instructions and events can be correctly emulated in software, then we simply emulate them. Otherwise, we should record the state changes and then during replay apply these changes.

Fortunately, for general instructions like data transfer (e.g., `mov`, `push`, `pop`), control transfer (e.g., `call`, `ret`, `jz`, `jmp`), and integer arithmetic (e.g., `add`, `shl`), it is fairly easy to emulate them correctly in software because of their semantics are simple and remain the same across processor series.[1] Moreover, these instructions are the vast majority in program execution. As a result, the efficiency of both recorder and replayer can be ensured. We believe that this is a valid assumption, because these common instructions are tested over and over again in many different application contexts. Even if there is an emulation bug in one of these common instructions, because of its simple semantics, it should be easy to fix.

External interrupts, memory-mapped IO (MMIO), port IO, direct memory access (DMA), and timestamp counter are inputs I to the guest system. Like other deterministic replay systems [10, 14, 36], we choose to record these events if they happen in the recording realm.

Software exceptions, model-specific registers, and the `cpuid` instruction are not generally treated as inputs in previous replay systems. However, it is extremely difficult to emulate them right. Software exceptions are triggered when certain condition checks fail in the processor. It is fairly complicated to emulate all these condition checks in the exactly same way as in the real processor, not to mention that a specific processor may have CPU bugs that raise incorrect exceptions. The behaviors of model-specific registers and the `cpuid` instruction are processor specific. It is a daunting task to correctly emulate all the specifics to support at least the common CPU series. Therefore, we choose to record and replay exceptions, model-specific registers and `cpuid` as state changes Δ.

Floating point instructions and SIMD (Single Instruction Multiple Data) instructions (e.g., MMX and SSE) are generally difficult to be emulated correctly as well. We could also record the results of these instructions as Δ. However, for programs that heavily perform these operations, the performance for both record and replay may greatly degrade. Alternatively, we choose to pass these instructions directly to the hardware processor during replay. This solution assumes that the replayer is running on a machine supporting this set of floating point and SIMD instructions. This assumption can easily hold by running the recorder and the replayer on the same kind of machines (or even the same machine).

[1] Note that these common instructions may still cause discrepancies in exceptions, which are handled separately.

Operation Type	Solution
Data Transfer / Control Transfer / Integer Arithmetic	Emulate
Interrupts / MMIO / Port IO / DMA / TSC	Replay as I
Exceptions / System Registers / CPUID	Replay as Δ
Floating Point / SIMD Instructions	Pass through

Table 1. Operations and Corresponding Solutions.

As a summary, Table 1 lists these operations and their corresponding solutions: *emulate*, *replay*, or *pass-through*. We emphasize that the platform following this design principle does not immediately become transparent completely, because we do not preclude that the emulation of some of these common instructions may still be buggy. However, once identified, these bugs can be easily fixed and the transparency of the platform will be further improved.

3. Transparent Recorder

For successful replay, we need to capture S_0, I and Δ in a transparent and efficient manner. In this section, we describe how we are able to achieve this goal by leveraging hardware virtualization.

3.1 Mediating Recording Realm

In order to monitor malware in various forms, including kernel modules, shared libraries and processes, the recording realm needs to be defined at the page-level granularity and the interaction between the recording realm and the rest of the system needs to be mediated.

In hardware virtualization, Two Dimensional Paging (TDP) is a memory virtualization mechanism. While the conventional page table pointed by CR3 in the guest will be used to translate a Guest Virtual Address (GVA) into its Guest Physical Address (GPA), the second-layer page table maintained by the hypervisor will translate the Guest Physical Address into the Host Physical Address (HPA). The maintenance of the second-layer page table is invisible to the guest. AMD and Intel have different implementations: Nested Page Tables (NPT) for AMD and Extended Page Tables (EPT) for Intel.

We take advantage of TDP to mediate the recording realm. Specifically, we create two TDP tables, which partition the guest physical memory into two memory spaces, one for the recording realm and the other for the rest of the guest system. The code pages that belong to the monitored malware will be loaded into the recording realm, such that the interaction of the monitored malware with the rest of the system can be mediated by the TDP page faults and other VMExit (transitions from the guest to the hypervisor) events. Once again, these VMExit events are invisible to the guest system.

This TDP-based recording realm is flexible enough to monitor a small code module, a full user process, and even the entire guest system, depending on what pages are loaded into the recording realm.

3.2 Basic Scheme

In the basic design, the two guest physical memory spaces are mutually exclusive. That is, each individual guest physical page can only be present in either the recording realm or the main realm, but not both. This basic design ensures mediating all the inputs and events for the recording realm is simple.

We use a simplified *adore-ng* rootkit [1] as an example to illustrate this basic scheme. The C source code and the corresponding disassembly are shown in Figure 2. Briefly speaking, the orig-

```
1. int adore_root_filldir(void *buf, char *name,
      int nlen, loff_t off, ino_t ino, unsigned x)
2. {
3.   struct inode *inode = NULL;
4.   int r = 0;
5.   uid_t uid;
6.   gid_t gid;
7.
8.   if ((inode=iget(root_sb[current->pid% 1024],
                     ino)) == NULL)
9.     return 0;
     //lines 10 to 20 are omitted for brevity
21. }
```

(a) C source

```
d88888550 <adore_root_filldir>:
550:  push  %ebp
 551...56C: //set up stack,
              //%eax = current @L8
56C:  mov   0x6c(%eax),%edx  //%edx=pid @L8
56F:  xor   %edi,%edi
571:  test  %edx,%edx
573:  mov   %edx,%eax
575:  jns   57d <adore_root_filldir+0x2d>
577:  lea   0x3ff(%edx),%eax
57D:  push  $0x0
57F:  push  $0x0
581:  and   $0xfffffc00,%eax
586:  sub   %eax,%edx
588:  pushl 0x1c(%ebp) //push ino @L8
58B:  pushl x0(,%edx,4) //push root_sb[...] @L8
            // call iget @ line 8
592:  call  593 <adore_root_filldir+0x43>
   //The rest is omitted for brevity.
```

(b) disassembly

Figure 2. adore_root_filldir

inal pointer to `root_filldir` has been replaced by a pointer to `adore_root_filldir` to hide certain files. Suppose we want to record the execution of this kernel rootkit. So we move the code page of this kernel module from the main realm to the recording realm. We may treat this first code page as S_0. Figure 3 (A) illustrates this situation.

When the guest system is about to call `root_filldir`, the execution is redirected to `adore_root_filldir`, with virtual address $0xd88888550$ and physical address $0x16876550$. Since the physical page $0x16876000$ is not present in the main realm any more, this control flow transition will trigger a TDP page fault. The recorder located in the hypervisor will capture this TDP page fault and switch the memory space to the recording realm, which is shown in Figure 3 (B). In addition, we record the current CPU state (all the registers and flags), as input I_0.

On fetching the first instruction in `adore_root_filldir`, two more TDP page faults will be triggered for the page table directory page (PD) and the page table entry page (PTc) respectively. This is because the TLB has been flushed during the realm switch, and the CPU needs to look up the page table for the physical address of the first instruction. We will also move these two pages in the recording realm and record their contents as another input. This is a desired behavior, because the replayer will need the page table pages for address translation. Moreover, by including the page table pages, the problem of page swapping and re-mapping is automatically handled in both recorder and replayer. Figure 3 (C) shows this moment.

This first instruction (`push %ebp`) writes onto the stack. As the stack page is absent in the recording realm, this operation triggers the TDP page faults for the stack page (MS) and the corresponding page table pages (PD and PTs). as shown in Figure 3 (D). In this example, PD has already been moved into the recording realm, so no TDP page fault happens for PD.

Then the execution continues without causing any VMExits until $0xd888856c$. This instruction (`mov 0x6c(%eax), %edx`) reads a data page (D), which is not present in the recording realm. Similarly, this data page (D) and its corresponding page table page (PTd) are moved into the recording realm and recorded (see Figure 3 (E)).

The execution further proceeds to the instruction located at $0x169f7592$. It calls a kernel function `iget`. A TDP page fault is raised because the jump target is absent in the recording realm. By checking the faulting EIP, we determine that this EIP does not belong to the malware module. So we decide to switch back to the main realm (see Figure 3 (F)). In addition, we record a "JumpOut" event at this point, indicating that the execution has transferred out of the recording realm.

The kernel function `iget` now resumes its execution in the main realm. While it accesses the parameters, another TDP page fault will occur because the stack page (MS) has been moved to the recording realm. So we will move the stack page (MS) and the corresponding page table pages (PD and PTs) back to the main realm. This behavior is also desirable, because next time when the recording realm reads one of these pages, we are able to capture it and record the new page content as a new input. Figure 3 (G) illustrates this situation.

When `iget` finishes and returns, a TDP page fault will occur because the jump target is not present in the main realm. Thus, we switch the memory space to the recording realm, which is shown in Figure 3 (H). The CPU state is recorded and the execution resumes in the recording realm. The subsequent execution of `adore_root_filldir` will follow a similar cycle.

3.3 Other Inputs

The previous example only shows how to capture inputs from CPU states and memory. We need to handle other kinds of inputs as well. Control transitions such as interrupts and exceptions are naturally handled in the basic scheme. As the Interrupt Descriptor Table (IDT) is never present in the recording realm, any interrupt or exception will trigger a TDP page fault. We will treat this fault as a control transition and a new CPU state will be recorded when the execution returns to the recording realm. When executing in the recording realm, instructions like `cpuid`, `rdmsr`, `in` and `rdtsc` need to be recorded as inputs I and state changes Δ. With hardware virtualization support, we are able to trap these instructions into the hypervisor, and record their results when they are executed in the recording realm. DMA transfers may change memory pages in the recording realm without CPU intervention. When DMA writes into a page resident in the recording realm, we need to record that page as a new input. The DMA controller is emulated in software. So we can intercept this DMA write and record this input.

3.4 Optimizations

The basic scheme enforces two mutually exclusive realms. In many cases, this is unnecessarily expensive. If two realms read a shared page alternately, the basic scheme would repeatedly remove that page from the recording realm, and later move it back and record it, even though the page contents have not changed. We employ several optimizations to allow these two realms to share pages.

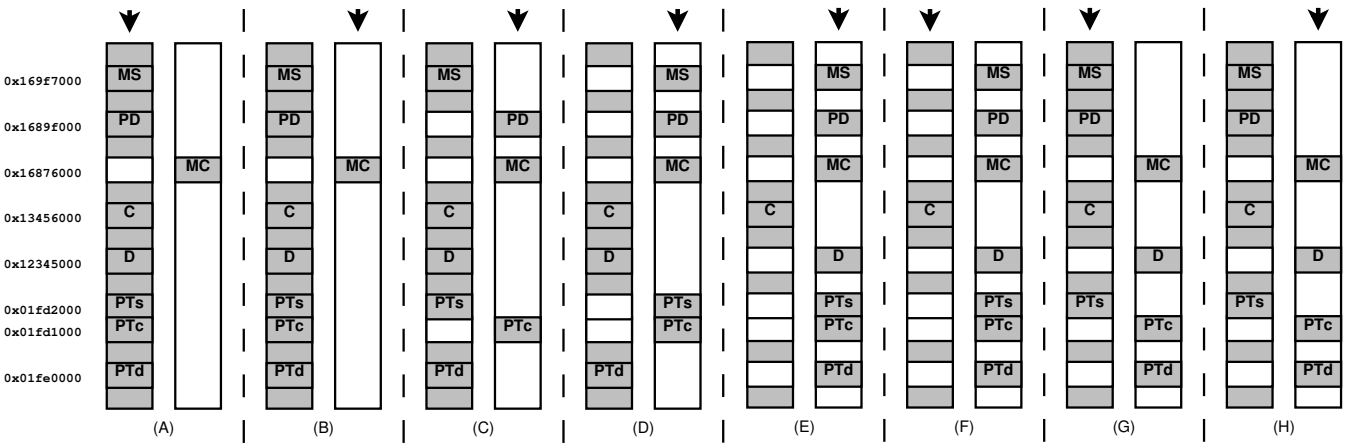

Figure 3. TDP snapshots for adore_root_fill. The two columns represent two guest physical memory spaces for the main realm and the recording realm respectively. A shaded block represents a present page, while a blank block indicates an absent page. The arrow on top signifies which realm is active.

Sharing Data Pages. To enable sharing data pages, we use a "Remove-On-Write" (ROW) principle, which is similar to Copy-On-Write. More specifically, we allow two realms to share pages, and these pages are set to be read-only. When one realm attempts to write to a page, a TDP write violation will be triggered, and that page will be removed from the other realm. This optimization works especially well for the page table pages, because both realms need these pages for address translation, and these pages do not change so often.

Sharing Code Pages. When recording a full process, we encounter a problem where both realms need to access shared library code pages. These code pages will be moved back and forth between the main realm and the recording realm, according to the basic scheme. Similar to sharing data pages, these code pages present in the both realms and are marked to be read-only. In addition, we manipulate the NX (Non-Execute) bit for these code pages, to capture the moment when the execution transitions into these code pages in the monitored process. More specifically, we monitor the context switch by intercepting CR3 writes. When the execution context switches to the recorded process, we switch from the main realm to the recording realm. However, there is a gap between this context switch and the user-level execution, because the context switch is performed in the kernel space and the execution will continue in the kernel space for a while before it transitions to the user space. In order to capture the entry point to the user space, we mark all the pages in the recording realm as Non-Executable. Although the kernel execution will trigger TDP page faults and in turn these pages will be loaded into the recording realm, we do not record these pages. We wait until we observe a TDP execute violation with the faulting EIP in the user space. Then we detect the entry point. The pages loaded during the kernel execution will be removed, so we can capture when the execution transfers back to the kernel later and pages marked NX will be restored. Fortunately, the kernel execution within this gap is normally very brief, so the overhead for capturing the entry point to the user space is very small.

3.5 Bridging the Semantic Gap

We usually specify which malware to monitor by its executable name, whereas the recording realm operates directly on guest physical pages. Therefore there exists a semantic gap. We leverage the VM introspection technique to bridge this semantic gap [17, 20]. More specifically, we intercept system calls and parse kernel data structures in the guest system to extract the OS-level semantics, such as the process list and the module memory map. By this way, we map the process name to the corresponding CR3, and module name to its virtual memory range. Then by looking up the guest page table, we further map the guest virtual address to the guest physical address.

The mapping from guest virtual to guest physical address may change over time due to page swapping. The newly mapped physical page will be captured and recorded when it is accessed later, but the physical page that is no longer mapped needs to be removed from the recording realm immediately. To do so, we need to capture the page table changes that affect the pages in the recording realm. According to our data page sharing mechanism, the page table pages associated to the recording realm are shared in both realms and set to be read-only. Therefore, any changes to these page table pages will be trapped to the hypervisor. Therefore, by checking which page table entry has been modified, we can determine which guest physical page needs to be removed from the recording realm.

3.6 Shadow Time Stamp Counter

As extra TDP page faults and other VMExits are needed for recording the malware execution, malware may detect the underlying recording behavior by examining the advance of the Time Stamp Counter (TSC). This can be done by `rdtsc` and reading the TSC model-specific register. We maintain a shadow TSC to hide this artifact.

The shadow TSC is an estimate of how much time the guest actually runs. We calculate it as follows: Let t_i be the value of the host TSC before VMRESUME[2] is executed. Let t_o be the value of the host TSC right after the CPU returns to the host. Let t_e and t_x be the time it takes to enter the guest and exit to the host and t_g be the actual execution time for the guest, then $t_o - t_i = t_e + t_x + t_g$. We approximate $t_e + t_x$ by turning on `rdtsc` and running `rdtsc` in a loop in the guest to obtain the average of $t_e + t_x$, which we used to calculate t_g. Then t_g is used to increment the shadow TSC, which is returned to the guest whenever the guest queries the TSC.

3.7 Implementation

We implemented the recording component in KVM in the Linux Kernel 2.6.32. The code base of KVM is well organized. While

[2] VMRESUME is the Intel instruction for entering the guest, non-root mode

`vmx.c` and `svm.c` contain the hardware specific code for Intel and AMD virtualization extensions respectively, `mmu.c` contains the memory management unit code that is common to both architectures. Within `mmu.c` is the `tdp_page_fault` function that is called by both VMX and SVM, and is where we implemented our realm control and enforcement logic. All memory based inputs are handled at this location. All non-memory based inputs to the *recording realm* are handled in the architecture specific implementation files.

Memory Management. For KVM, a virtual machine runs as a user process on the host. The guest physical memory is just virtual memory in that process. The guest physical pages can be swapped in and out, depending on the system workload. Therefore, the TDP table is rather dynamic. To tackle these dynamics, KVM registers *mmu_notifiers* with the Linux kernel. With these in place, any changes made to the page tables of that specific process, such as swapping a page or re-mapping a virtual page to another physical page, will notify KVM through *mmu_notifier*. As we manage two TDP tables, both tables need to be updated accordingly. Thus, we implemented our own versions of the *mmu_notifiers* so changes in the page table of the host process are reflected in the TDP page tables for both realms.

Logging. Being a kernel module, KVM cannot directly write to files. To enable logging, we implemented a user-level program that commits the changes to a log file. In the current implementation, when we load a page into the recording realm and record it, we record the full page content. Obviously, this is only necessary when the page is recorded for the first time. When the page is modified in the main realm and loaded back again, it is very likely that only a small portion of the page has changed. As a simple optimization, we may keep the old copy and only record the "diff". We did not implement this optimization. Instead, we simply zip the log. We found this simple solution sufficient in practice, because the compressed logs are usually small enough.

Event Landmark. We need a landmark to tell at what execution point a log event is recorded, such that we can replay the same log event at the same moment. Previous replay systems used the branch counter as a landmark. The branch counter increments when each branch instruction is committed and thus serves as an accurate landmark. In our current implementation, we simply used the CPU state (including the EIP, registers and flags) as the landmark. It is simple but might not be as accurate as the branch counter, because two execution points may happen to have the same CPU state. In practice, we found this simple solution good enough. As described later in Section 4, most of the events (such as memory accesses, regular control transitions, and special instructions) are replayed on demand, the role of the landmark is not as important as in the other replay systems.

4. Precise Replayer

We replay the execution using dynamic binary translation technique to provide good instrumentation support and analysis efficiency.

4.1 Dynamic Binary Translation

Briefly speaking, software emulation based on dynamic binary translation works as follows. When the emulator is about to run a block of code for the first time, it will translate that code block into a piece of code that can execute on the host and will store the translated code into the code cache. When the same code block needs to be emulated, the emulator can skip the translation procedure and directly fetch the translated code from the code cache and execute it. Special care is needed to emulate memory accesses. The *softmmu* is a software implementation of the Memory Management Unit, which looks up the page table and performs virtual-to-physical address translation. To speed up the address translation, the *software TLB* (Translation Look-aside Buffer) is implemented as a cache for the address translation results.

This design is mainly for improving emulation efficiency. However it deviates from the real execution in at least the following ways. First, a block-by-block translation procedure is introduced, which does not exist in the real processors. This translation procedure is normally invisible to the emulated execution, except when the block translation crosses the page boundary. This extra page access can be observed if it causes a page fault.

Second, for efficiency, dynamic binary translation often performs a lazy calculation of flags: a flag is calculated only when it is needed. For example, in "`cmpl $1, %eax; jz 0x401020;`", EFLAGS are not calculated on the first instruction, and then on the second instruction, only `ZF` is calculated to determine which branch to go. This lazy approach is good for efficiency but can be exploited to detect emulation.

Third, again for efficiency, interrupts are checked and served only at the block boundary. In contrast, on real hardware, interrupts may happen on any instruction. We have seen that realworld malware samples repeatedly measure the cpu cycles consumed by a simple `mov` instruction (using a sequence like "`rdtsc; mov %eax, %ebx; rdtsc;`") in a loop and only exit the loop when the number of cpu cycles is greater than a relatively big threshold to detect the presence of an interrupt in between.

In addition to the above discrepancies that are unique to dynamic binary translation, several more are common in software emulation. First of all, due to the complexity of x86 instruction set, some special-purpose instructions (e.g., System Management Mode and Trusted Execution Technology instructions) are hard to emulate and thus have not been implemented in software. We have seen that a malware sample executes `rsm` instructions and leads to a crash in QEMU. Moreover, accurate CPU timestamp emulation is nearly impossible. For simplicity, current CPU emulators (like QEMU) choose to fetch the time stamps on the host. It means that an instruction would consume much more CPU cycles in emulation than on the real hardware. Finally, the logic of checking and raising exceptions in the hardware is fairly complex and thus the software emulation for this logic is often error-prone.

4.2 Changes for Precise Replay

Considering all these challenges in software emulation and dynamic binary translation, we come up with several design changes in the work flow of software emulation to ensure precise replay. Particularly, we have modified the implementation of dynamic binary translation in QEMU to comply with these design changes.

New Translation Logic. According to Section 2.3, during dynamic translation, we classify the instructions into three categories: general-purpose, FPU, and others. General-purpose instructions include data transfer, control transfer, and integer arithmetic. They are translated according to their simple semantics. To avoid discrepancies in flag calculation, we disable the lazy flag calculation. That is, EFLAGS are immediately calculated after each instruction that changes EFLAGS. Assuming the control transition caused by exceptions be correctly replayed, the logic for checking and raising exceptions is completely removed, except for page fault. This is because we rely on the page fault logic to load memory pages at right moment.

Floating point and SIMD instructions execute on FPU. To ensure correctness, these instructions will be translated to wrapper functions that pass the operations directly to the real FPU. For example, in the software emulation approach, a floating point instruction `fadd %st1, %st0` would be translated to call a helper func-

tion `helper_fadd_ST0_STN`, in which this instruction is emulated in software. In our pass through approach, we will directly insert a piece of assembly code as: `__asm__("fadd %st(1), %st(0)")`. As this instruction takes two FPU registers. We can pass the same instruction to the FPU. If an instruction takes any operands from memory or the general-purpose registers, we need to copy the operands from the guest environment to the host or vice versa. For example, the instruction `fadds %0xc(%ebp)` adds a memory operand with `st(0)` and saves the result in `st(0)`. Since this memory operand is loaded in the guest system, we have to copy its value into a temporary value on the host and then performs the floating point operation natively on the host, as shown in the following code snippet:

```
unsigned long temp = ldl(A0);
__asm__("fadds %0;" : : "m" (float)temp));
```

These natively executed FPU instructions may raise exceptions. We register exception handlers to catch and ignore them. We expect to replay the exceptions from the execution log.

Other instructions will be translated into "nop", expecting that the results of these instructions be correctly replayed from the log. That is, no translated code will be generated except advancing the program counter to the next instruction.

While translating each instruction, we check and ensure the program counter should not across the page boundary. Otherwise, we will stop translating the current code block and leave the current instruction to the next block.

Replay Logic. As our page-level recording is based on TDP, we will need the same page table mechanism in software emulation to correctly replay log events on demand. We introduce a *physical page container* for this purpose. This physical page container indicates if a physical page has been loaded from the log and thus is present. Generally speaking, during replay the physical page container replicates the TDP page table of the recording realm. When the replayed execution accesses a page that is absent in the physical page container, we will load the missing memory page and update CPU states from the log at the right moment.

In addition, in the end of each instruction we will compare the current CPU state with the landmark of the next log event. If the landmark matches, we will replay this log event. This log event may be a control transition caused by interrupts or exceptions, or a state change made by special-purpose instructions.

4.3 Example Walkthrough

We use the same *adore-ng* example to walk through the replay logic. As the first log event, the code page MC is loaded in the physical page container. This initial state is the same as that of the right column in Figure 3(A).

Then the second event is the CPU state for the entry point of `adore_root_filldir`. The replayer updates the CPU state accordingly. The code block starting with the EIP *0xd8888550* needs to be translated and put into the translated code cache before executed. This translation triggers looking up the page table for the physical address of that EIP. Consequently, the page table pages (PD and PTm) are loaded from the log on demand, because they are not present in the physical page container.

Now we start to execute the translated code. The first instruction pushes onto the stack. At this moment, we will load the page table page PTs during page table lookup and then the stack page MS for the memory write. Similarly, we will load the page table page PTd and the data page D at the right moments.

At the end of the call instruction at $0xd8888592$, which is about to jump to the kernel function `iget`, we know that a control transition happens at this moment. This log event is followed by several events for removing pages (MS, PD, PTs). Subsequently, we see an event to update the CPU state. After executing all these log events, we skip the execution of `iget` and directly arrive at the instruction $0xd888885a7$, where `iget` returns to. Now we arrive at the same state as the right column of Figure 3 (H). As this point, we also need to flush the TLB, because changes may have been made to the page table during the skipped execution.

It is worth noting that the replayer literally replays the log. As this example describes the basic scheme, we can see that quite a few pages (such as page table pages) are removed and then loaded back later. Given a log recorded using the optimizations discussed in Section 3.4, the replay will proceed more efficiently.

4.4 Implementation

We implemented the replayer on TEMU, a dynamic analysis platform in the Bitblaze binary analysis infrastructure [37]. A handful of plugins have been developed on TEMU to perform malware analysis. TEMU is based on QEMU version 0.9.1. So we modified the existing dynamic binary translation code in QEMU to support precise replay.

With the modifications in TEMU, the existing analysis plugins should work automatically, except for a small change. Each regular TEMU plugin needs to check if the current execution is within the context of interest (e.g., if the current process is the malware's process). However, for a plugin that works with replay, all execution is of interest. This is ensured by the recorder. So we can remove this context checking in the plugin. In particular, we modified two plugins. The first plugin is an unpacker, which is the implementation of Renovo [21]. The second is an instruction tracing tool called tracecap, which performs taint analysis and dumps detailed information for each instruction. Having a detailed instruction trace with taint information has been demonstrated to be crucial in many malware analysis projects [7–9, 22].

5. Evaluation

We conducted a systematic experimental study for V2E from the following aspects: 1) we studied the existing emulation detection methods and examined the effectiveness of V2E against these methods; 2) we conducted a measurement study on the severity and prevalence of emulation-resistant malware; 3) we evaluated V2E's capabilities in tackling real-world emulation-resistant malware samples; and 4) we measured its performance impact in various settings.

Experimental setup. The host machine has a Core i7 860 Quad Core processor with 4 GB of memory running Ubuntu 10.04, with our modified 2.6.32.29 kernel. The guest systems are Windows XP SP2, Ubuntu 9.04 and Redhat 7.

5.1 Study of Existing Anti-emulation Attacks

We collected a list of existing anti-emulation methods that are available in the literature [16, 22, 31]. To evaluate if V2E can defeat these techniques, we tested both synthetic and realworld samples whenever available. For those techniques without samples, we speculated their internal mechanisms and then made careful decisions. Descriptions of these methods are listed in the Table 2.

Among these detection methods, six of them, including "cmpxch8b", "double fault", "reserved MSR", "instruction length", "alignment checking" and "invalid opcode", cause aberrant exception behavior. As V2E accurately replays the exceptions from the real hardware, these detection methods will be automatically taken care of. We verified *instruction length* and *invalid opcode* using synthetic samples.

Then, "cpuid", "MSR fingerprinting" and "rdtsc" have deviated values returned from certain instructions. These values are hardware specific and thus hard to emulate correctly. V2E defeats these

Description	Defeated?
`cpuid` returns processor specific information. QEMU returns generic information. [16]	✓
`rdtsc` returns the contents of the TSC. [16, 22, 31]	✓
`cmpxch8b` conditionally writes to the memory operand, but a #GP exception is always generated if the memory operand is not writable. QEMU only raises #GP when the memory is written indeed. [16]	✓
A *double fault* exception is generated if the #GP handler can't be retrieved from the IDT when a #GP occurs. QEMU generates #GPs repeatedly. [16]	✓
Writing to *reserved MSRs* should generate #GP, but QEMU does not. [31]	✓
A #GP is generated if the *instruction length* is more than 15 bytes, but certain prefixes like `rep` can break this rule in QEMU. [22, 31]	✓
If the *alignment check* flag is set, then an alignment exception is thrown when an unaligned memory address is accessed. This exception is not thrown in QEMU. [31]	✓
CPU bugs or *errata* results in behavior that are either unexpected or does not follow the published logic, thus the emulator does not handle these like specific hardware would. [31]	✓*
MSR fingerprinting uses the `rdmsr_safe` macro to find the MSRs supported by the CPU. The list of MSRs can be used to help determine the CPU model. [31]	✓
The *fnstcw* instruction pushes the FPU Control Word register onto the stack. As it turns out, bit 3 of this register is reserved in Intel's implementation but is always 1. QEMU always returns 0. [22].	✓
`icebp` is an *undocumented instruction* that simply raises an exception. In QEMU this instruction hangs the emulator. [22]	✓
`rep stosb` can be used to overwrite a range of memory with `nop`. When paired with a `jmp` to nowhere, QEMU and singlestepping will throw a segmentation fault. Hardware executes successfully. [16]	N/A

Table 2. Survey of Emulation Detection Techniques.

methods by recording the values from the real hardware. We further verified "cpuid" and "rdtsc" using both synthetic and realworld malware samples.

"fnstcw" causes a deviated state in the FPU. V2E passes FPU/MMX/SIMD instructions directly on hardware, so no deviations in FPU would be possible for V2E. We have verified using several realworld samples, thus we are confident that it is defeated.

The "CPU errata" method needs special considerations. If a CPU bug causes a totally unpredictable result, it would be extremely hard to handle. The CPU bugs used for emulation detection in the literature [31] all cause incorrect exceptions. These CPU bugs can be handled correctly by V2E, because exceptions are recorded and replayed. In any sense, the effectiveness of this detection method is very limited, because a CPU bug is specific to a CPU family.

The "rep stosb" detection method exploits a cache coherency bug for self-modifying code in earlier Intel processors. This bug has been fixed in all current Intel processors. Therefore, this method is no longer relevant.

5.2 Malware on Existing Malware Analysis Platforms

We would like to see how well the existing malware analysis platforms handle realworld malware. So we collected 150 realworld malware samples from a live malware repository (`http://malc0de.com/database`) and security researchers, and tested them on three malware analysis platforms: Anubis [2], CWSandbox [11], and TEMU [37]. While Anubis and TEMU are based on software emulation, CWSandbox uses API hooking technique. We found that out of 150 samples, 51, 88, and 14 crashed or exhibited no behaviors in Anubis, CWSandbox, and TEMU respectively. Note that all these samples run properly in KVM, which means that they intended to escape from either of these analysis platforms. Interestingly enough, 14 samples that are resistant to TEMU also escaped Anubis and CWSandbox. Evidently, emulation-resistant malware has already become a prevalent threat.

5.3 Analyzing Realworld Malware with V2E

To evaluate how well V2E handles realworld malware, we ran the these 14 emulation-resistant samples with V2E. We ran each sample for up to 2 minutes and then stopped recording. For each sample, we configured V2E to record the entire user-level process and spawned child processes if any. We chose the time-out threshold of 2 minutes to be consistent with the settings of Anubis and CWSandbox. We did not observe any sample installing kernel rootkits, but if indeed a kernel module is installed, the recorder can be configured to record the execution of that kernel module as well.

V2E was able to record and replay the malicious behaviors for all these samples. During replay, we tested three settings: 1) replay with no plugin provides a baseline for the replay performance; 2) replay with tracing produces a complete and detailed instruction trace for the recorded execution; and 3) replay with unpacking extracts hidden code and data from the packed malware. A summary of the results is presented in Table 3. For each sample, we list its MD5 hash, the executable size, and the size of the recorded execution log. Then the runtime for replay with no plugin is listed. With regards to tracing, we list the instruction count and the runtime for tracing. As for unpacking, we show the number of memory dump files, and the runtime for unpacking.

From Table 3, we can make the following observations. First, the execution logs (after compression) are fairly small (up to 55MB). It is worth noting that unlike the logs in the other execution replay systems, these logs are self contained with all necessary code and data included. We can directly feed these logs into the replayer for in-depth malware investigation, and no other environment setup (e.g., virtual machine images and configurations) is needed. Second, due to the efficiency of dynamic binary translation, the baseline performance of the replayer (with no plugin) is satisfactory, from less than 1 second to 79 seconds. The very short replay runtime (less than 1 second) on some samples indicates that these samples are mostly idle. This is reasonable because many of the samples are bots and the networking is disabled during recording. Note that some samples are very computation-intensive, with over 1.3 billion instructions executed within 2 minutes. Third, with good instrumentation support (especially the support for shadow

			Null	Tracing		Unpacking	
MD5SUM	**exe sz**	**log sz**	**runtime**	**runtime**	**# ins**	**runtime**	**dumps**
27eb815f101a9295fbb601986f393d01	105KB	29.07MB	76.76s	2h19m	1347M	96.5s	78
43de1618764daf7e5887bd8ac9cadb52	105KB	28.46MB	76.69s	2h18m	1346M	96.09s	79
03f322365b844d8faf9236aab34b4214	106KB	30.75MB	77.09s	2h19m	1349M	97.02s	79
4f12dfb4b613abc4ddf56d087223a868	115KB	35.93MB	78.87s	2h20m	1366M	98.8s	57
f01cdf6e5052aeb5c6510bd8f8d88636	103KB	29.95MB	77.21s	2h19m	1348M	98.94s	81
f068b4362c646dae42cc3b1b8fe20c12	110KB	30.13MB	77.2s	2h19m	1350M	97.17s	77
1686739bc81a407dd9944e2d9bbcf2e1	23KB	2.44MB	0.65s	46.8s	7.73M	0.71s	8
0b8b2c0926630c69a6c75bba67b24a3e	39KB	3.25MB	0.97s	99.7s	16.8M	1.2s	55
c5ff7232868333107fa3efe895f12361	245KB	55.36MB	29s	27m15s	248M	39.33s	39
36e5fdcdbe0bcdc59ea001b162bfb97d	243KB	37.48MB	20.91s	1150s	175M	30.56s	22
c1a66699820fdeb7242e884e6d2f8bcb	119KB	676KB	0.57s	55.7s	9.55M	0.66s	10
dabec78d489f1e783fb23d6e726bd1a4	108KB	2.00MB	0.19s	23.2s	4.09M	0.22s	1
ef0458e196fbd1b4cc1613ba2ca3c43b	280KB	3.30MB	0.36s	54.8s	9.51M	0.46s	1
7ce6cd9837e1a7837c2b491c21ff5b69	101KB	7.10MB	34.35s	294.4s	43M	35.4s	30

Table 3. Analyzing Realworld Emulation-Resistant Malware with V2E

memory), the unpacker built on top of the replayer demonstrated good efficiency. It was able to finish replaying 2-minute execution logs in up to 99 seconds, and at the same time successfully extract hidden code and data from the packed malware samples. Interestingly enough, all these samples are packed. Without V2E's support, it would not be possible to unpack them successfully. Finally, tracing is substantially more heavyweight than unpacking, because it has to disassemble each instruction, fetch instruction raw bytes and operands, and write these details into the instruction trace. We have obtained complete instruction traces for all the samples within a reasonably short period (from tens of seconds to a couple of hours).

5.4 Performance

Without ground knowledge about the realworld malware, it is difficult to accurately measure the performance of V2E. Instead, we choose to measure its performance in controlled experiments.

Recording adore-ng. This experiment shows how well V2E analyzes kernel rootkits. We installed the adore-ng rootkit in Redhat 7 and exercised it by decompressing the linux source with about 17,000 files. This workload took 3s without recording and 52s when recording was enabled, generating a 14MB execution log. A roughly 17x slowdown seems high, but is reasonable for this workload with frequent context switch between the rootkit and the rest of the kernel.

Recording Internet Explorer. We use this test to show how well V2E performs on analyzing a highly complex, computation-intensive, and interactive application. In this experiment, we measured the load time of IE with and without recording. Without recording, IE started up and loaded the MSN homepage in 2.5s. With recording, it took 13.8s. That is about a 5x slowdown. A 52MB execution log was generated. We expect that the recording performance impact decrease as the IE continues to run, because more pages would remain stable in the recording realm. While recording, the IE was very responsive, so it should not be a problem to record any user interactions. In general, as we design V2E in favor of the flexibility of page-level recording and the self-containment of execution logs, its recording performance is likely less efficient than other recording systems. We believe this trade-off is reasonable for malware investigation.

Comparing with Single Stepping. We then enabled single stepping on KVM and measured the performance overhead of single stepping by executing a loop with 8 million instructions. On KVM, it took approximately .008s vs. 25s when single stepping was disabled and enabled respectively. That is more than 3000x slowdown. We can anticipate the performance of obtaining a complete instruction trace based on single-stepping to be much worse. By contrast, using V2E, the same 8 million instructions were recorded with negligible performance penalty. The baseline replay runtime was 0.3s, and it took only 0.8s to perform unpacking analysis and 48s to obtain the complete instruction trace.

6. Discussion

We discuss the limitations and potential evasion techniques in this section.

Bugs in Common Instructions. To achieve transparency, we assume that common instructions be emulated correctly. This assumption does not necessarily hold in the current implementation of V2E. If malware exploits one of the emulation bugs in the common instructions, V2E cannot successfully replay the malware execution. Once the replay failure is found, we can identify the bug and fix its emulation code in V2E. Over time, V2E will be improved to be increasingly more transparent.

Attacking the Landmarks. As mentioned in Section 3.7, the landmark mechanism is not perfect because the CPU state is not a unique identifer of execution point. A malware author may take advantage of this limitation to force an imprecise replay. We leave it as future work to resolve this issue by implementing a branch counter landmark.

Multi-core Support. The current implementation of V2E only supports a single-core guest environment. Since we use two-dimensional paging for separating the main and recording realms and each virtualized core has its own TDP table, recording malware's execution on multi-core environment is feasible by design. We leave the multi-core support as future work as well.

Denial-of-Service Attack. It is feasible for the malware to induce a large number of exits (e.g., TDP page faults and exceptions) to the hypervisor, so as to launch a denial-of-service attack to the recorder. In addition, the malware could detect the analysis environment by measuring this slowdown using an external clock. In general, this kind of limitation is not unique to V2E. It is also shared by other platforms (like Ether). The analysts will have to analysts make case-by-case solutions once they actually arise.

7. Related Work

Malware Analysis Platform. Here we compare different analysis platforms with respect to the degree of transparency, the breadth of analysis view, the support for fine-grained instrumentation, and fine-grained analysis efficiency.

DynamoRIO [6], Pin [25], and Valgrind [28] provide convenient and efficient support for fine-grained instrumentation on a user-level program, by performing dynamic binary translation. However, as they can only instrument a single user-level process, they cannot analyze kernel malware. TEMU [37] is an analysis platform based on QEMU [5], which is an efficient CPU emulator by performing dynamic binary translation. With a whole-system view, TEMU can analyze both user-level and kernel-mode malware. Cobra [38] is a malware analysis platform in form of a Windows kernel module. It uses a technique called localized execution to instrument and inspect malware's behavior. The localized execution technique is in spirit similar to dynamic translation techniques. Since Cobra resides in the kernel, it also has a whole-system view. However, it is unclear if Cobra has support for instruction-level instrumentation. The downside for all the above systems that are based on dynamic binary translation is lack of transparency. Incapable of correctly emulating all the complexities of a real CISC computer system, these analysis systems can be easily detected [16, 26, 31]. By design, Cobra is able to defeat some known anti-analysis techniques. However, Cobra does not address the transparency problem in a general context.

Ether [13] makes use of hardware virtualization techniques to observe malware's execution in a stealthy manner. However, Ether is not an ideal platform for in-depth malware analysis, which requires instruction-level instrumentation. Although fine-grained instrumentation can be achieved through single-step mode, its significant performance overhead (thousands of times slowdown) is unacceptable for the context of in-depth malware analysis.

V2E outperforms all these analysis platforms by combining the advantages of both hardware virtualization and dynamic binary translation techniques, and thus can achieve the highest standard in all the four metrics.

Emulator Detection and Countermeasures. Researchers have studied the problem of detecting emulators extensively [16, 31] and identified numerous detection methods. We have carefully examined these methods, and confirmed that V2E is able to defeat them effectively.

To address the anti-emulation problem, several techniques have been proposed. Balzarotti et al. proposed an automatic method to detect the emulation-resistant behavior by comparing if a piece of malware behaves differently on the real hardware than in the emulated environment [3]. This technique is effective as a detection tool, but it does not help defeat this anti-emulation problem. In order to successfully emulate these emulation-resistant malware, Kang et al. proposed a differential analysis method by comparing two execution traces, one from Ether and the other one from QEMU [22]. By performing trace alignment, this technique is able to automatically detect the root cause for the divergence and generate a runtime patch. However, this approach is not scalable if malware has many anti-emulation checks in many different paths (which is often true in practice), because one runtime patch can only apply to one point in one execution path. In comparison, V2E takes a more radical approach to defeat this problem. With the assumption that the emulation for common instructions should be correct, it should be able to analyze all emulation-resistant malware correctly. This assumption may not be entirely true, but once a bug is found for one of these common instructions, we should be able to fix it easily.

In fact, EmuFuzzer can help discover emulation bugs by fuzz testing[26]. According to the results, a large number of deviations are found for exceptions and floating point operations for QEMU. As V2E directly replays exceptions and executes floating point instructions natively, these deviations are irrelevant. Most of deviations in CPU flags would also disappear, as V2E disables the optimization for flags calculation. The rest deviations (37 in total) are for CPU general registers and memory states, some of which may indeed be bugs in the common instructions. If so, we should fix them and the transparency of V2E would be further improved.

Execution Replay. Building a reliable and efficient execution replay system is a challenging task, especially in the malicious context. Some replay systems record and replay the program execution of a single user-level process [18, 19, 29, 33, 36]. So they cannot monitor the malicious activities happening in the kernel (e.g., kernel rootkits). Moreover, completely intercepting and recording all the inputs taken by a user-level process is also difficult. Other than system calls, a user-level program may also take inputs from CPU timestamps, and the memory regions shared with other processes and the kernel, etc. Most of the above systems cannot completely capture all these inputs, and thus they are only partial solutions. To solve this problem, Jockey [33] makes use of binary rewriting technique. The program to be recorded needs to be disassembled, and the problematic instructions are replaced with functionally equivalent functions that can be intercepted. This approach becomes ineffective when dealing with malicious code, because it is commonly known that correctly disassembling malicious code is an open problem.

Revirt [14], VMware [39], and Aftersight [10] record and replay the execution of an entire virtual machine. Thus, they can observe malicious activities in both user and kernel space. V2E shares some similarity with Aftersight in terms of heterogeneous replay. Aftersight records a whole virtual machine execution from VMware and replays it in QEMU, enabling heavyweight analyses (e.g., kernel bug detection) on production workload [10]. Unlike Aftersight, V2E is specially tailored for malware investigation. Instead of monitoring the entire virtual machine, V2E takes a targeted approach: it directly monitors malware's execution, and can potentially achieve higher efficiency. More importantly, V2E is able to record malware's execution in a transparent fashion and then during replay faithfully emulate the execution for heavyweight analyses.

8. Conclusion

In this paper we presented V2E, a new malware analysis platform that is both transparent and extensible. It first records an execution log for a given malware sample under hardware virtualization to achieve transparency goal, then it replays the execution using dynamic binary translation to facilitate custom code instrumentation and in-depth investigation. The key challenge is to ensure that this replay process is precise. We conducted well-rounded experiments by analyzing a large number of realworld emulation-resistant malware samples and comparing with other malware analysis platforms. While these emulation-resistant malware samples are able to evade the other analysis platforms, the successful analysis of these samples in V2E demonstrated the efficacy and efficiency of our approach.

Acknowledgements

We thank the anonymous reviewers for their insightful comments for improving this paper. This work is supported in part by the US National Science Foundation NSF under Grants #1018217 and #1054605. Any opinions, findings, and conclusions made in this material are those of the authors and do not necessarily reflect the views of the NSF or the Air Force Research Laboratory.

References

[1] adore-ng. adore-ng rootkit. `http://stealth.openwall.net/rootkits/`.

[2] anubis. Anubis: Analyzing Unknown Binaries. `http://anubis.iseclab.org/`.

[3] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient Detection of Split Personalities in Malware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2010.

[4] U. Bayer, P. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS 2009)*, 2009.

[5] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, April 2005.

[6] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization (CGO'03)*, March 2003.

[7] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM Conference on Computer and Communication Security (CCS'07)*, October 2007.

[8] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and Communication Security (CCS'09)*, Chicago, IL, Nov. 2009.

[9] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, San Diego, CA, Feb. 2010.

[10] J. Chow, T. Garfinkel, and P. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of 2008 Usenix Annual Technical Conference (ATC'08)*, June 2008.

[11] cwsandbox. CWSandbox::Behavior-based Malware Analysis. `http://mwanalysis.org/`.

[12] C. da Wang and S. Ju. The dilemma of covert channels searching. In *Information Security and Cryptology (ICISC'05)*, pages 169–174, 2005.

[13] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 51–62, 2008.

[14] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th symposium on Operating Systems Design and Implementation (OSDI'02)*, December 2002.

[15] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic Spyware Analysis. In *Proceedings of the 2007 Usenix Annual Technical Conference (ATC'07)*, June 2007.

[16] P. Ferrie. Attacks on virtual machine emulators. Symantec Security Response, December 2006.

[17] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of Network and Distributed Systems Security Symposium (NDSS'03)*, February 2003.

[18] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *Proceedings of the 2006 USENIX Annual Technical Conference (ATC'06)*, pages 27–27, 2006.

[19] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI'08)*, pages 193–208, 2008.

[20] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS'07)*, October 2007.

[21] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 5th ACM Workshop on Recurring Malcode (WORM'07)*, Oct. 2007.

[22] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song. Emulating emulation-resistant malware. In *Proceedings of the 2nd Workshop on Virtual Machine Security (VMSec'09)*, November 2009.

[23] kvm. Kernel Based Virtual Machine. `http://www.linux-kvm.org/`.

[24] A. Lanzi, M. Sharif, and W. Lee. K-Tracer: A system for extracting kernel malware behavior. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS'09)*, February 2009.

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'05)*, june 2005.

[26] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing cpu emulators. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09)*, pages 261–272, 2009.

[27] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy(Oakland'07)*, May 2007.

[28] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'07)*, pages 89–100, 2007.

[29] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*, Mar. 2009.

[30] qemulink. Qemu. `http://fabrice.bellard.free.fr/qemu/`.

[31] T. Raffetseder, C. Krügel, and E. Kirda. Detecting system emulators. In *the 10th Information Security Conference (ISC'07)*, pages 1–18, October 2007.

[32] R. Riley, X. Jiang, and D. Xu. Multi-aspect profiling of kernel rootkit behavior. In *Proceedings of the fourth ACM european conference on Computer systems (EuroSys'09)*, 2009.

[33] Y. Saito. Jockey: a user-space library for record-replay debugging. In *Proceedings of the sixth International Symposium on Automated Analysis-driven Debugging (AADEBUG'05)*, pages 69–76, 2005.

[34] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (Oakland'09)*, pages 94–109, 2009.

[35] M. Siper. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[36] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Annual Technical Conference (ATC'04)*, June 2004.

[37] temu. TEMU: The BitBlaze dynamic analysis component. `http://bitblaze.cs.berkeley.edu/temu.html`.

[38] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained malware analysis using stealth localized-executions. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (Oakland'06)*, pages 264–279, 2006.

[39] vmware. Vmware. `http://www.vmware.com/`.

[40] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communication Security (CCS'07)*, October 2007.

[41] H. Yin, Z. Liang, and D. Song. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.