Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

Automating Pre-Virtualization for Memory Objects

Yaowei Yang

Diplomarbeit

Verantwortlicher Betreuer: Prof.Dr. Frank Bellosa

Betreuender Mitarbeiter: Joshua LeVasseur

Stand: March 13, 2007

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Litraturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only citied sources have been used.

Karlsruhe, March 13, 2007 Yaowei Yang

Abstract

Virtualization is used to construct several virtual machines on a physical machine for high utilization and other benefits. The virtual machines should run independently within their own domain without disturbing the others' execution. In order to achieve this purpose, two things are considered: virtualization-sensitive instruction and sensitive memory operation. Since the virtualization-sensitive instructions and memory operations could change the actual state of physical resources — e.g. CPU, memory and I/O devices — the virtual machine must prohibit their use. The virtual machine monitor must fully control them and share them among many virtual machines.

Currently, virtualization technologies suffer from either performance penalty or high engineering effort to work around the virtualization sensitive instruction and sensitive memory operation. Pre-virtualization tries to balance these two factors by using code analysis, and successfully achieves high performance with low porting engineering effort for virtualizing virtualization sensitive instructions, but it lacks automation for the sensitive memory operations.

In this thesis, we analyze the technique used by pre-virtualization for automatically virtualizing virtualization sensitive instructions and find out that the limitation of this technique comes from the simple type system of assembly language. Therefore, we propose a solution for this problem that parses source code in a programming language that has a rich type system. By using this enhancement, pre-virtualization can achieve high performance and low engineering effort for virtualizating both virtualization sensitive instructions and sensitive memory operations.

Contents

Chapter 1

Introduction

With virtualization technology, one physical machine can be divided into several isolated domains, called virtual machines(VMs). The virtual machines have full or partial ability of the physical machine and host different applications. An underlying software layer, the virtual machine monitor (VMM), takes care of the running state of each VM and prevents them to from interaction undesirably. The VMM layer can be a general operating system or a specialized hypervisor. The applications running on the VMs can be of different kinds, e.g. device driver service or web service. They are normally called guest applications. An operating system running on VM is called a guest operating system(OS).

From the perspective of virtualization technology, requests for accessing physical resources from the guest applications can be divided into two kinds. One kind is innocuous access. It is the most common and passed directly to the physical resource. The other kind is sensitive access. If these accesses are passed directly to the physical resource, the other VM's execution will be undesirably affected, and even corrupted. Therefore the VMM must catch this kind of access and redirect them to the virtualized resource. Normally they will be held until they can be safely executed. The sensitive accesses include two types: virtualization-sensitive instructions and sensitive memory operations. Both of them have side effects on the hardware behavior.

Different virtualization technologies take different strategies to work around sensitive access. However, some of them bring huge performance overhead since they dynamically catch the sensitive accesses during guest OS execution in their virtualization environment; the others require a large engineering effort for porting the guest OS in their virtualization environment because they need static modification of OS source to avoid overhead from dynamically virtualizing. Pre-virtualization approach tries to combine the advantages from static modification and dynamic virtualizing by using code analysis tool so that virtualization can be freed from both performance penalty and high engineering effort. This technique is called "afterburning". The sensitive accesses in guest OS are tried to be statically(high performance) and automatically(low engineering effort) augmented with scratch spaces through afterburning. When the afterburned binary is loaded later, the scratch spaces are dynamically rewritten with the VMM-specific emulation codes(high performance and flexible). However, current assembly afterburning is not suitable for virtualizing sensitive memory operations.

The Weakness of Assembly

In order to catch the memory sensitive operations we must firstly identify the special memory object – memory sensitive object. From the view point of assembly, it is very difficult to find this kind of objects because assembly does not provide a high level abstraction for data. It deals with all memory accesses uniformly. The hardware side effect from the memory access will be automatically considered by hardware resource, when the corresponding memory access happens. Therefore, current pre-virtualization is only able to automatically afterburn the sensitive instructions. For memory sensitive operations, pre-virtualization has proposed to use "profile feedback" which would complicate the implementation of virtualization environment; or even manually find and replace them in a high level source codes, e.g. C language. The first solution is a little complex, imprecise, and may reduce the VM performance; the second solution increases the engineering effort for porting the guest OS to the virtualization environment.

High Level Afterburning

Most modern operating systems are not directly written in assembly language, but in a high level programming language,e.g. C. In contrast to the assembly language a high level programming language provides higher level abstraction for memory data. A memory object could be represented in different type with special name. Because of this high level abstraction, it is possible to automatically identify memory objects according to their types and replace the operation codes on these objects. In this thesis, we propose to construct a C code analyzer for pre-virtualization so that sensitive memory operations could also be automatically afterburned with the scratch space.

The Structure of This Paper

This paper consists of five sections. The following section creates the theoretical foundation for our approach. It explains most important concepts for further discussion. In section 3, our solution for automatically identifying and replacing the memory sensitive operations is proposed. We will describe it in detail from different aspects. Section 4 describes the implementation. Finally, in section 5, we will introduce how we constructed the experiment environment for our prototype system and show the feasibility of our approach through the experiment.

Chapter 2

Background and Related Work

In this chapter, we will introduce the basic concepts for this thesis. In the beginning, virtualization is briefly introduced. We will introduce what kind of problems are required to be solved in virtualization. Then different virtualization technologies are introduced and compared. From this discussion we will show our motivation in this thesis. Last of all, we make an introduction about code analysis. This is related to programming language and will also provide us a way to solve the problem.

2.1 Virtualization

Since 1960s', virtualization has become one of the discussion topics in the world of computer science [11, 24]. Although for a long time it did not attract many people's focus, it regains its attention recently. While there are many different kinds of virtualization, our discussion in this thesis will focus on the *platform virtualization*. For simplicity we use virtualization instead of platform virtualization in this paper. In short, virtualization is a technique used to replicate the functionality of real hardware platform so that one physical machine can host more than one system level software without conflict.

2.1.1 The Advantages of Virtualization

In current world more than one service are normally hosted by a single conventional operating system running on one physical machine. Thus one single service might crash the whole operating system and drop down the other healthy services. In order to overcome this problem, more physical machines might be bought for hosting the different important services with individual operating systems. However, this approach significantly increases the hardware cost and decreases the utilization of physical machine. With the enhancement from virtualization, a single physical machine can be virtualized to tens, even hundreds of virtual machines; every virtual machine is able to host one commodity operating system and ensures its execution not to be corrupted by the others. Therefore, important services do not need to be hosted by different physical machines any longer, but settle down in different virtual machines. This approach from virtualization is normally called service consolidation [5, 23]. It improves the security and robust of service without increasing any hardware cost and keeps, even increases, the utilization of physical machine.

Furthermore virtualization might also increase the portability and flexibility of services, since it is possible to migrate whole running environment of service dynamically from one virtual machine to another either within the physical machine or among different physic machines [5, 15].

For the developer of system level software, especially in the area of kernel development and operating system courses, virtualization also shows its value [11]. Since virtualization provides the virtualized full capability environment of a real machine, developers can test their system software in a virtual machine. They can get more debug information because the state of virtual machine can be more easily retrieved than from a physical machine. Moreover, virtualization makes it possible to co-host experiment system and mature system on the same machine. This decreases the risk for testing system in a real application environment because the mature system can take over the job when the experiment system crashes. This also improves the efficiency of experiment, because developer is encouraged to test experiment system in the real world with the help of virtualization technique.

2.1.2 The Challenges in Virtualization

Since platform virtualization is mostly used to host conventional operating system, we would like to shortly introduce the duty of conventional operating system. This is helpful for us to explain the major challenges in virtualization.

2.1.2.1 Conventional Operating System

A conventional operating system (OS) is usually the basic software layer between applications and the hardware platform. Its common duty is managing the hardware resources and distributing them to the various programs competing for them [29].

From the view of applications' developers, OS provides sufficient convenient interfaces for them to use the hardware resource, e.g. CPU, memory and I/O device. The interfaces achieve the following purposes:

1. ***Simplifying the usage of hardware [8].*** Applications don't need to face the low-level hardware interfaces any longer. A sequence of complex hardware operations may be abstracted into one simple interface provided by OS.
2. ***Centralizing the control of hardware.*** Since the interfaces are provided by OS, the behaviors of them are under the control of OS. Any hardware access is only possible with the permission of OS.

In another word, all applications except OS can't directly retrieve the hardware resources. They must obey the rules(interfaces) provided by OS to acquire the resources for their tasks. Interfaces not only simplify the usage of hardware but also limit the possibilities for communication between applications and hardware resources.

From the view of conventional OS's developers, OS is the lowest software management layer in the whole computer system. Only OS itself can directly control the critical behaviors of hardware. All applications running in OS is fully controlled through interfaces. Moreover, they don't need to care about conflict between two concurrently running OSes on one machine, since one OS is sufficient for managing the hardware resources so that concurrent execution of OSes is unnecessary (before the appearance of virtualization).

In conclusion, conventional operating system is designed as the unique resource manager in one physical machine. It prevents illegal access on the hardware resource, correctly allocates resources to the applications, and provides abstracted interfaces to simplify development. However, it has no ability to deal with unexpected changes in hardware's state and configuration.

2.1.2.2 Virtual Machine and Virtual Machine Monitor

From the introduction about conventional OS above, it is obvious that they are not designed to share hardware platform with each other. However, one promise by platform virtualization is allowing concurrent execution of many conventional OSes on the one physical machine. How to carry out this promise is the major problem for platform virtualization. Current infrastructure of platform virtualization usually divides whole virtualization environment into three layers. The lowest layer is the real hardware layer. The second layer is called virtual machine monitor (VMM). And the highest layer is the layer of virtual machine, which hosts the conventional OS. Figure 2.1 illustrates the infrastructure.

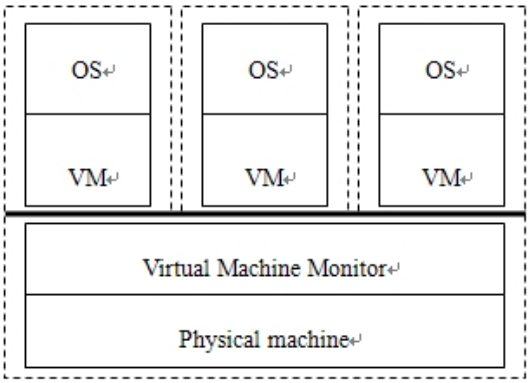

Figure 2.1: Infrastructure of Virtualization

Virtual machine, more exactly a system virtual machine, is a duplication of a real complete computer hardware system. It has normally its own virtualized CPU, memory system, storage system and even virtualized peripheral devices. In fact, virtual machine works like a cheater. Although conventional operating system is designed as unique resource manager in a computer system, this purpose cannot be achieved in the virtualization environment. So virtual machine was constructed to host(cheat) conventional OS and let it consider that it is unique and has the full control of hardware platform. But in reality, it shares the real hardware resources with the other operating systems, which are also hosted by virtual machines.

While conventional OS is no longer the real resource manager of hardware resources, a central resource manager is certainly necessary in every computer system. Virtual machine monitor(VMM) takes over this role in platform virtualization. In fact,

it works similarly to the conventional OS. It manages the resources, allocates them to virtual machines above it, and ensures that the OSes hosted by VMs are able to run concurrently. Moreover, as a central resource manager, it must keep all accesses on hardware resources under its control. This requirement brings more challenges in the layer of VMM. Although VMs are invented to cheat the conventional OS, they are usually transparent to the hosted OS(Who want to know that he/she is dealing with a cheater?!). Therefore, from the nature of conventional OSes, they still try to directly communicate with real hardware resources. If the messages that can affect the critical behaviors of hardware are not blocked and redirected to the cheater(VM), VMM will fail to correctly manage resources and to ensure the concurrent execution of OSes. So the VMM's other character is "redirector". It must find out all harmful messages from conventional OS and redirect them to the cheater so that these messages aren't able to corrupt the whole environment and can be dealt with safely and correctly. Thus, which kind of messages can control and affect the behavior of hardware will be next topic in our thesis.

2.1.2.3 Sensitive vs. Insensitive

In virtualization, the messages used by an OS to communicate with physical resources can be divided into two groups, sensitive group and insensitive group. In this thesis, all that can be safely executed without VMM's intervention are *insensitive*. In contrast, what can cause others' corruption and must be redirected is *sensitive*. We will cover the discussion of sensitive and insensitive over two kinds of most important hardware resources, CPU and memory. Although I/O device is another kind of most important hardware resources, the sensitive behaviors on I/O device are mostly the results of sensitive behaviors in CPU and memory.

Challenge 1: Sensitive Instructions

CPU is the central commander in today's computer system. It receives requests from the software layer and feeds them itself or dispatches them to the other components in the system. After the work is finished, it returns the result to the software layer if necessary. For this purpose of allowing the software layer to send requests, CPU provides a set of instructions(messages), called Instructions Set Architecture (ISA), to the software layer. Moreover, in order to help resource manager to control the hardware resource, CPU provides different execution modes to the software layer, called privileged mode and non-privileged mode. A program can have full access on the hardware resource in privileged mode, whereas the ability is limited when it is executed in non-privileged mode.

Past researchers on virtualization have given out classification for ISA from the respect of virtualization. The most famous classification came from Popek and Goldberg in 1974. In their papers [11, 24], instructions were classified according to two different aspects.

According to the first aspect, the execution mode, instructions are separated into non-privileged group and privileged group.

1. ***privileged instruction*** is the instruction that causes traps when it is executed in non-privileged mode.
2. ***non-privileged instruction*** is, in contrast, allowed to be executed in both privileged and non-privileged mode.

Although this characteristic of instruction is independent of virtualization process, it is used as a way to block sensitive messages in most virtualization technologies.

According to the second aspect, the sensitivity, instructions are grouped into control sensitive, behavior sensitive, and innocuous.

1. ***control sensitive*** instructions are those that attempt to change the configuration of resources in system.
2. ***behavior sensitive*** instructions are those whose behavior or result depends on the configuration of resources (the content of the relocation register or the processor's mode)
3. ***innocuous*** instructions are those that are neither control sensitive nor behavior sensitive.

From the opinion of Popek and Goldberg, both control sensitive instructions and behavior sensitive instructions executed by guest OS must be caught by VMM. In contrast, the innocuous instructions can be permitted to be executed directly on CPU.

The reason for catching every execution of control sensitive instruction is obvious, since they can unexpectedly change hardware behavior so that VMM's resource management is corrupted. However, some of behavior sensitive instructions are not essential to be directly caught, e.g. "MOV" in IA-32 architecture, because their sensitivity aren't their nature but dependent upon the object on which they operate. Moreover, these behavior sensitive instructions are usually used very frequently in program. To catch all of their executions significantly pulls down the performance of virtualization environment. Therefore, most of virtualization technologies don't directly monitor all executions of this kind of behavior sensitive instructions, but concentrates on the objects that cause instructions' sensitivity. Therefore, In this thesis we reorganize the instructions into two groups:

1. ***virtualization sensitive*** instruction is the instruction whose execution must be caught and redirected by VMM. This kind of instruction includes all control sensitive instruction and part of behavior sensitive instructions, e.g. "SIDT" in IA-32 architecture.

2. ***virtualization insensitive*** instruction is the instruction whose execution is not necessary or not always necessary to be caught and redirected by VMM. This kind of instruction includes all innocuous instructions and the behaviors sensitive instructions whose sensitivity is dependent on the their operand.

Challenge 2: Sensitive Memory Object

Memory is the major component for retrieving and storing data during the computer is running. The data stored in memory are usually abstracted to different kinds of "memory objects", represented by pieces of address space, according to their properties and usages. In the end of last section, we mentioned that some virtualization insensitive instructions' sensitivity is dependent upon their operands. The instructions used to operate on memory object, e.g. "MOV" in IA-32 architecture, are such virtualization insensitive instructions, because some kinds of memory objects can raise their sensitivity. We call the memory object, which can raise virtualization insensitive instructions' sensitivity, "*sensitive memory object*". The operations on sensitive memory object are called "*sensitive memory operation*". Since most of modern virtualization technologies prefer to monitor the appearance of sensitive memory object in order to catch-and-redirect sensitive memory operations, it is necessary to understand what kinds of memory object are sensitive.

Memory Mapped I/O

For communicating with I/O device, two ways without additional I/O processor are provided in modern computer system, i.e. port-mapped I/O (PMIO) and memory-mapped I/O(MMIO). While all accesses on I/O device through PMIO can be controlled by catch-and-redirecting virtualization sensitive instructions because CPU provides dedicate control sensitive instructions for PMIO, I/O communication through MMIO is accomplished by using virtualization insensitive instruction and brings the first kind of sensitive memory object in our thesis.

In MMIO system, I/O communication doesn't have a dedicated address bus, but shares the same address bus with memory system. CPU assigns a piece of space in the linear memory address space for the usage of I/O communication. This space could be dynamically assigned. After that, all accesses on MMIO device are accomplished through usual instructions for memory access. I/O device maps this addresses into its register and watches out the address bus. When a memory access happens on these addresses, the I/O device will automatically take corresponding actions. Since the accesses on the address space allocated to MMIO device can cause sensitivity, they are sensitive memory objects and must be cared about by VMM.

Memory Management in Virtualization

In non-virtualization environment a conventional OS manages the whole memory resource of a physical machine. It can access the whole physical address space; every physical page frame, segment frame are managed through certain data structures, called "*memory management data structure*". However, this can't be achieved in virtualization environment any longer because the real resource manager is not the guest OS, but VMM. On one side, VMM must ensure that guest OS can only manage the memory resources that are allocated to its hosting VM. On the other side, VMM must make guest OS to believe that they can access the whole physical address space and successfully manage all memory resource so that guest OS can run correctly in VM.

To achieve the purposes above, memory management data structures(MMDS) in guest OS become the key of solution. Since these data structures are the abstraction of physical memory system in OS, controlling all accesses on these data structures is sufficient to control the behaviors of memory management in guest OS. In another words, the accesses on MMDS is sensitive to VMM. Since these accesses are usually accomplished through virtualization insensitive instructions for memory access and MMDS raises their sensitive, all memory objects contained in MMDS are sensitive memory objects.

MMDS consists of different data structures according to different computer architectures. For example, in IA-32 architecture, MMDS includes page tables, segment tables and interrupt table, whereas in some architecture without segment support, segment tables disappear. interrupt table is included in MMDS because one entry of interrupt table is responsible to deal with page fault, which is an important concept used by OS to manage the memory allocation.

2.1.3 Conclusion

So far we have introduced the brief components and major challenges in virtualization technology. The virtualization environment consists of two parts, VMM and VMs. The major challenge in virtualization is to ensure the conventional OS to correctly run in their own hosting VMs on the same physical machine. The key to solve this problem is

to monitor the virtualization sensitive messages from the OS. These messages include two kinds of operations, virtualization sensitive instructions and the sensitive memory operations.

In the following sections we will introduce how the different virtualization techniques find their own keys to open the door to the world of virtualization.

2.2 Virtualization Technologies

Different virtualization technologies appeared during the development of virtualization. In order to show the motivation of this paper clearly, we would like to discuss and compare different major technologies of virtualization in this sections.

2.2.1 Pure-virtualization

Pure-virtualization is designed to host the original operating system in the virtualization environment. That means that the engineering effort for porting conventional OS to virtualization environment is zero.

2.2.1.1 Sensitive Instructions in Pure-virtualization

Traditional pure-virtualization, i.e. from the opinion of Popek and Goldberg, relies on the computer's architectural separation of privileged mode and non-privileged mode. Popek and Goldberg indicated that, in a *(classically) virtualizable* architecture, all sensitive instructions must be privileged, either the instruction itself must be executed in privileged mode or the sensitive memory objects are only accessible in privileged mode. That means that execution of any sensitive instruction, both control sensitive and behavior sensitive, in non-privileged mode will cause system trap. Since guest OS runs in non-privileged mode in virtualization environment, a classical VMM [3] can catch all illegal operations from guest OS by monitoring system traps and then emulate the corresponding operations for guest OS.

Unfortunately in some modern computer architectures this trap-and-emulate mode doesn't work. For example, x86 is not classically virtualizable. Some sensitive instructions, e.g. $POPF$ [25], are allowed to be executed in both privileged and non-privileged mode with different behaviors. Furthermore, trap-and-emulate mode brings significant overhead into virtualization environment and pulls down the performance. In order to virtualize the popular classically unvirtualizable architecture and improve the performance, *binary translation* is widely used in modern pure-virtualization technologies, e.g. VMware [3] and Virtual PC.

Binary Translation

The core component in binary translation is called binary translator(BT). BT works like an instruction interpreter, but more efficient. It translates a block of instructions instead of single instruction at once, e.g. maximal 12 instructions per block in VMware. The insensitive instructions are identically translated and passed to physical CPU, while the sensitive instructions are caught and emulated by VMM. Since all effects and results of an instruction are under control of BT, the lack of trap mechanism for some sensitive instructions on classically unvirtualizable architecture is overcome. Furthermore, since VMM can recognize the sensitive instructions and translate them into set of insensitive instructions before they are passed to physical CPU, the traps from illegal execution

of sensitive instructions can be avoided. Nevertheless, traps still exist in modern pure-virtualization in order to locate sensitive memory operations.

2.2.1.2 Sensitive Memory Objects in Pure-virtualization

Memory Traces

In section 2.1.2.3 we introduced that sensitive memory object must be protected from direct access of guest OS. In pure virtualization this is achieved by memory tracing, which bases on hardware page protect mechanisms provided by modern computer architectures. Particularly, sensitive memory object's access property is changed into write-protect or read-write-protect so that any direct access from guest on them is trapped and passed to VMM.

2.2.1.3 Problems in Pure-virtualization

Although memory traces can indicate all sensitive memory operations, it is possible to cause negative effect on performance. At first, data structures containing sensitive memory object, e.g. interrupt table, can be small enough to coexist with insensitive memory objects in the same page. Because sensitive memory object must be protected, the whole page, includes insensitive objects, must be set to write-protect or read-write-protect. If these insensitive memory objects are accessed very frequently by guest, they will cause a huge overhead from traps. Moreover, some sensitive objects are usually very frequently accessed, e.g. page table entry(PTE). Tracing every access on them is also a source of overhead.

Furthermore, pure-virtualization typically uses shadow data structure in VMM to represent the real state of MMDS of guest OS. VMM should prevent incoherence between shadow structures and their original data structures. For example, every page table in guest OS has a shadow page table in VMM. The accessed bit and dirty bit must be synchronized before they are accessed by guest OS. The synchronization can be executed during the tracing progress. As we described above, this method causes a heavy overhead because of traps from tracing. On the other hand, if they are not synchronized by tracing, either a large number of page faults in shadow page table or an expensive context switch to prevalidate shadow page tables for new context is required. The developers of modern pure-virtualization indicates themselves [3], "Striking a favorable balance in this three-way trade-off among trace costs, hidden page faults and context switch costs is surprising both in its difficulty and its criticality to VMM performance".

2.2.2 Para-virtualization

Para-virtualization [2] is originally designed to achieve high system performance. Before Xen [23] appeared, most para-virtualization technologies were designed for special guest OS which does not require full functionality of a hardware platform, e.g. Denali [4] and Disco [9]. Although they achieve the high performance, they can only be deployed in limited areas for special purposes. Since our discussion in this thesis focuses on the platform virtualization for hosting commodity operating systems and Xen is the first and most successful para-virtualization technology for this purpose, we will discuss the technique used in Xen in this section.

Unlike pure-virtualization, the basic idea of para-virtualization is to modify guest OS to fit in virtualization environment. In para-virtualization, the guest OS is modified to be able to run in a non-privileged mode; all necessary sensitive operations in a guest

OS are carried out by using *hypercalls* provided by *hypervisor*. We'd like to talk about this approach from both aspect of virtualization sensitive instructions and aspect of sensitive memory operations.

2.2.2.1 Sensitive Instructions in Xen

In pure-virtualization, catching and redirecting sensitive instruction is very important, since this ensures guest OS to function correctly on top of VMM. However, after OS is modified in para-virtualization, no sensitive instruction is necessary to be caught at runtime any more. Modified OS only uses the hypercalls instead of sensitive instructions to finish their sensitive work; hypercalls are fully under the control of hypervisor. From some aspects, the concept of hypercall is similar to the concept of interface in conventional OS introduced in section 2.1.2.1. Because no trap-and-emulate phase is required in para-virtualization, the overhead from it also disappears.

2.2.2.2 Sensitive Memory Objects in Xen

In order to efficiently work with sensitive memory objects, Xen uses a similar strategy, by modifying the original operations on them to fit in para-virtualization environment.

For direct MMIO accesses, Xen replaces them with hypercalls to communicate with I/O device so that guest OSes do not operate on MMIO sensitive objects any more but acquire the I/O service from Xen.

For MMDS sensitive objects, Xen doesn't adapt the concept of shadow data structure from pure-virtualization but allows guest OS to register their data structures directly with physical component. For simplicity, we will only discuss page table as an example in this section, since the other structures, e.g. segment table, are dealt with in similar way. In Xen, guest OS and Xen manage the same page table. However, in order to prevent guest from making unacceptable changes, guest OS is only allowed to directly read PTEs after it registered and initialized the page table. All updates in page table must be validated and accomplished by using hypercalls provided by Xen. This approach, on one side, eliminates the overhead of trapping and synchronization in pure-virtualization. For instance, in Xen guest OS can directly read the status bits in PTEs, whereas every access from guest on these bits must be validated by VMM in pure-virtualization because incoherence possibly exists between shadow page table and the page table used by guest OS. On the other side, hypercall for updating page table can be designed to be capable for batching update requests. With the modification on memory management strategy in guest OS, the performance can be more improved.

2.2.2.3 Problem in para-virtualization

Although para-virtualization achieves its purpose, high performance, this success is based on the modification of guest OS. The modification is typically carried out manually and, thus, increases the engineering effort for porting a conventional OS into the para-virtualization environment. Furthermore, since para-virtualization statically replaces the sensitive operations in OS source code with its hypercalls, it tightly couples virtualized guest OS and the targeted para-virtualization hypervisor. A paravirtualized OS can be executed neither in another para-virtualization environment nor directly on physical machine.

2.2.3 Pre-virtualization

So far it seems that to gain both performance and low engineering cost is an impossible mission for virtualization. But the appearance of *pre-virtualization* throws a light in the world of virtualization.

Pre-virtualization is a technique trying to unite the advantages from both pure-virtualization and para-virtualization [18]. Pre-virtualization, on one side, still uses the ideas of hypercall and modification on source code to keep high performance. However, it tries to reduce the engineering effort by finding a way to automating modification progress. On the other side, it borrows the idea of binary translation from pure-virtualization to provide flexibility of virtualization.

2.2.3.1 Afterburning

In order to automate the modification progress on guest OS's source code, pre-virtualization introduced a new concept, "afterburning".

Afterburning Virtualization Sensitive Instructions

The sensitive behaviors caused by virtualizing sensitive instructions are replaced in para-virtualization with hypercalls for high performance. However, this progress is difficult to be automated directly in the original source code. Therefore, pre-virtualization converts source code into assembly language level first and then uses an assembly language parser, called *afterburner*, to modify the assembly source code automatically. The nature of assembly ensures that virtualization sensitive instructions can be accurately located and replaced.

The modification works in manner of find-and-replace. At first, afterburn locates sensitive instructions by identifying their mnemonic name in assembly language. After finding a sensitive instruction, afterburner doesn't replace it directly with the hypercalls, but with a scratch space [17]. Every space has its start label and end label; this information is stored in a table section in the binary for later processing. After all assembly source codes are afterburned, they are compiled and linked into a pre-virtualization friendly binary.

Afterburning Sensitive Memory Objects

Although afterburner can efficiently find and replace virtualization sensitive instructions automatically, it seems to be helpless for finding memory sensitive operations without additional help. At first, using mnemonic name of instructions to locate sensitive memory operations is unavailable, since the instructions used in sensitive memory operations are virtualization insensitive. Furthermore, because of the simple type system in assembly language, it is difficult to recognize the sensitive memory objects through their special types. Therefore, beside manual modification current pre-virtualization proposes a solution called *profile feedback* to achieve the automatic modification on sensitive memory operations.

Profile feedback requires incomplete afterburned guest OS binary to be executed in virtualization environment and uses memory tracing to trap illegal accesses on sensitive memory objects. If a sensitive memory access happens, the trap handler first releases the access protection on memory object and accomplishes the access under control of VMM, then the information of the instruction that caused this access is recorded by a profiler. After that, the sensitive memory object is set to be access protected again in order to trap next illegal access. The collected information will be used later

by afterburner to locate and replace sensitive memory operations in assembly level. However, in order to carry out profile feedback approach VMM runtime must adapt more expensive approaches.

2.2.3.2 Binary Rewriting

Another purpose of pre-virtualization is to overcome the inflexibility of para-virtualization. Since inflexibility of para-virtualization is caused by static hypercall replacement in source code, pre-virtualization tries to use idea of binary translation to dynamically replace sensitive operations with hypercalls from different virtualization technologies. However, two differences exist between binary translation in pure-virtualization and that used in pre-virtualization. At first, binary translator in pure-virtualization accomplishes its work without any additional help from its target binary, whereas in pre-virtualization binary is afterburned and contains helpful information for translation. Moreover, binary translator in pure-virtualization translates binary during the whole life cycle of binary, whereas pre-virtualization's translation happens only during the loading-stage of binary. Therefore, we call binary translation in pre-virtualization "*binary rewriting*" in this thesis in order to distinguish it from binary translation in pure-virtualization.

Binary rewriting is accomplished by the most important component in pre-virtualization, In Place VMM(IPVMM). In section above, we introduced that afterburner creates scratch space tables for sensitive operations and stores the tables in special sections in binary. During the loading stage, IPVMM retrieves these scratch space tables and follows the information contained in table to locate the scratch spaces. The scratch spaces will be filled with codes that actually implement the sensitive accesses.

2.2.4 Hardware Virtualization

As we described in pure-virtualization, traditional x86 architecture is classically unvirtualizable. However, with the booming of virtualization technique, both Intel and AMD introduce hardware extensions to make their hardware products classically virtualizable. The virtualization technique based on these extensions is called *hardware virtualization* [3].

2.2.4.1 Sensitive Instructions in Hardware Virtualization

With the hardware extensions all sensitive instructions can be secured by validating their privilege property. In fact, an in-memory control block(VMCB), a new instruction, $vmrun$, and a new execution mode, *guest mode*, are added into x86 architecture. VMM sets the control bits, which are used to validate privileged conditions, in VMCB and executes $vmrun$ to start guest OS's execution in guest mode. When one of the conditions is filled, CPU will pause execution of guest OS, save its state in VMCB, leave guest mode, and pass control back to VMM.

2.2.4.2 Sensitive Memory Objects in Hardware Virtualization

However, current hardware extensions don't provide an efficient new way to deal with the sensitive memory objects. Nevertheless, CPU manufactures still suggest some solutions based on new extensions. Virtual TLB is one of these solutions.

Since in modern CPU architecture the TLB is used to cache the translation information between physical address and linear address, page table hierarchy is not in charge of all virtual address translations. CPU only fetches translation information from page table if it is missed in TLB. Therefore, if the TLB is big enough, it can cache all information in page table and take over all work of translation from the page table. Virtual TLB technique uses this idea and constructs a page table as a virtual TLB in the VMM to cache and control all translations existing in a guest's page table hierarchy so that the sensitive access on page table can be avoided.

To carry out Virtual TLB, control bits in VMCB about page fault, invalidating TLB entry, and $CR3$ writing must be set, so that the control can be passed back to VMM when these sensitive operations happen. Page fault is used to cache memory translation in virtual TLB for current page table hierarchy. Catching invalidating TLB entry is useful to eliminate invalid translation entries in virtual TLB. At last, since there is only one virtual TLB existing in VMM, it must be flushed as real TLB when $CR3$ register is modified, i.e. a task switch happens.

However, this solution is very expensive. Since virtual TLB is not a flat structure as real TLB, VMM is responsible not only to invalidate cached translation entry, when a trap of invalidating TLB entry happens, but also to validate whole page table that contains this invalid entry so that the potential empty page table can be deallocated. This checking mechanism is expensive. Furthermore, the frequent virtual TLB flushing and reconstruction caused by task switch also significantly pull down the performance of virtualization environment.

2.2.5 Conclusion

Till now we have introduced most popular software virtualization technologies and hardware virtualization technology. While how to eliminate the negative effects from virtualization sensitive instructions and sensitive memory objects is the major challenge in all virtualization technique, how to balance performance penalty and engineering effort for porting guest OS is also an important aim for different virtualization technologies. We conclude our discussion for virtualization with table2.2.5. In the table, virtualization sensitive instruction is shorten with VSI; sensitive memory operation is represented by SMO.

Table 2.1: Result of evaluation

	Performance Penalty		Engineering Effort		Hardware
	VSI	SMO	VSI	SMO	Acceleration
pure-virtualization	middle	high	low	low	no
para-virtualization	low	low	high	high	no
pre-virtualization	middle	low	low	high	no
harware virtualization	middle	high	low	low	yes

From the table above we figure out that efficiently virtualizing sensitive memory operation is more difficult than virtualizing virtualization sensitive instructions. The solutions from different virtualization technologies either bring significant performance overhead or cost huge engineering effort. Although pre-virtualization introduces "afterburning" and successfully balances the performance penalty and engineering effort for virtualizing virtualization sensitive instructions, it still fails to find an elegant way for virtualizing sensitive memory operations. This situation brings the motivation of

this thesis, to find a way to balance performance penalty and engineering effort for virtualizing sensitive memory operations.

2.3 Code Analysis

The root of afterburning technique in pre-virtualization is automatically analyzing assembly source code by using code analyzer. Although current afterburner cannot efficiently automate the find-and-replace progress for virtualizing sensitive memory operations, the idea, *code analysis*, behind afterburning is still useful for us to find our solution for the problem caused by sensitive memory objects. In this section, we will briefly introduce code analysis and some related works either referred or used in our proposed solution.

2.3.1 Overview of Code Analysis

Static code analysis, shortly code analysis, is the analysis of program's source code written in a certain programming language. The purpose of code analysis is automatically understanding the structure of source code, retrieving the useful information, and applying the changes on original codes. Tool used for code analysis is usually called *code analyzer*, and consists of a language *grammar parser* and a *code rewriter*. While grammar parser is in charge of correctly understanding target source code from the syntactic aspect, code rewriter is responsible to find the useful semantic meaning hidden in source code and use them to apply the modifications.

2.3.2 Grammar Analysis

Programming Languages are typically designed to be human-language-similar so that they can be used comfortably by programmer to describe their wishes. Therefore, like human language, programming language is defined through a set of lexical rules and grammatical rules.

Lexical Rules. The lexical rules are used to define the set of "vocabularies" in programming language. While some vocabularies have constant meanings and protected by programming language, e.g. "keyword", user can also name their own vocabularies by following lexical rules. Of course, the lexical system of a programming language is not as complex as human language, since the world of computer is much simpler than our human society.

Grammatical Rules. After programmer remembers the "vocabularies" in a programming language, they must know how to use them. While human uses grammar to constructs sentence, paragraph, and even article or book, programming language uses grammatical rules to construct the expression, statement, data structure, function, and some other abstracted structures. Grammatical rules in a programming language show the programmer all possible combinations of "vocabularies" and define their meanings. Programmers must obey the grammatical rules to construct their code.

Grammar analysis is the techniques used to understand lexical rules and grammatical rules in a programming language. Source code is input as a character stream into

grammar parser. Parser typically recognizes the legal vocabularies, also called *token*, and omit the redundancy information, e.g. white space, from the input stream according to lexical rules. After that, it uses grammatical rules to construct tokens into a tree structure, called abstract syntax tree(AST). This tree is finite, labeled, and directed. It provides organized and complete information about source code for code rewriter.

Parser Generator

However, constructing a correct and efficient grammar parser manually is a hard work. Fortunately, a tool, called a *parser generator*, appears to help user efficiently construct their grammar parsers. Usually, a parser generator accepts the syntactic rules, represented by a notation system, from user and generates source code of grammar parser. User can build parser directly or after hand-tweaking the source code. The popular notation systems used by parser generator include Backus-Naur form (BNF), Extended Backus-Naur form (EBNF), and Regular Expression(RE).

BNF was introduced by Backus and popularized by Peter Naur [30]. It is very simple but precise and powerful enough to describe any programming language [22]. A BNF specification is a set of derivation rules, written as:

```
<symbol> ::= <expression with symbols>
```

where `<symbol>` is a nonterminal, and the expression consists of sequences of symbols and/or sequences separated by the vertical bar, "|", indicating a choice; the whole being a possible substitution for the symbol on the left. Symbols that never appear on a left side are terminals.

EBNF and RE are actually derived from BNF. They add new operators so that user can more easily use them to describe language. The most popular and readable extension includes [10]:

grouping Parentheses are used to define the scope and precedence of the operators. For example, "hope—hold" can be shortened to "ho(pe|ld)".

quantification A quantifier after a character or group specifies how often that preceding expression is allowed to occur. The most common quantifiers are ?, $*$, and $+$:

- $*$ means that expression or symbol can be repeated 0 or more times
- $+$ means that expression or symbol can be repeated 1 or more times
- ? means that expression or symbol can be repeated 0 or 1 time

2.3.3 Code Rewriting

After successful grammar analysis, the next step in code analysis is using code rewriter to modify source code and outputting the result of rewriting if necessary. We have mentioned early that code rewriter needs to retrieve enough semantic information by analyzing AST tree so that it can correctly apply the modification. That means that the ability of a code rewriter is limited by how much semantic information it can get from the source code. The amount of semantic information is usually dependent on the kind of programming language.

2.3.3.1 Classification of Programming Language

Since our thesis focuses on recognizing sensitive memory object, which is dependent upon type system of programming language, we'd like to discuss the classification of programming language from the aspect of type system.

Type system helps programmer to represent raw data in computer system with different abstracted meanings, e.g. number and string. For our discussion, we classify type system into three abstraction level in this thesis.

Machine Abstraction Level This is the simplest type system of programming language. In this kind of type system, data are only recognized with binary type, i.e. "1" and "0". This kind of type system is typically only used by machine language.

Basic Abstraction Level The type system in basic abstraction level provides a set of low level abstraction on data, e.g. number; the available types in this kind of type system is limited. Furthermore, programmer can and only can use the type provided by type system.

Complex Abstraction Level The programming language uses type system in complex abstraction level provides richer kinds of types than those in basic abstraction level. In order to improve the flexibility of type system, they even allow user to define their own types.

According to the classification of type system above, we divide programming into two groups, called *simple type programming language* and *complex type programming language*. The simple type programming have a type system of either machine abstract level or basic abstraction level, while the complex type programming language's type system is in complex abstraction level.

2.3.3.2 Assembly v.s. C in Code Rewriting

As example, we'd like to talk about two programming languages that are widely used in operating system development, assembly language and C language. Particularly, the code rewriting in current afterburning technique is coupled with assembly language.

In assembly, Every numeric machine code has its mnemonic representative, called *mnemonic instruction*. Since there is only one-to-one correspondence between machine code and mnemonic instruction, the sensitivity and privileged feature of a machine code in virtualization are kept by its representative in assembly. Current afterburning technique uses this characteristic of assembly, constructs an assembly code rewriter, and successfully automates find-and-replace idea from para-virtualization. However, assembly language provides a very simple type system and belongs to the group of simple type programming language. All memory objects are represented as numeric addresses in assembly so that it is very difficult to distinguish sensitive memory objects from other harmless objects. This is why current afterburning technique cannot provides a satisfactory solution for rewriting sensitive memory operations.

In contrast, C is a complex type programming language. The rich type system in C can overcome the short comings of assembly language. Programmers can define data with various primitive data types in C, even with the type defined by programmers themselves. C doesn't mess up the data of different types and is able to recognize them accurately when they appear. In practice, the sensitive memory object is usually declared as a dedicated type. A C source code rewriter can simply differ a sensitive memory object from other common objects through the semantic meaning of its types.

Therefore, through a C source code rewriter, it is also possible to automate the find-and-replace progress for sensitive memory operations.

2.3.4 Related Works

Finally, we would like to introduce some related work before we finish this chapter. Our approach either borrowed ideas from them or used them in our implementation.

2.3.4.1 Aspect-Orient Programming

Aspect-Orient Programming(AOP) is a new programming concept to solve *crosscutting* problem that cannot be solved by procedure-orient programming(POP) and object-orient programming(OOP) [12]. The crosscutting problems are caused by the codes that are scattered and tangled, called crosscutting concerns. The traditional programming techniques can not separate these codes and modularize them. AOP introduces the concepts of *join point model* and *aspect* to improve the modularization of a programming language. The scattered and tangled codes are separated and modularized into different aspect, while join point model indicate where these aspects should be inserted into other functionality modules, which is modularized in traditional way, e.g. object-orient way.

AOP can be applied into different modern programming languages by either extending their original compiler or generating new code analyzer for AOP. We call both of them *AOP code analyzer*. The AOP code analyzer gets information from join point model and automatically combines the aspects with other source code. Some research are concentrating on applying AOP concept to improve the development of operating system. The range of these researches covered interrupt synchronization [7], and paging prefetching [31].

Although AOP is originally designed to enhance programming by changing programmer's coding style, i.e. using AOP concept instead of directly insert crosscutting concerns into different function modules, the concept of join point model and aspect are helpful in the progress of our research.

2.3.4.2 ANTLR - Another Tool for Language Recognition

Finally we want to introduce a concrete parser generator to finish our discussion in this chapter. The generator to be introduced here is named ANTLR.

In contrast, to some popular parser generator, e.g. YACC, ANTLR is a parser generator designed to create LL(k) grammar parser, which is able to do recursive-descent parsing the source code [21]. The most important idea in ANTLR is the predicate system. The predicate system helps user to overcome the difficulty from arbitrary expression by using the syntactic and semantic context. In practice, the introduction of this predicate system eliminates the hand-tweaking output of a parser generator and increase the ability of the parser to solve the difficult parsing problem.

ANLTR uses EBNF as is its notation system. As we described, EBNF is powerful and simple. This characteristic make it user-friendly. An additive expression can be described in ANTLR as follow:

```
additiveExpr
    : {1==1}? nr1:NUM PLUS NUM SEMI
      { printContent(nr1);}
    ;
```

```
NUM
    : ('1' .. '9')('0' .. '9')*
    ;
PLUS
    : '+'
    ;
SEMI
    : ';'
    ;
```

"additiveExpr", "NUM", "PLUS", and "SEMI" are the name of syntax rules where "additiveExpr" represents a grammatical rules and the three latter rules represent lexical rules. With the introduction in previous section, the meaning of these rules is not difficult to be understood. However some addition operators appear in this example [20]:

.. double dot is used to represent the meaning of "range". For instance, "1..9" means one of digits from 1 to 9.

{} the character sequences enclosed in curly braces are semantic actions.

: colon is used in ANTLR either to separate rule name and the rule definition or to define a representative for an Token. For example nr1 represents the token of "NUM" above.

Beside these operators there are two more important operators in ANTLR. Both of them are important parts of predicate system of ANTLR. Therefore we want to introduce them here:

? the question symbol is used after right curly brace and converts the code in curly braces into a condition selection. It is used for semantic predicate.

=> this combination of the $=$ and $>$ follows parentheses. The content enclosed in parentheses before "$=>$" will be used for syntax predicate. For example, in "$(a) => (ab)$" syntax "ab" is tied to be matched if and only if "a" is matched first.

One more advantage of ANLTR is the output system. ANTLR allows user to generate parser in different programming language, e.g. C++ and JAVA. This characteristic let user to user their familiar language so that they can quickly understand the codes generated by ANTLR and improve the efficiency of development.

Chapter 3

Proposed Solution

3.1 Problem Review

In this chapter we will discuss our approach in detail. However, first of all, we would like to link the theoretical introduction with our approach, as well as to define the problem more precisely.

The motivation of this thesis attempts to find a way to balance performance penalty and engineering effort for virtualizing sensitive memory operations. One possible way is using the afterburning technique in pre-virtualization. However, current afterburning bases on code analyzing in assembly level. While the one-to-one mapping between machine codes and mnemonic instructions simplifies the mission to find out and virtualize virtualization sensitive instructions, the basic abstraction level type system makes it difficult to recognize the sensitive memory objects in assembly codes. Thus, current pre-virtualization avoid negative effects from sensitive memory operations not by using afterburning technique but by manually modifying source code.

In this thesis, we keep using the idea of afterburning technique, but enhance it through a C code analyzer. With this enhancement we can automate the process of manually modifying sensitive memory operations in pre-virtualization. We call this enhancement "*C afterburning*".

3.1.1 Solution Overview

The compilation of C source code has normally an intermediate stage so that the translation from C source codes into the machines codes can be simplified. From the side of programmer, they are allowed to control this intermediate stage and even insert new stage around the intermediate stage for their own purpose. While assembly afterburning used this characteristic and modifies the intermediate assembly before OS binary is created, we also used this feature for C afterburning.

In assembly afterburning, a stage for transforming C source codes into assembly code is added before the source codes are passes to formal compilation progress. Since our code manipulation level is higher than assembly afterburning, we add our codes manipulation stage before codes are passed to pre-virtualization's assembly parser. In this additional stage, a C code analyzer is in charge of parsing and annotating the input source codes. After it finishes its afterburning job, the result for later assembly afterburing was exported in C style again. After we introduce our C level afterburning

stage, the whole compilation progress for a C source file in pre-virtualization looks like Figure 3.1.

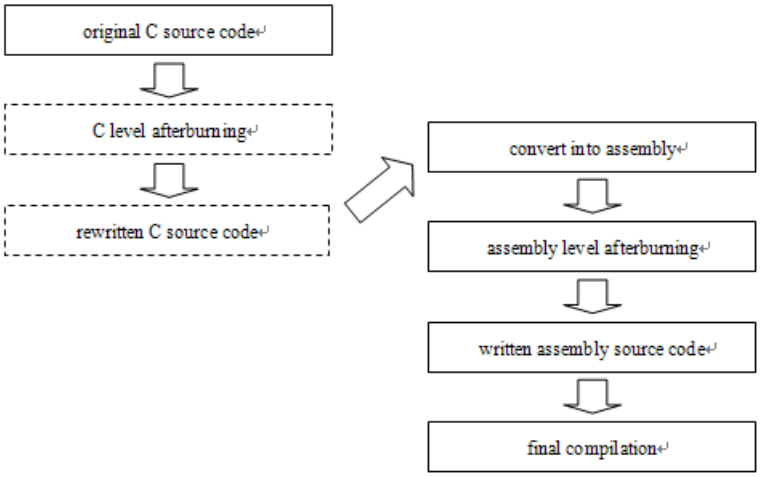

Figure 3.1: New progress of afterburning

As we described in section2.3, code analyzer is usually made up with a grammar parser and a code rewriter. In our C code analyzer, the grammar parser consists of a *lexical component* and a *grammatical component* for correctly parsing C source code; code rewriter actually accomplishes the search-and-replace mission. Furthermore, in order to help the parser recognize the targets, we also designed a simple system for user to indicate additional information.

3.1.2 Terminologies

In order to simplify the discussion, we'd like to unify the terminologies we used in the following. We call the data types that are used to identify the classes of sensitive memory objects, *sensitive type*, while the other types are called *insensitive type*. The variables used to represent a sensitive memory object are called *sensitive variable*. The operations on sensitive variable are named *sensitive operations*.

3.2 Rule Definer

The simple system for user to indicate additional information is called "*rule definer*". Although C as a high level language can classify data into different types, but to the code analyzer, they cannot distinguish sensitive types from other insensitive type. In other words, we still need to give code analyzer semantic principle so that they can exactly know which kinds of types represent our targets. Moreover, we also must tell C code analyzer which action we must take when it meets the sensitive variables. The rule definer plays the role of communication channel for these kinds of information between user and code analyzer.

In our solution, the rule definer accepts two kinds of rules. One kind of rule indicates the search-and-replace principle for the sensitive variables. We call this kind of rule "*object replacement rule*", whereas the other declares the principles for function's

replacement, called "*function replacement rule*". We shall introduce the reason for this division in next section. Now we only inform the structure that we designed for user to input these kinds of information.

Before we start to introduce the first kind of rules, we would like to shortly recall the type definition principles in C. In C language a type can be classified into *primitive type* and *user defined type*. Primitive type represents the most common simple types, including integer, double, char and so on, while user defined type allows user to construct their own types with all possible types by following C's syntactic rules. For example, user can define a type of 'STRUCT', which can contain a mixture of members from both basic type and user defined type. Since a sensitive memory object is possibly an object of a user defined type with some members and it is sometimes necessary to apply some special replacement operations on certain member, we classify the "object replacement rule" in two parts. One part tells parser how to replace actions on a variable of sensitive type, while the other part indicates what parser should do on the members if the sensitive memory object is a sensitive variable declared by a user defined type with members. However, the members are not recognized by their types but by their names. Some members are declared with insensitive types but still sensitive since they are part of sensitive object.

The second kind of rule's definition is much simpler than the first one, since it focuses only on the functions in C language. The user is allowed to define the target function's name, destination function's name and some additional information of destination functions, e.g. parameter list and return type. Parser will follow these parameters to find-and-replace the target functions in source codes.

In addition we also write a rule parser for our "rule definer" system. Since we define our rule system like a small language with limited syntactic rules, we use the same language parser generator to help us construct our rule parser. In our experimental system, the rule parser is integrated into the code rewriter of our C code analyzer. More information will be discussed in next section.

3.3 C Code Analyzer

In order to automate the C level afterburning progress, a good C code analyzer is very necessary and important. We had two choices in the beginning; either we can borrow the code analyzing component in an existing C compiler or we write a C language parser by ourselves.

There are some obvious advantages if we choose the first possibility. Firstly, a parser from a mature C compiler is usually more reliable than an analyzer written by ourselves, since it has been tested by thousands of programmers. Moreover customizing a parser from an existing compiler may save our time to understand the C language's syntax and write code for analyzing the input stream. Lastly, we also do not need to worry about how to deal with the individual extensions from different C compiler, since the code analyzing component in each compiler is certainly capable to recognize their own special definitions. However, in order to customize a code analyzing component we may need to understand the whole organization of the compiler. This is disadvantageous because the components in a compiler usually cooperate tightly with each other. Moreover, we must apply our modification for each possible C compiler, which can be used to compile the OS' source code. As different compiler normally has different design psychology and implementation strategies, to apply the modification for each of them increases significantly the engineering effort. In the worse case, we

may need to adjust our modification for different version of the same compiler, since the development of a compiler brings also many changes. Thus to modify the parser component from an existing compiler seems to be inflexible and inefficient.

Now proceeding to the second choice, although writing a C language parser costs us time to understand the syntax and semantic of C, the amount of this time is comparatively smaller, because C's syntactic rules are designed as simple as possible. You can also easily get the complete syntactic and semantic definition of C from elsewhere, e.g. internet. Furthermore, with the help from modern parser generator, constructing a C code analyzer is much easier than imagined. Most source codes of our parser can be automatically generated, when parser generator the correct syntactic and semantic rules. While it is impossible to totally eliminate analyzer's dependency upon different compilers because of the individual extensions, writing our own analyzer decreases this dependency greatly, since our analyzer can be designed to be simply extended. Furthermore, writing an analyzer saves much time for understanding the existing system's structure. Except a little more effort for learning the grammar of C, writing a code analyzer is more flexible and efficient than modifying the parser component in compiler.

As a conclusion of the discussion above we decided to write our own C code analyzer.

As we described in previous chapter, a classic code analyzer normally has two standard components, a grammar parser, which is made up with lexical component and grammatical component, and a code rewriter. At first the lexical component splits the input stream into separated tokens, then grammatical component further puts the tokens into a user defined data structure, usually a token tree. At last code rewriter accepts the tree, does the modification and writes out the result to a destination file.

3.3.1 Lexical Component

In common, the lexical component is responsible to convert the meaningless character stream into meaningful token stream. In our grammar parser, this component is not only able to recognize the C language tokens, but also some tokens from preprocessing system. C compiler uses some macro system to help programmer more comfortably and efficiently finish their work. Although this system is independent from the C's syntactical and semantic rules, macro codes are mostly mixed with the C style codes in one source code file. While this makes the whole language system more programmer-friendly, it also increases the complexity of parser. The existence of macro codes in a C source code file can even breaks the C syntactic rules. Two examples are:

```
#ifdef 0
while (1) {
#else
for (;;) {
#endif
}
```

```
typedef struct {
    int mem1;
#ifdef 0
    char mem2;
#else
    long mem2;
} test_struct;
```

In the left example, unpaired curly appears, which is illegal in C syntax. The second example shows another illegal situation raised by macro. Variable "mem2" is declared twice with different type, which isn't allowed in C. Therefore if we don't eliminate negative effect from macro system for C language analyzing, we may not retrieve the token stream correctly.

Similar to the discussion on C code analyzer, we also have two choices to eliminate the negative effect from macro system, either asking for help from the existing compiler or writing a preprocessor ourselves. But this time we make our decision on the former choice. In most modern compiler, on one hand, they allow user to separately use their preprocessor. The source code file with preprocessing information can be converted into well defined (almost) "pure" C style file after they are processed by these preprocessors. On the other hand, although C language recommends a standard set of preprocessing rules, different compilers may have their special additions. Furthermore in our approach, we do not need to modify the macro system since our work focuses only on C level analysis. Besides, there are also some problems that can complicate the implementation of a preprocessor. For example, to implement "$\#include$" rule in macro rules, we must involve the analysis of file system. Thus using an existing preprocessor is much simpler than to write one preprocessor.

However, our lexical component still has a limited ability for analyzing some preprocessing style codes, since some preprocessors do not perform their tasks very clearly and add some additional preprocessing information codes in the preprocessed source code for helping later compilation. In conclusion, our lexical component is not able to parse the complete preprocessing codes but still have ability to recognize some tokens and do corresponding actions on them. The original source code is preprocessed by preprocessor from existing compiler before they are passed to our lexical component.

3.3.2 Grammatical Component

The grammatical component was designed to accept the token stream from lexical component. As usual our component reads token one by one and tries to fit them into one of the pre-defined grammatical rules by using some parsing algorithms. After a match is found, the grammatical component will construct them into an AST subtree structure (or an AST node), and attaches this subtree (or the tree node) into the main tree. When the whole token stream is processed, a complete AST appears. This tree is prepared for code rewriter in next stage.

In design of this component some points must be mentioned. Firstly we did not follow the C language's grammatical rules very strictly, as we do not require compilation. In our parser, we only need to catch the targeted reading and writing operations on special objects; other information can be ignored. This tolerance decreases the complexity of implementation.

Secondly, we must consider the extensions in different C compiler. We must find a way to add these extensions into our parser, since they generally do not disappear after the preprocessing. As different C compiler has different extension sets, it is not feasible to mix them with the standard C grammatical rules. If we do so, we need to change our parser every time when the target C compiler is changed. So we have to find a more flexible method.

Since the most modern parser generators use Object-Orient(OO) programming language to construct the source code of target parser, this gives us a chance to simply extend a standard C language parser into a parser with special language extensions. In OO language there is a concept called *class inheriting*. This attribute of OO language allows one CLASS to inherit all properties and methods from another CLASS. Furthermore, you can even individually overload some methods of the previous CLASS. In our solution the grammatical component was implemented by parser generator as a CLASS. Thus, to match different compiler, we only need to write an extended CLASS from our standard C grammatical component CLASS.

3.3.3 Key Component - Code Rewriter

At last, we will introduce the key component in our parser, the code rewriter, which carries out our actual purpose in this thesis — automatically search-and-replace sensitive operations.

After the token stream is wholly converted into a AST tree, the tree is passed to code rewriter. In the previous stages, every token has its lexical meaning and is organized into some pre-defined forms according to grammatical rules. But they still lack semantic meaning for our purpose. The code rewriter will first recognize their basic semantic meanings, then use this information for search-and-replace progress. In order to hold semantic meaning of every interesting token, we introduced an information recorder into code rewriter.

3.3.3.1 Information Recorder

The information recorder must hold following information:

The information of all sensitive types We have discussed in section 3.2 that we designed a system to help parser recognize sensitive types. This helpful information will be stored in our recorder for later processing.

The sensitive variables While sensitive type is only an abstraction representation of one kind of data, we must find the real object of this class. In C "variable" represents an actual object. Since variable name may vary, we must record all variables of sensitive types for later rewriting progress.

The members and their types in a user defined type In rule definer, user can indicate which members in a user defined type to be replaced through matching the member name. On the other hand, the members in a user defined type can also be an object of sensitive types. The operation on these kinds of members should also be mentioned. Although user can find the name of these members and add them into rule definer, this usually requires the user to manually analyze the OS' source code, which increases the difficulty of pre-virtualization. Thus we recorded all information of user defined types so that rewriting engine can automatically use this information to find and replace the potential sensitive member in user defined type.

The position of sensitive operations This is the help information for debugging. By recording the position of sensitive operations, we can simply find the place the rewriting engine acts.

The functions' information Function is also a potential target for rewriting engine since they can return the object of sensitive object. Moreover the parameters of a function can involve sensitive objects. Thus the information about every function declared in source should be recorded.

Code rewriter uses these recorded information during its processing progress for identifying and replacing target.

3.3.3.2 Rewriting Strategy

Theoretically every sensitive operation is able to be caught by identifying the presence of sensitive memory variable. However, this is not as simple as we thought.

First of all, type cast in the C language brings in problems. Some sensitive objects may be converted into insensitive common type before some actions happen on them. Especially when they are passed into a function, they may be automatically converted into the type of function's parameter. To catch and recursively replace the operations on converted object is very difficult. Furthermore, in MMIO system, the number of types for representing different sensitive object is huge. Every device driver may have its own strategy to name its MMIO sensitive object. It not only costs a lot of time to look for them in the source code but also requires us to fill more information in rule definer so that code rewriter can recognize them. Since MMIO uses virtual memory address to represent their registers, the type of these registers may also be the same as those insensitive memory objects, e.g. "unsigned long" object. This makes the analysis more complicated.

Particularly, we noted that system programmers usually use two ways to implement sensitive operations.

Primitive Function In order to increasing the readability of codes programmer usually constructs simple and short functions that carry out some very primitive functionalities. We call this kind of function *primitive function*. For example, a function in charge of returning a member's value in a special user defined type is a primitive function. Since primitive functions are dedicated to some functionality, they are typically named according their functionalities and can be simply recognized.

Primitive Operator The primitive operator is the operator a language provided to accomplish basic read and write operations. Automatic type casting can happen when two operands are in different but convertible type classes.

In well-formed OS's source code, a kind of I/O operations are usually defined as a pair of primitive functions, one for 'in' operation and the other for 'out' operation. Although the developers of MMIO system are allowed to access directly the virtual memory address through general read/write way, e.g. primitive operator '=', they are recommended to use the I/O primitive functions provided by OS developer. Moreover, all I/O operations in standard device driver provided by OS are finished through I/O pimitive functions, since some platforms require synchronization instructions in conjunction with the memory instruction. Using primitive function can decrease the dependency upon different platforms of OS implementation, since primitive function encapsulates the platform-dependent codes in the scope of its implementation. So, on one side, if we rewrite the pair of I/O primitive functions directly, all implemented sensitive MMIO operations can be controlled through the pair. On the other side, the types of MMIO operations are few. For example, on IA-32 architecture there are only three kinds of MMIO operation for byte-wise, word-wise and double word-wise registers. With different direction of I/O operations, there are in total of only six I/O functions need to be defined. Of course, finding and replacing 6 functions is more efficient than rewriting sensitive operations by monitoring all sensitive MMIO object. Therefore, we divided our rewriting strategy into two classes.

Function Rewriting Class In our approach, we put the functions, which directly implement primitive sensitive operations, e.g. I/O primitive function, into the function rewriting class and replaced these functions with the scratch space for pre-virtualization.

Object Rewriting Class However not all sensitive operations are defined as primitive functions. Although some of them are defined as macros, but after the preprocessing stage, these macros may be replaced by primitive operators. These sensitive operations must be caught by recognizing the sensitive memory object. Therefore, for this kind of sensitive objects, we used the object rewriting strategy. We identify every sensitive memory object by comparing their type with the predefined sensitive type. Then we check their behaviors in each of their occurrence and take corresponding action.

3.3.3.3 Output and Code Formatting

Outputting the result of rewriting is also the work of code rewriter. Although as an intermediate stage, it is not essential to output result in a well formatted form, we still added a formatting system into our rewriter, so that the output codes will be readable for system evaluation and optimization.

Chapter 4

Implementation

In this chapter, we will describe in detail how we implemented a prototype system for our proposed solution. This system has covered all things we discussed in previous chapter and is planned to integrate into current pre-virtualization environment for experiment. In following sections, we will firstly introduce the environment we used to carry out our idea, and then we discuss the components in our system one by one.

4.1 The Environment of Implementation

In order to implement our prototype system, we need to consider three things. Firstly we must confirm the target source code for our prototype system. Although pre-virtualization is designed to be applied on most popular operating systems, it is currently developed and tested with Linux, since Linux is mature and its source codes can be retrieved simply from internet. Thus we used Linux as our code manipulation's target. This decision also made our solution to be able to fit in current pre-virtualization's development.

Secondly we should consider the target C compiler, with which our language parser works together, because we must take care of the extensions in different compiler. In fact, after we selected Linux as our analysis's target we had no other choices except GCC, because Linux is dependent on this compiler's extensions to optimize its codes.

At last, we still need a parser generator to help us generate our parser components. As we introduced in section 2.3.4.2, ANTLR is a very good modern parser generator. It can generate not only both lexical component and grammatical component in a standard language parser but also the customized processing component for analyzing the result of grammatical component. Therefore, ANTLR was selected to be our parser generator. ANTLR can generate the source code of parser in different high-level language, e.g. C++ and JAVA.We selected C++ as the output format of parser's source code, because the program written in C++ is usually more efficient than in JAVA and does not need Java Runtime Environment for compilation and execution. Although C++ implementation with ANTLR is short on official documentation, there are some inofficial documents in the internet fortunately. Moreover, reading the source codes generated by ANTLR is also a good way to understand the mechanism and helps us to find out useful C++ API in ANTLR's C++ library.

4.1.1 Terminologies

We retain the meaning of "sensitive type", "sensitive variable" and "sensitive operation" from previous chapter in this chapter. Furthermore, we use some new terminologies for simplify our discussion in this section:

"address object" Address object is the memory object that contain the memory address of the other memory object.

"pointer variable" The variable in C language represents a address object.

"(sensitive) operator write" It indicates the (sensitive) write operation accomplished by using primitive assignment operators, e.g. "=".

"(sensitive) primitive read" All (sensitive) read operations without involving a primitive read function are called sensitive read.

"replacing table" The replacing table contains the information of replacing for a certain type.

"replacing table list" It is a list structure for store all entry address of replacing table.

"type renaming table" It is a table used to record the renaming information of a type in source code.

"complex type table" It records the information of a complex data type, e.g. STRUCT data type. The information involves type's name, members' name and members' type.

"variable table" It is a table for recording all information retrieved from variable declaration.

"scope stack" The scope stack is a stack structure to represent the scope concept in C language.

"function table" A table used to record information about all declared function in source code.

4.2 The Implementation of Rule Definer

Rule definer is in charge of reading the helpful information from the outside. As introduced in section 3.2, rule definer must accept descriptions from the user, especially for the sensitive types. In order to organize the information, we also designed a small bunch of lexical and grammatical rules for this input component. Therefore, the rule definer looks like a small language parser to analyze the codes from user and retrieve the helpful information. we also use ANTLR to help us generate rule definer.

4.2.1 Lexical Rules of Input Component

Lexical rule indicate the vocabularies that can be used in rule definer. We defined the following keywords:

Table 4.1: Keywords

`#typename`	`#member`

#auto	#funcname
#returnType	#para
#single	#basic
#mbonly	#ptonly
#ptsingle	#assign
#y	#n

Each keyword is started with '#'. Since '#' is used for identifying macros by C compiler, it cannot be used for naming type, function or variable. Therefore, using '#' to identify the keyword can avoid the conflict between the user input information and the keywords. Beside these keywords our lexical system still includes some other tokens. For example, ':' is used to split the information so that the user can input the same type information several times; ';' represents the end of a grammatical rule. All special characters used in our lexical system have special meaning in C language for the same reason as our selection for '#'.

4.2.2 Grammatical Rules of Input Component

The grammatical rules are divided into two groups because there are two kinds of information allowed to be accepted from user. One class represents information about function rewriting, called *function rewriting information*. The function rewriting information can be made up with three grammatical rules. The first rule must start with '#funcname' keyword. Then two function names are appended. The former is for the old function to be replaced; the latter represents the new function. They are separated by ':' and ended by ';'. The second rule is used for indicating *return type* of the new function. Although the return type of new function is usually the same as that of old function, we still require the user to input it, since doing so can avoid the effort to dynamically considering return type in our implementation. The last rule allows us to define the list of parameters used by new functions. This rule is optional because some function does not need to explicitly declare their parameters. Function rewriting information might look like:

```
#funcname:old_function:new_function;
#returnType:return type;
#para:first_para;
#para:second_para;
```

The other kind of information is about the sensitive types. This kind of information must start with '#typename' keyword. It indicates that the rules following it tell the code rewriter which type need to be monitored. It also indicates how to work around with different situations happening on the variables declared with this type. It is more complex than the information about function rewriting. Thus, we divide it into three parts for discussion.

The first part of information is about the properties of the type and the corresponding actions on their variables. A type has four properties in our rule definer. '#basic' property is used to indicate whether the type is a primitive type of C language or a user defined type. Since non-primitive type in C can be only used after its first declaration, the rewriter needs this information for inserting declaration of the replacing functions into source code. The second property is '#mbonly'. It is useful for some types, which require the rewriter to care about their members but not the variable of types themselves. If it is set as '#y', then the rewriter will only monitor the members of this type, whose name and replacing rules are listed in the second part of information.

Furthermore in this situation the variable of the types themselves will not be mentioned during the rewriting phase. '#ptonly' indicates that the rewriter needs to care about the pointer variable of this type and ignore operations on the variables of type. The detailed description of the actions on point variable is placed in the third part of information. The last property is '#assign'. It determines whether the rewriter replaces the operator write on the variables of this type. If it is set to '#n', rewriter will ignore all operator write on the variables of type. Beside the properties, two kinds of replacing rules are allowed in this part. After keyword '#single', the name of replacing function, which is used for single read operation on the variable of this type in source code, appears. The other kind of rules for replacing operate write are listed in form:

```
assignOp:replacing function name
```

The second part of information is dedicated to the members of type, which must be individually monitored by rewriter. The '#member' keyword tells the rewriter where to find the member name. As in first part, the function name for read operation on member follows the keyword '#single' and the rules for replacing operator write are also organized in the form of list introduced above.

The last part of information is the information about the actions on address objects of type. Unlike the former two parts, it does not have naming mechanism, since its name is the name of type plus a star. But '#auto' keyword is used to tell rewriter whether it needs to create the replacing table for address objects of this type. If '#auto' is set, a new table structure is created and inserted into the global replacing table list. Furthermore, we use '#ptsingle' to inform rewriting component where to find the name of replacing function for primitive read operation on the point variable, whereas we still use the same form of list to declare the function name for primitive assignment operator. Moreover, '#assign' can also be used here to inform rewriter ignore all operator write on address object. The information block for type replacing might look like:

```
#typename:TYPE;
#basic:#-;
#mbonly:#-;
#ptonly:#-;
#assign:#-;
#single: function name for 'read' on variables of TYPE;
Op1: function name for Op1;
Op2: function name for Op2;
 ...
#member:MEMBER NAME;
#single: function name for 'read' on member;
Op1': function name for Op1';
Op2': function name for Op2';
 ...
#auto:#-;
#ptsingle: function name for read on address object;
#assign:#-;
Op1'': function name for Op1'';
Op2'': function name for Op2'';
 ...
```

Above, '-' means either 'y' or 'n'; '...' means that there could be more.

4.3 Implementation of Lexical and Grammatical Components

During the development of our prototype system, we found a C grammar parser from John Mitchell and Monty Zukowski, which is written within ANTLR. Their parser carries out almost complete C language's lexical and grammatical rules, and also covers GCC's specification, although it misses few extended rules in GCC. Therefore, we decided to borrow the core EBNF representation for lexical and grammatical rules in their work and applied our own modification to fit it in our approach.

4.3.0.1 Type Name Recorder

Since some stuffs in C are dynamically created, pure EBNF representation for lexical and grammatical rules can not drive the grammar parser work correctly. For instance, C allows user to define own type name for an existing type and even define a new type. Therefore, we must keep these "user defined" types during the grammatical parsing so that we can recognize them in correct manner, e.g. declaration statement. In order to keep them, we constructed a simple table, called *type name table*. Since in this stage we do not need to do much work, the table only records the name of user defined type. When parser encounters an unknown token in a declaration statement, it looks up the table and tries to match the token's name with a user defined type name in the table. If the unknown token cannot be matched, the parser will stop parsing and throw an exception.

4.3.0.2 Modification for Later Processing

In order to cooperate with the code rewriter better, we still need to modify some grammatical rules' definition in Mitchell and Zukowski's implementation. We separated the rules that are important for our code analyzing and put them into AST tree as individual tree nodes. These modifications can help the code rewriter to locate the important expressions and statements efficiently.

Our first modification is about the rule of function call. In Mitchell and Zukowski's work they treat function call expression as one kind of postfix expressions. Since function call is one of the target expressions to be monitored in our approach, we must pay more attention on it. Therefore, we separated the function call expression into an individual grammatical rule and defined a dedicated node type for function call, called "funcExpr".

Beside function call, we also do some additional work on increase and decrease expressions. Increase operator, "++", and decrease operator, "", are two special operators for assignment in C++. In practice, these two operators can either prefix or postfix a variable. Therefore, Mitchell and Zukowski treated the increase expression and decrease expression with post-fixed operators as postfix expression; the expressions with pre-fixed increase and decrease operators were put into the rule for unary expressions, in which operator prefixes the operand. For the pre-fixed expressions we kept Mitchell and Zukowski's rule, because we can isolate them from unary expression simply by constructing a new AST node for them. Unfortunately, we can not apply the same strategy on post-fixed increase and decrease expression, since the definition of postfix expression is more complicated than unary expression in their work. Thus, we also picked them out and defined a new rule, "autoPostExpr", for them.

Some other rules, including the rules for assignment expression, primary expression, and cast expression, are also modified in our approach. To modify grammatical rules is not as simple as we thought in the beginning. One rule could be involved by other rules, so the related rule must also be modified to fit previous modification.

4.4 Implementation of the Code rewriter

From this position we will concentrate on the most important component in our approach, the code rewriter. In order to simplify terminology, we use *rewriter* instead of code rewriter in following discussion. The rewriter is in charge of three major tasks. The first task is to recognize the AST tree from grammar parser. The second task is finding all possible appearance of sensitive objects, and applying the replacing strategy on them. Lastly, outputting the rewritten source code is its third mission. We will also divide our discussion into three parts according to these three tasks of the rewriter.

4.4.1 Retrieving Information from the AST Tree

In the previous stage lexical and grammatical components have analyzed the input stream and converted it into an AST tree. Now our rewriter will use this AST tree to finish our automatic annotation purpose. In order to use the tree, the rewriter must recognize the structure of tree and retrieve the information from every tree node. Fortunately, this progress in ANTLR is very similar to the grammatical analysis in grammatical component. We even reused most of the grammatical rules defined in grammatical component to retrieve the information from AST tree in our rewriter. One major difference between grammatical component and the rewriter is that grammatical component faces a bunch of undetermined information, while the rewriter faces a well constructed information tree. You may assume that input tokens are different kinds of fishes and grammatical component is a fisher. The fisher checks the fishes caught by him and classifies them into different container (the grammatical rules) labeled with corresponding fish name. The rewriter is just like a cooker. He knows how many containers there are and they are full of fishes. What he needs to do is only getting the fish from a container and applying the recipe on them according the fish names. Respectively retrieving the information is like getting fish from a container and looking the label on the container so that cooker (rewriter) can determine which recipe (replacing strategy) he must use.

Certainly, we can not totally reuse the grammatical rules from grammatical components. For instance, there are not individual rule for pre-fixed increase and decrease expressions in grammatical component, but we must add one in our rewriter in order to simplify the application of rewriting strategy. Furthermore the rules for some expressions, e.g. function call, assignment, and etc., are also modified to fit our replacing strategy.

4.4.2 Applying the Replacing Strategy

After the rewriter is able to accept the AST tree and to retrieve the code information prepared by grammatical component, it needs to process them according to the “recipe” defined by rule definer. As we introduced previously, rule definer gives user a way to tell our rewriter what kinds of types are sensitive and how to briefly deal with them. We define a replacing table list in our rewriter to record the information accepted by rule definer. The replacing table list is indexed by the name of sensitive types, every entry

in the table points to a replacing table that stores all related information on the corresponding sensitive type. When the rewriter meets the sensitive type during processing source code, it looks for corresponding replacing table by sensitive type and retrieves the useful information, e.g. name of replacing function, from replacing table.

The rule definer only indicates the wish from the rewriter's "customer". In order to carry out the wish, we must collect more information from the AST tree and organize them into certain data structure. Since the rewriter concentrates on sensitive object, it must find out and record all possible sensitive objects at first.

4.4.2.1 Type and Variable Recorder

In lexical and grammatical components we introduced a simple type recorder to record all user defined name of type. Unlike that recorder, the recorder system in the rewriter is more complex and complete, because it is one of the most important keys to ensure correct rewriting. Beside the information about types, the recorder system in the rewriter also records the properties of a declared variable since they are the real targets that the rewriter must take care of. So we call the recorder system in the rewriter "*type and variable recorder*". In following we will discuss our recorder from four different aspects.

Renaming System of C

At first we would like to take a look at renaming system in C language. As we introduced, C allows programmer to give a new name to an existing type. This feature is normally used to assign new semantic meaning to the existing type. For instance, "long" is the primitive type in C. Many memory objects can be represented by it. But it is a very low-level abstraction from the view of machine. From the view of programmer, the objects that are represented by "long" may be able to be classified into different classes, e.g. objects for "size", objects for "count", and even objects as "lock". So programmer can assign new name "size_t" to "long" for objects representing "size", "count_t" for objects representing "count", and "lock_t" for objects representing "lock" so that they can be much more clearly separated and recognized.

While this feature improves the programmer-friendliness of C, it also increases the effort to catch the sensitive objects. Since sensitive type can also be possibly renamed, the objects declared by new name is surely also the sensitive object. In order to catch all possible sensitive, we must record the relationship between the new type name and original type name. We constructed the mapping of new type name and original type name and store the mapping in type renaming table. The new type name is used as index in the table; the original type name is stored in the entry. For primitive type, we used "basic" as their original type name. To determine whether an object is sensitive, the rewriter stops searching the type renaming table till the original type name is "basic" or the name of a sensitive type.

User Defined Type

Beside the renaming system, C also allows programmer to construct own type, including "UNION" and "STRUCT". Currently our implementation concentrates only on the "STRUCT". "STRUCT" is made with several variables of existing types, called members or properties in "STRUCT", so a sensitive variable, which we call *sensitive member*, is possibly included in a "STRUCT's" definition. Certainly, the operations on

the sensitive member is also the target of our rewriter. In order to determine if a member in a "STRUCT" is sensitive, we recorded all members' definition in a "STRUCT" by using complex type table. When the rewriter encounters an operation on a "STRUCT" member, it gets the information of this "STRUCT" from its complex type table and checks the definition of this member.

Recording Information of Variables

We recorded two kinds of information for variable. First, we recorded the name of variable's type in variable table, since we need this information to determine if the variable is sensitive. Furthermore, we must pay attention to the concept of "*scope*" in C language.

C language allows programmer to define different scopes by using paired curly. The variables defined in one scope are only valid in that scope and its sub-scope, and destroyed (at least logically) on the end of the scope; variables declared with same name can co-exist in different unrelated scopes. Furthermore, a variable in sub-scope is also allowed to be declared with the name that is used in current scope. For instance, "a" is used to identify a memory object in scope "A", and "B" is the sub-scope. Normally "a" is visible and valid in "B". But, if another variable is defined with name "a" in scope "B", "a" in scope "A" isn't valid in scope "B" any longer. The variables defined in the inner scope always have higher priority than those defined outside it. Therefore, we must create and destroy information about variable correctly to avoid the ambiguity for the name of variable.

We used scope stack to simulate the scope concept in C language. When the rewriter encounters left curly, it allocates a new variable table in the recorder and pushes the former variable table on the top of stack. After the right curly, the rewriter destroys current variable table and pops the top of scope stack. This design ensures the variables declared in one scope are only valid when that scope exists. To determine a variable's type, the rewriter searches it in current variable table first. If it cannot be found, the rewriter searches the stack from top to button. This search strategy ensures that the rewriter always obtains the highest prior definition for this variable's name. Moreover, the stack concept also avoids the possible conflict between two scopes in the same scope level because they never co-exist in our recorder.

Recording the Information of Function

Beside variables, we also need to collect the information of functions for function rewrting. The function declaration includes three parts, i.e. return type of function, name of function, and parameters of functions. We dedicated a class to store information of function and constructed them into function table indexed by the name of functions. One more thing, which must be care about, is that the parameters appear in the declaration of a function also belong to the local scope of this function. Therefore we must construct new variable table before a function finishes its parameter declaration and put its parameters into the new variable table.

4.4.3 Rewriting the Source Code

Now, we have enough information for rewriting. The rewriting should be applied on two kinds of targets in source code, i.e. sensitive functions and sensitive variables. We will discuss them separately in the following.

4.4.3.1 Inserting the Declaration of Replacing Functions

In C language, a function can only be used after it is correctly declared. Since our parser uses several replacement functions, which did not exist in original source code, we must insert their declaration correctly into source code.

The declarations are divided into two kinds. In the first kind of declaration only primitive types are used. So they can be inserted in the beginning of all processed source codes. In contrast, the other kind of declaration contains customized types. So they must be inserted exactly after the corresponding customized types are declared. Fortunately, every replacement function in our approach depends upon at most one customized type. The rewriter can find the declarations of their customized types and insert them directly after these declarations.

4.4.3.2 Rewriting Sensitive Primitive Functions

We have explained that we used a separate grammatical rule in grammatical component to recognize and label function call expressions. Now the rewriter gets the tree node labeled as "funcExpr" and retrieves the function name from the AST node for this function call. According to the name of function call, the rewriter searches the replacing table that contained the rewriting rules from user. If one function name that needs to be replaced matches the name of current function call, the rewriter retrieves the function name of rewriting's destination from that entry and replaces current function name with this new function name.

Beside the function name, we also need to take a look at the return type and parameters of function. Normally these two things are the same between source and destination. But sometimes they are different. We have recorded all function declarations in function table, now the rewriter can retrieve the information of the function in this function call expression through the function name and compare them with the information gotten from rule definer. If return type and parameter's type is not the same between source and destination, we may apply suitable conversation on them.

4.4.3.3 Rewriting Sensitive Operations

Our major work of implementation is annotating and rewriting the operations on sensitive variables. There are total two kinds of operations in C, i.e. read and write. In C "write" operation can be recognized through special operators, called assignment operators. However, no operator is dedicated to "read" operations. Every appearance of a variable is a read operation except some special positions, e.g. in the left of an assignment operator. We will first introduce the technique for rewriting operator "write" and then discuss about the "read" operations.

Annotating and Rewriting The "write" Operations

Since the operator "write" can be recognized through the assignment operators, our lexical and grammatical components have already found and labeled most of them in AST tree. The rewriter can easily use this labeled information to rewrite the operations. The rewriter checks first if the operand on the left of assignment operator is a sensitive variable. If it is, the rewriter retrieves the rewriting information from the replacing table according to the type of left operand and the assignment operator. Then rewriter eliminates the assignment statement and fills the blank space with corresponding replacing

function call. In the following, we will first focus on how the rewriter checks the sensitivity of a variable, and then discuss about some special forms of "write" operation in C language.

Identifying Sensitive Variables

The rewriter only rewrites the assignment statement for sensitive variables, so whether they can be correctly identified is the first important step to successful rewriting. The left operand of an assignment statement is normally divided into four kinds including simple variable, postfix expression, casting expression, and unary expression.

To identify a simple variable's sensitivity is the simplest among four kinds. The rewriter uses the variable's name to get the type of this variable in the variable table. Then the rewriter checks whether the type is a sensitive type in the sensitive type table. If it is, rewriting will happen. Otherwise, the rewriter keeps the assignment statement and continues his analysis on next AST node.

To identify postfix expression's sensitivity is a little complex. Our grammatical component defines postfix expression as the access form on a member of an "STRUCT" or "UNION", i.e. "`a->b`" or "`a.b`". Therefore, the postfix expression is sensitive when the member is sensitive. The first question is how to identify the member's sensitivity. Furthermore, postfix expression can be multiple levels, e.g. "`a->b->c->d`". This brings the second question, which member is necessary to be identified. We would like to answer the second question first. Our rewriter currently only checks the last member's sensitivity in the postfix expression, since the operator write only happens on the last member. Now we answer the first question. In our approach the sensitivity of a member can be either indicated by user through rule definer or determined through the type, with which it is declared. Nevertheless, both of these two possibilities require the rewriter to know the definition of STRUCT or UNION that includes the last member of postfix expression. The rewriter finds out the information of STRUCT by identifying the left operand of last postfix operator. For one-level postfix expression, rewriter can simply get the STRUCT information by using the name of left operand, since it must be a declared variable in variable table. For multi-level postfix expression, the rewriter must identify the type from the first operand and then recursively identify every member's type till the last left operand, and then use its type name to retrieve the STRUCT information for last member. After the rewriter has the STRUCT information, it first uses the STRUCT name and the last member's name together to check the member's sensitivity through the replacing table. If there is no match, it continues to get the type name of last member from the STRUCT information and checks the replacing table.

For casting expression, the identification must be applied on both result of conversion and the original variable. The identification for original variable is similar to the identification for the simple variable name. If the original variable is sensitive, the whole casting expression must be sensitive. Otherwise, the rewriter continues to identify the result of conversion. The key part of casting expression is the casting prefix. It determines the result of conversion. The prefix is consisted of types that programmer can use in the declaration of variable. Therefore, the rewriter reads all type names contained in prefix and checks them in the replacing table. If one sensitive type is matched, the rewriter considers that the result of conversion is sensitive.

At last the rewriter must identify the unary expression's sensitivity. Only two operators in unary expression are important in our approach, i.e. "*" and "`&`", since they change the target of assignment statement. "*" redirects the "write" operation on the

memory object that is pointed by pointer variable, while "`&`" redirects the "write" operation from the original variable to the address object pointing to it. For identifying unary operation the rewriter focus only on the result of unary expression. For "*" the rewriter decreases one "*" symbol in original variable's type, where as it increases one "*" symbol. The rewriter tries to find a matched entry in sensitive type table for the changed type specifier. If it found, the unary expression is identified to be sensitive.

Different possibilities of assignment

Beside the normal form of assignment expression, i.e. form "`a=b;`" some other possibilities of assignment are allowed in C. We need to deal with them carefully in our approach.

The first possibility is the assignment operation during the declaration of a variable, called initialization of a variable. Since we want to use a function call to replace the "write" operation in source code and C only allows the initialization through assignment operator, we cannot directly rewrite the initialization with our function call. Our solution for this situation is to break up the declaration phase and initialization phase. For instance, "`int a = 0;`" will be broken into two statements, i.e. "`int a; a = 0;`", so that the rewriter can treat initialization phase as normal assignment operation. However, some initialization phase cannot be simply reorganized in this way, especially the initialization for a variable of "STRUCT". We must organize them into more complex form by using the cast expression or even anonymous functions. Fortunately, the initialization phase of a variable is normally not very important in annotating the sensitive memory object, because they are mostly *fake sensitive*. We will discuss this in the section about "optimizing rewriting".

The second form of assignment is the continued assignment expression. In C, programmer can connect several variables of the same type with assignment operator and give out the assignment value in the end of expression so that these variables can be assigned together with the same value, e.g. "`a=b=c=d=value;`". The replacing function normally takes only two parameters, so we must find a way to deal with continued assignment expression. We took the similar strategy for the continued assignment expression in the beginning. We broke the continued assignment expression into individual assignment expressions. For example, the rewriter will use "`a=value;b=a;c=a;d=a;`" to replace the continued assignment expression above. Then the individual assignment expressions can be treated as usual. However, this brought problem. If "`a`" above is replaced with "`a[i++]`", then "`b`", "`c`", and "`d`" will be assigned with false value. We cannot do like "`a=value; b=value; c=value; d=value;`", either, since there can be a function call, which returns different values for every invocation, in the position of "`value`" above. So we propose to create a temporary variable to store "`value`", and to assign all variables in the continued assignment expression from the temporary variable. For example, we can rewrite the codes above with

```
__typeof__(a) temp;
temp = value;
a = value;
b = value;
c = value;
d = value;
```

and the rewriter will check every individual assignment expression to check their sensitivities.

The third possibility of assignment is the assignment with increase and decrease operators. Unlike the normal assignment operators, increase operator and decrease operator include both "read" and "write" operation on a variable. Furthermore, according to the position where increase and decrease operators appear, i.e. after the operand or before the operand, they have different behaviors, especially for "read" operation, and must be separately treated. Here, we only discuss how we treated the "write" property of increase and decrease operators. We left the "read" property to the later discussion about rewriting "read" operations. In fact, the rewriter deals with "write" property of increase and decrease operators quite simply. It identifies the variable's sensitivity operated by increase and decrease operators. If the variable is sensitive, it searches the replacing function for increase operator or decrease operator of this sensitive type from the replacing table and passes the variable as the unique parameter.

4.4.3.4 Annotating and Rewriting Read Operation

Similar to the progress of rewriting "operator write", the progress of rewriting "read" operation is divided into two phases, identifying and replacing. For identifying the sensitivity of a variable, we used the same technique as in rewriting "write" operation. After the rewriter finds a sensitive object, it needs to check the replacing table according to the name of current sensitive object's type and get function name for rewriting sensitive "read" operation. In our rewriter, function "`replace_single`" is responsible to this work. Beside identification functionality, "`replace_single`" also accomplishes the replacing action for a sensitive variable.

4.4.4 Shadow Object

A shadow object is the copy of a sensitive memory object. We introduce this concept in our implementation for two purposes, working around *rewriter-invisible* sensitive operations and reducing the unnecessary synchronization function call from guest OS to VMM. In order to achieve these purposes, our replacing functions for sensitive read operations return only address of corresponding shadow objects.

We do not need to create shadow object for every sensitive memory object. Every type of sensitive memory object needs only one shadow object. For instance, we created one shadow object for all memory objects of "pte_t". Every intercepted sensitive "write" operation refreshes the value of corresponding shadow object with the actual value of sensitive object. On the other side, "read" operation must not refreshes the value of shadow object. It refreshes the value only if the "read" operation changes its target. Otherwise, it keeps its value and returns immediately. Moreover, "read" operation is in charge of applying the changes in shadow object from invisible sensitive operations. It compares the original value of real sensitive object with the value of shadow object each time when it is called. If they are different, it applies immediately the changes back to the real object.

We would like to introduce how we achieve the first purpose in this section and put the discussion of second purpose in section about optimization in next chapter.

4.4.4.1 Rewriter-invisible Sensitive Operations

C allows automatic typecasting for parameter in function call statement. For example:

```
void dummy( unsigned long *pt )
```

```
{
    unsigned long tmp;
    tmp = *pt;
}
typedef struct {
    int pte_dummy;
} pte_t;
int func()
{
    pte_t pte;
    dummy(&pte);
}
```

Assume that "pte_t" is sensitive type. Although we pass a "pte_t" pointer into "dummy", the pointer is automatically converted to "unsigned long" pointer and appears as "pt" in the scope of "dummy". Since "unsigned long" is not sensitive type, any further operations on the object pointed by "pt" is insensitive from the view of the rewriter, whereas these operations happen actually on a sensitive object. These operations are called rewriter-invisible sensitive operations.

Of course, in order to catch these missed sensitive memory operations we can replace "dummy" with our customized function. Nevertheless, "dummy" is usually a general purpose function, which can be also used on insensitive objects. Constructing a suitable replacing function for general purpose functions is not a simple work, because it must determine whether the passed parameter is sensitive. In following, we would like to call functions like "dummy" *sensitive-invisible* function.

In our approach, our rewriter compares the type of every passed address object with the type of parameter declared in corresponding function call. If the types are not matched and the passed pointer variable is related to a sensitive object, we rewrite the pointer variable with replacing function for "read" operation on corresponding sensitive object. By using this replacing function we do not pass the real address of sensitive object, but the address of corresponding *shadow object* into the sensitive-invisible functions. Then the sensitive-invisible function does not operate directly on the real sensitive memory object. Any potential change is made on shadow object. After the function calls return, we check the value of shadow object immediately. If the shadow object has been changed, the real sensitive memory object is updated with the value of shadow object so that the corresponding side-effect could take place. By doing so, we can fully control the modification on sensitive object from sensitive-invisible functions.

4.4.4.2 The Limitation of Shadow Object

Although shadow object seems to solve the problem raised by rewriter-invisible sensitive operations, it still has its limitation. At first, the side-effect of a sensitive operation is obviously delayed. The sensitive object changes only after the function finishes.

Furthermore, some primitive functions that raise invisible sensitive operations require to be accomplished atomically. For instance, the set-and-clear operation on a memory object is usually atomic so that the value of memory object is always consistent. Now we passed shadow object into set-and-clear function, the atomic ability of function is only ensured for shadow object.

In order to eliminate inconsistency totally we move our sight to the functions that contain sensitive-invisible functions, e.g. "func" in the example above. We can replace them instead of replacing sensitive-invisible functions directly. The annotator can scan

the source code and record them in a file in first round. User can check the file and manually compose special functions for these functions after the first round. Then the annotator can use the new replacing functions to rewrite them in the second round. If two-round annotation is too expensive, it is also possible to generate all replacing function dynamically. However, this work is beyond the scope of our thesis due to the time limitation.

4.5 Optimization for Rewriting Sensitive Object

In former sections we introduced the general implementation of our parser for automating the annotation of sensitive object. Although the general implementation can ensure all sensitive objects are caught, it is not very clever and brings a lot of unnecessary rewritings. Because we used function call to replacing assignment statement and the overhead of a function call is much heavier than a simple assignment statement, the unnecessary rewritings can significantly pull down the OS's performance. Therefore we must optimize our rewriter so that it is clever enough to avoid the unnecessary rewriting.

4.5.1 Fake Sensitive Object

The first unnecessary rewriting is the rewriting for fake sensitive object. In general purpose implementation, our rewriter recognizes a sensitive object by checking whether its type is the sensitive type defined in rule definer. However, this detection cannot help the rewriter to distinguish the fake sensitive object from real sensitive object.

Some memory objects are declared with sensitive type but are not really sensitive. We call these memory objects *fake sensitive objects*. During implementation we found that the memory objects that are declared with sensitive type in a local scope, e.g. during a function call's life cycle, are fake sensitive objects. In C language these objects are created on the stack and mostly destroyed after the end of function call. They are mostly used to temporarily store a copy of real sensitive object's value for some calculation purpose, so the changes made in them do not really affect the real sensitive object and raise the hardware side-effect until their values are written back to the real sensitive object. Therefore it is safe to assume all local declared sensitive objects are fake sensitive and eliminate the rewriting on them. It is not difficult for our rewriter to consider whether a variable is local. We have introduced above that we constructed a stack for storing variable in different scope-level so that the rewriter can get the type information about a memory object. Beside this stack there is a space for saving global variables declared in source codes. Since checking all local variables in the stack of local scopes is slow, we let our rewriter check global space to determine whether current memory object is local. If it is local declared, we eliminate all rewritings of "read" and "write" on them.

Chapter 5

Experiment

In previous chapter we discussed in detail the implementation of our prototype system, which is used to automate annotation of sensitive objects. In this chapter, we will introduce how we applied our prototype system in current pre-virtualization system and show the result of our prototype system.

5.1 Automatic C Afterburning

The current pre-virtualization of sensitive memory object has three steps. The first step is to annotate the operation on sensitive memory objects and to change them manually into corresponding macros and function-calls for pre-virtualization. This work costs time and does not cover all direct 'read' operations on sensitive objects currently. After this step, a file named "annotate.h" is created, which contains implementation of macros and function-calls that are used to replace old sensitive operations in first step. The last step in current pre-virtualization is shipping the changes in source codes with pre-virtualization project. Pre-virtualization generates patch-files for changed source code. They must be applied each time before the OS binary is being built.

After we introduced our tool for automating pre-virtualization of sensitive object, we must change the pre-virtualization's steps. Obviously the first step, in which manually annotating sensitive operations happen, is not necessary any more. Our annotator will take over the job of annotating and be integrated into the compilation of source codes. The integration is really simple, since we know the way in which assembly afterburner is integrated.

Linux uses the script "`Makefile.build`" to control major progress of binary's generation. Pre-virtualization inserts command lines for assembly afterburner at the location before the command line that converts source code to object file. Therefore, we found the place where afterburner appears, and inserted command lines for annotator before the command line for assembly afterburner. There are in total of two command lines, which needs to be inserted. The first command line uses the "`-E`" option provided by GCC to convert original source code into pre-processed source code. The second command line passes pre-processed source code to our annotator and prepares the result of annotating for assembly afterburner.

For the second step in previous pre-virtualization of memory sensitive object, we converted "annotator.h" into "annotator.c". Since our annotator uses preprocessed source code and retains the output code in the same format, we need to rewrite all

macros defined in “annotate.h” with corresponding function-calls. This conversion brings additional overhead, because the execution of function call is more expensive than the execution of macro.

In C programming language, compiler needs to store some environment states before a function call is invoked, and to restore them after the function call. On the other hand, it simply replaces all occurrences of macro with the codes that defines macro; no environment save-and-restore phase is required. Furthermore, compiler uses “call” instruction, which is very expensive from the respect of performance, to jump to the first instruction in a function call. Fortunately, our optimization in annotator has eliminated most of unnecessary rewriting, and keeps the number of function calls as small as possible. Besides, we also introduced new mechanism to decrease the overhead during the progress of processing caught sensitive operations.

The last step for generating patch file is kept, because pre-virtualization links Linux kernel to a lower address and changes its clock rate to fit in virtualization environment. Furthermore, Instead of the content of “annotate.h”, it contains the content of “annotate.c” so that annotate.c can be dynamically generated each time the Linux kernel is going to be built.

5.2 Optimization

In order to improve the performance of processing intercepted sensitive operations, we optimize its progress. As we described above, the overhead comes mainly from function call. After our annotator rewrote the sensitive operations, every sensitive operation generates two function calls. The first is the calling of function that is used to replace sensitive operation in source codes. GCC provides a chance to decrease the overhead caused by this function call, i.e. “inline function”. Unfortunately, our current designation cannot take advantage of this feature in GCC, since we implemented all functions for replacing sensitive operations in a separated file, “annotator.c”, and used the functions by linking the object file created from that file with the OS binary. However, it is possible for us to add functionality in annotator so that the implementations of replacing functions are able to be directly inserted into annotated source codes and the “inline” feature of GCC can be used correctly.

The second function call exists inside the replacing functions. In pre-virtualization, replacing functions are only the preparer, which creates scratch spaces for dynamical rewriting during the loading of afterburned binary. The actual work of processing sensitive operation is given to the special functions, called *frontend sync-function*, in IPVMM. They are dynamically filled into scratch spaces. We mentioned that the sensitive operations focus on the same sensitive memory object for a while in some situations. Not every operation is necessary to raise the synchronization in IPVMM. For example, two continued “read” operation on the same sensitive object only requires one synchronization in IPVMM. Therefore, we introduced some mechanisms in order to reduce unnecessary calling of sync-functions.

At first, we recorded the address of last operated sensitive memory object. This together with shadow object in section 4.4.4 can help us avoid the unnecessary invocation of “read” sync-function. When the replacing function for sensitive “read” is called, the address of currently operated sensitive object are compared with the address of previous operated sensitive memory object. If the address is the same, calling of “read” sync-function is skipped and replacing function still returns the result of last calling of sync-function, which is stored in shadow object. Usually the overhead of

comparison here is very small. For "write" operation the "write" sync-functions are always invoked, since "write" always changed the value of a sensitive memory object.

However, negative effect can happen between two "read" operations on the same sensitive object. If a TLB flush happens between two intercepted "read" operations, the inconsistency between shadow object and real sensitive object can appear. The solution for this problem is to synchronize shadow object and real sensitive object each time when a TLB flush happens. TLB flush is a virtualization sensitive instruction, so that it will be caught and replaced with customized TLB flush instruction by pre-virtualization. Since the IPVMM monitors the execution of customized instructions in pre-virtualization and sensitive activities in user space, it is not very difficult to make shadow objects visible to IPVMM and synchronize them with corresponding sensitive objects during execution of customized TLB flush instruction.

5.3 Evaluation

We have made two experiments for evaluating our approach. The first experiment co-operated with L4Ka environment and just tried to prove the feasibility of our approach. The second evaluation cooperated with XEN hypervisor with pre-virtualization environment.

5.3.1 Experiment on L4Ka

The environment of first evaluation is relative simple. It consists of the following parts:

L4ka Pre-virtualization provides possibility to adapt different hypervisors. We used the configuration for L4ka in our evaluation, since we are more familiar with it than Xen. Nevertheless, the rewriting progress is similar for these two configuration.

Linux 2.6.9 We selected Linux 2.6.9 as target source code of our C parser. Linux 2.6.9 has replaced a lot of Macro codes in previous version with primitive inline functions, especially the I/O functions. This is important for our parser, since we could use function replacing strategy in Linux 2.6.9.

QEMU The QEMU is a simulation software, which simulates IA-32 architecture. It is useful to briefly examine the accuracy of compiled binary.

GCC 4.1.2 We used gcc 4.1.2 as our compiler to compile the rewritten codes.

General P4 machine The building progress is finished on a standard P4 machine, on which SUSE 10.1 is installed.

We evaluate our first experiment from three aspects.

5.3.1.1 Overhead

The first aspect is the overhead brought by our C code analyzer. Here, we only talk about the overhead occurring during the building progress. The overhead occurs during the execution of binary is not involved. As the result, our parser brings a heavy overhead in the building progress. The building progress takes three times longer as

before. The problem is that the implementation of our search-and-replace is not optimal. A lot of redundant search progresses exist in prototype. Our parser's performance can be significantly improved when these redundant progresses are eliminated. However, due to time factor, the work for optimization is beyond the scope of this thesis.

5.3.1.2 Replacement

We have collected information about replacement happened during building stage. Total of 466 files were annotated by our C code analyzer. Among them, replacement happened in 329 files. We list the replacement according to the sensitive types and expressions in Table 5.3.1.2.

Table 5.1: Result of evaluation

TypeName	Sum	Normal	Unary	Postfix	Cast
pgd_t	12	0	1	11	0
pgd_t*	360	359	0	1	0
pmd_t	104	0	28	76	0
pmd_t*	1	1	0	0	0
pte_t	61	0	33	28	0
pte_t*	2309	669	0	1640	0

We recorded only the result of three sensitive types, i.e. pte_t, pmd_t, and pgd_t. They represent page table entry, middle page table directory entry, and global page table directory. In table, "Normal" shows the number of sensitive operations that happened directly on the variables of corresponding sensitive type. "Unary" means that the operand, which brought sensitive operation, is the result of an unary expression. "Postfix" represents how many operations happened on sensitive members. Lastly, "Cast" shows the number of sensitive operations that were caused by result of casting expression.

From the Table 5.3.1.2, no replacement happened in column "Cast". That means no sensitive object is casted from an insensitive type's object. This is possible, since casting insensitive type's object to sensitive object is not usual. However, to clarify the situation, it is better to manually inspect this situation and consider whether our parser works correctly. Due to the time limitation, we would like to accomplish this mission in future. As most sensitive memory objects are operated through their address object in Linux, only entries represent types of address object in column "Normal" have values. Furthermore, all sensitive operations on result of unary operator "*" on sensitive address object were counted into the entries of corresponding sensitive types in column "Unary", so it is not strange that all entries of sensitive address types in column "Unary" are empty.

Through the value's distribution in table, the parser seems works well. However, without executing the generated Linux binary, we can not ensure the positive conclusion.

5.3.1.3 Binary Execution

At last, we tried to start pre-virtualization and Linux with new generated Linux kernel binary. The number of scratch space replacings during the loading stage of pre-virtualization decreased, because we used the function call instead of Macros defined

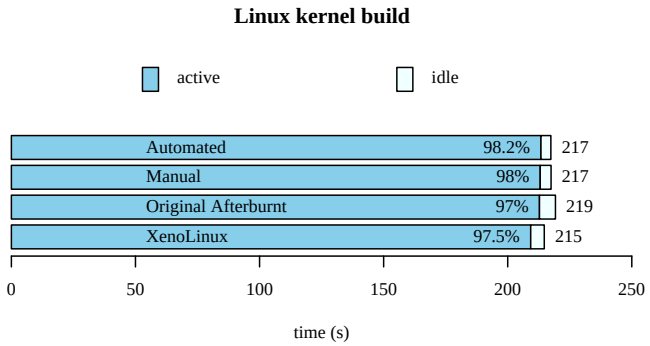

Figure 5.1: The performance of the Linux kernel build benchmark in four different virtualization environment, running on Xen 2.0.2 hypervisor. Shorter times are better. They all achieved nearly full CPU utilization, which is expressed as a percentage of total duration.

in original pre-virtualization. The scratch space in a function is only rewritten once, whereas scratch space in Macro definitions will be put in every place where Macros appear and replaced multiple times by IPVMM. Finally, the Linux was successfully brought up.

5.3.2 Experiment on Xen

In order to better evaluate our approach, we made a second experiment. This time, we put our approach in a stricter environment: pre-virtualization for Xen. We used Xen 2.0.2 as our underlying hypervisor and ran our automatically modified kernel with the IPVMM for Xen 2.0.2. The test bed was a 2.8 GHz Pentium 4 machine with an installed Debian 3.1 operating system. Besides booting the modified Linux kernel binary, we also made a benchmark for our approach. In the benchmark, we evaluated the performance of Linux kernel build on a virtual machine (256M RAM) booted with the following four different kernels:

XenoLinux kernel: This kernel is modified for Xen para-virtualization environment and comes from the Xen project.

Afterburnt Linux kernel: This is the original kernel for pre-virtualization environment on Xen.

Manual Linux kernel: We manually annotate sensitive operations in the kernel source code and replace them with our replacing function calls.

Automated Linux kernel: The kernel source code is rewritten automatically with our rewriter and compiled.

For each kernel, the Linux kernel build was executed 5 times. The result of the benchmark is shown in Figure 5.1. The data represent the mean values of five independent benchmark runs, with an approximate 95% confidence interval not exceeding 1.17% of the mean values (calculated using Student's t distribution).

The result shows that XenoLinux has the most efficient performance. The other three kernels are less efficient, since IPVMM brings additional overheads for the sys-

tem. Furthermore, we use function calls instead of macros for manual kernel and automated kernel, which also pulls down the performance slightly in both systems.

Chapter 6

Conclusion

Because assembly is a simple type programming language, sensitive memory operation can't be annotated through recognizing sensitive memory object in assembly level. Therefore, pre-virtualization annotate all sensitive memory operations manually in current solution, which is inefficient and inflexible.

Therefore, we move our sight to the C language. C is a complex type programming language, which has a rich type system and can accurately distinguish memory object according to their types. It is also used widely in development of operating system. Furthermore, because of its simple syntax, constructing a C code analyzer is also feasible. Therefore, we enhance the afterburning technique with a customized C code analyzer and successfully automate the virtualizing progress of sensitive memory operations.

Our code analyzer is made up with a grammar parser and a code rewriter. Furthermore a "rule definer" was designed to help our parser recognize targets. The grammar parser was used to convert input character stream into a well constructed information tree, called AST tree. The tree is accepted by the code rewriter. It retrieves the useful semantic information from AST tree and stores them in its own information recorder for further processing. Through cooperation with "rule definer" rewriter can find out the sensitive operations by locating the position of sensitive objects. We also optimized the way to identify the sensitive object and filtered out the fake sensitive objects.

At last we proved through experiment that using C language parser to automate annotation of sensitive memory operation and memory object is feasible. We also showed the result of our experiment in Chapter 5.

Future Work

Currently, performance of our parser is not satisfactory. Our future work will focus on optimizing the performance of our parsers. We will also add the inline function support in our parser so that the function call overhead can be eliminated. Furthermore, we are interested in the optimization in IPVMM for sensitive read and write operations, after they are caught and replaced by our parser.

Beside C parser, we are also interested in working around with C++ language, since C++ is also a language that could be used to write operating system. C++ provides operator overriding to redefine the meaning of operator. This characteristic might be able to simplify the replacing progress for sensitive operations.

Bibliography

[1] *IA-32 Intel Architecture Software Developer's Manual*.

[2] M. Shaw A. Whitaker and S. D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. Technical report, February 2002.

[3] Keith Adams and Ole Agesen. A comparison of Software and Hardware Techniques for x86 Virtualization. 2006.

[4] Marianne Shaw Andrew Whitaker and Steven D. Gribble. Scale and performance in the denali isolation kernel. In *Proceedings of the 5th Symposium on Operating System Design and Implementation*, Boston, MA, December 2002.

[5] Steven Hand Christopher Clark, Keir Fraser and etc. Live migration of virtual machines. In *Proceedings of USENIX NSDI 2005*, Boston USA, 2005.

[6] Andrew Cooke. An Introduction to Programming Languages. 2003.

[7] Andreas Galz Daniel Mahrenholz, Olaf Spinczyk and Wolfgang Schröder-Preikschat. An aspect-oriented implementation of interrupt synchronization an aspect-oriented implementation of interrupt synchronization. In *Proceedings of the 5th ECOOP Workshop on Object Orientation and Operating System*, June 2002.

[8] Edsger Wybe Dijkstra. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[9] Scott Devine Edouard Bugnion and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th Symposium on Operating System Principles*, Saint-Malo, France, October 1997.

[10] Lars Marius Garshol. BNF and EBNF: What are they and how do they work? 2003.

[11] R. P. Goldberg. Survey of virtual machine research. *Computer*, 1974.

[12] Anurag Mendhekar Chris Maeda Cristina Videira Lopes Jean-Marc Loingtier Gregor Kiczales, John Lamping and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, June 1997.

[13] System Architecture Group. The l4ka::pistachio microkernel white paper. Technical report, Fakultät für Informatik, Universität Karlsruhe (TH), May 2003.

[14] J. G. Hansen and E. Jul. Self-migration of operating systems. In *Proceedings of the 11th ACM SIGOPS European Workshop*, September 2004.

[15] Jacob G. Hansen and Asger K. Henriksen. Nomadic operating systems. Master's thesis, Dept. of Computer Science, University of Copenhagen, Denmark, 2002.

[16] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., 1988.

[17] Joshua LeVasseur. Pre-virtualization internals, March 2006.

[18] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical Report 2005-30, Fakultät für Informatik, Universität Karlsruhe (TH), November 2005.

[19] Joshua LeVasseur, Volkmar Uhlig, Ben Leslie, Matthew Chapman, and Gernot Heiser. Pre-virtualization: Uniting two worlds, October 23–26 2005.

[20] T. J. PARR and etc. *ANTLR Reference Manual*.

[21] T. J. PARR and R. W. QUONG. ANTLR: A Predicated-LL(k) Parser Generator. *SOFTWARE-PRACTICE AND EXPERIENCE*, 25.

[22] Richard E. Pattis. EBNF: A Notation to Describe Syntax. 1994.

[23] Keir Fraser Paul Barham, Boris Dragovic and etc. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating System Principles*, October 19–October 22 2003.

[24] Gerald J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17, 1974.

[25] John Scott Robin and Cynthia E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of 9th USENIX Security Symposium*, Denver, Colorado, USA, August 2000.

[26] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating System Design and Implementation*, Boston, MA, December 2002.

[27] James E. Smith and Ravi Nair. An overview of virtual machine architectures. 2004.

[28] James E. Smith and Ravi Nair. The architecture of vritual machines. *Computer*, 2005.

[29] Andrew S.Tanenbaum and Albert S. Woodhull. *Operating System: Design and Implementation*, pages 3–5. PRENTICE HALL, 1997.

[30] Terrence W.Pratt and Marvin V.Zenlkowitz. *Programming Language Design and Implementation*, pages 80–89. PRENTICE HALL, 1997.

[31] Mike Feeley Yvonne Coady, Gregor Kiczales and Greg Smolyn. Using AspectC to Improve the Modularity of PathSpecific Customization in Operating System Code. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT symposium on Foundations of software engineering*, 2001.