ExecRecorder: VM-Based Full-System Replay for Attack Analysis and System Recovery

Daniela A. S. de Oliveira   Jedidiah R. Crandall   Gary Wassermann
S. Felix Wu   Zhendong Su   Frederic T. Chong
University of California at {Davis, Santa Barbara}
{oliveira,crandall,wassermg,wu,su}@cs.ucdavis.edu, chong@cs.ucsb.edu

Abstract

Log-based recovery and replay systems are important for system reliability, debugging and postmortem analysis/recovery of malware attacks. These systems must incur low space and performance overhead, provide full-system replay capabilities, and be resilient against attacks. Previous approaches fail to meet these requirements: they replay only a single process, or require changes in the host and guest OS, or do not have a fully-implemented replay component. This paper studies full-system replay for uniprocessors by logging and replaying architectural events. To limit the amount of logged information, we identify architectural nondeterministic events, and encode them compactly. Here we present ExecRecorder, a full-system, VM-based, log and replay framework for post-attack analysis and recovery. ExecRecorder can replay the execution of an entire system by checkpointing the system state and logging architectural nondeterministic events, and imposes low performance overhead (less than 4% on average). In our evaluation its log files grow at about 5.4 GB/hour (arithmetic mean). Thus it is practical to log on the order of hours or days between checkpoints. It can also be integrated naturally with an IDS and a post-attack analysis tool for intrusion analysis and recovery.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection–*invasive software*; B.8.1 [*Performance and Reliability*]: Reliability, Testing, and Fault-Tolerance

General Terms Security, virtual machines, invasive software

Keywords replay, recovery, malware, worms, virtual machines

1. Introduction

Log-based recovery [2, 3, 5, 11, 19, 23, 24] and replay [7, 10, 13, 17, 20, 21, 27, 30] systems are important for system reliability and system security. Recent work has also used a replayer to perform postmortem analysis of malcode attacks [10]. After an attack, one may replay the attack sequence using off-line analyzers. Replay systems can also be used for recovery from malware attacks by integrating them with an intrusion detection system (IDS) and an analysis tool. Upon detecting an attack (with an IDS) and discovering the execution point at which the attack happened (with an analysis tool), one can roll-back the system execution to an earlier checkpoint and disable particular effects of the attack.

Replay and recovery systems are generally based on three components: checkpoint, log and replay. The checkpoint component captures a snapshot of the current state of a system at specific times. The log component records the nondeterministic events that affected system execution since the checkpoint was taken. The replay component uses the information logged along with the checkpoint to deterministically replay the system execution during that specific run (the sequence of states a system passes through during execution, represented by the partial ordering of events sent and received and also local events [2, 3]).

These systems must incur low space and performance overhead, provide full-system replay capabilities, and be resilient against attacks. Previous approaches fail to meet these requirements. Most replay only a single process or application [13, 17, 21, 27]. Those that address the whole system require changes in the host and guest OS [10], or do not have yet a fully-implemented replay component [30].

This paper studies full-system log-and-replay for uniprocessors by logging and replaying architectural events. In order to limit the amount of information that needs to be logged we characterize architectural nondeterminism, both by identifying nondeterministic events and by encoding them compactly. Working at the level of architectural events enables a full-system replay that is flexible with respect to the OS and the applications.

We have implemented our system as ExecRecorder, a full-system, VM-based log and replay framework for post-attack analysis and recovery. It can replay the execution of an entire system (not only a process or a distributed application in isolation) by checkpointing the complete system state (virtual memory and CPU registers, virtual hard disk and memory of all virtual external devices) and logging all architectural nondeterministic events. The checkpoints can be taken at any time and the replay does not need to start from a powered-off machine. Our strategy for checkpointing the hard disk (HD) is efficient (based on copy-on-write), and is achieved by using committable/rollbackable disk images.

Virtual machines (VMs) are considered the ideal place, in terms of security and reliability, where an IDS or a post-attack analysis tool should be placed. This is because even when the monitored guest OS is compromised, an attacker cannot easily control the actions of an IDS or the integrity of logged data. Because ExecRecorder runs as part of a VM, it is not easily accessible to malware, yet still gives a detailed view of the system execution. In order to perform post-attack analysis and recovery there should be integration and cooperation among the IDS, analysis tool and replay system. We believe that this can be best achieved when all of these are running under the complete control of a VM.

ASID'06 October 21, 2006, San Jose, California, USA.
Copyright © 2006 ACM 1-59593-576-2…$5.00.

ExecRecorder imposes low performance overhead (beyond that of the VM). In our evaluation, which includes multiple workloads in both Windows and Linux, its log files grow at about 5.4 GB/hour (arithmetic mean). Thus it is practical to log on the order of hours or days between checkpoints. It can also be integrated naturally with an IDS, such as Minos [8], to determine when an exploit has occurred. An analysis tool, such as DACODA [9], can point where the exploit was detected in order to understand the exploited vulnerability.

The rest of the paper is organized as follows. Section 2 surveys related work. This is followed by Section 3 which characterizes architectural nondeterministic events and provides a compact format encoding. Then Section 4 presents ExecRecorder and details its implementation, followed by Section 5 which presents our experimental evaluation. In Section 6 we describe an example of post-attack analysis using ExecRecorder, and conclude with a discussion of future work in Section 7.

Our contributions are as follows. First, we provide a characterization of architectural nondeterminism for uniprocessors and a compact format encoding. Second, we present a low-overhead, full-system log and replay framework for uniprocessors that runs integrated with a VM and does not require any modification in the guest or host OS. Third, we show with a practical example how such system can streamline the analysis of a zero-day exploit.

2. Related Work

Instant Replay [17] is a deterministic replay for highly parallel programs to help the debugging process. During program execution the relative order of significant events is saved without recording any data associated with them. It is well-suited for replaying non-interactive applications.

Flashback [27] is a lightweight OS extension for software debugging that provides replay capabilities for an application. The main idea is to use shadow processes to capture the in-memory state of a process at a specific execution point and log the process' interactions with the system. ExecRecorder can replay the execution of an entire system in the same environment as it happened during log and does not require any changes in the OS. In Flashback, a replayed application may be executed in a different environment.

Rx [21] proposes an interesting technique to survive software failures by treating bugs as allergies: their manifestations can be avoided if the execution environment is changed. As in Flashback [27], it also uses shadow processes and the application replay happens in a changed environment.

FDR [30] is a low-overhead, full-system hardware recorder for software debugging. Our system is different from FDR in that it records architectural events inside a VM and does not address multiprocessors and DMA, while FDR is an actual design of a hardware recorder for multiprocessors. On the other hand, FDR only enables replay intervals of approximately 1 second, does not capture disk state, and does not provide a fully-implemented replayer. ExecRecorder allows replay windows of any length (provided that there is enough disk space to store the log files) and checkpoints the disk with copy-on-write.

BugNet [13] is a log and replay architecture for debugging. It records the register file contents at any point in time, and the load values that occur after that point. As it focuses on application level bugs, it cannot replay the full-system execution.

Dunlap et al. proposed ReVirt [10], a logging and replay system for analyzing intrusions that runs integrated with a VM and performs the logging in the host OS. After an attack, it can replay the whole VM process for analysis. Our work is different from ReVirt because our application requires fine-grained control and more flexibility in the choice of the guest OS. ExecRecorder does not require any changes in the guest or host OS. This is important because most Internet worms attack Windows, and many worms or attacks must be caught with a specific version of an OS. For our studies we need to be able to run Windows XP, Windows 2000, Windows Whistler, as well as a variety of Linux, FreeBSD, and OpenBSD distributions, and interface with these using a Pentium hardware interface. ReVirt is implemented as a set of modifications in the host kernel and the guest OS must be ported to run on their VM (UMLinux). Second, ExecRecorder gives us complete control of a system under replay, which is necessary for future work on attack recovery and replay-based entropy control. In post-attack recovery, where we want to recover the system by disabling particular effects of the attack and replaying the modified run, we need full-control of each event being replayed. ReVirt, by just replaying the VM process from the perspective of the host OS, does not offer such level of control. While Bochs limits the performance of our implementation, we can do very sophisticated analysis without prohibitively affecting system performance by using the low-overhead ExecRecorder module, and then do the analysis during replay with much more expensive modules such as DACODA.

3. Nondeterminism

Models of real-world systems necessarily exclude some details of the systems they model. Within a model, a nondeterministic event is one that causes a state transition but is not fully determined by the previous state, i.e., the event could not have been predicted with certainty from knowledge only of the model's previous state(s). For systems based on uniprocessors, the nondeterministic architectural events are hardware interrupts and input events.

3.1 Hardware Interrupts

External devices such as the HD or the keyboard generate interrupts asynchronously with regard to the processor clock [18]. Although it is possible to model the behavior of a certain device with great accuracy, the exact time at which an interrupt is generated is unpredictable.

We encode an interrupt as an integer representing the difference between the tick of the last event and the tick at which the interrupt occurred, and an integer encoding the interrupt value, i.e., the IRQ line number. In our VM, the tick means the number of instructions executed before that event happened [31]. Logging just the tick difference allows us to considerably reduce the amount of logged data. For example, if the tick at which an interrupt occurred is 4294967298 and the tick of the last event is 4294967297 (both requiring 8 bytes for recording), we log just 1 as the tick information for the interrupt event. This approach decreases the number of bytes required to record the event's timing from 8 bytes to a maximum of 4 bytes (we have observed that the tick difference does not exceed 2^{32}).

In this study we have considered the following external devices: PIT, CMOS, HD, keyboard, mouse, network card and serial and parallel port devices. PIT interrupts are generally regarded as deterministic events. We are characterizing them as nondeterministic because, although most of its interrupts occur at a fixed frequency, there is some nondeterminism coming from noise in the device's crystal-controlled oscillator and from our VM's PIT implementation. As a result, these interrupts are not entirely predictable.

3.2 Input Events

These are events where data from a device is read to main memory or a register where the CPU can process it. The exact time at which these data become available to the CPU is also unpredictable, even though the behavior of external devices can be modeled. Moreover, the bytes coming from such events are also nondeterministic because they can come from user input, network or the environment (for example, the current time). An important exception are the bytes coming from the HD. The value of these bytes is based solely on the contents of the disk and the disk requests made by a program. Thus, we characterize input events coming from the HD as deterministic. We encode an input event as an integer representing the tick difference and an integer representing the byte(s) read.

4. The Log and Replay Framework

4.1 Log-Based Rollback Recovery

Log-based rollback recovery is a technique used to achieve fault-tolerance in distributed systems and also to allow the replay of an application or system for debugging or post-attack analysis. The main idea is to combine checkpoints with a log file. The checkpoint is a snapshot of the system state. The log file contains enough information to reproduce all nondeterministic events that occurred during a run. All messages received by the system during a run are classified as nondeterministic. Log-based rollback recovery relies on the *piecewise deterministic* assumption [2, 11] which states that all nondeterministic events that a process executes can be identified and all information necessary to reproduce these events can be logged. This set of information needed to reproduce a nondeterministic event is called the *determinant* of the event [3].

Besides the checkpoint and the log components, there is also a replay component. The log component continuously records determinants of nondeterministic events in a non-volatile medium. The replay component recovers the system state captured by a checkpoint (usually the latest one) and uses the determinants in the log file to reproduce the execution of a run. The amount of data recorded plays an important role on the performance of such systems. It will directly impact the overhead incurred by the log and replay components.

4.2 Logging and Replaying Inside a VM

A VM is a software layer running on top of a real machine or a host OS to support running a system or a process in a target architecture [25]. The code providing virtualization is usually called virtual machine monitor (VMM).

VM's were first developed more than 30 years ago to provide timesharing capabilities to mainframes. Today, they have been receiving renewed attention by academia and industry [6, 12, 16, 22, 25, 29] due to the advantages they provide for present applications, especially in the security field. VM's are suitable for system replication because they allow several virtual machines execute different systems or applications concurrently on a single real machine. They can also provide software compatibility and portability by allowing software written for a certain architecture be executed on another [25]. Further, they can isolate systems and applications from one another, thus, improving reliability and security: a crash, a bug or a security breach of a certain system or application will not impact others being executed in different VM's.

A system-level VM supports an entire guest OS along with its applications and can be classified as native-VM, hosted-VM or dual-hosted-VM. A native-VM is installed directly on hardware and has the best performance among the three. A host-VM is installed on top of a host OS and has the worst performance due to the extra level of software. A dual-hosted-VM represents a hybrid solution by having some of its modules running in priviledged mode (as in a native-VM) and others in user-mode (as in a host-VM).

Although native-VM's present better performance, hosted-VM's lend themselves better for post-attack analysis and recovery. They provide isolation for the host OS: we can have an IDS and a set of applications running in the host OS even if a guest OS has been compromised by an attack.

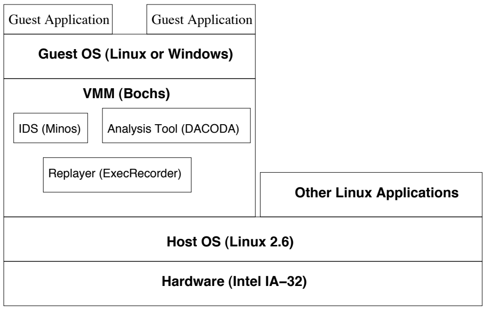

Figure 1. Our VM architecture.

4.3 Implementation

We have used Bochs [31] as the VM of our experiments. It is a hosted-VM that emulates the IA-32 Pentium architecture. The host OS in these experiments is Linux 2.6. Our VM is currently integrated with Minos [8], a microarchitecture to catch attacks, and DACODA [9], a post-attack analysis tool. We have designed this system to run as a honeypot. Figure 1 illustrates this architecture. ExecRecorder has three main components: checkpoint, log and replay.

4.3.1 Checkpoint

This module is executed immediately before the logging of a system run. It is responsible for saving the system state (virtual main memory, CPU registers, HD and memory from external devices) at the current instant of time. We have implemented it by duplicating the Bochs VM process via the *fork* system call. After the fork, the parent process waits in the background for a SIGUSR1 signal while the child process continues its execution. The suspended parent represents the frozen state of the system at the time the checkpoint was taken.

The checkpoint of the virtual HD is achieved by using the undoable disk mode of Bochs. An undoable disk is a committable/rollbackable disk image [31]. It is based on a read-only disk image combined with a file, called *redolog*, that contains all changes made to the read-only image. After a run the *redolog* file can be merged to the read-only image or simply discarded. ExecRecorder always starts the VM with the read-only disk image. When a checkpoint is taken, the child process continues its execution with a new *redolog* file, which is initialized with the contents of the parent process *redolog* file.

4.3.2 Log

The log component records in the host HD enough information about the nondeterministic events happening in the system so that they can be later replayed.

In order to correctly replay input events we need to log more information than just the characterization of nondeterministic input events given in Section 3.2. Although an input event can be solely characterized by the time or tick at which it occurred and the bytes read, our replay component needs enough information about the input instruction itself to correctly reproduce it. For example, for the Intel IA-32 architecture [14] we have input instructions to transfer a (or a string of) byte(s), word(s), or a double word(s) between an I/O port and a CPU register. To simulate an I/O port instruction, our VM implementation requires knowledge of the number of bytes being transferred and whether it is to a register or memory. In theory, this type of information is not required in the log file.

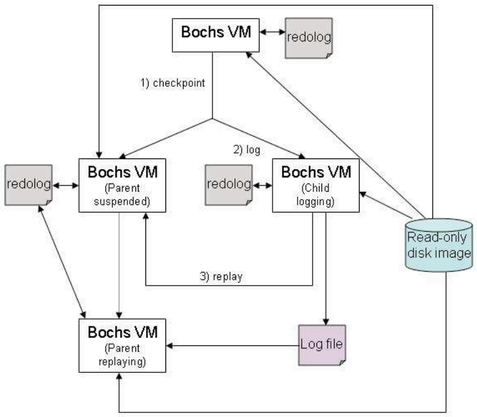

Figure 2. ExecRecorder.

Although we have considered all input events from the HD as deterministic, the replay component still needs information about the HD input instruction to reproduce it. The reason is that, during replay, we need to make the disk requests synchronized with the tick at which an HD interrupt occurs. If we do not log and replay such instructions, it may occur that (at least in our Bochs VM) a HD interrupt is raised before the intended bytes are read from the disk. Note that we do not log the bytes read from the disk but only information about the input instruction. FDR [30], on the other hand, logs all values returned from I/O loads.

The format of our log files is as follows:

- event type (1 byte): input event, interrupt or the handling of an interrupt by the CPU (the last one is Bochs-specific);
- tick difference (4 bytes);
- Input events-specific: port address (2 bytes), bytes (4 bytes, not logged for HD), flags (1 byte encoding information about number of bytes being transferred and whether it is to a register or memory), memory address (4 bytes, logged only if bytes are being transferred to memory);
- Interrupts-specific: IRQ number (1 byte).

4.3.3 Replay

After the logging of a run this module can be called to reproduce the system execution from a certain checkpoint. The child process wakes up the parent process with a SIGUSR1 signal. The parent process, which captures the system state at the point the checkpoint was taken, resumes its execution. The virtual disk image used is also the one at the time the checkpoint was taken. However, all interrupts or input events that may be generated are disabled and do not affect the state of the system. Being in replay mode, the VM uses all information recorded in the log file to reproduce the events at the tick at which they happened during the log phase. Figure 2 illustrates how ExecRecorder works.

The log and replay framework also acted as an oracle in validating our characterization of architectural nondeterministic events. We have validated our characterization of nondeterminism by trying to remove each type of nondeterministic event and then replaying the system execution. The exclusion of a certain type of nondeterministic event would prevent a successful replay.

4.3.4 Multiprocessors and DMA Discussion

Our proof-of-concept implementation currently does not address multiprocessors and DMA. However, our approach can be extended, in principle, to include them. A first direction for this future work is to extend our VM to model the cache subsystem, DMA and the bus. From these models we can extend ExecRecorder to log DMA writes and the minimal subset of memory races according to the algorithm proposed in the FDR design [30].

5. Experimental Results

In our experiments we have analyzed, for Linux and Windows, the size of the log files generated by ExecRecorder and the log files' growth rate when we varied the workload in the guest OS. We have also analyzed the performance overhead incurred by the logging component. We have studied the system in the following situations: running a Web server which is receiving a burst of requests in a noisy campus network, executing intensively its CPU and disk, and running multitask activities, and idle.

We have selected a set of publicly available applications as our workloads. For each one of our experiments we ran each workload three times and averaged the results obtained. The workloads chosen for Linux and Windows were independent from one another because we have used publicly available workloads or benchmarks and, in general, they are developed for a specific OS. The experiments were executed on a Pentium 4 SMP with 2 3.2 GHz CPU's and 1 GB of RAM.

5.1 Linux

We have selected two workloads for our Linux 2.4.21 guest OS. The first one tests the system running the Apache Web server [4]. It generates, from an external network, a burst of 2000 requests to fetch a 3K html document.

The second workload was UnixBench [28], which is a benchmark suite for Linux that integrates CPU, file I/O, process spawning and other workloads. The following tests were performed: Dhrystone 2 using register variables, arithmetic, system call overhead, pipe throughput, pipe-based context switching, process creation, execl throughput, file system throughput, concurrent shell scripts, compiler throughput, and recursion.

5.2 Windows XP

We have selected three workloads for Windows. The first is the same used to test Linux as a Web server [4]. We tested the Apache Web server in Windows by generating 200 requests to fetch a 3K html document. The requests were also generated from an external network, where, in this case, we have generated one request per second. We have inserted a light load in our Web server for Windows, because it could not handle well more than 200 HTTP requests per second.

The second workload was Microsoft SQLIO [26], a disk subsystem benchmark tool. It generates disk workload so as to simulate aspects of the I/O workload of the Microsoft SQL Server. In our tests, we had one thread reading for approximately two minutes from a file using 2 KB IO's over 128 KB stripes with 64 IO's per run.

The third workload was an implementation of the Sieve of Erastosthenes. Our goal was to generate a CPU-intensive workload. We have chosen to use this algorithm not only because it is usually part of several well-known CPU benchmarks, but also because publicly available CPU benchmarks for Windows were interactive and we did not want the user response time to influence our results.

5.3 Results

Figure 3 shows how the size of our log files varied for each considered workload for Linux and Windows, and Figure 4 presents the corresponding log file growth rate in GB/hour.

Although our choice of applications does not represent a characterization of a certain type of workload, we observe that I/O-intensive applications, especially those that extensively use the HD,

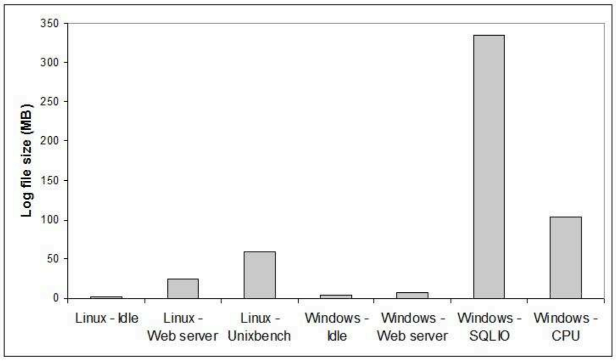

Figure 3. Log size (MB) for different workloads - Linux and Windows.

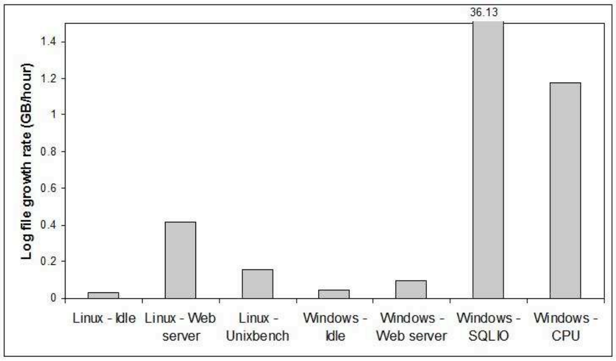

Figure 4. Log file growth rate.

tend to have a larger log file growth rate. Also, our results show that ExecRecorder is feasible and practical for different types of workloads provided that the frequency at which checkpoints are taken is chosen appropriately, considering the amount of disk space available for logging. As the cost of HD's is relatively low, checkpoints can be taken every hour, twice a day or every day, depending on the demand of the application. Although the redolog file (Section 4.3.1) should count as part of the non-volatile storage necessary, we did not consider it as part of the ever-growing log. This is because the redolog file can be at most the size of the original HD no matter how long we run a benchmark.

Figure 5 illustrates the performance overhead of the logging component for our selected workloads. For all cases the overhead due to logging is low (less than 4% on average). We have not shown performance results during replay because according to Elnozahy and Alvisi [11], it has been observed that in replay mode the system can run considerably faster than in normal execution. During normal execution a process may block waiting for I/O events while during replay all events can be immediately replayed.

6. Post-Attack Analysis

Here we describe a practical example of using ExecRecorder to perform post-attack analysis. In this experiment we have Minos as our IDS and DACODA as our analysis tool, according to the architecture shown in Figure 1.

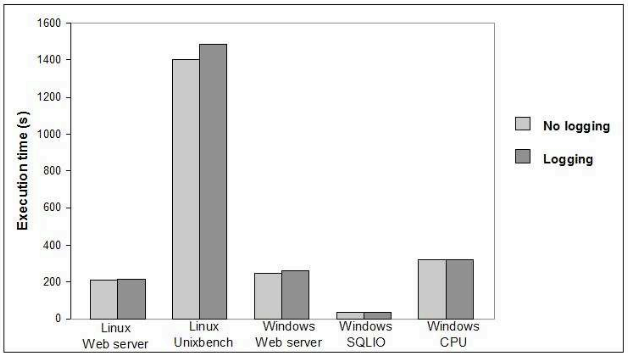

Figure 5. Performance overhead due to logging - Linux and Windows.

Minos is a security-enhanced microarchitecture that prevents attacks that hijack program control flow. Every 32-bit word of memory and every 32-bit general purpose register in the x86 architecture is augmented with one tag bit which represents the integrity level of this word (zero meaning low integrity and one high integrity). This bit is set by the kernel based on the trust it has for the data. The basic assumption is that any control transfer (instructions such as jump, call, and return) involving untrusted data is a system vulnerability and a hardware exception traps to the kernel whenever this occurs.

DACODA is a tool that analyzes attacks using symbolic execution. It labels each byte coming from the network with a unique identifier and tracks these bytes in the system during their lifetime. When Minos catches an attack, DACODA provides information about it, such as processes involved, if the attack involved kernel or user processes, tokens that compose the attack trace and the predicates found. A limitation of this tool is the performance overhead it incurs, because for each instruction executed, DACODA has to perform symbolic execution. This overhead is exacerbated for exploits that require considerable amount of computation such as Code Red II and ASN.1. IntroVirt [15] also uses predicates to detect intrusions. The difference between the two is their goals. IntroVirt checks if a system has been exploited in the period between vulnerability discovery and patch release, while DACODA analyzes and generates signatures for zero-day exploits.

ExecRecorder, Minos and DACODA currently run as extensions to Bochs. To integrate ExecRecorder with them we just need to log and process more information. For Minos we have to log the integrity bit of every word transferred in input events and for DACODA we have to log all incoming network packets because their bytes need to be labeled in the order that they were received by the network card and not in the order delivered to the CPU (the bytes are usually reordered in the network card).

A solution is to turn off DACODA in our honeypot and only execute it off-line using ExecRecorder. Our honeypot executes Minos along with ExecRecorder in log mode, which incurs very low performance overhead. When Minos catches an attack, we use the log file generated since the last checkpoint and analyze the attack off-line with DACODA.

Here we analyze, for an exploit of the wu-ftpd 2.6.0 vulnerability [1], the size of the log file generated and the execution time of the attack in three situations: when it executes with only Minos on, when it executes with Minos and ExecRecorder in log mode, and when it executes with Minos and DACODA on. Figure 6 shows the exploit execution time for these three situations. The log file for the exploit is 1.769 MB.

DACODA provided us with the following information about this attack: (1) it has a total of 2888 predicates and all of them were found in user space, (2) the process involved is wu-ftpd, (3) the longest signature token has 283 bytes, and (4) the token length histogram as "Number(size in bytes)" is 4(283),4(119),4(11),1(10),1(9),1(6),4(5),3(4),4(3),10(2),41(1).

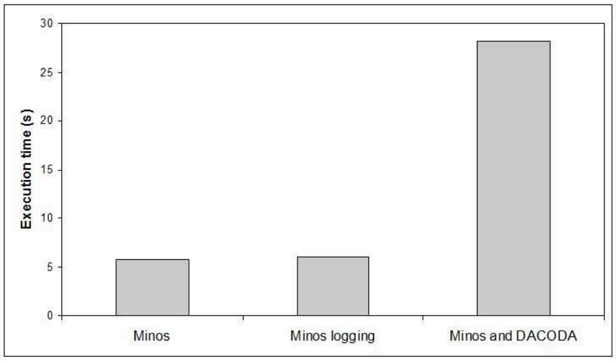

Figure 6. Execution times for wu-ftpd 2.6.0 exploit.

7. Discussion

In this work we presented ExecRecorder, a full-system, VM-based log and replay framework for uniprocessors to perform post-attack analysis and recovery. It addresses the limitations found in current replay systems by providing full-system replay capabilities and low performance overhead without requiring any OS changes.

The lessons we have learned can be summarized as follows. We can considerably decrease the amount of logged data by recording the tick or time difference of an event and the last one, instead of its absolute timing value. Also, although an input event can be characterized by its timing and bytes only, a replayer usually will need information about the input instruction itself to correctly reproduce the event. This extra information will vary depending on the system architecture or the VM implementation. Input events from the HD are deterministic but we still need to log information about their associated input instruction to synchronize the event with its corresponding interrupt. The HD bytes, however, do not need to be logged.

As future work, we intend to extend ExecRecorder for multiprocessors and DMA use and to improve it by allowing checkpoints to be saved in a non-volatile medium. Also we plan to implement a post-attack recovery strategy using ExecRecorder, our IDS, Minos, and our analysis tool, DACODA. We are also using ExecRecorder to analyze covert channels through repeated replays with varying confidential data.

8. Acknowledgments

This work has been supported by grants 0335299, 0520269, 0627749 from NSF. We would like to thank the anonymous reviewers for their helpful comments and the developers of the Bochs project.

References

[1] http://x82.inetcop.org/h0me/papers/free-ur-mind.pdf.

[2] L. Alvisi. *Understanding the Message Logging Paradigm for Masking Process Crashes*. PhD thesis, Cornell University, 1996.

[3] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, Causal, and Optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, February 1998.

[4] Web benchmark. `http://www.serverwatch.com/news/article.php/10824_1133391_2`.

[5] T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault Tolerance. *ACM TOCS*, 14(1):80–107, February 1996.

[6] P. M. Chen and B. D. Noble. When Virtual is Better than Real. *HotOS*, May 2001.

[7] J.-D. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. *ACM SIGMETRICS SPDT*, pages 48–59, August 1998.

[8] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. *MICRO*, pages 221–232, December 2004.

[9] J. R. Crandall, Z. Su, S. F. Wu, and F. T. Chong. On Deriving Unknown Vulnerabilities from Zero-Day Polymorphic and Metamorphic Worm Exploits. *ACM CCS*, pages 235–248, November 2005.

[10] 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. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[11] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *University of Michigan Technical Report CSE-TR-410*, 34(3):375–408, September 2002.

[12] T. Garfinkel and M. Rosenblum. When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments. *HotOS*, June 2005.

[13] Z. Gutterman and B. Pinkas. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. *ISCA-32*, pages 284–295, June 2005.

[14] Intel. IA-32 Intel Architecture Software Developer's Manual. Volumes 1, 2 and 3.

[15] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting Past and Present Intrusions through Vulnerability-specific Predicates. *ACM SOSP*, pages 91–104, October 2005.

[16] S. T. King, G. W. Dunlap, and P. M. Chen. Operating System Support for Virtual Machines. In *USENIX*, 2003.

[17] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, 36(4):471–482, April 1987.

[18] R. Love. *Linux Kernel Development*. 2005.

[19] D. E. Lowell and P. M. Chen. Discount Checking: Transparent, Low-Overhead Recovery for General Applications. *University of Michigan Technical Report CSE-TR-410-99*, 1998.

[20] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. *ISCA-30*, pages 110–121, June 2003.

[21] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating Bugs as Allergies—A Safe Method to Survive Software Failures. *ACM SOSP*, pages 235–248, October 2005.

[22] M. Rosenblum and T. Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *IEEE Computer Society*, 38(5):39–47, May 2005.

[23] J. Slye and E. Elnozahy. Supporting Nondeterministic Execution in Fault-Tolerant Systems. *FTCS*, 1996.

[24] J. H. Slye and E. N. Elnozahy. Support for Software Interrupts in Log-Based Rollback-Recovery. *IEEE Transactions on Computers*, 47(10):1113–1123, October 1998.

[25] J. E. Smith and R. Nair. *Virtual Machines - Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.

[26] Microsoft SQLIO. `http://www.microsoft.com/downloads/`.

[27] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. *USENIX*, June 2004.

[28] UnixBench. `http://www.tux.org/pub/tux/benchmarks/System/unixbench/`.

[29] A. Whitaker, R. S. Cox, M. Shaw, and S. D. Gribble. Rethinking the Design of Virtual Machine Monitors. *IEEE Computer*, 38(5):57–62, May 2005.

[30] M. Xu, R. Bodik, and M. D. Hil. A Flight Data Recorder for Enabling Full-System Multiprocessor Deterministic Replay. *ISCA-30*, pages 122–133, June 2003.

[31] bochs: the Open Source IA-32 Emulation Project (Home Page). `http://bochs.sourceforge.net`.