Analysis of Virtual Machine Record and Replay for Trustworthy Computing

Julian B. Grizzard and Ryan W. Gardner

Many effective techniques for defending against computer attacks are impractical because they would significantly impair application performance. Enforcing a property called control-flow integrity, for example, can ensure that a program's execution does not diverge from a specific, predetermined set of paths, but doing so diverts computational resources from the program as it runs. To enable practical uses of these techniques, we have implemented the virtual machine record and replay (RnR) prototype for modern x86 computers. RnR separates live execution from analysis by recording an executing virtual machine at speed and conducting computationally intensive analysis separately on a replay of the virtual machine. Our findings are that the performance overhead of the recording mechanism is minimal (less than 5% for common workloads) and that, therefore, the recording mechanism provides a resource-friendly way to deploy previously impractical techniques.

INTRODUCTION

The goal of trustworthy computing is to build systems that do exactly what they are supposed to do and nothing else, but the complexity of modern software systems makes this a very difficult task. One way to address this challenge is to monitor execution at runtime. Essentially, a relatively simple monitor can analyze a more complex program as it executes to make sure some predetermined property holds. For example, a monitor could verify that each control-flow transfer of a program does not diverge from a predetermined control-flow graph (a property known as control-flow integrity).[1]

Computationally expensive runtime analyses add an unacceptable performance penalty for most applications. However, using a technique called record and replay, a program can be recorded at speed and then analyzed in the background with great precision without slowing down the live execution.[2] Record and replay was first applied to debugging,[3] but more recent work has suggested

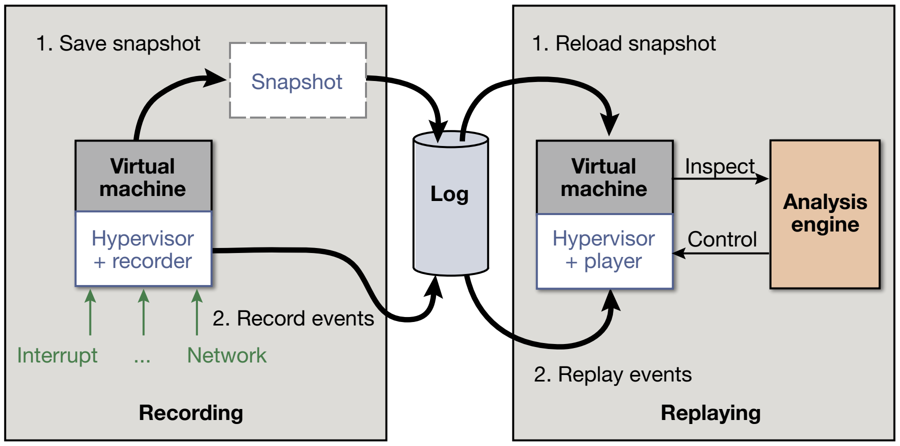


Figure 1. To separate execution from analysis, RnR records execution of a virtual machine at speed as illustrated on the left. Then, RnR replays the recorded execution for offline, detailed analysis as illustrated on the right.

that it be used for intrusion detection and analysis.[2,4] Although previous results are promising, researchers do not have access to a reliable, open, and extensible record and replay prototype to use to experiment with new types of analyses. To further research in this area, we have implemented the virtual machine record and replay (RnR) prototype as illustrated in Fig. 1 by extending the open-source Kernel-based Virtual Machine (KVM)/Quick EMUlator (QEMU) virtualization software.[5,6]

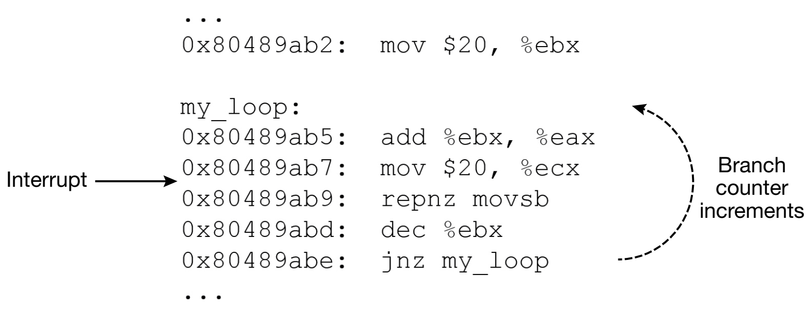
The key questions we seek to answer are as follows: How practical is it to record and replay a virtual machine? How trustworthy is the record and replay mechanism itself? What types of trustworthy computing applications and technologies are enabled by RnR? To answer these questions, we present an overview of the RnR technique and implementation (the *Record and Replay Technique* section), provide an analysis of the overhead of RnR (the *RnR Experimental Results* section), and discuss possible applications of RnR using previously impractical defensive techniques (the *Trustworthy Computing Applications* section). Finally, we outline remaining challenges to overcome and summarize our findings (the *RnR Challenges and Future Work* and *Summary* sections).

RECORD AND REPLAY TECHNIQUE

The basic record and replay technique records all activity in a virtual machine so that it can be replayed and analyzed in detail offline. The recording is intended to be precise enough to recreate every state in the recorded system. To use RnR in production systems, however, it is critical to maximize efficiency.

The key observation behind efficient record and replay is that most computer operations are deterministic.[3] Given an initial state, a computer executes a predictable sequence of instructions until some unpredictable event occurs (such as when an external interrupt occurs). Consider the example sequence of x86 software instructions in Fig. 2, but assume that no interrupts occur. These instructions begin with a `mov` at memory address `0x80489ab2` that initializes the value of the register `ebx` to 20. Next, a short loop (`my_loop`) adds the value of `ebx` into another register `eax` (instruction `0x80489ab5`), copies data from one fixed location in memory to another (instructions `0x80489ab7` and `0x80489ab9`), decrements the value of `ebx` by one (instruction `0x80489abd`), and then repeats the loop if the value of `ebx` is not zero (instruction `0x80489abe`). This instruction sequence is deterministic given the initial state, assuming no interrupts occur. Therefore, to replay the instructions, in this case, we only need to know the initial state of the system. This observation is very powerful when there are hundreds of thousands of sequential instructions that can be replayed from an initial state.

Of course, not all execution is deterministic. Instead, unpredictable events occur, such as input from the keyboard, network, and external interrupts. Consider the sequence of instructions in Fig. 2 again, but this time consider that an interrupt occurs (e.g., due to a key press) after three iterations of my_loop just before the *repnz* instruction is executed. The interrupt will divert the flow of execution temporarily to software known as the *interrupt handler* (not shown). The handler will service the interrupt and then resume execution of the loop. In this case, the sequence of instructions is not deterministic (which includes the interrupt handler


```
...
0x80489ab2:  mov $20, %ebx

my_loop:
0x80489ab5:  add %ebx, %eax
0x80489ab7:  mov $20, %ecx
0x80489ab9:  repnz movsb
0x80489abd:  dec %ebx
0x80489abe:  jnz my_loop
...
```

Figure 2. Example sequence of instructions executed in an x86 computer.

instructions) because the timing and type of interrupt are unknown. So, to record these instructions fully, one needs to capture the initial state and also record the interrupt event.

To generalize the record and replay concept, there are two important types of data to record: *initial state* and *nondeterministic events*. To record, the *recorder* must first capture the initial state of the *target* (software that is being recorded or replayed). Then, the recorder must log all nondeterministic events that occur as the target executes. To replay, the *player* must first restore the initial state. Then, as the target executes, the player replays the nondeterministic events at the appropriate points during the execution.

Virtual Machine Background

Figure 3a illustrates a model of *virtual machine* systems. The *hypervisor* instantiates one or more virtual machines on a single physical machine. Each virtual machine consists of a virtual central processing unit (CPU), virtual random access memory (RAM), and virtual devices so that an unmodified operating system and set of applications can execute inside the virtual machine as it would on a dedicated physical machine. In this way, physical resources of one machine can be shared by many virtual machines, maximizing resource utilization, decreasing costs, reducing size, and saving power.

We distinguish between the *host domain*, meaning executing within the context of the physical machine, and the *guest domain*, meaning executing within the context of a virtual machine. Software that manages virtual machines executes in the host domain (i.e., the hypervisor, drivers, and emulated hardware), whereas the operating system and applications that execute inside the virtual machine execute in the guest domain (the *guest operating system* and *guest applications*). The physical machine and software executing in it are collectively referred to as the *host*. The virtual machine and software executing in it are collectively referred to as the *guest*. There can be many separate guest domains but only one host domain per physical machine.

For our RnR implementation, we have modified KVM/QEMU (one of many virtual machine systems).[5,6] Figure 3b illustrates the basic architecture of the unmodified system (see the *Recording Virtual Machines* and *Replaying Virtual Machines* sections for details on our modifications). The system consists of a Linux kernel module called KVM, which provides an interface to CPU virtual machine extensions, and a user space process called QEMU, which emulates most of the virtual devices (i.e., network card, video card, hard disk, etc.). Both components execute in the host domain. A guest operating system and set of applications (not shown) execute in a guest domain.

Recording Virtual Machines

To record a virtual machine, we leverage existing functionality in KVM/QEMU to capture the virtual machine's initial state and add additional functionality to log nondeterministic events. The initial state includes the CPU registers, virtual memory, virtual hard disk, and other devices that are part of the virtual machine. The nondeterministic events are summarized in Table 1.

There are two classes of events that must be recorded: synchronous and asynchronous. Each synchronous event occurs at a point in "time" that is deterministic given all previous events, so we do not need to record the "time" for these events. However, each asynchronous event occurs at a point in time that is not deterministic, so we must log the time of the event. For both types of events, the data associated with the event are not deterministic, so they must be logged.

We use a point in the instruction sequence to identify time, which we refer to as *execution time* to distinguish

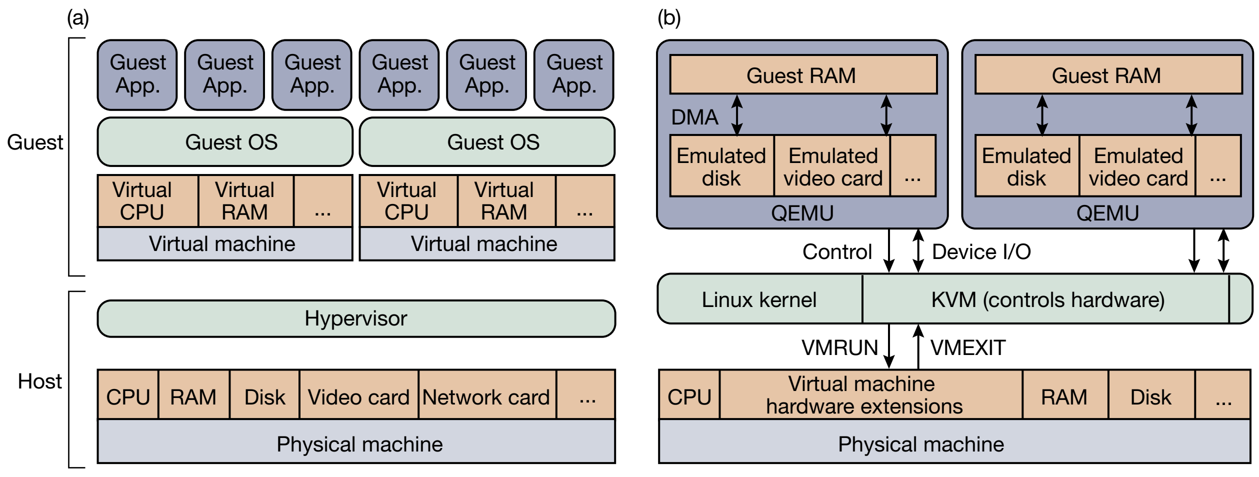


Figure 3. (a) Architecture model of virtual machine system versus (b) KVM/QEMU implementation. DMA, direct memory access; I/O, input/output; OS, operating system; VM, virtual machine.

Table 1. Types of events recorded and replayed

Type	Class	Data Size
Port input	Synchronous	1 or 4 bytes (data read)
Memory mapped input	Synchronous	4 bytes (e.g., advanced programmable interrupt controller data)
Rdtsc instruction	Synchronous	8 bytes (number of CPU clock cycles)
Virtual mouse input	Synchronous	8–48 bytes (e.g., mouse movement)
Interrupts	Asynchronous	1 byte (vector)
DMA	Asynchronous	Varies (e.g., network packets)

it from wall-clock time. We choose execution time as opposed to wall-clock time because it is not feasible to precisely identify a point in the instruction sequence based solely on wall-clock time (due to complexities of modern hardware such as memory caches). RnR logs execution time by recording the instruction pointer and the number of branches executed to account for executing the same instruction multiple times.

Each type of event listed in Table 1 has input data associated with it. The size of the data ranges from bytes to tens of bytes per event except for the direct memory access (DMA) events, which can be on the order of hundreds of bytes. This size of DMA data varies depending on the device. For example, the network card may copy 1500 bytes of data to main memory (length of a typical network packet) per event. The amount of data for any one event is small, which is why it is very efficient to record and replay virtual machines even if there are thousands of events per second.

Replaying Virtual Machines

To replay a virtual machine, RnR first loads the initial snapshot stored by the recording process and detaches the virtual machine from any virtual devices. Then, RnR starts re-executing the virtual CPU, allowing it to re-execute the same sequence of instructions executed while it was recorded up until the execution time of the first event. At that point, RnR reads the first event from the log and injects the input into the virtual machine. This process repeats for each sequential event in the log.

When a synchronous event occurs, such as execution of the *rdtsc* instruction, the virtual machine stops execution and transfers control to RnR. RnR determines why the execution stopped and reads the next event from the log to inject it into the virtual machine. For the *rdtsc* event, for example, RnR copies the value of the clock cycle count as recorded in the log into the virtual machine rather than using the current CPU clock cycle count.

RnR replays asynchronous events by first determining the execution time of the next event and then configuring the virtual machine to stop once it reaches that execution time. Once the virtual machine stops, data from the event are copied into the virtual machine. For example, if the event is DMA, then RnR copies the data from the log directly into the virtual machine's memory. After injecting the event, RnR resumes execution of the virtual machine.

Dynamic Analysis of Virtual Machine Replay

The value of RnR lies in detailed analysis of the replay. For example, consider a virtual machine equipped with RnR that has been recording all activity in the virtual machine for the past 24 h, during which time malware was installed on the system. One question a forensics investigator might ask is when was the malware first installed? From there, the investigator may want to know what the malware did. To answer the first question, the investigator could generate a log of all processes that executed by replaying execution and logging the time, name, and identification of each process. Then, the investigator could search the list of processes to determine the first point in time that the malware began executing. There may be other important things to analyze in the infected system, such as how the malware was installed, what data were accessed or tampered with, and so forth. All of the questions can be answered by analyzing the replay.

To perform analysis, RnR uses a technique known as virtual machine introspection (VMI) to read memory and CPU registers in the virtual machine as it replays. The VMI technique was first introduced as a way to inspect live execution of a virtual machine.[7] Essentially, VMI provides a way to interpret the state of the virtual machine at higher levels of abstraction (e.g., kernel data structures, processes, open files, etc.) at any point in time. For analysis in RnR, the same VMI technique is applied to a replay of the virtual machine. The advantage of this architecture is that live execution is unaffected by the performance overhead of any of our analysis techniques.

To generalize analysis, we have designed our architecture in a way that supports many different dynamic analysis techniques implemented as *Analysis Engines* (illustrated in Fig. 4). There are various components to the VMI application programming interface that provide details of what is happening in the replay with increasing levels of abstraction. For example, a forensics Analysis Engine may list all processes currently executing in the replay. The *Process VMI* subsystem in our VMI application programming interface layer provides this ability with a function called *list_all_processes()*.

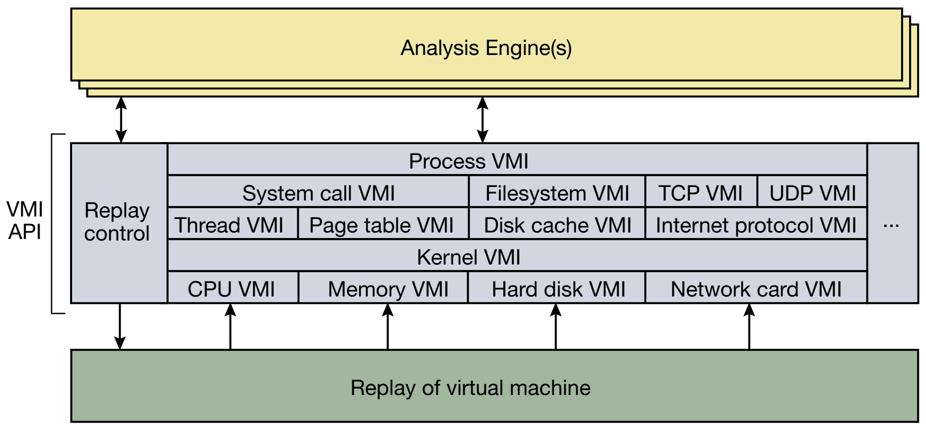


Figure 4. Software design for dynamic analysis of replay. An Analysis Engine controls replay of the virtual machine and inspects the replay using a common VMI application programming interface (API). TCP, Transmission Control Protocol; UDP, User Datagram Protocol.

Meanwhile, lower-level functionality could convert a guest virtual address to a guest physical address (i.e., the *rnr_guest_va_to_pa()* function).

RnR EXPERIMENTAL RESULTS

There are two important requirements of record and replay that we have stipulated to help determine how practical it is to record and replay a virtual machine. First, the performance overhead added by the recorder must be small. Second, the log growth rate must be small. This section details the results of our analysis for these two aspects of RnR.

Experimental Setup

To conduct performance experiments, we compared a virtual machine executing without our recording logic as a baseline (KVM/QEMU) with the same virtual machine extended with our recording logic (RnR). To perform the experiments, our test system consisted of an Intel Core 2 Duo CPU running at 3.00 GHz with 4 GB of RAM. All tests were conducted by installing a 64-bit version of Ubuntu 10.04 into the virtual machine. The virtual machine was configured with 256 MB of RAM and one CPU.

Performance Overhead

Figure 5 shows recording performance overhead of RnR by comparing the time to execute five different workloads in a baseline virtual machine (unmodified KVM/QEMU) with the time to execute the same workloads while recording them with RnR. For each experiment, our results are based on an average of ten runs of the experiment. The experiments and results were as follows:

- "Compute Pi": We installed a program in the guest system that would compute the first 2 million digits of pi. The performance overhead for this experiment was 1.4%.
- "Compile kernel": We copied the source code for a Linux kernel onto the experiment machine and compiled the kernel using a default configuration. The performance overhead for this experiment was 4.1%.
- "100 Mb/s Send": We installed an Apache web server in the guest system along with a web page consisting of ~8 MB of HTML pages and PDF files. From a computer connected by a 100-Mb/s network link, we ran a script to download the web page 1000 times with no delay. The performance overhead was 7.2%.
- "100 Mb/s Recv": We installed an Apache web server in a separate physical machine. The web server was configured as described for the previous experiment. The same script from the previous experiment was executed on the recorded machine instead of the secondary machine. The performance overhead was 7.9%.
- "100 Mb/s Recv bursty": We used the same setup as the "100 Mb/s Recv" experiment except that the download script was configured to download a web page 100 times with 1-s periodic idle periods. The performance overhead was 1.4%.

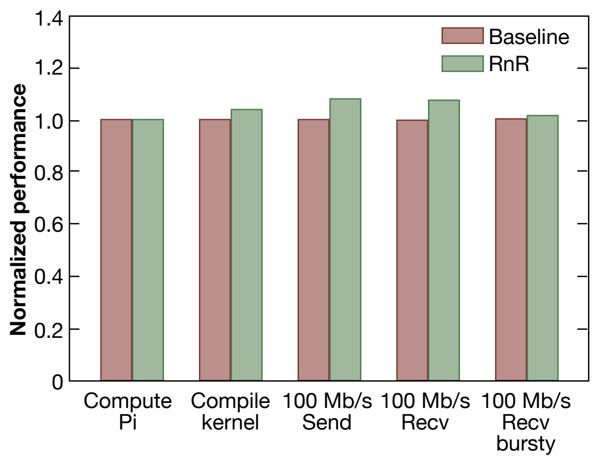


Figure 5. Performance overhead of KVM/QEMU baseline versus RnR for various workloads.

The experiments represent a CPU intensive test, a CPU and input/output intensive test, a network upload test, a network download test, and a simplified model of an aggressive, interactive web session. Only the "100 Mb/s Send" and "100 Mb/s Recv" workloads showed a performance overhead greater than 5% (7.9% and 7.2%). The other three workloads showed a performance overhead of less than 5%.

The experiments were designed to characterize the worst-case results for a range of common activities on desktop and server platforms. On the basis of these results, we expect that on average, the performance overhead will be less than 5% because the two network intensive workloads are not common in production systems. This indicates that the performance overhead of the recording mechanism is practical for many applications.

Log Growth Rate

Figure 6 shows the uncompressed log growth rates for four workloads. For these experiments, we selected a representative run and plotted the log growth rate over time. The log consists of "CPU" and "DEV" (i.e., device) data. The CPU data account for any data that are read from instructions executed on the CPU and data recorded for interrupts. The "DEV" data account for data directly copied to the virtual machine memory by a device. "DEV + CPU" is the summation of both and characterizes the total log growth rate. The experiments and results were as follows:

- "Idle": We recorded the virtual machine while it was idle for 5 min. The log growth rate was constant at 710 bytes per second.
- "Compute Pi": We recorded the virtual machine as it computed the digits of pi for 5 min. The log growth rate was 58 K/s.
- "Compile kernel": We recorded the virtual machine compiling a Linux kernel for 5 min. The log growth rate was 243 K/s. However, the log growth rate for just the CPU data was 61 K/s. In theory, only the CPU data must be recorded for this experiment, but our current RnR implementation does not support this functionality.
- "100 Mb/s Recv bursty": We recorded a virtual machine while downloading a web page with 1-s periodic idle periods for 5 min. The log growth rate was 1.23 MB/s.

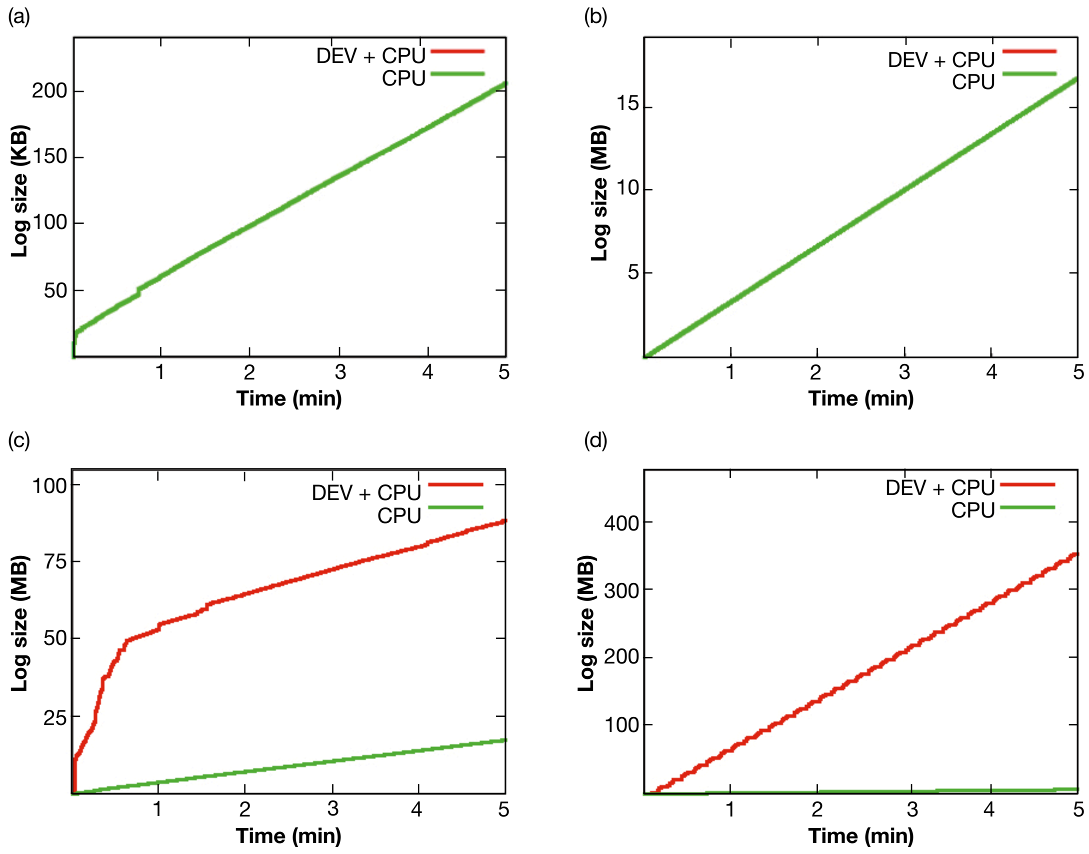


Figure 6. Uncompressed log growth rate of recorder for various workloads: (a) idle (CPU overlaps DEV + CPU); (b) compute Pi (CPU overlaps DEV + CPU); (c) compile kernel; and (d) 100 Mb/s Recv bursty.

These experiments were designed to characterize extreme workloads ranging from nearly no activity (i.e., the idle workload) to CPU, input/output, and network intensive workloads. For the network workload, the log growth rate was dominated by the DEV data, which consist mostly of inbound network packets. The results of this experiment indicate that log storage overhead is reasonable for many applications. However, for workloads that require excessive network input, more storage is necessary.

TRUSTWORTHY COMPUTING APPLICATIONS

To answer the question of what types of trustworthy computing applications are enabled by RnR, this section highlights a few potential technologies. As a first example, RnR could be used to record a highly detailed log of all activity in a host in support of a possible forensics investigation as illustrated in Fig. 7a. For this purpose, all application software in a host would be installed in a virtual machine that is equipped with our recorder. In the event of a compromise, a forensics investigator could analyze the log to determine exactly what happened and initiate appropriate recovery procedures.

As shown in Fig. 7b, RnR could also be used as an intrusion detection monitor—similar to Aftersight[2]—in a host. For this purpose, a monitor would operate in a separate process to perform near-real-time monitoring of applications executing in the host. As the host executes, the log of activity would immediately be sent to the separate process that analyzes the execution in the background.

As an example analysis based on the system illustrated by Fig. 7b, consider software written in the C programming language. One problem with the C programming language is that it lacks memory safety (a property that guarantees editing one variable will not corrupt another variable by going past the bounds of the first variable), which can lead to vulnerabilities. Variants of C, such as Cyclone, add extra robustness to the language, but the added assurance requires performance overhead of up to 60%.[8] As an alternative, RnR could be used to record the execution of software at speed and then analyze the execution in the background to verify that memory safety violations do not occur. Although analysis of the execution is slow, the analysis can "catch up" to the live execution during idle periods. There is a short window of risk until the background process detects a violation, but this greatly increases the difficulty of an effective attack.

More advanced systems are illustrated in Figs. 7c and 7d. Figure 7c illustrates a technique to parallelize analysis by sending the recorded event log to multiple analysis systems. Beyond attack detection, Fig. 7d illustrates a system that is capable of automatically detecting, diagnosing, and recovering from an attack. Additional research is necessary to realize this type of system, but RnR can potentially enable such techniques.

We have outlined out a few different applications of RnR in this section. However, there are potentially many areas of research that can build on RnR. We intend to explore these research areas and collaborate with others to expand the potential applications of RnR.

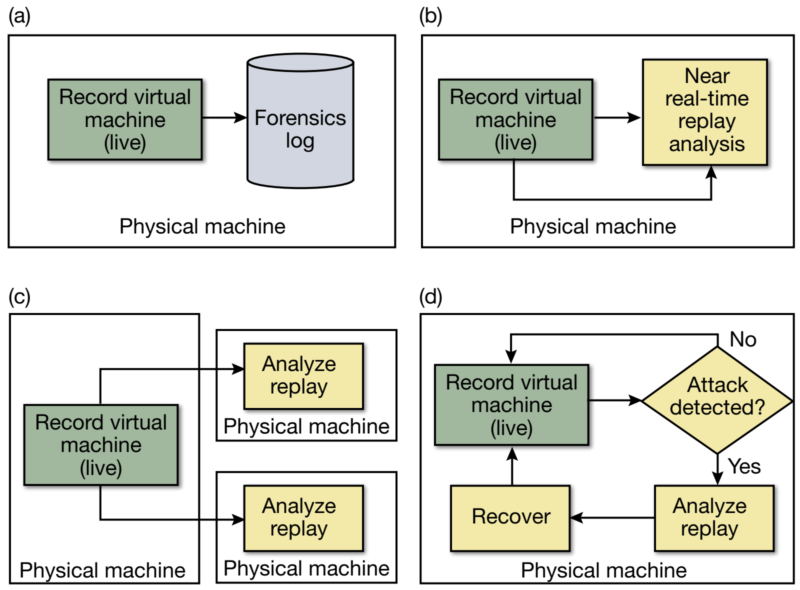


Figure 7. Overview of various applications of RnR: (a) forensics logging of host: (b) monitoring single host in near real time; (c) monitoring host in a cloud (distributed analysis); and (d) self-healing host.

RnR CHALLENGES AND FUTURE WORK

Several challenges remain to be solved for RnR. One of the most important challenges is determining whether RnR itself is vulnerable to attack. If RnR is vulnerable, then any software built on top of it is vulnerable. In our current prototype, RnR executes as part of a host Linux operating system with a considerable amount of complexity. A better design might be to strip down the host operating system and isolate different components of RnR so that if one component fails, the system can recover.

A second important challenge for RnR is supporting analysis during the replay with VMI as discussed in the *Dynamic Analysis of Virtual Machine Replay* section. Although there has been a lot of work with VMI, there are important challenges that remain and a lack of extensive tools available. One of the most important challenges is that the guest is not trustworthy. This means that techniques that inspect guest data structures need some way to guarantee or detect that the guest has not intentionally altered the data structures in a way that would confuse (or worse, compromise) the introspection mechanisms.

A third critical challenge for using RnR is privacy. The recording mechanism will record all data that have been input into the machine and all data that are processed by the machine. This includes sensitive user data such as passwords or other authentication credentials. These data are transient in modern systems, but with RnR, the data could be stored indefinitely. To properly address this concern, various cryptographic schemes need to be explored along with careful system design for specific applications.

A fourth important challenge we highlight is adding support to record and replay multicore virtual machines. Our current prototype can only record a uniprocessor virtual machine. Although there are many useful applications of RnR for uniprocessor virtual machines, extending the functionality to support multicore virtual machines is an important area of future research. The challenge of supporting multicore virtual machines is that shared memory access between cores introduces nondeterministic input for each core because it is difficult to predict correct ordering of reads and writes from each core.

SUMMARY

RnR is a promising technology that enables many new techniques in trustworthy computing. The key concept is that RnR separates live execution from dynamic analysis of the execution. Experimentation shows that performance overhead of recording is less than 5% for typical workloads and that the log growth rate is reasonable.

There are several important challenges to solve to take advantage of potential applications of RnR. These challenges include increasing the trustworthiness of RnR itself, developing replay analysis tools and techniques, creating schemes to properly address privacy concerns, and discovering practical techniques to record multicore virtual machines. We plan to continue looking at ways to address these challenges as we leverage RnR as a research platform for trustworthy computing.

REFERENCES

[1]Abadi, M., Budiu, M., Erlingsson, Ú., and Ligatti, J., "Control-Flow Integrity Principles, Implementations, and Applications," *ACM Trans. Inf. Syst. Secur.* **13**(1), 4:1–4:40 (2008).

[2]Chow, J., Garfinkel, T., and Chen, P., "Decoupling Dynamic Analysis from Execution in Virtual Environments," in *USENIX 2008 Annual Technical Conf.*, Boston, MA, pp. 1–14 (2008).

[3]LeBlanc, T. J., and Mellor-Crummey, J. M., "Debugging Parallel Programs with Instant Replay," *IEEE Trans. Comput.* **36**(4), 471–482 (1987).

[4]Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A., and Chen, P. M., "ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay," *SIGOPS Oper. Sys. Rev.* **36**(SI), 211–224 (2002).

[5]Kernel Based Virtual Machine Main Page, www.linux-kvm.org/page/Main_Page (accessed 31 July 2013).

[6]QEMU Open Source Processor Emulator, wiki.qemu.org/Main_Page (accessed 31 July 2013).

[7]Garfinkel, T., and Rosenblum, M., "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proc. Network and Distributed Systems Security Symp.*, San Diego, CA, pp. 191–206 (2003).

[8]Trevor, J., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., and Wang, Y., "Cyclone: A Safe Dialect of C," in *Proc. USENIX Annual Technical Conf.*, Berkeley, CA, pp. 275–228 (2002).

The Authors

Julian B. Grizzard is the Principal Investigator of APL's virtual machine record and replay research. He is a research scientist in APL's Asymmetric Operations Department where he focuses on solving national cybersecurity challenges with a particular interest in trustworthy computing. **Ryan W. Gardner** is a research scientist in APL's Asymmetric Operations Department and a member of the virtual machine record and replay research team. He is interested in discovering practical solutions to cybersecurity problems that do not sacrifice usability goals. For further information on the work reported here, contact Julian Grizzard. His e-mail address is julian.grizzard@jhuapl.edu.

The *Johns Hopkins APL Technical Digest* can be accessed electronically at **www.jhuapl.edu/techdigest**.