Research for Fault Isolation of Memory Protection in Virtual Machines

Jiye Wang*, Chunlei Zhou, Lingchao Gao, Junfeng Bai
State Grid Corporation of China Big Data Centre, Beijing, 100031, China

a Email: Sgcdbd@163.com
*Corresponding Author

Keywords: Memory Protection, Fault Isolation, Virtual Machine

Abstract: This Paper is on How to Implement Memory Protection Technology and Do More Research on It. One Way to Realize Fault Isolation among Cooperating Software Modules is to Place Each in Its Own Address Space. However, for Tightly-Coupled Modules, This Method Would Be Lead to Context Switch Overhead. to Solve This Problem in Virtual Machine, a Kind of Technology is Introduced, Which Presents a Software Approach to Implementing Fault Isolation within a Single Address Space. It is to Be Checked That This Method is Efficient in Software-Based Fault Isolation, and This Technology Have Been Successfully Applied in an Actual Virtual Machines.

1. Introduction

In Order to Ensure the Mapping of an Address Space and Its Implementation, Several Features in the Memory Architecture Must Be Simulated in the Virtual Machine. in Particular, the Abi (Application Binary Interface) Memory Architecture Which Also Means User-Level Applications Memory Architecture Should Take the Following Three Characteristics into Consideration:

(1) The Global Structure of the Address Space. for Example, Whether the Memory Architecture is Divided into a Segment Structure or an Entire Linear Address Space. Our Topics for Discussion is Largely Limited to Linear Address Spaces, as Most Popular Abis Adopt Linear Address Spaces (Although They May Be Further Divided into Heaps and Stacks). and Virtual Segment Memory Technology Can Be Built into These Linear Address Spaces.

(2) Types of access rights supported. Some ABIs support read, write, and execute (R, W, E) permissions, and some are limited to read and write only.

(3) Protection and distribution particle level. This means the minimum size of the memory block that the operating system can allocate and the memory size of the smallest protected particle level that the operating system can hold. The particle level of memory allocation and protection is the same in most systems, however, strictly speaking, they do not have to be the same.

An application can usually have extensible features that are achieved through the collaboration of independently developed software modules. However, if there is a fault code in these extensible code, the software system will be no longer reliable or even in danger. For example, this fault code can destroy permanent data. To increase the reliability of scalable software, the operating system can provide services that prevent fault codes in untrusted modules from corrupting application data. This fault isolation service makes software development more convenient by confirming errors in the original system [1].

This paper describes how to reduce the cost of implementing fault isolation technology so that system module developers can ignore the disadvantages of their performance when choosing which modules to place in separate fault domains. Fault isolation techniques are useful in many situations, such as cross-domain function which is called very frequently and each call contains reasonable overhead [2]. It is impractical to reassign logically independent modules to their respective address spaces due to overhead across hardware boundaries.

This paper only covers simple switching techniques, called sandboxing, which is at the expense of slightly increasing the time it takes to execute the modified object code.
2. The Research of Memory Protection in Virtual Machine

We only discuss the memory protection at runtime in this article. At runtime, some softwares (including code and data) share the same address or space with the client application softwares, so that memory must be protected by the client application. For example, if an emulator tries to access an runtime address area from an error, it should be reported as a memory error in the emulated application. In order to maintain compatibility and make correct decisions, the client program should not be allowed to read and write the memory area at runtime.

An efficient solution is employed in the Omniware VM virtual machine (Lucco, Sharp, and Wahbe 1995; Wahbe et al. 1993), in which host hardware is used to support address translation, but protection verification is still implemented by software. For the protection verification of the streamlined software, the client’s data and code are divided into segments of power of 2, by relying on such segmentation, only a single shift (extracting the address bits of the segment) and comparison is needed to finish verification work efficiently. At any time if only one segment of data is active, the address bits of the currently accessible segment can be stored in the host’s registers, making the comparison faster [3]. Further, by analyzing the program control flow, such optimization as used in HLL virtual machines that can reduce null pointer and array range checking can also be applied. This can reduce the extra time overhead, which is about 10% of the total (and sometimes less). Although this technique protects the runtime out-of-bounds access from the clients, the forced power-of-two size segment may diminish substantial compatibility because these restrictions placed in the memory address space are not available on the local platform.

A technique for both address translation and protection verification employing the underlying hardware can be seen in the Dynamo system (Bala, Duestewald, and Banerjia 2000). In this way, the runtime system is responsible for calling the appropriate host operating system routines to set memory protection, when there will be two execution modes. In emulation mode, the converted client code is executing; at other times, including the time the binary converter generates the code, the virtual machine is executed in runtime mode [4]. Only when the virtual machine is in emulation mode, the runtime code and data must be protected by the converted code. Thus the memory protection in both two modes will be different, as shown in Figure 1.

![Fig.1 Use Cached Conversion Code for Memory Protection Settings](image)

Figure 1 depicts memory protection in runtime mode and emulation mode. In runtime mode, the runtime code is executable; the runtime data structure is read-write accessible, and the code cache Cache is readable and writable because it must be accessed and modified during the conversion process. In emulation mode, all runtime data structures become inaccessible, and at this point the code cache Cache is only executable; only data in the client’s memory image is accessible.

To change this type of execution-based protection, we can take advantage of a system call at runtime. For example, Linux can be called by mprotect() to change the runtime data and code to be inaccessible and the code cache Cache to execute only. Of course, the switching of the mode can also be performed similarly. When control returns to run, it can re-establish read and write permissions. It also raises the problem of the emulation software attempting to jump or branch to
the runtime area, which can be checked out during the interpretation phase or the binary translation phase. During the interpretation phase, the purpose of the branch and jump can be accurately checked out by the interpreter [5]. For code that has already been converted, all branches and jumps in the code block will be directly addressed inside the code block. All indirect jumps and branches in the code block operate correspondingly by the mapping table or connection pointer in the block. All of these are written at runtime and can be verified when written [6].

It is an effective method, but it will take a relatively long time when there is a mode switch. However, if the efficiency of the code cache is high enough (usually it is), the time overhead can be ignored when the client's instruction set is translated and converted.

3. Research on Fault Isolation Technology Implemented by Software

In this section, we list several software packaging techniques to limit untrusted modules to their own fault areas. We first introduce a technique that allows users to locate errors in software modules, and then we introduce a technique called sandboxing that isolates untrusted modules at the expense of a small increase in execution time. Finally, we present a software encapsulation technique that allows memory areas associated with each other to share memory. In this part of the discussion we use the RISC load/store architecture, and of course this technique also can be applied to CISC. Part 4 describes how to improve the performance and security of RPC across fault areas.

We divide the virtual address space of an application into segments, and all virtual addresses in a segment are aligned, that is, share a unique high-order address pattern called a segment identifier. A fault domain consists of two segments, one for the code of the untrusted module and the other for storing his static data and stack [7]. The specific segment address is determined at load time.

Software packaging technology is the object code that changes an untrusted code module so that he can only jump to the corresponding target address in his own code segment and can only write to the corresponding address in his data segment. Therefore, all legal jump destination addresses in the untrusted module have an identical high bit pattern (segment identifier); similarly, all legal data addresses generated by untrusted modules also share the same segment identifier. In order to prevent the module from modifying his code segment, it is necessary to separate the code segment from the data segment. It may also be illegal for an address with the correct segment identifier. For example, if the address refers to a non-mapped page, this exception is captured by the operating system’s page fault interrupt mechanism.

An unsafe instruction is an instruction whose jumped or stored address cannot be statically verified to be in the correct segment. Most control transfer instructions, such as branch instructions relative to the program counter, can be statically verified. Static variable store instructions that utilize immediate addressing mode are also statically validated. However, with the help of register jump instructions, these instructions are typically used to implement function call returns, and store instructions that use registers to store the target address cannot be statically verified.

A straightforward way to avoid exploiting illegal addresses is to insert a verification code in front of each unsafe instruction [8]. The verification code determines if the destination address of the unsafe instruction has the correct segment identifier. If the verification fails, the inserted verification code will cause a system error, which of course is outside the fault area of the untrusted module. We call this software packaging technology a segment match.

In a typical RISC instruction architecture, segment matching requires four instructions to implement. Pseudocode 1 lists the pseudo-code snippets for segment matching. The first instruction in the fragment places the storage target address in the dedicated register. Each dedicated register is only used by the inserted verification code and is not used by the code in the untrusted module. It is necessary because there may be a jump somewhere in the untrusted code module, which jumps directly to this unsafe storage instruction and bypasses the inserted verification code. Therefore, we arrange all unsafe store and jump instructions to use the dedicated register.

All software package technologies described in this article require special registers. (For architectures such as 80386 [Int86], which limits the number of registers, the packaged module can also use non-reserved registers. In this case, you need to limit the control in the fault area. Stream
Segment matching techniques require four special registers: one for the address reserved in the code segment, one for the address reserved in the data segment, one for the reserved segment offset, and one for the reserved segment flag symbol [9].

Pseudocode 1: The assembly pseudocode for segment matching is as follows:

```assembly
dedicated-reg <= target address
scratch-reg <= (dedicated_reg >> shift-reg)
The address is shifted to the right to get the segment identifier.
The scratch-reg register is not a special register.
The shift-reg register is a special register.
Compare scratch-reg register and segment-reg register Segment-reg register is a special register
trap if not equal
There will be interruption if the memory address is free of segment space.
The dedicated-reg register will be used to store instructions.
```

3.1 Address Sandbox

The segment matching technique has the advantage that it can point out “unpleasant” instructions. It is useful in the software development process. We can further reduce runtime overhead, but at the expense of not providing information about the source of the error.

In front of each unsafe instruction, we can simply insert the code so that the high bit of the target address is set to the correct segment identifier. We call this the address sandboxing, or the sandbox address. The sandbox cannot capture illegal addresses; it can only prevent illegal addresses from damaging any faulty areas, rather than creating an address like a segment match.

Address sandbox technology requires the insertion of two arithmetic instructions in front of unsafe jump instructions and store instructions. The first inserted instruction clears the bits of the segment identifier and stores the result in a special register. The instruction inserted in clause 2 sets the segment identifier to the correct value. Pseudocode 2 lists the pseudocode that does this. As with segment matching, we use special registers to modify unsafe jump instructions and store instructions. Since we are using special registers, the code of the untrusted module cannot generate an illegal address, even if there is an instruction in the module that jumps to the second instruction of the sandbox sequence. This is because the high order of the special register already contains the correct one. The segment identifier, when the second instruction can still be executed normally.

The address sandbox technology requires five dedicated registers. One is used to store the segment mask, two are used to store the segment identifiers of the code and data segments, and two are used to store the code and data sandbox addresses.

Pseudocode 2: The address sandbox assembly pseudocode implemented with the target register target-reg is as follows:

```assembly
dedicated-reg <= target-reg & and-mask-reg
Use the special register and-mask-reg to clear the ones digit of the segment identifier dedicated-reg <= dedicated-reg | segment-reg
Use the special register segment-reg to set the ones digit of the segment identifier
Execute a store instruction with the dedicated-reg register
```

3.2 Optimization

The execution time of the software package can be reduced by traditional compilation techniques. In this article, the author uses loop invariant code and instruction scheduling optimization. In addition to these traditional optimization methods, we have also applied some optimization techniques specifically for software packaging.

The execution overhead of the software encapsulation mechanism can be reduced by avoiding the method of arithmetically calculating the target address. For example, many RISC architectures...
include a register plus offset instruction pattern where offset is an immediate constant within a certain range. This offset ranges from -64 K to +64 K in the MIPS architecture. Consider the store instruction: store value, offset(reg).

The address offset (reg) here uses the register plus offset address mode. The sandbox instruction requires three instructions to be inserted: one to put reg+offset in the special register and two to specify the segment identifier for the special register.

In this article, the author applies this optimization to simply sandbox the register reg without sandboxing the actual target address reg+offset, saving one instruction. In order to support this optimization, a protected area must be created at the top and bottom of each segment, rather than a protected area, so that there is no time-consuming overhead for page breaks due to page faults. However, this also reduces the size of the available area of the segment. Note that the sandboxing register reg also modifies the size of the segment identifier. The segment with the protection area is shown in Figure 2 (the size of the protection area covers the register plus offset instruction).

![Fig.2 Segment with Protected Area](image)

We can also treat the MIPS stack pointer as a special register to reduce runtime overhead. We can avoid using the stack pointer in a sandbox way, but we can sandbox the registers at any time. Since the use of the stack pointer to form an address is much more efficient than changing the stack pointer, this optimization will significantly improve performance.

Further, the modified stack pointer can be prevented by sandboxing with a small constant offset as long as the modified stack pointer is used as part of the load or store address before the next control branch instruction arrives. If the modified stack pointer has been moved into the protected area, using its store or load instruction will cause a hardware address error.

### 3.3 Low Latency Communication Across Fault-Areas

The goal of this work is to reduce the overhead of fault isolation between modules that are collaborative and distrustful. Figure 3 illustrates the main functional components of RPC across faulty areas between trusted and untrusted areas of failure. This section focuses on three aspects of technology across fault areas: ① Explain a simple mechanism that allows a fault area to safely call a trusted stub routine outside the region; then the stub routine can safely call the target area; ② Discuss how the parameters are effectively passed between the fault areas; ③ Discuss registers and other machine states in the process of cross-fault-do-main RPCs on how it is managed to ensure fault isolation. In this situation, the derivation function and the naming function protocol between the fault areas are independent of our technology.

The only way to control the departure from the fault area is to go through a jump table, as shown in Figure 3. Each jump entry is a control transfer instruction, and the target address of the transfer is a legal entry point outside the current domain. By using these instructions, the target address of the instruction is immediately encoded, so that the jump table does not have to rely on the use of special registers. Since the jump table is kept in the (read-only) code segment, it can only be modified by trusted code module.
The dedicated call stubs and return stubs for each pair of fault regions are established for each of the derived functions. Currently these stubs are not generated by the stub generator, but are created manually. These stubs run under unprotected and run outside of the calling domain and the tuned domain. They are responsible for copying cross-region parameters and managing machine state across regions.

Since the stub routine is trusted, we can copy the call parameters directly to the target area. The traditional RPC performs three types of copy work for transferring data when implementing cross-address space: the parameters are organized into a message, the kernel copies the message to the target address space, and finally the caller splits the message into parameters. By communicating the call and the callee by sharing a buffer, the LRPC can transfer data between regions using a single copy.

The stub routine is also responsible for managing machine state. Any register that is used by the caller in the near future, that may be used by the caller in the near future, or that may be modified by the potential callee must be protected. Only these registers need to be saved - marked by the architectural convention as a register to reserve process cross calls. When optimizing, we can avoid save operations if we do not include those instructions that modify the reserved registers in the callee area. Karger uses a trusted connector to perform this type of optimization between address spaces [10]. In addition to saving and restoring registers, the stub routine must switch the execution stack, establish the correct register context for the software encapsulation technology being used, and verify all special registers.

UNIX signaling facility is used to identify such errors, then end the outstanding call and notify the caller of the fault area. If all of the fault zone applications use the same operating system thread, there must be a way to terminate calls that last a long time, such as those caused by infinite loops. The trusted module can use a time facility to periodically interrupt execution and decide if a call needs to be terminated.

4. Conclusion

Firstly, this paper discusses the memory protection of virtual machines and takes Win32ABI as an example in order to explain that the runtime and user processes in the virtual machine must share the address space in a transparent manner. The following major discussion is the memory protection issue at runtime. In the Omniware VM virtual machine, the hardware supports address translation, and the software completes the protection verification work. The segmentation technology here greatly improves the operation efficiency, but at the expense of compatibility. In the Dynamo system, address translation and protection verification are all supported by hardware. It describes the two execution modes in the system and how the protection types of each mode work. If the code cache is efficient enough, the time overhead this technology brings is acceptable.

Secondly, this paper makes an in-depth study on the related technologies of fault isolation, namely the fault isolation technology in the cooperative software modules. In this paper, the author discusses the technology that can be transplanted and independent of the programming language. By implementing this technique in a single address space, the mechanism for communicating across fault areas is more efficient than any RPC available today. In order to prevent the impact of
untrusted modules from exceeding the fault area, the software packaging technology, sandbox technology, is introduced here, which also increases the time overhead by 4%. This overhead is spent executing untrusted code, and also makes software-based Fault isolation technology has greatly improved the performance of the overall software. In the optimization technique, this paper presents a method to effectively optimize the instruction mode such as register plus offset, which is to open the protection area at the top and bottom of each segment, so that there will be no time overhead in page change due to page missing. If a protected area is opened outside the segment, there will be time overhead in page loss, but there is no need to modify the register. The specific strategy can be determined on a case-by-case basis. At last, a scheme for low-latency communication across fault areas is presented. Compared with traditional RPC, LRPC can transfer data between regions with a single copy."

Acknowledgement

In this paper, the research was sponsored by the 2019 year China State Grid Corporation Big Data Center self built technology project: “Research on Key Technologies of isolated sandbox environment based on high concurrent multi-user competition mode”.

References


