USENIX Association

Proceedings of the 5th Symposium on Operating Systems Design and Implementation

Boston, Massachusetts, USA
December 9–11, 2002

ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay

George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, Peter M. Chen

Department of Electrical Engineering and Computer Science
University of Michigan
covirt@umich.edu, http://www.eecs.umich.edu/CoVirt

Abstract

Current system loggers have two problems: they depend on the integrity of the operating system being logged, and they do not save sufficient information to replay and analyze attacks that include any non-deterministic events. ReVirt removes the dependency on the target operating system by moving it into a virtual machine and logging below the virtual machine. This allows ReVirt to replay the system's execution before, during, and after an intruder compromises the system, even if the intruder replaces the target operating system. ReVirt logs enough information to replay a long-term execution of the virtual machine instruction-by-instruction. This enables it to provide arbitrarily detailed observations about what transpired on the system, even in the presence of non-deterministic attacks and executions. ReVirt adds reasonable time and space overhead. Overheads due to virtualization are imperceptible for interactive use and CPU-bound workloads, and 13-58% for kernel-intensive workloads. Logging adds 0-8% overhead, and logging traffic for our workloads can be stored on a single disk for several months.

1. Introduction

Improving the security of today's computer systems is an urgent and difficult problem. The complexity and rapid rate of change in current software systems prevents developers from verifying or auditing their code thoroughly enough to eliminate vulnerabilities. As a result, even the most diligent system administrators have to cope routinely with computer break-ins. This situation is likely to continue for the foreseeable future—statistics from the CERT® Coordination Center show a steady increase over the past 4 years in the number of incidents handled, the number of vulnerabilities reported, and the number of advisories posted [CER02].

The infeasibility of preventing computer compromises makes it vital to analyze attacks after they occur. Post-attack analysis is used to understand an attack, fix the vulnerability that allowed the compromise, and repair any damage caused by the intruder. Most computer systems try to enable this type of analysis by logging various events [Anderson80]. A typical Unix installation may record login attempts, mail processing events, TCP connection requests, file system mount requests, and commands issued by the superuser. Windows 2000 can record login/logoff events, file accesses, process start/exit events, security policy changes, and restart/shutdown events. Unfortunately, the audit logs provided by current systems fall short in two ways of what is needed: integrity and completeness.

Current system loggers lack *integrity* because they assume the operating system kernel is trustworthy; hence they are ineffective against attackers who compromise the operating system. One way current loggers trust the operating system is by keeping their logs on the local file system; this allows attackers who compromise the kernel to hide their activities by deleting past log records [CER01a]. Even if the existing log files are kept safely on another computer or on write-once media, attackers can forge misleading log records or prevent useful log records from being saved after they compromise the operating system. The absence of useful log records after the point of compromise makes it very difficult to assess and fix the damage incurred in the attack. It is ironic that current loggers work best when the kernel is not compromised, since audit logs are intended to be used when the system has been compromised!

Villains can attack kernels in many ways. The easiest way is to leverage the capabilities that the kernel provides to the superuser account. An attacker who has gained superuser privileges can change the kernel by writing to the physical memory through a special device (/dev/mem on Unix), by inserting a dynamically loaded kernel module, or by overwriting the boot sector or kernel image on disk. If an administrator has turned off these capabilities, an attacker can instead exploit a bug in the kernel itself. Kernels are large and complex and so tend to contain many bugs. In fact, a recent study used an automated tool to find over 100 security vulnerabilities in Linux and OpenBSD [Ashcraft02].

Current system loggers also lack *completeness* because they do not log sufficient information to recreate or understand all attacks. Typical loggers save only a few types of system events, and these events are often insufficient to determine with certainty how the break-in occurred or what damage was inflicted after the break-in. Instead, the administrator is left to guess what might have happened, and this is a painful and uncertain task. The attack analysis published by the Honeynet project typifies this uncertainty by containing numerous phrases such as “may indicate the method”, “it seems reasonable to assume”, “appears to”, “likely edited”, “presumably to”, and “not clear what service was used” [Hon00].

More secure installations may log all inputs into the system, such as network activity or keyboard input. However, even such extensive logging does not enable an administrator to re-create attacks that involve *non-deterministic* effects. Many attacks exploit the unintended consequences of non-determinism (e.g. time-of-check to time-of-use race conditions [Bishop96])—recent advisories have described non-deterministic exploits in the Linux kernel, Microsoft Java VM, FreeBSD, NetBSD, kerberos, ssh, Tripwire, KDE, and Windows Media Services. Furthermore, the effects of non-deterministic events tend to propagate, so it becomes impossible to re-create or analyze a large class of events without replaying all earlier events deterministically. Encryption is a good example of this: encryption algorithms use non-deterministic events to generate entropy when choosing cryptographic keys, and all future communication depends on the value of the these keys. Without logging non-deterministic events, encrypted communication can be decrypted only if the attacker forgets to delete the key.

The goal of ReVirt is to solve the two problems with current audit logging. To improve the integrity of the logger, ReVirt encapsulates the target system (both operating system and applications) inside a virtual machine, then places the logging software beneath this virtual machine. Running the logger in a different domain than the target system protects the logger from a compromised application or operating system. ReVirt continues to log the actions of intruders even if they replace the target boot block or the target kernel.

To improve the completeness of the logger, ReVirt adapts techniques used in fault-tolerance for primary-backup recovery [Elnozahy02], such as checkpointing, logging, and roll-forward recovery. ReVirt is able to replay the complete, instruction-by-instruction execution of the virtual machine, even if that execution depends on non-deterministic events such as interrupts and user input. An administrator can use this type of replay to answer arbitrarily detailed questions about what transpired before, during, and after an attack.

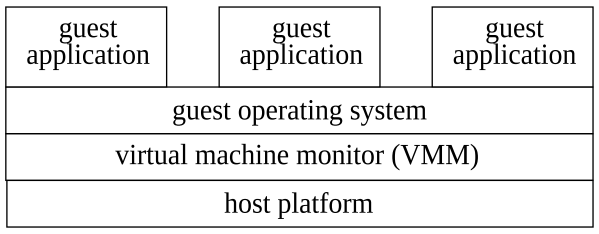

Figure 1: Virtual-machine structure.

2. Virtual machines

A virtual-machine monitor (VMM) is a layer of software that emulates faithfully the hardware of a complete computer system (Figure 1) [Goldberg74]. The abstraction created by the virtual machine monitor is called a virtual machine. The hardware emulated by the VMM is very similar (often identical) to the hardware on which the VMM is running, so the same operating systems and applications that run on the physical machine can run on the virtual machine. The host platform that the VMM runs on can be another operating system (the host operating system) or the bare hardware. The operating system running in the virtual machine is called the guest operating system to distinguish it from the host operating system running on the bare hardware. The applications running on top of the guest operating system are called guest applications to distinguish them from applications running on the host operating system (of which the VMM is one). The VMM runs in a separate domain from the guest operating system and applications; for example, the VMM may run in kernel mode and the guest software may run in user mode.

Our research group (CoVirt) is interested in enhancing security by running the target operating system and all target services inside a virtual machine (making them guest operating system and applications), then adding security services in the VMM or host platform [Chen01].

Of course, even the VMM may be subject to security breaches. Fortunately, the VMM makes a much better trusted computing base than the guest operating system, due to its narrow interface and small size. The interface provided by the VMM is identical or similar to the physical hardware (CPU, memory, disks, network card, monitor, keyboard, mouse), whereas the interface provided by a typical operating system is much richer (processes, virtual memory, files, sockets, GUIs). The narrow VMM interface restricts the actions of an attacker. In addition, the simpler abstractions provided by a VMM lead to a code size that is several orders of magnitude smaller than a typical operating system, and this smaller code size makes it easier to verify the VMM. As we will see, the narrow interface of the VMM also makes it easier to log and replay.

Virtual machines can be classified by how similar they are to the host hardware. At one extreme, traditional virtual machines such as IBM's VM/370 [Goldberg74] and VMware [Sugerman01] export an interface that is backward compatible with the host hardware (the interface is either identical or slightly extended). Operating systems and applications that were intended to run on the host platform can run on these VMMs without change. At the other extreme, language-level virtual machines like the Java VM export an interface that is completely different from the host hardware. These VMMs can run only operating systems and applications written specifically for them.

Other virtual machines such as the VAX VMM security kernel [Karger91] fall somewhere in the middle—they export an interface that is similar but not identical to the host hardware [Bellino73]. These types of VMMs typically deviate from the host hardware interface when interacting with peripherals. Virtualizing the register interface to peripherals controllers is difficult and time consuming, so many virtual machines provide higher-level methods to invoke I/O. A guest operating system must be ported to run on these VMMs. Specifically, the device drivers in the guest kernel must use the higher-level methods in the VMM; e.g. a disk device driver might use the host system calls `read` and `write` to access the virtual hard disk. The work required to port a guest operating system to these types of VMMs is similar to that done by device manufacturers who write drivers for their devices.

3. UMLinux

ReVirt uses a virtual machine called UMLinux [Buchacker01].[1] UMLinux falls in the last category of virtual machines; the VMM in UMLinux exports an interface that is similar but not identical to the host hardware. The version of UMLinux described and used in

1. Note that UMLinux is different from the similarly-named User-Mode Linux [Dike00].

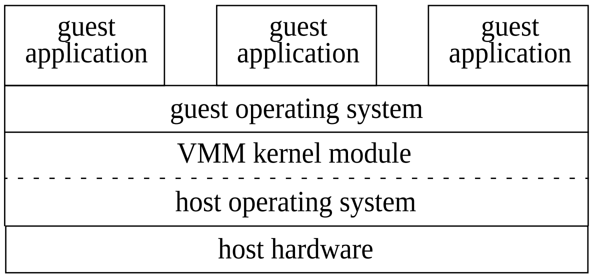

Figure 2: UMLinux OS-on-OS structure. Our version of UMLinux is implemented as a loadable kernel module in the host operating system. The device and interrupt drivers in the guest operating system use host services such as system calls and signals.

this paper is modified from code developed by researchers at the University of Erlangen-Nürnberg. Our version of the UMLinux VMM uses custom optimizations in the underlying operating system to achieve an order of magnitude speedup over the original UMLinux [King02].

3.1. UMLinux structure and operation

The virtual machine in UMLinux runs as a user process on the host. Both the guest operating system and all guest applications run inside this single host process (the virtual-machine process). The guest operating system in UMLinux runs on top of the host operating system and uses host services (e.g. system calls and signals) as the interface to peripheral devices (Figure 2). We call this virtualization strategy *OS-on-OS*, and we call the normal structure where target applications run directly on the host operating system *direct-on-host*. The guest operating system used in this paper is Linux 2.4.18, and the host operating system is also Linux 2.4.18.[2]

The VMM in our version of UMLinux is implemented as a loadable module in the host Linux kernel, plus some hooks in the kernel that invoke our VMM module. The VMM module is called before and after each signal and system call to/from the virtual-machine process.

Most instructions executed within the virtual machine execute directly on the host CPU. Memory accesses are translated by the host's MMU based on

2. The guest and host operating systems can also be different. We use the same operating system for guest and host to enable a more direct comparison between running applications on the UMLinux guest and running applications directly on the host.

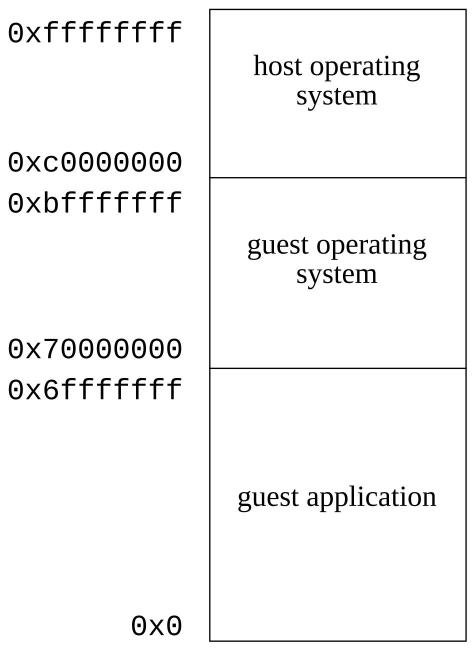

Figure 3: UMLinux address space. As with all Linux processes, the host kernel address space occupies [0xc0000000, 0xffffffff], and the host user address space occupies [0x0, 0xc0000000). The guest kernel occupies the upper portion of the host user space [0x70000000, 0xc0000000), and the current guest application occupies the remainder of the host user space [0x0, 0x70000000).

Host component or event	**Emulation mechanism in UMLinux**
hard disk	host raw partition
CD-ROM	host /dev/cdrom
floppy disk	host /dev/floppy
network card	TUN/TAP virtual Ethernet device
console	TCP to host application
video card	none (display to remote X server)
current privilege level	VMM variable
system calls	SIGUSR1 signal
timer interrupts	timer + SIGALRM signal
I/O device interrupts	SIGIO signal
memory exception	SEGV signal
enable/disable interrupts	mask signals

Table 1: Mapping between host components and UMLinux equivalents.

translations that are set up via the host operating system's `mmap`, `munmap`, and `mprotect` system calls.

Figure 3 shows the address space of the virtual-machine process. Host memory protections are used to prevent guest applications from accessing the guest kernel's address space.

UMLinux provides a software analog to each peripheral device in a normal computer system. Table 1 shows the mapping from each host component or event to its software analog in the virtual machine. UMLinux uses a host file or raw device to emulate the hard disk, CD-ROM, and floppy. Our version of UMLinux uses the TUN/TAP virtual Ethernet device in Linux to emulate the network card. UMLinux uses a small X application on the host to display console output and read keyboard input; this application communicates with the guest kernel's console driver via TCP. UMLinux uses no video card; instead it displays graphical output to a remote X server (which would typically be the host's X server).

UMLinux provides a software analog to the computer's current privilege level. The VMM module maintains a virtual privilege level, which is set to *kernel* when transferring control to the guest kernel, and is set to *user* when transferring control to a guest application. The VMM module uses the current virtual privilege level to distinguish between system calls issued by a guest application and system calls issued by the guest kernel.

System calls issued by a guest application must be redirected to the guest kernel's system-call trap handler. When a guest application executes a system-call instruction (`int 0x80`), the host CPU traps to the host kernel's system-call handler, which then transfers control to the VMM kernel module. If the current virtual privilege level is set to *kernel*, then the VMM knows the guest kernel made the system call (typically to access a host device or change memory translations). In this case, the VMM checks that this system call is one that a UMLinux guest kernel is expected to make, then passes it through to the host kernel. If the virtual privilege level is set to *user*, then the VMM knows a guest application made the system call. In this case, the VMM module notifies the guest kernel by sending it a signal (SIGUSR1). The VMM module passes the registers at the time of the trap to the guest kernel's signal handler. The SIGUSR1 signal handler in the guest kernel is the equivalent of the system-call trap handler in a normal operating system.

SIGALRM, SIGIO, and SIGSEGV signals are used to emulate the hardware timer, I/O device interrupts, and memory exceptions. As with SIGUSR1, the host kernel delivers these signals to the registered signal handler in the guest kernel. These signal handlers are the equivalent of the timer-interrupt, I/O-interrupt, and memory exception handlers in a normal operating system.

UMLinux emulates the enabling and disabling of interrupts by masking signals (using the `sigprocmask` system call).

3.2. Trusted computing base for UMLinux

All the virtualization strategies described in Section 2 depend on the trustworthiness of all layers below the guest operating system (the VMM and host platform in Figure 1). For UMLinux, the trusted computing base (TCB) is comprised of the VMM kernel module and the host operating system. UMLinux's TCB is larger than the TCB for virtual machines that run directly on the hardware, such as IBM's VM/370 or VMware's ESX Server. UMLinux's TCB is similar to other virtual machines that cooperate with a host operating system, such as VMware Workstation.

A common question is whether a security service that is added to the host operating system in an OS-on-OS structure is more protected from attack than a security service that is added to the host operating system in a direct-on-host structure. For example, while the logging in an OS-on-OS structure does not depend on the integrity of the guest operating system, doesn't it still depend on the integrity of the host operating system?

We contend that the logging in an OS-on-OS structure is much more difficult to attack than the logging in a direct-on-host structure, because the TCB for an OS-on-OS structure can be much smaller than the complete host operating system [Meushaw00]. While both OS-on-OS and direct-on-host depend on the host operating system, the avenues a villain can use to attack the host differ greatly between the two structures.

Assume for this comparison that the villain has gained control over all target applications and can send arbitrary network packets to the host. A villain can launch attacks against the host operating system from two directions. First, a villain can attack from above by causing application processes to invoke the host operating system in dangerous ways. In a direct-on-host structure, the attacker has complete freedom to invoke whatever functionality the host operating system makes available to user processes. The attacker can control multiple application processes, access multiple files, and issue arbitrary system calls. In an OS-on-OS structure, an attacker who has gained control of all application processes can use these same avenues to attack the *guest* operating system. However, even if the attacker gains control over the guest operating system, he/she is still severely restricted in the actions he/she can take against the *host* operating system. The guest kernel needs only a small subset of the functionality available to general-purpose host processes, and the VMM can easily disallow functionality outside this subset [Goldberg96]. For example, an attacker who has gained control over all target applications and the guest operating system still controls only a single host process (the virtual-machine process), can access only a few host files/devices (the virtual hard disk, the virtual CD-ROM, and the virtual floppy), and can make only a few system calls.

Second, a villain can attack the low level of the network protocol stack by sending dangerous network packets to the host (e.g. ping-of-death). As with attacks from above, less of the host operating system is exposed to dangerous packets with an OS-on-OS structure than a direct-on-host structure. Without virtual machines, packets traverse through the entire network stack and are delivered to applications; villains can thus craft packets to attack any layer of the network stack. With virtual machines, packets need only traverse a small part of the network stack.

The portion of the host operating system included in UMLinux's TCB is the host OS code that the guest kernel or incoming packets can invoke (plus the VMM, which disallows invocations outside this portion). We have yet to measure the size of this code rigorously, but early indications suggest that this portion is significantly smaller than the entire host operating system. For example, our VMM restricts the guest kernel to use fewer than 7% of the system calls available to general host processes, and network traffic to the virtual machine is processed mostly by the guest operating system's TCP and UDP stacks (only a small IP-layer packet filter is used in the host operating system).

The TCB of our current UMLinux prototype, while smaller than the complete host operating system, is not yet as small as it could be. The host operating system in our prototype runs other processes which could be attacked (e.g. the X server), and network messages to these host processes traverse the entire host network stack. Our future work includes measuring and reducing the size of the host operating system used to support UMLinux. For example, we could further restrict the system calls issued by the guest kernel to use only certain parameter values, and we could move the X server into another virtual machine.

4. Logging and replaying UMLinux

4.1. Overview

Logging is used widely for recovering state. The basic concept is straightforward: start from a checkpoint of a prior state, then roll forward using the log to reach the desired state. The type of system being recovered determines the type of information that needs to be logged: database logs contain transaction records, file system logs contain file system data. Replaying a process requires logging the non-deterministic events that affect the process's computation. These log records guide the process as it re-executes (rolls forward) from a checkpoint. Most events are deterministic (e.g. arithmetic, memory, branch instructions) and do not need to be logged; the process will re-execute these events in the same way during replay as it did during logging.

Non-deterministic events fall into two categories: time and external input. Time refers to the exact point in the execution stream at which an event takes place. For example, to replay an interrupt, we must log the instruction at which the interrupt occurred. External input refers to data received from a non-logged entity, such as a human user or another computer. External input enters the processor via a peripheral device, such as a keyboard, mouse, or network card.

Note that output to peripherals does not affect the replaying process and hence need not be saved (in fact, output to peripherals will be reconstructed during replay). Non-determinism in the micro-architectural state (e.g. cache misses, speculative execution) also need not be saved, unless it affects the architectural state. Replaying a shared-memory multiprocessor requires saving the fine-grained interleaving order of memory operations and is outside the scope of this paper [LeBlanc87].

4.2. ReVirt

This section describes how we apply the general concepts of logging to enable replay of UMLinux running on x86 processors. ReVirt is implemented as a set of modifications to the host kernel.

Before starting UMLinux, we checkpoint the state by making a copy of its virtual disk. We currently require replay to start from a powered-off virtual machine, so the virtual disk comprises all state in the virtual machine. We envision checkpointing being a rare event (once every few days), so copying speed is not critical.

Log records are added and saved to disk in a manner similar to that used by the Linux `syslogd` daemon. The VMM kernel module and kernel hooks add log records to a circular buffer in host kernel memory, and a user-level daemon (`rlogd`) consumes the buffer and writes the data to a log file on the host.

ReVirt must log all non-deterministic events that can affect the execution of the virtual-machine process. Note that many non-deterministic host events do not need to be logged, because they do not affect the execution of the virtual machine. For example, host hardware interrupts do not affect the virtual-machine process unless they cause the host kernel to deliver a signal to the virtual-machine process. Likewise, the scheduling order of other host processes does not affect the virtual-machine process because there is no interprocess communication between the virtual-machine process and other host processes (no shared files, memory, or messages).

ReVirt does have to log asynchronous virtual interrupts (synchronous exceptions like SIGSEGV are deterministic and do not need to be logged). Before delivering a SIGALRM or SIGIO host signal (representing virtual timer and I/O interrupts) to the virtual-machine process, ReVirt logs sufficient information to re-deliver the signal at the same point during replay. To uniquely identify the interrupted instruction, ReVirt logs the program counter and the number of branches executed since the last interrupt [Bressoud96]. Because the x86 architecture allows a block memory instruction (repeat string) to be interrupted in the middle of its execution, we also must log the register (`ecx`) that stores the number of iterations remaining at the time of the interrupt.

x86 processors provide a hardware performance counter that can be configured to compute the number of branches that have executed since the last interrupt [Int01]. The `branch_retired` configuration of this performance counter on the AMD Athlon processor counts branches, hardware interrupts (e.g. timer and network interrupts), faults (e.g. page faults, memory protection faults, FPU faults), and traps (e.g. system calls). We use another hardware performance counter to count the number of hardware interrupts and subtract this from the `branch_retired` counter. Similarly, we instrument the host kernel to count the number of faults and traps and subtract this from the `branch_retired` counter. We configure the `branch_retired` counter to count only user-level branches. This makes it easier to count the number of branches precisely, because it keeps the count independent of the code executed in the kernel interrupt handlers.

In addition to logging asynchronous virtual interrupts, ReVirt must also log all input from external entities. These include most virtual devices: keyboard, mouse, network interface card, real-time clock, CD-ROM, and floppy. Note that input from the virtual hard disk is deterministic, because the data on the virtual hard disk will be reconstructed and re-read during replay. One can imagine requiring the user to re-insert the same floppy disk or CD-ROM during replay, in which case reads from the CD-ROM and floppy would also be deterministic and would not need to be logged. However, we do not expect data from these sources to be a significant portion of the log, because these data sources are limited in speed by the user's ability to switch media.[3]

The UMLinux guest kernel reads these types of input data by issuing host system calls `recv`, `read`, and `gettimeofday`. The VMM kernel module logs the input data by intercepting these system calls. In general, ReVirt must log any host system call that can yield non-deterministic results.

The x86 architecture includes a few instructions that can return non-deterministic results, but that do not normally trap when running in user mode. Specifically, the x86 `rdtsc` (read timestamp counter) and `rdpmc` (read performance monitoring counter) instructions are difficult for us to log. To make the virtual-machine process completely deterministic during replay, we set the processor control register (`CR4`) to trap when these instructions are executed. We remove the guest kernel's `rdtsc` instructions by replacing them with a `gettimeofday` host system call (and scaling the result); it would also be possible to leave these calls in the guest kernel, then trap, emulate, and log the `rdtsc` instruction. We disallow `rdpmc` in the guest kernel and guest applications.

During replay, ReVirt prevents new asynchronous virtual interrupts from perturbing the replaying virtual-machine process. ReVirt plays back the original asynchronous virtual interrupts using the same combination of hardware counters and host kernel hooks that were used during logging. ReVirt goes through two phases to find the right instruction at which to deliver the original asynchronous virtual interrupt. In the first phase, ReVirt configures the `branch_retired` performance counter to generate an interrupt after most (all but 128) of the branches in that scheduling interval. In the second phase, ReVirt uses breakpoints to stop each time it executes the target instruction. At each breakpoint, ReVirt compares the current number of branches with the desired amount. The first phase executes at the same speed as the original run and is thus faster than the second phase, which triggers a breakpoint each time the target instruction is executed. The second phase is needed to stop at exactly the right instruction, because the interrupt generated by the `branch_retired` counter does not stop execution instantaneously and may execute past the target number of branches.

Replay can be conducted on any host with the same processor type as the original host. Replaying on a different host allows an administrator to minimize downtime for the original host.

4.3. Cooperative logging

Most sources of non-determinism generate only a small amount of log data. Keyboard and mouse input is limited by the speed of human data entry. Interrupts are relatively frequent, but each interrupt generates only a few bytes of log data. Of all the sources of non-determinism, only received network messages have the potential to generate enormous quantities of log data.

We can reduce the amount of logged network data with a simple observation: one computer's received message is another computer's sent message. If the sending computer is being logged via ReVirt, then the receiver need not log the message data because the sender can re-create the sent data via replay. This technique is used commonly in message-logging recovery protocols [Elnozahy02] and can be viewed as expanding the domain of the replay system to include other computers. Thus the receiver need not log data sent from computers that can cooperate in the replay; the receiver need only log a unique identifier for the message (e.g. the identity of the sending computer and a sequence number).

Cooperative logging can reduce the amount of logged network data dramatically in certain cases. For example, if all computers on a LAN participate, then only traffic from outside the LAN needs to be logged, thus reducing the maximum log growth rate from LAN bandwidths to WAN bandwidths.

3. If the CD-ROM is switched by an automated jukebox, then the jukebox can participate in replay and CD-ROM reads can be considered deterministic.

While cooperating logging can reduce log volume, it complicates replay and requires that cooperating computers trust each other to regenerate the same message data during replay. We have not yet implemented cooperative logging in ReVirt.

4.4. Alternative architectures for logging and replay

We considered several strategies for building a logging/replay system before settling on the virtual-machine approach described above. In particular, we started by implementing a direct-on-host system, where the host kernel logged and replayed all its host processes. As discussed in Section 3.2, the direct-on-host approach is not as secure as a virtual-machine approach. We also found it to be much more difficult to log and replay all host processes than to log and replay a virtual-machine process. Interestingly, the narrow interface (between UMLinux and the host kernel) that makes an OS-on-OS approach more secure than a direct-on-host structure also makes an OS-on-OS system easier to replay.

The general approach for replaying a direct-on-host system is similar to that used in ReVirt: the system must log and replay all non-determinism. The same types of non-determinism exist for multiple host processes as for our virtual-machine approach (interrupt timing, external input).

However, it is much more challenging to log and replay a direct-on-host structure than a virtual-machine process, because a direct-on-host structure involves multiple host processes while an OS-on-OS approach involves only a single host process. (While the scheduling order between guest processes in UMLinux is non-deterministic, this is an abstraction above the virtual machine and is replayed deterministically as a result of deterministic signal delivery to the virtual-machine process.)

Replaying multiple host processes can be done in two ways, both of which are problematic. First, one can replay the communication channels between processes, but replaying a shared-memory communication channel requires complex instrumentation of the executing code and adds significant overhead [Netzer94].

Second, one can replay the scheduling order between host processes [Russinovich96]. This strategy is difficult because a host process can be interrupted while executing in kernel mode (e.g. while executing a system call). It is hard to identify the point in the kernel where an interrupt occurred, yet identifying this point is critical for replaying the exact scheduling order. The hardware performance counters we used to identify the exact interrupt point in ReVirt do not work well when the interrupt point is in the kernel, because we configure them to count only user-mode instructions. Configuring them to count both user and kernel instructions also leads to difficulties—the kernel does not execute deterministically, so the instruction counts would differ during replay.

A few solutions are possible, though none is appealing. First, one could change the host kernel to only allow interrupts at a few well-defined points and log which of these points was interrupted. This would require widespread changes to the host kernel. Second, one could try to replay the entire host kernel. This would require changing the interrupt handlers to log and replay hardware interrupts, and adjusting the hardware performance counters for the different code paths executed by the interrupt handlers during logging and replay.

In addition to coping with scheduling order between multiple host processes, a direct-on-host approach must cope with a large number of non-deterministic interfaces. There are a large number of system calls, including some (e.g. `ioctl`) that have a very wide variety of possible parameters. Replaying a direct-on-host system requires one to identify, log, and replay non-determinism in each of these system calls. In contrast, ReVirt only needs to handle the few systems calls used by UMLinux.

4.5. Using ReVirt to analyze attacks

ReVirt enables an administrator to replay the complete execution of a computer before, during, and after the attack. Two types of tools can be built on this foundation to assist the administrator to understand the attack.

The first type of tool runs inside the guest virtual machine. ReVirt supports the ability to continue live (i.e. non-replaying) execution at any point in the replay. An administrator can use this ability to run new guest commands to probe the virtual machine state. For example, the administrator can stop the replay after a suspicious point and use normal guest commands to edit the current files, list the current processes, and debug processes. However, the virtual machine cannot switch back to replaying after being perturbed in this manner, because the instruction counts will not apply to the revised state. To continue the replay beyond the perturbed point, the analyst should checkpoint the process before perturbing it or start the replay over and let it continue to the later point.

Second, tools such as debuggers and disk analyzers can run outside the guest virtual machine and analyze the state of a virtual machine (address space, registers, and disk data). The advantage of these off-line tools is that they do not depend on the guest kernel or guest applications. For example, an off-line tool can inspect the contents of a directory even if the attacker has replaced the command that normally lists the directory.

A particularly useful tool that runs outside the guest is one that re-displays the original graphical output. Recall that UMLinux uses a remote X server (perhaps running on the host) as its graphical display. The replaying virtual-machine process faithfully recreates the stream of network packets being sent to the X server. However, the X server is not under the control of the replay system and will likely send back different packets to the virtual machine (e.g. due to different mouse movements). The packets being sent to the virtual machine do not affect replay, because the replaying machine gets its input packets from the log. However, the TCP protocol at the X server may expect different replies to the packets it sends to the virtual machine and may be confused by the virtual machine's resent packets. We address this with a simple X proxy on the host that opens a *new* TCP connection to the X server. The X proxy's goal is to act as a new X client that happens to send the same display messages to the X server as the virtual machine did during logging. The X proxy accomplishes this by receiving the packets being (re)sent from the replaying virtual machine, stripping off the Ethernet, IP, and TCP headers from these packets, reconstituting the X window data stream, and sending the data stream to the X server. Fortunately, the X protocol is largely deterministic and does not require the client to reply to messages sent from the X server (the sole exception is the X authentication protocol, and the X proxy can be written to navigate through this protocol).

5. Experiments

This section validates correctness and quantifies virtualization and logging overhead for our modified UMLinux and the ReVirt logging and replay system. All experiments are run on a computer with a AMD Athlon 1800+ processor, 256 MB of memory, and a Samsung SV4084 IDE disk. The guest kernel is Linux 2.4.18 ported to UMLinux, and the host kernel for UMLinux is a modified version of Linux 2.4.18. The virtual machine is configured to use 192 MB of "physical" memory. The virtual hard disk is stored on a raw disk partition on the host to avoid double buffering the virtual disk data in the guest and host file caches, and to prevent the virtual machine from benefitting unfairly from the host's file cache.

We evaluate our system on five workloads. All workloads start with a warm guest file cache. *POV-Ray* is a CPU-intensive ray-tracing program. We render the benchmark image from the POV-Ray distribution at quality 8. *kernel-build* compiles the complete Linux 2.4.18 kernel (make clean, make dep, make bzImage). *NFS kernel-build* is the same as *kernel-build* with the kernel stored on an NFS server. *SPECweb99* is a benchmark that measures web server performance; we use the 2.0.36 Apache web server. We configured *SPECweb99* with 15 simultaneous connections spread over two clients connected to a 100 Mb/s Ethernet switch. Both workloads exercise the virtual machine intensively by making many system calls. They are similar to the I/O-intensive and kernel-intensive workloads used to evaluate Cellular Disco [Govil00]. We also used ReVirt and UMLinux as the first author's desktop machine for 24 hours to get an idea of the virtualization and logging overhead for day-to-day use.

Each result represents the average of three runs (except for the daily-use test, which represents a single 24-hour period). Variance across runs is less than 3%.

5.1. Virtualization overhead

Our first concern is the time overhead that arises from running all applications in the UMLinux virtual machine. We compare running all applications within UMLinux with running them directly on a host Linux 2.4.18 kernel. The host and guest file systems have the same versions of all software exercised in the tests (based on RedHat 6.2).

Table 2 shows the results. UMLinux with our host optimizations adds very little overhead for compute-intensive applications such as *POV-Ray*. We also perceive no overhead when using UMLinux for interactive jobs such as e-mail, editing, word processing, and web browsing.

The overheads for *SPECweb99*, *kernel-build*, and *NFS kernel-build* are higher because they issue more guest kernel calls, each of which must be trapped by the VMM kernel module and reflected back to the guest kernel by sending a signal. In addition, *kernel-build* and *NFS kernel-build* cause a large number of guest processes to be created, each of which maps its executable pages on demand. Each demand-mapped page causes a signal to be delivered to the guest kernel, which must then ask the host kernel to map the new page.

Workload	**UMLinux runtime (normalized to direct-on-host)**
POV-Ray	1.01
kernel-build	1.58
NFS kernel-build	1.44
SPECweb99	1.13
daily use	≈ 1

Table 2: Virtualization overhead. This table shows the overhead caused by running applications in UMLinux. Runtime is normalized to the runtime when running directly on the host.

We believe the overheads for using UMLinux are low enough to be unnoticeable for normal desktop use. While overheads are higher for workloads that use the guest kernel intensively, we believe that even an overhead of 58% is not prohibitive for sites that value security.

For reference, VMware Workstation 3.1 has a normalized runtime of approximately 1.25 for *kernel-build*, UMLinux without our modifications to the host kernel has a normalized running time of 26, and a recent version of User-Mode Linux (configured to protect the guest kernel memory from guest applications) has a normalized runtime of 14. The low overhead of VMware and its acceptance in production settings indicate that virtualization can be made fast enough to enable services such as ReVirt.

5.2. Validating ReVirt correctness

Our next goal was to verify that the ReVirt system faithfully replays the exact execution of the original run. For these runs, we add extensive error checking to alert us if the replaying run deviated from the original. At every system call and virtual interrupt, we log all register values and the `branch_retired` counter and verify that these values are the same during replay. In addition, ReVirt's mechanism for replaying interrupts verifies that the branch count at the interrupted instruction matches the branch count seen at that instruction during logging.

We first run two micro-benchmarks in the virtual machine to verify that virtual interrupts are being replayed at the same point at which they occurred during logging. The first micro-benchmark runs two guest processes that share an mmap'ed memory region. Each guest process increments a shared variable 10,000,000 times, prints the resulting value, then repeats. Because the two guest processes share this variable, the output of process A depends on how many iterations process B executed by the time process A prints the value. The second microbenchmark runs a single process that increments a variable in an infinite loop. The process prints the current value when it receives a signal. This test verifies that the guest kernel re-delivers the signal at the same point as during logging.

We ran each micro-benchmarks 5 times, and each time the output during replay matched the original output, and all error checks passed.

We next run a macro-benchmark to verify that ReVirt faithfully plays back input from external systems and to exercise the system as a whole for longer periods. During the macro-benchmark, we boot the computer, start the GNOME window manager (displaying on a remote X server), open several interactive terminal windows, and concurrently build two applications (`freeciv` and `mup`) on a remote NFS server. The logging run of this benchmark generates 15,000,000 system calls and 55,000 virtual interrupts. ReVirt replayed this run without any deviation from the original run.

For the other tests used in this paper, we disable the extra error checking mentioned above. However, ReVirt always checks that the branch count at the interrupted instruction matches the branch count seen at that instruction during logging, and we have found this to detect errors effectively while we were developing ReVirt.

5.3. Logging and replaying overhead

Next we seek to quantify the space and time overhead of logging. We do not include the time and space overhead to checkpoint the system, since we expect a checkpoint to be amortized over a long period of time (e.g. a few days). Table 3 shows the time and space overhead for logging on the *POV-Ray*, *kernel-build*, *NFS*

Workload	**Runtime with logging (normalized to UMLinux *without* logging)**	**Log growth rate**	**Replay runtime (normalized to UMLinux *with* logging)**
POV-Ray	1.00	0.04 GB/day	1.01
kernel-build	1.08	0.08 GB/day	1.02
NFS kernel-build	1.07	1.2 GB/day	1.03
SPECweb99	1.04	1.4 GB/day	0.88
daily use	≈ 1	0.2 GB/day	0.03

Table 3: Time and space overhead of logging and replay. Logging slowdown shows the overhead caused by logging, relative to running UMLinux without logging. Log growth rate shows the average rate of growth of the log during the workload. Replay runtime is normalized to the runtime of UMLinux with logging. Replay runtime values less than 1 indicate that replay ran faster than logging, due to replay's ability to skip over periods of idle time.

kernel-build, and *SPECweb99* workloads. Logs are stored in a compressed format using `gzip`.

Table 3 shows that the time overhead of logging is small (at most 8%).

The space overhead of logging is small enough to save the logs over a long period of time at low cost. Workloads with little non-determinism (e.g. *POV-Ray, kernel-build*) generate very little log traffic. Note that the log data needed to replay local compilations takes much less space than the disk data generated in compilation.

The log growth rate for *NFS kernel-build* and *SPECweb99* is higher because of the need to log incoming network packets. However, it is still not prohibitive. For example, a 120 GB disk can store the volume of log traffic generated by *NFS kernel-build* for 3-4 months. If the file server used ReVirt, using cooperative logging at the client would reduce the log volume generated by *NFS kernel-build* to that of *kernel-build*.

We also used ReVirt and UMLinux as the first author's desktop machine for 24 hours to get an idea of the virtualization and logging overhead for day-to-day use[4]. We experienced no perceptible time overhead relative to running directly on the host, and the log occupied 0.2 GB after one day.

Table 3 shows that workloads typically replay at the same speed as they ran during logging. It is possible to replay a workload faster (sometimes much faster) than it ran during logging because replay skips over periods of idle time, such as that encountered during the non-working hours of the daily use workload.

4. This test was run using Linux 2.2.20 as the guest operating system.

5.4. Analyzing an attack

Finally, we demonstrate the ability of ReVirt to help analyze a non-deterministic attack that involves a kernel-level vulnerability. We re-introduced into our guest kernel the ptrace race condition that was present in Linux kernels before 2.2.19 [CER01b]. A villain exploits this bug by running a setuid process and attaching to it via `ptrace`. The vulnerability is non-deterministic because it depends on a time-of-check to time-of-use race condition. The attack is successful only if the file is not currently in the file cache, and the file cache state depends on the scheduling order and behavior of prior processes.

We exercised the vulnerability until compromising the system, then we added a trojan horse to /bin/ls and a backdoor to /etc/inetd.conf. ReVirt successfully replays the attack and allows us to find out how the attacker compromised the system and assess all damage done after the point of compromise. We were able to stop the replay after each point in the attack, run guest programs that examined the system state, and diagnose the method and effects of the intrusion.

6. Related work

Bressoud and Schneider's work on hypervisor-based fault tolerance [Bressoud96] shares several techniques with ReVirt. Bressoud and Schneider use a virtual machine for the PA-RISC architecture to interpose a software layer between the hardware and an unchanged operating system, and they log non-determinism to reconstruct state changes from a primary computer onto its backup.

While ReVirt shares several mechanisms with Hypervisor, ReVirt uses them to achieve a different and new goal. Hypervisor is intended to help tolerate faults by mirroring the state of a primary computer onto a backup. ReVirt takes some of the techniques developed for fault tolerance and applies them to provide a novel security tool. Specifically, ReVirt is intended to replay the complete, long-term execution of a computer. To illustrate the difference between these goals, compare the usefulness of checkpoints for each goal. Recovering a backup to a prior point in time can be accomplished either by checkpointing the primary's state periodically or by logging the primary's operations. On the other hand, checkpoints are not sufficient for intrusion analysis because they do not show how the system transitioned between checkpoints; checkpoints can only be used to initialize the replay procedure.

Besides a difference in goals, Hypervisor and ReVirt also differ in several design choices. Because Hypervisor only seeks to restore the backup to the last saved state of the primary, it discards log records after each synchronization point. In contrast, ReVirt enables replay over long periods (e.g. months) of the computer's execution, so it must save all log records since the last checkpoint. Another difference is that Hypervisor defers the delivery of interrupts until the end of a fixed number of instructions (called an epoch), while ReVirt delivers interrupts as soon as they occur (or when the guest kernel re-enables interrupts). Hypervisor also logs more information than ReVirt (e.g. Hypervisor logs disk reads).

There are several virtual machines that are similar to UMLinux. User-Mode Linux [Dike00] shares many of the same goals as UMLinux [Buchacker01]. We chose UMLinux because the virtual machine is contained in a single host process, whereas User-Mode Linux uses a separate host process for each guest application process (this speeds up context switching between guest processes). SimOS's direct-execution mode is also similar to these systems but is targeted at an architecture that is easier to virtualize than the x86 [Rosenblum95].

ReVirt shares a similar philosophy of security logging with S4 [Strunk00]. Both ReVirt and S4 add logging below the target operating system to protect the logging functionality and data from compromised applications and operating systems. ReVirt adds logging to a virtual machine, while S4 adds it to disk drives. The logging in ReVirt captures different information than the logging in S4. ReVirt enables replay of the entire computer's execution, while S4 logs and replays disk activity. ReVirt and S4 save different data to the log (ReVirt saves non-deterministic events, S4 saves disk data), so a comparison of log volume generated will depend on workload.

7. Future work

Our near-term work is to make checkpointing faster and more convenient. We plan to accelerate the disk copy done during checkpointing using copy-on-write. We plan to enable the VMM to checkpoint a running virtual machine by saving and reconstructing the host-kernel state for the virtual-machine process [Plank95].

We also plan to build higher-level analysis tools that leverage ReVirt's ability to replay detailed, long-term executions. Whereas current techniques in computer forensics can only analyze the evidence left behind by careless intruders, ReVirt allows an analyst to watch any intrusion in arbitrary detail.

Finally, we plan to use ReVirt as a building block for new security services. ReVirt's ability to recover to an arbitrary state may enable us to recover a system automatically and to analyze or prevent key events in an attack.

8. Conclusions

ReVirt applies virtual-machine and fault-tolerance techniques to enable a system administrator to replay the long-term, instruction-by-instruction execution of a computer system. Because the target operating system and target applications run within a virtual machine, ReVirt can replay the execution before, during, and after the intruder compromises the system. This capability is especially useful for determining and fixing the damage the intruder inflicted after compromising the system. Because ReVirt logs all non-deterministic events, it can replay non-deterministic attacks and executions, such as those that trigger race conditions. Finally, because ReVirt can replay instruction-by-instruction sequences, it can provide arbitrarily detailed observations about what transpired on the system.

ReVirt adds reasonable time and space overhead. The overhead for virtualization ranges from imperceptible for interactive and CPU-bound applications to 13-58% for kernel-intensive applications. The time overhead of logging ranges from 0-8%, and logging traffic for our workloads can be stored on a single disk for several months.

9. Acknowledgments

We are grateful to the researchers at the University of Erlangen-Nürnberg for writing UMLinux and sharing it with us. In particular, Kerstin Buchacker and Volkmar Sieh helped us understand and use UMLinux. We would also like to thank Brian Noble, Landon Cox, the anonymous reviewers, and our shepherd Jay Lepreau for their helpful feedback. This research was supported in part by National Science Foundation grants CCR-0098229 and CCR-0219085 and by Intel Corporation. Samuel King was supported by a National Defense Science and Engineering Graduate Fellowship.

10. References

[Anderson80] James P. Anderson. Computer Security Threat Monitoring and Surveillance. Technical report, James P. Anderson Co., April 1980. Contract 79F296400.

[Ashcraft02] Ken Ashcraft and Dawson Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, May 2002.

[Bellino73] J. Bellino and C. Hans. Virtual Machine or Virtual Operating System? In *Proceedings of the 1973 ACM Workshop on Virtual Computer Systems*, pages 20–29, 1973.

[Bishop96] Matt Bishop and Michael Dilger. Checking for Race Conditions on File Accesses. *USENIX Computing Systems*, 9(2):131–152, 1996.

[Bressoud96] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, February 1996.

[Buchacker01] Kerstin Buchacker and Volkmar Sieh. Framework for testing the fault-tolerance of systems including OS and network aspects. In *Proceedings of the 2001 IEEE Symposium on High Assurance System Engineering (HASE)*, pages 95–105, October 2001.

[CER01a] CERT/CC Security Improvement Modules: Analyze all available information to characterize an intrusion. Technical report, CERT Coordination Center, May 2001.

[CER01b] Linux kernel contains race condition via ptrace/procfs/execve. Technical Report Vulnerability Note VU#176888, CERT Coordination Center, March 2001.

[CER02] CERT/CC Overview Incident and Vulnerability Trends. Technical report, CERT Coordination Center, April 2002.

[Chen01] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the 2001 Workshop on Hot Topics in Operating Systems (HotOS)*, pages 133–138, May 2001.

[Dike00] Jeff Dike. A user-mode port of the Linux kernel. In *Proceedings of the 2000 Linux Showcase and Conference*, October 2000.

[Elnozahy02] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[Goldberg74] Robert P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, pages 34–45, June 1974.

[Goldberg96] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 1996 USENIX Technical Conference*, July 1996.

[Govil00] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 18(3):226–262, August 2000.

[Hon00] Report on the Linux Honeypot Compromise. Technical report, Honeynet Project, November 2000. http://project.honeynet.org/challenge/results/dittrich/evidence.txt.

[Int01] The IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide. Technical report, Intel Corporation, 2001.

[Karger91] Paul A. Karger, Mary Ellen Zurko, Douglis W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11), November 1991.

[King02] Samuel T. King. Operating System Extensions to Support Host-Based Virtual Machines. Technical Report CSE-TR-465-02, University of Michigan, September 2002.

[LeBlanc87] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant

Replay. *IEEE Transactions on Computers*, pages 471–482, April 1987.

[Meushaw00] Robert Meushaw and Donald Simard. NetTop: Commercial Technology in High Assurance Applications. *Tech Trend Notes: Preview of Tomorrow's Information Technologies*, 9(4), September 2000.

[Netzer94] Robert H. B. Netzer and Mark H. Weaver. Optimal Tracing and Incremental Reexecution for Debugging Long-Running Programs. In *Proceedings of the 1994 Conference on Programming Language Design and Implementation (PLDI)*, June 1994.

[Plank95] James S. Plank, Micah Beck, and Gerry Kingsley. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of the Winter 1995 USENIX Conference*, pages 213–224, January 1995.

[Rosenblum95] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: the SimOS approach. *IEEE Parallel & Distributed Technology: Systems & Applications*, 3(4):34–43, January 1995.

[Russinovich96] Mark Russinovich and Bryce Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the 1996 Conference on Programming Language Design and Implementation (PLDI)*, pages 258–266, May 1996.

[Strunk00] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A.N. Soules, and Gregory R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 2000 Symposium on Operating Systems Design and Implementation (OSDI)*, October 2000.

[Sugerman01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Technical Conference*, June 2001.