

藉由虛擬機器實現精確的系統執行狀態重播

FREE: A Fine-grain Replaying Executions by Using Emulation

許家維
交通大學資訊工程學系
Chia-Wei Hsu
Department of Computer Science
National Chiao Tung University
vicnobody.cs96g@nctu.edu.tw

謝續平
交通大學資訊工程學系
Shiuhpyng Shieh
Department of Computer Science
National Chiao Tung University
ssp@dsns.cs.nctu.edu.tw

Abstract

Replaying of execution sequence and state transition of a system is very useful for software testing, malware analysis and post-attack recovery. However, existing system logging and replaying techniques have restricted abilities and hence cannot be applied widely. Most of them are unable to perform a general whole-system analysis for the following reasons: 1) It can only replay a single process's running. 2) Modification needs to be done in OS kernel 3) Non-deterministic events such as interrupts and context switches cannot be replayed. 4) An intrusive analysis might influence the replaying result. This paper proposed a general whole-system VM-based logging and replaying mechanism. To record efficiently, our scheme only takes non-deterministic information into account such as most hardware interrupts and non-deterministic data from external I/O devices. Based on the recorded data, the accuracy of the replaying is assured. The state transition of the whole-system can be perfectly replayed; even the execution sequence of all instructions is preserved.

1. Introduction

Replaying of virtual machine execution [1-11] is very useful to be applied to software testing and

malware analysis. Many aspect of software testing make it become a complicated work because following reasons. First, non-deterministic event always occur in process execution such as hardware interrupts and software exceptions. Sometimes those non-deterministic events would not be recorded in system logs or debug messages. What's worse is that it is hard to reproduce these events in a short time, such as race conditions and deadlocks. In some cases, the event cannot be reproduced at all because it's timing-related. To review the information, replaying the whole-system is the only option we have. Second, intrusive analysis would affect its result. No matter how the analysis is performed, instruction must be injected, which would always affect timing, even scheduling. Virtual machines could help us analyze the target process non-intrusively.

Malware analysis is another application of replaying execution [2, 5]. For example, some malwares would only be activated at a specific instant. After that, they may delete themselves to erase evidence of existence. Mostly, people analyze the malware after attacks. With replaying ability, programmer can analyze the attack process of malware on the fly. Some powerful analyzers use emulation, which require a large amount of computation. The overhead makes the analysis environment different from the real world.

Consequently, we need a transparent replaying tool for malware analysis.

Replaying has been drawing much attention due to its applicability. Numerous works has been done already. However, we believed that a successful replaying tool should provide following functionalities.

Whole-system recording & replaying

Conventional execution replaying techniques are defective on account of whole-system replaying. Some works [12] record system calls history for replaying and ignore hardware interrupts. Thus they are heavily operating system dependent. It is a cumbersome work to maintain different recorder for different versions of system calls. There were also works which focus on replaying a single process by recording reads from IPC pipes, files, and all system calls. However, in modern software design a process may interact with numerous processes, libraries, or modules to accomplish its task. Without whole-system replaying mechanism, we can only acquire limited information. It may perform whole-system replaying by replaying all of them simultaneously; however the interaction and synchronization would be cumbersome.

Another reason for taking hardware interrupts into consideration is correctness. Without hardware interrupts recorded, the dependency of threads would be impossible to be replayed perfectly. Besides, some of instructions can return non-deterministic or timing-dependent values directly without invoking system calls such as rdtsc. Neither system call trapping nor single process replaying can replay such execution.

Non-intrusive recording

Recording of execution sequence causes reasonable time and space overhead. Hence, intrusive system

recording [1, 8] would impact on results of execution dramatically. To avoid unnecessary influence, we need to place recording component outside the system. VM-based recording could achieve this goal with little performance overhead. Besides, a virtual machine gives us full control over the whole-system; therefore, timing inaccuracy caused by recorder can be eliminated. Another advantage of using a virtual machine is that we need only record interrupts and external I/O of VM, since does not all internal events are deterministic.

Replaying correctness

To perform a deterministic replay with correct sequence of execution, all inputs from external devices and unpredictable hardware interrupts must be recorded since they are non-deterministic factors. All non-deterministic events and their timing should be carefully logged, so we can replay them at the exact instant as in previous execution. By doing this, there is no non-deterministic factor in the replaying process, so that replaying correctness could be assured.

In this paper, we proposed a whole-system, transparent and deterministic replaying system, which satisfy requirements listed above. It can replay the whole-system states instruction-by-instruction. Besides, we have a deterministic replay because of recording non-deterministic event in our logs. We implemented our system in QEMU [13], a faster dynamically binary translator, and can be used to integrate analysis tool for many purposes. We can replay the transition of whole-system states with low space and compute overhead. Simultaneously, it is a transparent replay without any non-deterministic events.

The rest of paper is organized as follow. Section 2 gives introduction to related works. In section 3 we formalize the replaying problem definition.

Architecture of our system and implementation details are given in Section 4. Section 5 is experiences to evaluate our replay system. Some possible future works are shown in section 6. Finally, we conclude our work in section 7.

2. Related Work

Rollback system recovery and VM-based log analyzer are very popular in recent years. With zero-day attack raise, computer security faces an unprecedented challenge. Virtual machine can provide an isolated environment for analyzing malware. It is also used for standalone system debugging as honeypot.

Jockey [12] is a record and replay tool for single process debugging in Linux. It does not need to change the target binary and any programming language or API. It can record the non-deterministic data by pre-loading as module. Jockey segregates resource of target program for avoiding being compromised. But it cannot record hardware level non-deterministic event such as memory access races, thread scheduling interrupt and interrupts. Jockey only focus on one simply process debugging but cumbersome whole-system debugging.

Flashback [14] provides rollback and replay ability for software debugging. It forks a new process as shadow processes for checkpoint. Then it captures the memory state at specific execution point and interaction between processes. Nevertheless, Flashback cannot replay thread dependency correctly because it is hard to trap the interrupt of thread scheduling. Without the correctly context switching, the dependency of thread would be change at re-execution. Thus, even Flashback is a lightweight replay for software debugging; it cannot debug some problems with hardware interrupts.

Revirt [8] logs enough information for replay even in long-term system. Because of based on virtual-machine monitor (VMM), it needs to modify the kernels of guest OS. Revirt consist of two parties to monitor, one is guest user, and another is guest kernel. Both of them are building on host system as processes. By delivering signal SIGUSR1, the guest kernel can trap the system called by guest user. Additionally, it records non-deterministic events to follow a set pattern by using SIGIO and SIGSEGV. Revirt also replace some instruction can return non-deterministic results. Specifically, the rdtsc (read timestamp counter) and rdpmc (read performance monitoring counter) get CPU's information directly. It replaces that functionality by using other time-related system call. Thus the environment of whole-system would be some differences from real.

XenLR [11] is achieved on a lightweight VM (Mini OS) replay. It causes a little time and space overhead to log the keystroke and time updating on Mini OS. XenLR do not think about file system and process interaction because of Xen, a vitalization VM so that many non-deterministic events cannot be capture. But in real system, the file system and threads are usually an essential part of the system.

Based on Bochs, ExecRecorder [5] perform hardware interrupts and whole-system replay. It can replay the executions of entire system by checkpoints and logs of non-deterministic events. A checkpoint is a duplicate of Bochs VM process via the fork system call. When replaying the system, ExecRecorder invokes the suspended child process by SIGUSR1. Same as our system, the implementation of ExecRecorder does not address DMA and multiprocessors. But Bochs has heavy computation overhead in emulation so that it is hard to be applied in large-scale analysis. Moreover,

they do not mention about the consistency of instruction sequence between recording and replaying.

3. Problem Definition

We construct formal model for replaying system as following. A machine state s is a set of states of register bank r , memory region m , and hardware disk content hd . Let's denote $s = \{r, m, hd\}$. We state that $s = s'$ if and only if s and s' have same values in their register bank, memory and hard disk. Using the above definition of s , the state of a virtual machine M at the time i could be denoted as s_{Mi} . A special state s_{M0} stands for the initial system state. After executing n serious instructions, the trace of the state transition can be symbolized as $S_M = \{s_{M0}, s_{M1}, \dots, s_{Mn}\}$. Now, we can formally define the replaying problem:

Given M 's trace of the state transition $S_M = \{s_{M0}, s_{M1}, \dots, s_{Mn}\}$, we say that machine N **replays** M 's execution S_M if and only if $s_{Mn} = s_{Nn}$. Note that above definition does not require $s_{Mi} = s_{Ni}$ for all $1 \leq i \leq n$. However, we believed that $s_{Mn} = s_{Nn}$ is sufficient to imply that in most cases.

Obviously, s_0 is the only information required to replay a system M on N without any non-determinism. Unfortunately, it is not the common case. A useful computational system always interacts with external data and commands. To perfectly replay such a system with non-deterministic inputs, we need this information, too. Let's denote all non-deterministic inputs happened during M 's execution as I_M .

In this work, we tried to implement a system which can:

- (1) Record I_M during M 's execution.
- (2) Replay M 's execution S_M on N by feeding N with s_{M0} and I_M .

4. System Overview

In this chapter, we introduce kinds of non-deterministic events would be introduced following. Finally, we divide our system design into two components such as record and replay component.

4.1. Non-determinism

A non-deterministic event is one of the reasons for transiting system states. In our system, we considerate all the deterministic event, or you can call them arithmetic, are in the black box instead of non-deterministic event. The states of external devices, such as hardware disk, network card, keyboard and mouse, can be ignore because they are too complicated to synchronize their states as previous execution. However, it is also impossible to re-generate a real hardware signal during replay. Thus we record the data in VM hardware simulation, and then reply the return data at exact time during CPU emulation. By mentioned restrictions above, our system only cares about hardware interrupt, port I/O, memory-mapped I/O and DMA (direct memory access). We discuss in details following.

4.1.1. Hardware Interrupt

Computers need interrupts to communicate with each external device. Numerous tasks are accomplished by them. For example, context-switching, disk I/O, DMA data transfer, etc. Interrupting is a way to disturb CPU execution, and hence becomes a source which generates non-deterministic events. However, not all of them are non-deterministic. For example, exceptions and software interrupts could never be non-deterministic. Exceptions are generated by deterministic execution of instructions existing in memory which remains constant before any hardware

interrupt. A software interrupt can only be generated by instructions existing in memory, which is internal data storage. In other words, we can ignore all the exceptions and software interrupts and reduce time and space usage when recording.

QEMU translates the binary to translate blocks for execution. It would not be interrupted at anytime, so it checks whether interrupts are peddling or not before execution function. We modify this function for recording our interrupt, including instruction counter, which represents how many instructions are executed, interrupt number, the number indicate what should the PIC (programmable interrupt controller) handled.

4.1.2. External Input

An external input event is one of the reasons for system state transitions. In our system, we only take account of the difference of execution between runs and record factors of change. However, system transition is arithmetic event so that the identical inputs will result in the same outputs. Thus we do not considerate all the outputs of every devices. Instead, we only care about external inputs which come from any simulated devices. The states of devices outside VM, such as hardware disk, network card, keyboard and mouse, can be ignore because they are too complicated to synchronize when replaying. For example, we can simply generate a keyboard interrupt in VM without synchronizing the state of real keyboard. It has no influence if we do not synchronize their states. Therefore, we record all non-deterministic events during original execution, and then reply them at exact time. Our system only takes hardware interrupts, port I/O, memory-mapped I/O and DMA

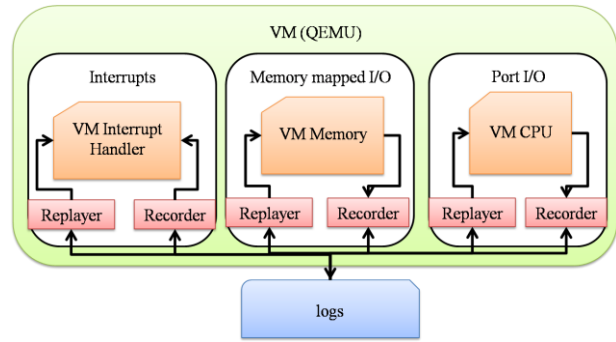


Figure 1 Implementation of our replay system

(direct memory access) into account to record and replay efficiently.

Port I/O and memory-mapped IO are the same concept of the system inputs. Almost non-deterministic inputs of the system are from user or other computer. For example, a user types commands as input or receives a packet from other computer. Programs cannot decide when those data coming, this is a reason that external inputs are non-deterministic.

DMA is another non-deterministic event in our system. DMA allows devices within the computer to access system memory independently of the CPU. We have to handle with this, because it would affect the integrity of data for read and write. This event is very difficult to synchronize between original executions and replay of recorded. When DMA happening, QEMU would read host disk for simulating the guest system's disk. In the real world, it is unlikely for accessing disk in the same time. Therefore, we do not address this problem into our implementation.

4.2. System Design

In our system design, there are two components compose system replay ability record and replay. Following are our descriptions in detail.

4.2.1. Record

The record component logs the entire non-deterministic event into files. The component is made up many modification of QEMU. This part would record every non-deterministic event with corresponding instruction counter. We segregate this component into many simulated I/O or interrupt functions. When these I/O function or interrupts taking place, we check the VM system state whether is recording or replaying. If the state of VM is recording, we will record current data from I/O port or interrupt handler by our record function.

Moreover, there are many data type such as hardware interrupts, port I/O and memory-mapped I/O in non-deterministic events. In order to replay correctly and efficiently, we also record additional information corresponding to each data type. For example, the IA-32 we have input instructions to transfer a byte, word or a double words.

Instruction counter is very important for recording. It counts how many instructions are the CPU computed. Otherwise, it stands for the related timestamp. Not only we can replay the instruction transparency, but also the virtual related time. It is very significant for some time dependent event such as rdtsc, rdpmc. We combine all above information for recording, a very simple format for decreasing record space.

4.2.2. Replay

We implemented transparent and deterministic replay in our system. All of the recorded data are corresponding to instruction counter for ensuring time dependency. Replay of the system state will be the same as previous recorded, including thread scheduling, interactions of process, hardware interrupts from external device and even data of packets from network.

But we do not take the transparency replay of external devices states into account. One of the reasons, it's hard and also impossible for synchronizing their states. For example, you can regenerate a packet from a website, but change the website status to send the packet like previously. In other words, states of external devices are not in our concern so that their states would not be transparent.

Non-deterministic events are the determinants for achieving deterministic replay. A non-deterministic event causes a state transition without corresponding to previous states. By checking instruction counter, system states can be replayed predictably because of removing the entire non-deterministic factor. However, we let all the events be deterministic.

5. Evaluation

Our replay system can be evaluated by two aspects; one is correctness, another is performance. First of all, we describe our experiment environment and design in detail. Secondly, correctness of system replaying can be verified by multi-threads scheduling. Finally, we evaluate the space overhead of recorded data.

5.1. Evaluation Method

To evaluate our system, we load a VM image of Ubuntu 9.04 which is one of popular Linux distributions for our experiment environment. It provides completely functionality for normal users, so our experiment result can be very close to real world system replaying. Besides, Ubuntu is convenient to evaluate replaying correctness, and replaying common operating system is more convinced than only replaying specified or small linux system. Our replay system only connects keyboard and mouse. The host

system runs on Intel 2.5 GHz Dual core and 8GB of RAM.

To evaluate correctness of system replaying, we design two experiments to confirm that instruction dependency can be replayed. We write a testing program and the outputs will be delivered to host system. After replaying, guest system will create another copy in host system. By doing this, we can compare the content of two logs to ensure correctness of replaying. Simultaneously, we monitor the increase of log size and set host timer to calculate how much

```

1. void *producer( void *arg){
2.     while( go == 0 );
3.     item++;
4. }
5. void* consumer(void * arg){
6.     while( go == 0 );
7.     item--;
8. }
9. int main()
10. {
11.     Initialize();
12.     for (i=0; i<n; i++){
13.         my_thread_create( producer, i);
14.         my_thread_create( consumer, i);
15.     }
16.     OtherCode();
17. }

```

Figure 3 Example of Producer-Consumer problem

time does the experiment take. We evaluate correctness, space-overhead and computation –overhead in detail following.

5.2. Correctness Evaluation

For general purpose, our replay system can replay many Linux command such as *ps* (lists all the active processes state), *ls* (lists directory content) and *date* (shows the system time). To replaying the foregoing commands, we must replay correct interrupts, user behavior, external inputs and time-related event (rdtsc). However, those events are instruction-dependency senseless that cannot highlight the accuracy of our

replay system. Thus we design further complicated evaluation for instruction-level replaying.

Instruction dependency can be reduced into multi-threads dependency. Because of thread is the smallest computing unit in most of operating system. Our system faithfully replays the timing of context-switch, and feeds recorded input value for correct condition branch. Therefore, we will have the same instruction dependency as recording runs.

We implemented two experiment of multi-thread competition. Firstly, printing TID (thread identity) is a

Computation overhead related to native emulated VM	
Small Linux	2.2x ~ 2.5x
Windows keystroke	2.5x ~ 3.5x
Ubuntu	2.5x ~ 3.5x

Figure 2 Computation overhead during replaying

simply way to show the multi-threads ordering. Because it is hard to predict which thread will be selected, the sequence of TID is non-deterministic in real system. The possibility of re-generating the sequence is $1/n!$, in other words, it is impossible to create two identical sequence when n is a large number. In our system, all of non-deterministic events are recorded including context-switch, thus we can replay the scheduling problem even with 100 threads.

Another experiment is thread race-condition. The different instruction dependency will cause different result of access competition. Figure 2 is producer-consumer problem example code. Producers increase the share instance, and consumers decrease it. All of created threads modify the same variable item. Because it cannot be guaranteed that access variable item is atomic, some modification failures are due to other threads overriding. This experiment is repeated many

times with 40 threads and save those output into log. Then we confirm whether the output is matched during replaying.

5.3. Performance & Storage

Our replay system takes a few space-overhead and reasonable computation-overhead. Space-overhead of log are very small because of selective recording. Our replay system does not record all the executed instruction and all the I/O. We only take non-deterministic events into account for efficient recording. The log size average 350KB per minute; on the other hand, the recorded data increases 5.3 bytes per thousand of instructions.

The computation-overhead depends on the complexity of guest OS. We randomly execute linux command on small linux guest system, and execution time will cost about double times than native VM. Additionally, we also replay user behavior on Windows XP guest system, but the performance is not as good as small linux. The reason of computation slow down is that our replay system executes many condition branch and instructions. To count computed instructions, every instruction on guest system need to append an addition to instruction counter. Otherwise, the number of non-deterministic events will affect the replaying performance. The more data is recorded, the slower replaying is.

6. Future Work

Our system is extendable for replay of DMA and malware analysis. Even replay of DMA is a difficult problem, but it still has a solution for replaying. We can halt the VM for waiting the host system data access finish. By doing this, guest system can access the data correctly to keep system states in a good shape.

Additionally, we plan to integrate a security analyzer in our system for security analysis tool such as dynamically taint analysis or intrusion detection system. It would be helpful for any kind of expensive compute consumption analysis.

7. Conclusion

Replaying execution sequence and recording the state transition of a system are very useful for software testing, malware analysis and post-attack recovery. Unfortunately, current approaches cannot provide complete replaying functionalities therefore they cannot be applied widely. Whole-system replay can recover the complete situation of previous execution time. Non-intrusive recording can prevent the result from being interfered. Replaying correctly can guarantee that the sequence of executions is identical with that of the previous execution.

We implement a system based on QEMU for transparent and deterministic whole-system replay. Moreover, we can save the space of logs by only caring about the impacts of non-deterministic. The sequence of system replay must be deterministic if every factor of states transition is predictable. Our system is extendable by integrating other security analysis tool, thus enables its wide application in the future.

8. Acknowledgement

This work was supported in part by TRUST Center of UC Berkeley, ITRI, Institute for Information Industry, Chung Shan Institute of Science and Technology, Chunghwa Telecomm., Investigation Bureau, Dlink, the International Collaboration for Advancing Security Technology (iCAST) and Taiwan Information Security Center (TWISC), respectively.

9. References

- [1] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging Operating Systems with Time-Traveling Virtual Machines," 2005, pp. 1--15.
- [2] Jim, "Decoupling Dynamic Program Analysis from Execution in Virtual Environments," in USENIX '08, 2008, pp. 1--14.
- [3] P. Montesinos, L. Ceze, and J. Torrellas, "DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently," in ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture, Washington, DC, USA, 2008, pp. 289--300.
- [4] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in CCS '08: Proceedings of the 15th ACM conference on Computer and communications security, New York, NY, USA, 2008, pp. 51--62.
- [5] D. A. S. de Oliveira, J. R. Crandall, G. Wassermann, S. F. Wu, Z. Su, and F. T. Chong, "ExecRecorder: VM-based full-system replay for attack analysis and system recovery," 2006.
- [6] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution replay of multiprocessor virtual machines," in VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, New York, NY, USA, 2008, pp. 121--130.
- [7] M. Xu, R. Bodik, and M. D. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," in ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture, New York, NY, USA, 2003, pp. 122--135.
- [8] 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., vol. 36, pp. 211--224, 2002.
- [9] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan, "Rx: Treating bugs as allergies---a safe method to survive software failures," ACM Trans. Comput. Syst., vol. 25, p. 7, 2007.
- [10] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou, "Triage: diagnosing production run failures at the user's site," in SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, New York, NY, USA, 2007, pp. 131--144.
- [11] H. Liu, H. Jin, X. Liao, and Z. Pan, "XenLR: Xen-based Logging for Deterministic Replay," in FCST '08: Proceedings of the 2008 Japan-China Joint Workshop on Frontier of Computer Science and Technology, Washington, DC, USA, 2008, pp. 149--154.
- [12] S. Yasushi, "Jockey: a user-space library for record-replay debugging," in Proceedings of the sixth international symposium on Automated analysis-driven debugging Monterey, California, USA: ACM, 2005.
- [13] F. Bellard, "QEMU, a fast and portable dynamic translator," in ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference, Berkeley, CA, USA, 2005, pp. 41--41.
- [14] Sudarshan, C. R. Ula, and Y. Zhou, "Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging," 2004.