IQ: A Whole-System Instrumentation Framework for Post Analysis

by

Wenhao Xu

B.Eng, Sichuan University, 2006

M.Eng, Tsinghua University, 2009

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

The University Of British Columbia

(Vancouver)

October 2012

Abstract

Analyzing operating systems is a hard problem. Instrumentation can be used to insert analysis code into executions of an operating system. IQ is a dynamic instrumentation framework for instrumenting the whole system. It records executions of an operating system running in a virtual machine and decouples analysis from executions of the operating system. IQ tools can do heavyweight analysis during replays of executions and refine the analysis through continuous replay of the same execution. IQ is a fine granularity framework that provides an API appropriate to instrument operating systems. To our best knowledge, IQ is the first post-analysis framework for instrumenting the whole system.

Preface

Deterministic replay described in Chapter 3 is a collaboration work with Mihir Nanavati.

Table of Contents

List of Figures

Acknowledgments

I am grateful to my supervisors, Andrew Warfield and Norm Hutchinson. Thanks to their help, I had a great research experience and life in UBC.

I would like to thank Geoffrey Lefebvre from whom I learned lots of system knowledge. Thanks to Mihir Nanavati for the collaboration work on deterministic replay.

Thank you to my classmates in NSSL, who makes NSSL a fun place to work.

Thank you to my parents, sister, and girl friend. You always encourage me and make me happy. I cannot do this without you.

Thank you to Brian Oki for reading my thesis draft and giving me good advice.

Thank you to all my friends. The time with you guys are always fun.

Chapter 1

Introduction

1.1 Motivation

System programmers spend much time understanding legacy systems and the interactions between their code and such legacy systems. The usual practice is to run the system, observe what happens, debug the code and repeat the above steps. When it comes to understanding the whole operating system, it is difficult to follow the standard practice for the following reasons:

1. *Non-Deterministic Behaviors*. The execution of an operating system is non-deterministic. Therefore, it is challenging to refine the analysis by reproducing the interesting facts or problems of operating systems.

2. *Asynchrony*. There are many asynchronous events going on in the system. Manually stopping and debugging the system is problematic.

3. *Context Interleaving*. Different Contexts interleave and interact with each other. For example, an interrupt context interleaves with a process execution context. It is hard to analyze the causal relationship between these contexts.

System programmers could use state-of-the-art instrumentation tools to write analysis tools to gather information from either operating systems or applications. These tools, however, are not post-analysis tools. They are either unable to reproduce the execution of an operating system or lack the capability to refine analysis

after the operating system execution.

IQ (Instrument-able QEMU) is an instrumentation framework for post-analysis that helps system programmers to understand the interesting facts from executions of an operating system. IQ makes use of deterministic replay to reproduce the execution of an operating system. The analysis is written and executed afterward during replay. Programmers can refine their analysis through continuous replay over the same execution.

IQ is designed to be easy to use so that most programmers could benefit from it. IQ achieves this by building itself on top of a fast machine emulator, QEMU [8]. It is easy to set up to reduce the barrier for programmers to start using it. In addition, instrumentation tools run in the user space. All the libraries available in the user space can be used by IQ tools.

IQ provides fine-granularity instrumentation APIs to enable tools to instrument the whole operating system. For example, tools can instrument every basic block executed in the operating system by using the translation instrumentation point. Also, IQ provides the context instrumentation point to make it easy to write context aware analysis.

1.2 Contribution

To our best knowledge, IQ is the first post-analysis framework for instrumenting the whole system. It does not only decouple the analysis from the execution of the operating system, but also provides fine granularity APIs to instrument the recorded execution. Therefore, IQ fits well for programmers to understand the execution of operating systems.

Many IQ tools can be written independently from the operating system running inside IQ. This reduces the programmers' effort to port an analysis tool to another operating system.

1.3 Paper Organization

We assume readers of this paper to be familiar with virtualization technology and operating system concepts. Chapter 2 describes the design of the instrumentation framework of IQ. Chapter 3 discusses the implementation of deterministic replay

of IQ. Chapter 4 evaluates IQ by describing several applications built on top of it. Chapter 5 describes related work. Chapter 6 summaries our contributions.

Chapter 2

Instrumentation Design

2.1 Overview

There are usually two ways to analyze a running system, introspection and reflection. Introspection, also called self-observation, is a way that the system reads its own state to do analysis. For example, Dprobe [16] is an introspection tool that runs inside the kernel and analyzes the kernel. Reflection is a way to analyze the running system from the underlying layer, also called the interpreter layer. Because the underlying layer presents an abstraction to the upper layer, it can read the state of the upper layer. For example, the virtual machine could know the page table organization of a running operating system.

IQ instruments the guest operating system in a reflective way. This has the following advantages:

1. Programmers don't need to insert analysis code into the system before hand. Thus, it enables refining the analysis afterward and analyzing the system without available source code.

2. Instrumentation tools can be portable to different operating systems because they can be built without depending on the specific guest operating system.

3. Instrumentation tools don't interfere with the normal execution of the guest operating system.

Building IQ in a reflective way presents many challenges to us. First, we have to decide which instrumentation points IQ will present to tools. The goal is to make instrumentation points easier to use and to enable fine granularity tools. We discuss instrumentation points in Section 2.3.

Secondly, programmers usually need to figure out the current context of the guest operating system to do analysis. For example, in order to figure out the return value for a function whose return value is stored in the register *EAX*, tools could read *EAX* at the moment the function returns. In order to do so, programmers could match the *esp* value of the calling point and the returning point. However, this must be done in the same process stack. In order to facilitate this, IQ provides a context instrumentation point to ease the task of doing context-aware analysis. We discuss the context instrumentation point in Section 2.3.5.

Thirdly, IQ has to provide a way for tools to inspect the machine state. If tools need to read the value of a variable when an interrupt happens, tools need to figure out the guest virtual memory address of this variable before reading it. Mapping a variable to the virtual memory address could be done by looking up the debugging information available in the binary. IQ provides a library for reading debugging information from DWARF [1] debugging information available in ELF [2] executable. In Section 2.4, we discuss the API for reading machine state.

IQ's instrumentation engine is built on top of QEMU [8]. Although it is not as efficient as a virtual machine hypervisor like Xen [7] or KVM [12], QEMU perfectly fits our post-analysis purpose [8, 10, 18, 21]. As an improvement, we are working on recording in Xen and replaying in QEMU [10] to improve the recording performance.

2.2 QEMU

Because IQ is built on top of QEMU v0.12.4, we briefly introduce the internals of QEMU's i386 emulation in this section to provide some background to the reader.

QEMU [8] is a fast machine emulator that can run many operating systems inside it. It runs in user space and emulates various CPUs and peripheral devices. Hypervisors, such as Xen [7] and KVM [12], also make use of QEMU's device emulations.

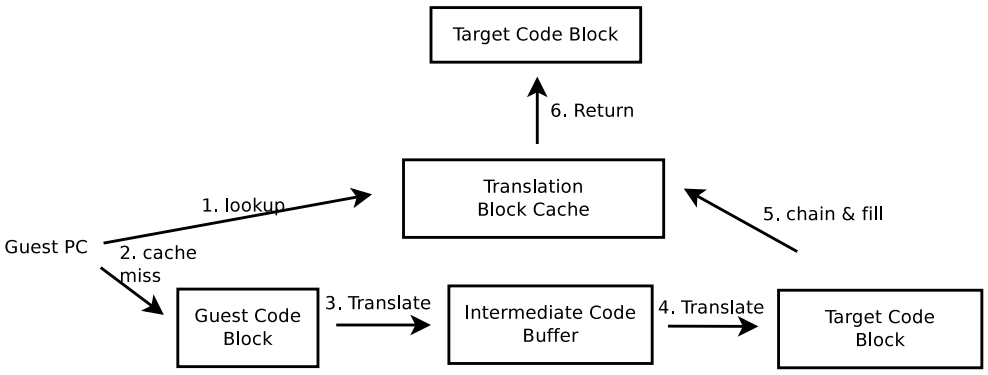

Figure 2.1: QEMU Binary Translation

QEMU's emulation includes three major parts, CPU emulation, memory emulation and device emulation. In Section 2.2.1, we introduce the binary translation [20] which is the core of the CPU emulation of QEMU. In Section 2.2.2, we introduce how QEMU does the memory translation from the guest virtual address into the host virtual address. In Section 2.2.3, we introduce how I/O devices are emulated in QEMU.

2.2.1 Binary Translation

QEMU makes use of binary translation [20] to translate guest code blocks into the instructions of the host architecture. QEMU has a Tiny Code Generator (TCG) to do the translation. Figure 2.1 shows the process of QEMU's binary translation. Given a guest program counter (PC) value, the output of binary translation is the translated target code block ready to execute.

1. QEMU looks up the translation block cache. If a translation block is found for the program counter (PC) value, go to step 6. Otherwise, continue with step 2.

2. QEMU reads the guest code block corresponding to PC.

3. The TCG frontend translates the guest code block into the TCG intermediate code. In QEMU, a translation block ends at a branch instruction, I/O instruction or when the translation buffer is full.

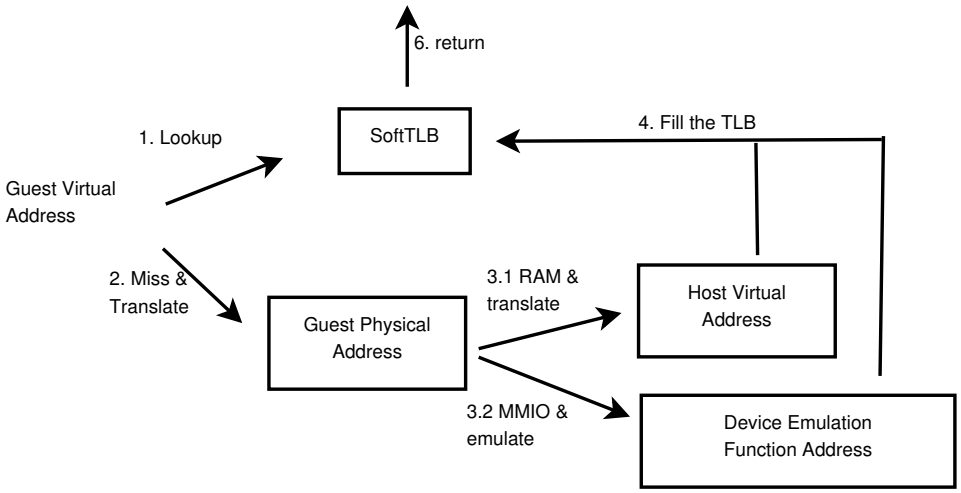

Figure 2.2: QEMU Memory Translation

4. The intermediate code is passed to the TCG backend and translated into the target code block.

5. QEMU puts the target code block into the translation block cache and tries to link the current code block with the calling block. By chaining the target code blocks together, the cost of returning to the emulator is largely reduced.

6. Return the translated block.

2.2.2 Memory Translation

Figure 2.2 is an overview of QEMU's memory translation process. Given a guest virtual address, the goal of memory translation is translating it into either a host virtual address or a device emulation function address based on the guest's physical memory mapping.

1. QEMU looks up the address in the SoftTLB. If there is a mapping for the guest virtual address, the host virtual address or the device emulation function address is directly calculated from it. Otherwise, QEMU will go through the following translation process.

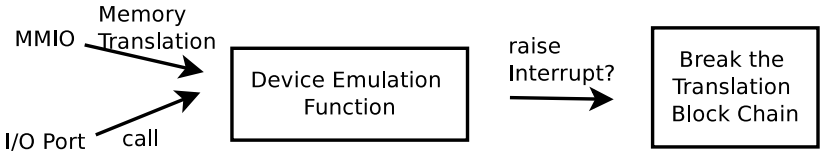

Figure 2.3: QEMU Device Emulation

2. QEMU translates the guest virtual address into the guest physical address by following the guest page table. If a page entry is missing during the process, QEMU stops the translation and injects the page fault into the guest.

3. QEMU judges if the guest physical address is a RAM address or MMIO address. QEMU maintains a radix tree for physical memory page description. By looking up this tree, the usage of the physical page could be known. If it is the RAM address, a host virtual address will be obtained. Otherwise, a device emulation function address will be obtained.

4. QEMU inserts the address mapping into the SoftTLB and return it.

2.2.3 Device Emulation

In x86 architecture, there are two ways to do I/O: I/O port and memory mapped I/O. In either way, QEMU translates an address into a device emulation function call, as shown in Figure 2.3.

1. Map the I/O port or MMIO address to the device emulation function. Emulated devices register emulation functions for I/O ports and MMIO ranges to QEMU during machine bootstrap. For I/O ports, QEMU maintains a lookup table for all 64K I/O ports. Therefore, mapping the I/O port to the device emulation function is a direct array access. How MMIO address is mapped to the emulation function is described in Section 2.2.2.

2. If the device needs to raise an interrupt, QEMU will break the current translation block chain so that the interrupt can be injected into the guest as soon as possible. This means that QEMU only injects the interrupt into the guest between two translation blocks. Although there is potentially a delay for

interrupt delivery, this design works well in practice and simplifies the implementation.

2.3 Instrumentation Points

IQ provides tools with several instrumentation points: translation, memory access, function return and context switch.

Translation instrumentation is used to instrument the intermediate code translated by the TCG frontend. With this instrumentation point, tools can instrument every basic block executed in the system. The instrumentation point enables fine granularity instrumentation.

In addition, to ease the instrumentation of the whole system, IQ provides memory access, function return and context switch instrumentation for tools. Memory access instrumentation allows tools to instrumentation the access to any memory address. Function return point is used to track the return point of a function. Context switch instrumentation point is used to track the current context of operating system.

We describe each instrumentation point in the following sections.

2.3.1 Translation

```
/* Register a basic block callback to intercept the basic block translation */
int dbi_register_bb_hook(void (*ev_it_bb)(dbi_tcg_context_t *));
```

Listing 2.1: Translation Block Instrumentation

By registering a callback at this instrumentation point, tools can intercept the binary translation process and instrument the intermediate code translated by the TCG frontend. The API to register the callback is shown in Listing 2.1. The parameter *dbi_tcg_context_t* contains the intermediate code buffer and the pointer to the guest code block.

Implementation. IQ creates a buffer for storing the intermediate code. After QEMU translates the guest code into the intermediate code, IQ passes the buffer to the tools if tools registered the instrumentation point. The tools could instrument the code in the TCG intermediate language and return the instrumented code buffer

to IQ. After that, IQ passes the code to QEMU's TCG backend, which translates the intermediate code into the target code.

IQ supports on-demand emulation, which means the tools could add or remove instrumentation for a translation block anytime. IQ provides an API for tools to remove a translated block from the translation block cache to force the block to be translated again. Thus, the tools can add or remove instrumentation for the translation block on demand.

2.3.2 Memory Watchpoint

As shown in Listing 2.2, tools can register a memory watch callback by calling *dbi_register_watch_mem_hook*. The callback will be called when the memory address, watched by calling *dbi_watch_mem*, is accessed by the guest operating system. *dbi_watch_mem* is used to watch either read or write access to a memory address. When the callback is called, the guest virtual address, length of the access, data read from/written to the address, and the direction of the access are passed to the tools.

Implementation. IQ registers a "watchpoint device" to QEMU during bootstrap. As shown in step two in Figure 2.2, when translating the guest virtual address, IQ checks if the virtual address is watched. If it is watched by tools, IQ marks the address as a MMIO address which points to the "watchpoint device" emulation functions. The MMIO address is inserted into the SoftTLB. So when a watchpoint address is accessed, the watchpoint device emulation function will be called. In the device emulation functions, the callback registered by tools will be called.

```
/* Watch a guest virtual memory address. reason: read, write*/
int dbi_watch_mem(target_ulong addr, int reason);

/* Register a memory watch callback. */
/* When the watched memory is accessed, ev_watched_mem will be called.*/
/* The first parameter is the guest virtual address.
/* The second parameter is the length of this access. */
/* The third parameter is the value read from or written to this location. */
/* The fourth parameter indicates if this access is a read or write. */
int dbi_register_watch_mem_hook(
            void (*ev_watched_mem)(target_ulong, int, uint32_t, int));
```

Listing 2.2: Memory Watchpoint Instrumentation

2.3.3 Memory Tracing

By registering a callback at this instrumentation point, tools can intercept all memory accesses of the guest. This is mainly used to analyze the memory access patterns. The API is shown in Listing 2.3. *dbi_register_trace_mem_hook* is used to register functions which will be called when the guest CPU is reading from or writing to the memory. The guest virtual address, guest physical address and the length of the access are passed to the tools when the function is called. *dbi_register_mem_async_hook* is used to register functions which will be called when the guest devices are doing memory copy in DMA operations. The guest physical address and the length of the access are passed to the tools when the function is called. *ev_mem_read* and *ev_mem_read_async* are called when the data is read from the memory. *ev_mem_write* and *ev_mem_write_async* are called when the data is written to the memory.

Implementation. If the tools register this instrumentation point, IQ intercepts the SoftTLB lookup process. Because every memory access goes through SoftTLB, IQ can intercept all the memory accesses.

The tools can also implement memory tracing by using the translation instrumentation point. However, the memory tracing instrumentation point makes it easier to write memory analysis tools. We describe a memory analysis tool in Section 4.2.

```
/* Register R/W callback for tracking every memory access */
/* The first parameter of callbacks is guest virtual address */
/* The second parameter of callbacks is guest physical address */
/* The third parameter of callbacks is the length of access */
int dbi_register_trace_mem_hook(
                void (*ev_mem_read)(target_ulong, target_ulong, int),
                void (*ev_mem_write)(target_ulong, target_ulong, int));

/* Register the callback for tracking every asynchronous memory access, e.g. DMA */
/*  The first parameter of the callback is the guest physical address */
/*  The second parameter of the callback is the length of access */
int dbi_register_mem_async_hook(
                void (*ev_mem_read_async)(target_ulong, int),
                void (*ev_mem_write_async)(target_ulong, int));
```

Listing 2.3: Memory Tracing

2.3.4 Function Return

```
/* Register the callback for 'ret' instruction. */
/* The parameter of the callback is the current stack pointer */
void dbi_register_ret_hook(void (*ev_ret)(target_ulong));
```

Listing 2.4: Function Return Instrumentation

As shown in Listing 2.4, *dbi_register_ret_hook* is used to register a callback *ev_ret* which will be called when an *ret* instruction is executed. The parameter of *ev_ret* is the current stack pointer, *ESP*. The purpose of this instrumentation point is to ease tracking the returning point of a function.

Implementation. IQ intercepts the translation of each *ret* instruction and inserts a function call after translating the *ret* instruction. In this function call, the callback registered by the tools will be called.

Tools can also use translation block instrumentation point to instrument the *ret* instruction. This instrumentation point, however, is easier to use and more efficient. Without this instrumentation point, the tools have to scan every intermediate code block for the *ret* instruction.

2.3.5 Context Switch

As shown in Listing 2.5, *dbi_register_context_hook* is used to register a callback *ev_context* which will be called when a context switch event happens. The parameters of *ev_context* are the context switch event type and the value of *ESP0* which is the pointer to the kernel stack.

IQ defines five context switch events:

1. *interrupt*: an interrupt is going to happen.
2. *iret*: the interrupt handling is finished.
3. *stack switch*: the kernel stack switch happens.
4. *sysenter*: the operating system enters to kernel mode.
5. *sysexit*: the operating system leaves kernel mode.

Implementation When QEMU injects interrupts into the guest, IQ intercepts it and invokes *ev_context* to indicate *interrupt* happens. When translating any one of the *iret*, *sysenter* or *sysexit* instructions, IQ inserts a function call after translating the instruction. The function will be called after the instruction is executed.

In order to detect the *stack switch* event, IQ watches the memory location of *ESP0*. In x86, *ESP0* points to the kernel stack of a task. In Linux, writing to *ESP0* means a process switch happens. The memory location of *ESP0* is obtained by reading the *Task Register*. When the *ESP0* is written, IQ generates a *stack switch* event and invokes *ev_context* registered by the tools.

```
/* Register the context callback. */
/* When a context switch happens, the callback will be called */
/* with the switch reason and current esp0 as parameters */
void dbi_register_context_hook(
                void (*ev_context)(context_reason_t, target_ulong));
```

Listing 2.5: Context Switch Instrumentation

2.4 Machine State Inspection

IQ provides an API for tools to inspect the machine state in instrumentation. As shown in Listing 2.6, *get_cur_cpu_state* is used to get the current CPU state. Tools

can read the state of the CPU from *CPUState*. *read_guest_mem* is used to read the content from the virtual guest memory address. If the physical page for the virtual address is not present in the memory, IQ returns *-1* to tools. If tools are doing source level analysis, the tools can obtain the virtual memory address of a variable from the debugging information, such as DWARF [1] and then call *read_guest_mem* to read the value from the memory address.

virt_to_phys is used to translate a guest virtual address into a guest physical address. *phys_ram_to_host* is used to translate a guest physical memory address into the host virtual address.

```
CPUState *get_cur_cpu_state(void);
int read_guest_mem(target_ulong vaddr, uint8_t *buf, int len)
/* -1: physical page not present */
target_ulong virt_to_phys(target_ulong virt_addr);
/* -1: if paddr is not a RAM or does not exist */
target_ulong phys_ram_to_host(target_ulong paddr);
```

Listing 2.6: Machine State

Chapter 3

Deterministic Replay

3.1 Overview

IQ replays a uniprocessor virtual machine. IQ records non-deterministic events into logs and injects them into the guest during replay.

There are two categories of non-deterministic events that IQ must handle. The first category is the asynchronous events, such as hardware interrupts. For these events, the events and the time when they happen are recorded into the interrupt log. During replay, these events are injected into the guest by IQ based on the time recorded in the interrupt log.

The second category is the data read from external devices. For example, the data received from network card is non-deterministic. Some instructions, such as *rdtsc*, are also non-deterministic. For these events, the data and the time when they happen are recorded into the data log. Recording the time is for debugging and quickly discovering the deviation during replay.

In order to reduce the data log size, IQ doesn't record the data read from disks. This is because disk operations can be replayed if the disk starts from the same initial state and the same data are written to the disk I/O ports and MMIO space. IQ makes a snapshot of the whole disks before recording and loads the snapshot before replay so that the disk is in the same initial state in record and replay. And because the command and data sent to disks are deterministic, disk operations can be replayed. IQ currently has two mechanisms of making snapshots of virtual disks.

One is making use of the QCOW2 [5] image format and its snapshot mechanism. The other one is making use of Btrfs [4] to snapshot virtual disks in RAW image format. The latter one is much faster than the first both in the I/O performance and snapshot loading speed.

3.2 Timestamp

In order to replay asynchronous events, the time of events needs to be recorded. ReVirt [11] and VMware Workstation [15] use a triple containing branches retired since the last interrupt and the value of the ECX register and the EIP register to identify the time. Branches retired since last interrupt is used to identify the basic block when the event happens. EIP is used to identify the instruction. ECX is used to identify the progress of a string operation [3]. The triple uniquely identifies a time in the execution. This information can be read from the CPU. For example, branches retired since last interrupt could be read from hardware performance counters.

QEMU [8] doesn't emulate hardware performance counters. IQ could emulate the hardware performance counters, but the cost is too high. Instead, IQ uses a different kind of timestamp to identify the time. As described in Section 2.2.3, QEMU only generates interrupts at translation block boundaries. So the number of translation blocks executed can be used to replace branches retired since last interrupt. Together with EIP, the pair could be used as the timestamp. ECX is not necessary because QEMU does not generate interrupts when a string operation is in progress.

3.3 Record

As shown in Figure 3.1, IQ inserts a Record and Replay (*RR*) layer between the emulation world and the guest world. The emulation world does the binary translation and the device emulation. And the guest world is where the guest code executes.

IQ counts the number of blocks executed since the last interrupt by inserting a piece of code into the intermediate code buffer while translating a guest code block.

When the emulation world injects an interrupt to the guest world, the *RR* layer

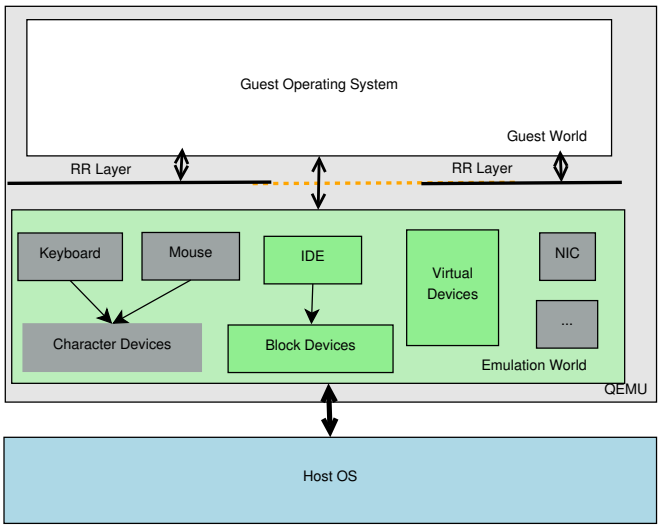

Figure 3.1: Deterministic Replay

intercepts it and records it into the interrupt log. For non-deterministic instructions, such as *rdtsc*, IQ records the value returned by the instruction in the data log. When the guest world reads data from an I/O port or a Memory Mapped I/O (MMIO) address, IQ records the value returned by the emulation world into the data log unless it is an I/O port or MMIO address of disks or virtual devices. Because disks and virtual devices are still emulated in the replay, it is not necessary to record events from them.

Virtual devices, used to improve the performance of the guest, usually write to the emulated CPU state or guest memory directly. The side effect must also be replayed. As far as we've seen, virtual devices are deterministic when they write to the guest memory or the CPU state. Like handling disk non-determinism, IQ doesn't record the side effect made by virtual devices because the side effect can be replayed given the same initial device state.

Recording Direct Memory Access (DMA) events in QEMU is a challenge because it is asynchronous. The DMA emulation in QEMU involves a memory copy, which is usually performed by another thread or the kernel when using asynchronous I/O. IQ does a simplification here by not recording the exact time of the

memory copy with the assumption the memory buffer used for DMA will not be touched by the operating system before the DMA controller raises an interrupt. This assumption is always true for modern operating systems, such as Linux.

3.4 Replay

During replay, when the guest reads data from an I/O port or a MMIO address, if it is not for disks or virtual devices, the *RR* layer intercepts it and returns the data from the log. Also, when the guest writes data to an I/O port or MMIO address which is not an address of a disk or a virtual device, the *RR* layer intercepts the write operation and discards it. So in replay, only disks and virtual devices are still emulated.

IQ counts the number of translation blocks executed since the last interrupt. Based on the interruption log, when it is time to inject an interrupt, *RR* layer will break the current translation block chain and inject the interrupt into the guest.

Disk DMA events need special handling in order to ensure the right order between the memory copy and the DMA interrupt injection. Because the disk is still emulated in replay, the DMA controller may not finish copying data into the memory yet when it is the time to inject the DMA interrupt. In this case, IQ pauses the execution of the guest operating system and waits for the completion of the DMA operation. After the memory copy is done, IQ injects the interrupt and resumes the execution of the guest operating system. Replay is usually faster than record because of the time compaction [10].

Chapter 4

Evaluation

In this chapter, we describe several tools built on top of IQ. In Section 4.1, we describe how to track the return value of a function in the operating system. In Section 4.2, a tool for analyzing the memory access pattern of the whole system is described.

4.1 Function Return Value

Tracking the function return value is common in many IQ tools. It can be done using the following two steps:

1. *The tool figures out the return point of a function*. If there is only one stack in the system, the return point could be obtained by matching the stack pointer (ESP) of the calling point and return point. However, when there are many stacks in an operating system, the match must be done in the same stack. Thus, the tool must register three types of instrumentation points: translation, function return, and context switch. With the translation instrumentation, the tool can figure out the calling point of a function and record the ESP value. With the context switch instrumentation point, the tool can track stack changes to make sure the function return is on the same stack as the calling point. The function return instrumentation point is used to instrument the *ret* instruction.

2. *The tool reads the return value from either a register or a memory location.*

Where the return value is stored can be found in the debugging information for the binary.

Listing 4.1 shows a tool that prints out the return value of the *__alloc_pages* function in the Linux kernel. The tool initializes itself by registering two instrumentation points: translation and context switch. The translation instrumentation callback is *it_bb*. It checks if the first instruction of a translation block matches the address of function *__alloc_pages*. If so, it means it is the entry point of *__alloc_pages*. It then instruments the intermediate code block by inserting a function call to *helper_it_call*. So when *__alloc_pages* is executed, *helper_it_call* will be called. *helper_it_call* records the current stack top into *alloc_esp* and the current stack pointer into *alloc_esp0*. It also registers a function return instrumentation point callback *it_ret*. When a *ret* instruction is executed, the *it_ret* callback will be called. *it_ret* checks if it is the same context by comparing the stack pointer with *alloc_esp0*. If so, it further checks if it is the return point of *__alloc_pages* by comparing the current stack top with *alloc_esp*. If it is the return point, the return value can be read from the register EAX and printed out.

```
target_ulong esp0;
target_ulong alloc_esp;
target_ulong alloc_esp0;
static void it_context(context_reason_t reason, target_ulong esp0)
{
        /* Record the current esp0 */
        esp0 = esp;
}
static void it_ret(target_ulong esp) {
        if( alloc_esp0 == esp0 && alloc_esp == esp){
                /* If the esp0 value matches, it means it is in the same stack. */
                /* If the esp value matches, it means it is the matching ret instruction. */
                /* When the function returns, the return value is in EAX. */
                printf("__alloc_pages return %ld\n", cpu_single_env->regs[R_EAX]);
        }
}
/* This function will be called when __allloc_pages is called */
static void helper_it_call(target_ulong pc)
{
        /* Record the current context(esp0) into alloc_esp0. */
        alloc_esp0 = esp0;
        /* Record current stack pointer at calling point into alloc_esp. */
        alloc_esp = cpu_single_env->regs[R_ESP];
```

```
        /* Register the return hook. */
        dbi_register_ret_hook(it_ret);
}
static void it_bb(dbi_tcg_context_t *context)
{
        /* Get the pc of first instruction */
        target_ulong fipc = get_first_ipc(context);
        if( fipc == __ALLOC_PAGES ){
                /* If it is the first instruction of __alloc_pages, */
                /* generate a call to helper_it_call */
                gen_helper_it_call(tcg_const_i32(fipc));
        }
        /* Copy left instructions to output buffer. */
        copy_in_left_to_out(context);
}
int dba_init(int argc, char **argv){
       dbi_register_bb_hook(it_bb);
       dbi_register_context_hook(it_context);
}
```

Listing 4.1: Print __alloc_pages Return Value

gen_helper_it_call is the macro to generate the intermediate code to do the function call to *helper_it_call*. *get_first_ipc* returns the virtual guest address of the first instruction in the current translation block. *copy_in_left_to_out* copies the left intermediate code to the output buffer.

4.2 Memory Analysis

We've built a tool to study how each memory page is used in the system and what is the access pattern of less utilized pages. IQ records the execution of a guest running Ubuntu 8.04 with 512MB memory. During recording, we perform a user's daily operations by executing commands in a terminal and opening some web pages in the Firefox browser.

In order to get the access pattern of memory pages, the tool tracks every access of the memory by registering a callback at the memory trace instrumentation point. The tool creates a bitmap for the physical memory space. Each bit in the bitmap denotes if the byte is accessed. When the byte is accessed, the tool marks it in the bitmap. The utilization and spatial access pattern of each page can be calculated from this bitmap.

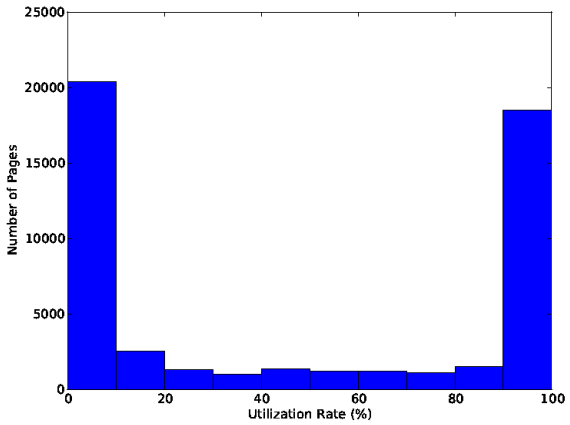

Figure 4.1: Utilization Distribution

The tool treats each allocation of the same physical page as different pages. In order to do so, the tool instruments the "*__alloc_pages*" and "*__free_pages*" functions to know when a page is created and freed. When "*__alloc_pages*" returns, it means pages get created. The bitmap of allocated pages is cleared. When "*__free_pages*" is called, it means the pages are freed. The bitmap of freed pages is stored into the trace log.

Readers could refer to Section 4.1 for how to instrument the "*__alloc_pages*" function. The "*__free_pages*" function can be instrumented in the same way.

Figure 4.1 shows the utilization distribution of pages. The tool traces 143052 pages, of which there are 93093 pages that are never touched. Pages never touched are filtered out.

Interestingly, most pages are either highly utilized or little utilized. There are 41.42% pages that are less than 10% utilized and 36.61% pages that are more than 90% utilized.

Figure 4.2 shows the spatial pattern of pages with utilization rates between 1% and 10%. The x-axis is the byte offset inside a page. The y-axis is the identifier of physical pages sorted based on utilization rate. From bottom to top, the utilization is increasing. 43.1% of pages are unique physical pages, which means many physical pages are short lived: allocated and then freed quickly. Figure 4.3 plots the

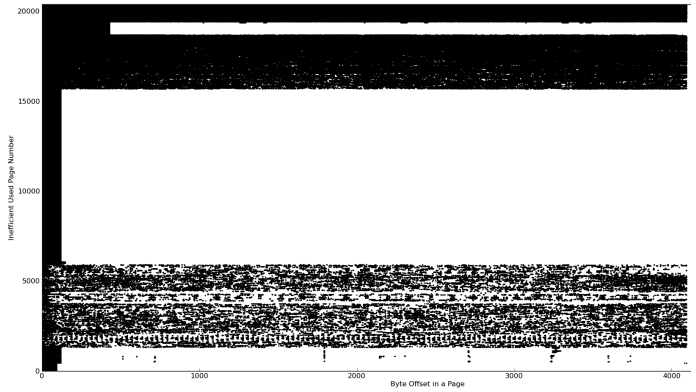

Figure 4.2: Spatial Pattern - Include Reused Pages

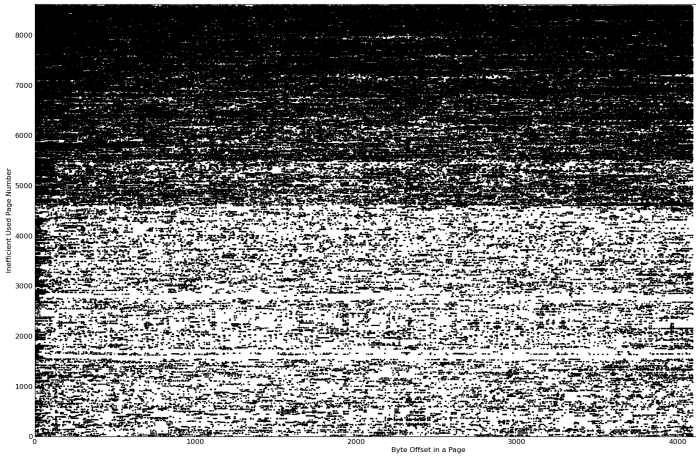

Figure 4.3: Spatial Pattern - Exclude Reused Pages

spatial pattern by excluding short-lived pages.

By comparing Figure 4.2 and Figure 4.3, we can see that long-lived pages are sparsely accessed, while the short-lived pages are densely accessed and then quickly freed.

The bitmap method used for utilization and spatial pattern analysis doesn't help to analyze the temporal pattern. Such an analysis requires recording the order of every memory access. We further refine the analysis by writing a tool to record every memory access of 12 pages randomly chosen from the less utilized pages. The tool does not record the order of every memory access, but the order of every 128

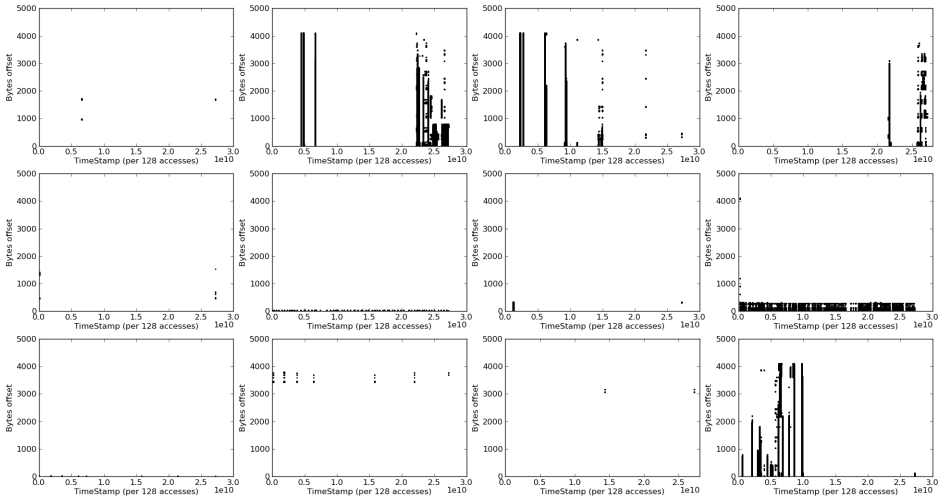

Figure 4.4: Temporal Pattern

memory accesses in order to reduce the log size. The granularity of 128 accesses is fine enough to observe the temporal pattern of memory access. Figure 4.4 shows the temporal pattern of the 12 pages. The x-axis is the time and the y-axis is the byte offset in the page.

4.3 Retroactive Aspects

Retroactive Aspects [19] is a tool built on top of IQ. It presents a novel way to analyze the execution of an operating system. Retroactive aspects make it much easier to instrument the system when the source code is available. Programmers can refer to the function name or variable name directly in the tool. Interesting readers could refer to the paper Retroactive Aspects [19] for more details.

4.4 Instrumentation Overhead

IQ's instrumentation framework brings little overhead to the replay of the virtual machine. The overhead is mainly caused by additional functions calls brought by the instrumentation framework. However, comparing to the QEMU's emulation

overhead, the overhead of IQ is neglectable.

4.4.1 Experiment Setup

The experiment setup is shown in Table 4.1.

Table 4.1: Experiment Setup

Host OS	CentOS 6.2
Host File System	EXT3
Guest OS	Ubuntu 10.04 Server
Virtual Machine Image	QCOW2

4.4.2 Overhead

We record the process of compiling Linux kernel in the virtual machine. We then instrument the execution at different instrumentation points. These instrumentation points will be hit frequently during the execution of the operating system.

Table 4.2 shows the time of the replay process with different instrumentation points enabled. As we can see, the overhead of IQ's instrumentation framework is very small.

Table 4.2: Instrumentation Overhead

Instrumentation	Time(s)	Slowdown
None	630.9	0%
Translation Block Instrumentation	633.0	0.33%
Context Switch Instrumentation	633.8	0.4%
Memory Tracing	631.0	0.02%

Chapter 5

Related Work

5.1 VAssert

VAssert [6] is a debugging tool that executes analysis during replay. Programmers insert VAssert calls into their source code. These VAssert calls have no effect during recording in order to reduce the overhead. When programmers try to analyze the program during replay, the analysis will be run. On the one hand, VAssert is similar to IQ in that it decouples the analysis from normal execution of the system and performs the analysis during replay. On the other hand, the difference is that VAssert inspects the guest operating system introspectively by embedding the analysis code into the source code. IQ inspects the guest operating system from the outside, which requires no modification of the source code. In addition, VAssert is a pre-analysis tool that requires programmers to insert analysis code before recording, while IQ is a post-analysis tool. IQ is more suitable for analyzing legacy code or when programmers don't know in advance what analysis will be necessary.

5.2 Aftersight

Aftersight [10] is a tool that decouples the analysis from the normal VM execution. Aftersight records execution in VMware workstation and replays the execution in QEMU. Recording is more efficient than in IQ. However, Aftersight does not define an API to do instrumentation and access the machine state. Further, it is more

focused on a production system. On the other hand, IQ focuses on understanding the system offline. The instrumentation design of IQ makes it a powerful framework to write post-analysis tools.

5.3 PinOS

PinOS [9] is a whole operating system instrumentation framework built on top of Xen. PinsOS's API is very similar to Pin [14], while IQ's translation block instrumentation is more like Valgrind [17]. Writing PinOS tools is more difficult than writing IQ tools because PinOS tools are running in the PinOS space, while IQ tools are running in the user space. All the user space libraries in Linux could be used by tools. Besides, PinsOS doesn't support deterministic replay to reproduce problems.

5.4 Deterministic Replay

IQ's methodology of deterministic replay is similar to work, such as *Revirt* [11] and VMware Workstation [15]. Both deterministically replay the guest running inside it. Recording in VMware Workstation is much more efficient than in IQ because VMWware workstation is much faster than QEMU. *ReVirt* and VMware Workstation use hardware performance counters to read the branches retired since last interrupt. However, IQ is built on top of QEMU and uses a different timestamp.

ArgosReplay [18], built on QEMU version 0.9.1, deterministically replays a guest operating system. It does not record all the non-deterministic events, but eliminates all the non-deterministic sources. ArgosReplay disables timer interrupts and uses instruction count to emulate the timer interrupt. After executing a fixed number of instructions, a timer interrupt is injected. Besides, ArgosReplay replaces the asynchronous I/O operations with the synchronous ones that are also deterministic. ArgosReplay is slower than IQ and may distort the system's behavior by eliminating non-deterministic events.

5.5 Tralfamadore

Tralfamadore [13] is a post-analysis tool. It records every translation block and its side effect to memory and register state into a trace log. Post execution tools can reconst the system state from the trace log. The Tralfamadore log is huge because it records every instruction. IQ only records non-deterministic events. Thus, log is much smaller. Both Tralfamadore and IQ tools run in the user space. Tralfamadore tools could easily walk back and forth through the trace file or even do parallel analysis over the log. It can be inefficient for Tralfamadore tools to read from a memory address because the system state must be reconstructed from the trace log. IQ tools can easily read from the guest state.

Chapter 6

Conclusion

We have presented IQ, a post-analysis instrumentation framework for analyzing operating systems. By recording executions of operating systems, IQ tools can instrument the executions afterward and refine analysis through continuous replay over the same execution.

IQ is easy to setup and use. IQ provides tools with fine granularity instrumentation points and an API to inspect the virtual machine state. IQ adds neglectable overhead to the replay.

IQ tools run in the user space and can be portable to analyze different operating systems. By decoupling execution from instrumentation, IQ tools can do heavyweight analysis without disturbing the execution of an operating system.

IQ is not designed to be used in production systems. It is designed to help programmers to understand executions of operating systems.

Bibliography

[1] Dwarf debugging information format. 1993. → pages

[2] Tool interface standard (tis) executable and linkable format (elf) specification, version 1.2. 1995. → pages

[3] Volume2: Instruction set reference. intel architecture software developer's manual. 1999. → pages

[4] The btrfs filesystem. 2007. → pages

[5] The qcow2 image format. 2008. → pages

[6] Vassert programming guide. 2008. → pages

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. ACM, ACM, 2003. → pages

[8] F. Bellard. Qemu, a fast and portable dynamic translator. USENIX, 2005. → pages

[9] P. Bungale and C. Luk. Pinos: a programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 137–147. ACM, 2007. → pages

[10] J. Chow, T. Garfinkel, and P. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 1–14. USENIX Association, 2008. → pages

[11] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002. → pages

[12] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007. → pages

[13] G. Lefebvre, B. Cully, M. Feeley, N. Hutchinson, and A. Warfield. Tralfamadore: unifying source code and execution experience. In *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009. → pages

[14] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN conference on Programming language design and implementation*. ACM, 2005. → pages

[15] X. Min, M. Vyacheslav, S. Jeffrey, V. Ganesh, and W. Boris. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2007)*, 2007. → pages

[16] R. Moore. A universal dynamic trace for linux and other operating systems. *Proceedings of the FREENIX Track*, 2001. → pages

[17] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007. → pages

[18] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *ACM SIGOPS Operating Systems Review*, 40(4): 15–27, 2006. → pages

[19] R. Salkeld, W. Xu, B. Cully, G. Lefebvre, A. Warfield, and G. Kiczales. Retroactive aspects: programming in the past. In *Proceedings of the Ninth International Workshop on Dynamic Analysis*, pages 29–34. ACM, 2011. → pages

[20] R. Sites, A. Chernoff, M. Kirk, M. Marks, and S. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, 1993. → pages

[21] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. *Information Systems Security*, pages 1–25, 2008. → pages