Transparent driver-kernel isolation with VMM intervention

ZHENG Hao*, WANG Endong†, WANG Yinfeng‡, ZHANG Xingjun*, CHEN Baoke*, WU Weiguo*, and DONG Xiaoshe*(Corresponding author)
Email: xsdong@mail.xjtu.edu.cn
*Xi’an Jiaotong University, †State Key Laboratory of High-End Server & Storage Technology, ‡Shenzhen Institute of Information Technology

Abstract

How to satisfy the on-demand environment while providing highly dependable services with minimum cost is a challenging issue. Improvements in the reusability of virtualization technology have enabled operating system’s adaptability, which helps users customize their application environments by using various types and versions of operating systems and drivers. However, driver faults in virtual machine are a critical obstacle to achieve reliable user environment, and may even harm the reliability of the entire server. This paper describes Chariot which transparently isolates drivers in a virtual machine without affecting the reusability of the virtualization environment. An isolation loading mechanism links an isolated driver with monitoring wrappers in a virtual machine, which avoids modifying the VM kernel and drivers. According to the monitoring information, Chariot not only instantaneously updates the access control table which records the memory used by the driver, but also sets the write protection of the shadow page table which is corresponding to the whole kernel space of the virtual machine. As a result, the write operations of a driver can be captured and examined in advance. Experimental results show that Chariot can effectively isolate driver faults and improve the reliability of the operating system in a virtual machine. Furthermore, Chariot can be easily extended to isolate new drivers and ported to other versions of operating systems.

1 Introduction

Virtualization technology has become an important means to integrate and optimize resources of high-performance servers. Users can rent server resources to customize their own virtual machine environment. But due to performance requirements, the operating system’s kernel and drivers are still run in the same address space and at the same privilege level in the VM environment. Furthermore, existing research has shown that driver faults have become the main source of OS crashes [7][15][27][39] and also exploited [18][22][25] by attackers seriously affecting the reliability of services offered to users. Currently, there are a variety of solutions to solve this reliability problem. However, three problems prevent these methods from being directly used in a virtualization environment.

First, most solutions are not fully transparent to the kernel and drivers. For example, hardware isolation methods [42][38][36] aren’t transparent to the kernel. They are complex and need to modify the kernel. Software isolation methods [14][29][40] and some language-based methods [44][10] need to add runtime checks to drivers. Type-safe languages [12][21][4][3] and micro-kernel architectures [19][41] even need to rewrite the kernel and drivers. So they have a compatibility problem. The best result [38] of the existing methods is transparent only to drivers and still needs to modify the kernel. If these technologies are used directly, the important advantage of reusability would be impractical.

Second, most solutions are complex which impact their expandability and portability. For example, Nooks [38] needs to modify the kernel to ensure that each interaction between the kernel and the driver switches write permissions; It also needs to synchronize page tables and data between the driver and the kernel; If a programmer writes bug-containing isolation code, debugging a new isolated driver can easily cause a system crash. Consequently, extending the framework to isolate a new driver isn’t easy, and these solutions are less likely to be reused in different versions and types of operating systems.
Third, some solutions use the isolation property of virtualization technology to isolate drivers [7][15][27][39]. However, they don’t solve the reliability problem of drivers in the VM. They isolate the driver at VM granularity. Although they are transparent and expansible, they just make the VM, rather than the host OS, suffer system crashes, in exchange for improving the reliability of the whole server. Thus, there still exists the reliability problem of driver faults in the VM. If a driver fault causes a VM kernel crash, not only the services provided by a faulty driver but also other services provided by other drivers in the same VM will be interrupted.

Thus, this paper describes Chariot, which is a transparent driver isolation architecture in the virtualization environment. The architecture can transparently isolate drivers in fine-grained without modifying the VM kernel and drivers. By monitoring a driver’s running information, Chariot maintains an access control table (ACT) which records the memory that a driver can write. It additionally captures a driver’s write operations by setting the VM’s shadow page table in the VMM to read-only. Then combined with the ACT, a driver’s write operation can be examined and the spreading of driver faults can be prevented. Experimental results show that Chariot can effectively detect and isolate driver faults, improve the reliability of the VM kernel and maintain the availability of the services provided by the VM.

Chariot combines the shadow page table setting and the ACT to isolate the driver. It ensures that the isolation property of the architecture is strong and fine-grained. Capturing the write operation by setting the shadow page table, Chariot inherits the advantage of hardware isolation methods [42][38][36][37][43] and achieves strong isolation. At the same time, Chariot can achieve fine-grained isolation by controlling driver’s write operations by the ACT, similar to the software isolation method [5].

By setting the shadow page table and building the ACT in the VMM, Chariot moves the isolation component down to the VMM. Thus, operations and interactions between the kernel and the driver in the VM don’t need to be changed. By monitoring the driver via the monitoring component in the form of a kernel module, Chariot can transparently and timely update the ACT and set the shadow page table in due course. So the reusability of the virtualization technology is inherited. In contrast, both software isolation methods [5][28] and language methods [44] need to modify the driver to add runtime checks.

To facilitate the expandability of the architecture, Chariot offers a debugging mode to debug a new isolated driver. A user merely needs to inject known permissions to maintain the ACT in monitoring wrappers, and orders the driver to run in debugging mode. Then Chariot can automatically add the permissions required by the driver to the ACT while notifying the user to improve their wrappers. Compared with Nooks [38], debugging a new driver in isolation style is relatively complex. It needs to modify the kernel to ensure each interaction between the kernel and the driver switches page tables, and to change the driver’s memory management to restrict the driver running in a protection domain. Any privilege switching error may yield a system crash. Moreover, the transparency property of Chariot allows it to be easily ported to different types and versions of operating systems by a small number of changes to the monitoring wrappers.

In Chariot, the driver’s write operation capturing is guaranteed by setting the shadow page table in the VMM to read-only in appropriate time rather than by modifying the page table in the VM kernel, or by creating a private page table for the driver. Compared with some hardware isolation methods [36][38], Chariot does not need to maintain additional page tables, which avoids the overhead of page table switching and decreases the complexity of the architecture as well.

The contributions of this paper are summarized as follows: 1) Propose a transparent driver isolation architecture that can easily isolate drivers in the VM; 2) Distinguish the memory that a driver can access for maintaining the driver’s real-time ACT; 3) Present a method to efficiently capture a driver’s write operations in the virtualization environment in order to examine its correctness; 4) Describe a transparent interposition approach using a monitor module to obtain the driver’s running information between the VM kernel and drivers without modifying them; 5) Introduce a method to isolate a new driver using Chariot; 6) Verify the architecture to show that it has a strong feasibility indicators of reliability, performance, expansibility and portability.

Our architecture has been implemented on an X86 machine with KVM virtualization environment. We have isolated 10 different drivers in Linux. Experiments show that Chariot can isolate 80% of system crashes with less than 20% performance loss and can be ported to new VM kernels with only a few modifications to the monitoring code. The rest of this paper is organized as follows. Section II reviews the goals and the architecture of Chariot. Section III clarifies how to establish a driver’s ACT. Section IV describes how to capture and examine the driver’s write operation by using the shadow page in the VMM. Section V discusses how to monitor the driver’s running information. Section VI describes the isolation loading mechanism to insert the monitoring wrappers between the VM kernel and driver. Section VII describes the isolation process of a new driver. Section VIII presents the evaluation of the architecture. Section IX analyzes and compares existing solutions that address the driver reliability problem. We conclude our work in Section X.
2 Overview

To get a transparent driver isolation architecture in the virtualization environment, in this section, we first describe the goals of Chariot, then clarify the system components to realize these goals.

2.1 Goals

The main reason of driver-forced kernel crashes is write operation faults associated with errors of null pointers and invalid pointers [9][38]. Thus, Chariot aims to address this problem caused by drivers’ write operations. Similar to Nooks [38], Chariot follows two basic principles: 1) addressing the driver bug rather than the malicious driver; 2) trying to isolate driver faults as much as possible, rather than solving all driver fault problems. By using the full virtualization technology, one can run various types and versions of operating systems in the VM without modifying the kernel and drivers. In order to inherit this advantage and achieve driver isolation in the virtualization environment, Chariot needs to achieve the following goals:

Goal 1: fully transparent to the kernel and drivers. Since the situation, that the monolithic kernel is commonly used in large-scale servers, will not be changed in the short term, Chariot needs to be compatible with the existing kernel and driver architecture. In order to inherit the advantage of reusability of the full virtualization technology, Chariot can’t modify the VM kernel and drivers and should be fully transparent to the VM environment.

Goal 2: fine-grained memory isolation. No matter a write operation is executed by a function of the kernel driver interface or an indirect reference pointer in a driver, both should be captured and examined by Chariot, with its error spreading being prevented. Likewise fine-grained isolation is capable of avoiding some leaks of write operation error against coarse granularity.

Goal 3: excellent expandability and portability. For the huge number of existing drivers, the architecture should have a good expandability and be easy to isolate new drivers. Due to the good property of reusability in virtualization technology, there are various types and versions of operating systems running in the VM. Thus, to take advantage of the full transparency of Chariot, it should be easily ported to a different VM kernel environment.

Goal 4: moderate performance loss. The system suffers some performance degradation from isolating drivers in the VM. Since less performance loss are better, Chariot should be finely designed to avoid too large overhead. For example, setting the shadow page table should not affect normal operations of other services of the VM kernel, the query time to ACT can’t be too long, the driver monitoring overhead can’t be too large, etc.

2.2 Architecture

In order to achieve the stated goals, we design the architecture of Chariot, as shown in Figure 1. It consists of three components which run with different privileges: the interposition component, the monitoring component and the isolation component.

The interposition component is used to provide an interface to the users, wherein the importance is providing a driver isolation loading mechanism. Under the premise of transparency, the mechanism can establish the connection between an isolated driver and the monitoring component. So a driver loaded by the interposition component will be monitored by Chariot.

The monitoring component is responsible for monitoring an isolated driver transparently. It mainly includes four tasks: 1) obtaining the memory information that a driver’s normal operations need; 2) obtaining a driver’s running states; 3) obtaining the trusted kernel ranges relative to an isolated driver; 4) rapidly recovering a faulty driver for ensuring normal operations of other parts of kernel.

The use of the isolation component is to provide a special operation environment that restricts write permissions of a driver. The environment just has the smallest set of memory resources that a driver needs, and is composed of an isolated driver’s ACT and the write protection of the corresponding shadow page table of the VM. The correctness of each write operation should also be scrutinized by the isolation component. When a driver fault is detected, the error handling in the monitoring component is triggered.

The goals of Chariot cannot be achieved unless all these components cooperate with each other correctly. The relationship of the main components of Chariot
is shown in Figure 2. The connection between a driver and monitoring wrappers in the monitoring component is established by the interposition component. Thereafter, monitoring wrappers are inserted between an isolated driver and the kernel transparently, and the driver is running in the isolated mode with memory usage being monitored timely. The monitoring component reports memory information to the isolation component which updates a driver’s ACT. The isolation component captures driver’s write operations by setting the corresponding shadow page table of the whole kernel space of the VM to read-only. When a write operation of a driver is captured, the isolation component examines its correctness. If a driver failure is detected, the monitoring component tries to recover the faulty driver. Currently, the architecture can isolate various types of drivers effectively. We believe that the approach can be extended to other kernel extensions as well.

3 Access Control Table

Chariot uses an ACT for each isolated driver to record the memory which is needed by a driver’s normal operations to achieve fine-granted driver isolation. In order to show how to construct the ACT, we analyze the types of memory that are necessary for a driver and their monitoring approaches first, then describe the construction of an ACT.

The ACT records all the driver’s accessible memory resources that mainly include four categories:

1) Heap memory: memory that is allocated or released by a driver via calling the kernel’s memory management functions;
2) I/O memory: memory that is mapped or released by a driver via using the kernel’s port mapping functions;
3) Stack memory: memory occupied by the stack of a process which calls a driver’s functions;
4) Grant memory: memory used by kernel objects that kernel functions pass to a driver by parameters.

The heap memory includes the page memory, the slab memory and the high memory. The driver requests the heap memory by the kernel’s memory management functions. For example, the high memory is applied by the `vmalloc` function. The I/O memory is requested by the kernel’s I/O management functions. For example, the `ioremap` function maps the hardware’s I/O resources, such as registers and device memory, to the kernel space memory.

When a driver hasn’t been loaded into the kernel, kernel functions corresponding to the heap and I/O memory are in the form of undefined symbols in a driver module file. When a driver is loaded, they are granted actual kernel addresses. If the kernel addresses granted to these undefined symbols are the monitoring wrappers’ addresses, these two types of memory are monitored. As shown in Figure 3, the wrapper first calls the original kernel function. After successfully allocating or releasing the requested memory, the wrapper records the memory’s start address and size, and informs the ACT (Line 5). We call this wrapper a kernel wrapper. When a driver requests the heap or I/O memory, the kernel wrapper is automatically triggered. Thus the usage of the heap and I/O memory is transparently monitored.

The stack memory is used for the calculation of local variables. It consists of one or more memory page(s). The stack of a process that executes driver functions can be obtained by dedicated functions, for example `current_thread_info()` in Linux or by addresses of local variables which are included in the stack’s memory page. Driver functions may be called by multiple processes, all these stacks should be recorded to the ACT.

The grant memory is often in the form of parameters in driver interface functions. Driver interface functions are function pointer fields of driver interface structures which record the actual addresses of driver functions. For example, the `probe` field of the USB interface

```c
void *chariot_k___kmalloc(size_t size, gfp_t flags)
{
    void * result;
    result = __kmalloc(size, flags);
    CHARIOT_ADD_PERM(result, size);
    return (result);
}
```

Figure 3: The kernel wrapper for monitoring the heap memory and the I/O memory.

3 Access Control Table

Chariot uses an ACT for each isolated driver to record the memory which is needed by a driver’s normal operations to achieve fine-granted driver isolation. In order to show how to construct the ACT, we analyze the types of memory that are necessary for a driver and their monitoring approaches first, then describe the construction of an ACT.

The ACT records all the driver’s accessible memory resources that mainly include four categories:

1) Heap memory: memory that is allocated or released by a driver via calling the kernel’s memory management functions;
2) I/O memory: memory that is mapped or released by a driver via using the kernel’s port mapping functions;
3) Stack memory: memory occupied by the stack of a process which calls a driver’s functions;
4) Grant memory: memory used by kernel objects that kernel functions pass to a driver by parameters.

The heap memory includes the page memory, the slab memory and the high memory. The driver requests the heap memory by the kernel’s memory management functions. For example, the high memory is applied by the `vmalloc` function. The I/O memory is requested by the kernel’s I/O management functions. For example, the `ioremap` function maps the hardware’s I/O resources, such as registers and device memory, to the kernel space memory.

When a driver hasn’t been loaded into the kernel, kernel functions corresponding to the heap and I/O memory are in the form of undefined symbols in a driver module file. When a driver is loaded, they are granted actual kernel addresses. If the kernel addresses granted to these undefined symbols are the monitoring wrappers’ addresses, these two types of memory are monitored. As shown in Figure 3, the wrapper first calls the original kernel function. After successfully allocating or releasing the requested memory, the wrapper records the memory’s start address and size, and informs the ACT (Line 5). We call this wrapper a kernel wrapper. When a driver requests the heap or I/O memory, the kernel wrapper is automatically triggered. Thus the usage of the heap and I/O memory is transparently monitored.

The stack memory is used for the calculation of local variables. It consists of one or more memory page(s). The stack of a process that executes driver functions can be obtained by dedicated functions, for example `current_thread_info()` in Linux or by addresses of local variables which are included in the stack’s memory page. Driver functions may be called by multiple processes, all these stacks should be recorded to the ACT.

The grant memory is often in the form of parameters in driver interface functions. Driver interface functions are function pointer fields of driver interface structures which record the actual addresses of driver functions. For example, the `probe` field of the USB interface
Figure 4: The driver wrapper for monitoring the stack memory and the grant memory.

struct usb_driver is a driver interface function. The kernel transmits the address of a kernel object to the driver by a pointer parameter of a driver interface function, so that a driver can operate the kernel object pointed by the parameter and other kernel objects pointed by fields of the kernel object. For example, the parameter intf in the interface function probe transmits the struct usb_interface type kernel object to the driver.

Both the stack memory and the grant memory are related to interface functions. The former one is related to local variables, and the latter one is related to parameters. The values of function pointer fields of driver interfaces are assigned when the driver is being registered. Thus, if the pointer fields are assigned to the monitoring wrappers, two types of memory are monitored. As shown in Figure 4, in order to ensure the driver's normal operation, the wrapper called driver wrapper first sends the stack memory (line 5) and the grant memory (line 6, 7) information to the ACT, and then calls the original interface function. When the driver function is called by the kernel, the usage of the stack and grant memory is transparently monitored by the driver wrapper.

By using kernel wrappers and driver wrappers, a driver’s memory usage can be timely obtained, and a driver’s ACT can be maintained. Thus, the wrapping of kernel and driver functions is the key method to obtain the driver’s memory information. Since an isolated driver’s write operations are examined by its ACT, in order to rapidly check write operations’ correctness and guarantee the stability of the system, the ACT’s query time should not be too long. Therefore, Chariot organizes the elements of the ACT by using both a red-black tree and an ascending ordered link list. When the element number is small, a query is completed on the list. Otherwise, it is done on the red-black tree. Currently, the ACT just records the write permission. Moreover, it can be easily extended to control other types of permissions.

4 Driver’s write operation

In order to capture a driver’s write operations, Chariot sets the shadow page table to read-only. By doing this and combining with ACT, the correctness of a write operation of driver can be examined. In this way, Chariot moves the isolation component to the VMM without changing the execution and interaction between the kernel and drivers in the VM, thus avoids destroying the reusability of the virtualization technology.

4.1 Write Operation Capturing

To capture a writer operation of a driver before its execution, the shadow page table corresponding to the whole kernel space of the VM should be set to read-only, as shown in Figure 5(a). However, keeping the shadow page in read-only may cause other problems. First, when the isolated driver isn’t called, the kernel will be executed as usual. Since the kernel is trusted, its write operations do not need to be captured and examined. Thus the shadow page do not need to be kept in read-only under this situation. Otherwise, it will have negative effect.
on system performance. Second, when the isolated driver is called, in order to ensure every write operation of the driver can be sustainably captured, the write permission of the shadow page that the driver’s legal operation opened should be reset to read-only after the legal operation. If the reset page is a frequently used page, its write permission will be closed and opened alternately. This will seriously affect system performance. Finally, setting the entire shadow page table corresponding to the whole kernel space of the VM is time-consuming. Thus alternately setting the entire shadow page table to writeable or read-only will affect system performance seriously.

In order to avoid these adverse factors, Chariot improves the shadow page setting approach by the principle of locality, as shown in Figure 5. When an isolated driver is called at the first time, Chariot sets the entire shadow page table related to the kernel space of the VM to read-only for capturing write operations of the driver, as shown in Figure 5(a). During the running of the driver, the opened shadow page’s information is pushed into the opened shadow page cache. The cached shadow page will temporarily not be reset to read-only after the writing. If the cache is full, the oldest cached shadow page will be popped and reset to read-only, as shown in the left part of Figure 5(b).

When the driver isn’t called, the previous setting shadow pages are still read-only. Since setting the entire shadow page table to writeable is time-consuming, we just take the page fault into account to open the write permission as needed. At this point, the opened shadow page is still pushed into the opened shadow page cache, but the popped cache is sent to the opened shadow page pool, as shown in Figure 5(c). Once the driver is called again, only shadow pages in the pool have to be changed into read-only, which reduces the setting overhead and avoids setting the last used page to read-only.

Finally, when the driver is called at the second time or more, the opened shadow pages have already been recorded in the pool and the driver’s write operations can be captured by just setting pages in the pool to read-only, as shown in the right part of Figure 5(b). The process of opening and resetting of the shadow page is still the same as that at the first time, as shown in the left part of Figure 5(b).

The shadow page is set in the VMM, but the driver state information can’t be obtained in the VMM. In order to take different setting strategies to the shadow page, the driver state information (Refer to section 5 in detail) is obtained by the monitoring component and is injected to the VMM. Furthermore, the cached shadow pages shouldn’t be larger than 1% of the VM’s total shadow pages to ensure capturing of the driver’s write operations effectively.

4.2 Write Operation Examination

When a write operation triggers a page fault and is captured, Chariot examines its correctness before the original processing of the shadow page fault. The running of a driver can be interrupted by other processes, and a driver can also call kernel functions. In order to reduce the complexity of kernel wrappers, we consider the process of calling a driver function as the driver running and the driver’s write operations during this time interval are all examined.

Since a captured write operation may be from the driver or the kernel, we should first determine the source of a write operation. If a write operation comes from the trusted kernel, Chariot directly sets the page to writeable, allows the write operation and records the opened page in order to set it back to read-only later. If a write operation comes from the driver, the ACT of the driver is needed. If a write address is in the ACT, Chariot allows execution of the write operation. Otherwise, a driver fault is detected. Chariot records the error address, the error instruction and the faulty driver information and so on, and then reports all these information to the monitoring component in the VM for eliminating the faulty driver.

The ACT can be in any granularity, and the correctness of write operations can be checked in any granularity too. While the memory is opened in a page granularity, if the driver’s usage data and other protected data are in the same page, the protected data may be corrupt by the driver in theory. As the writeable time of the opened page is finite, we don’t consider it as a serious problem.

4.3 Error Handling

When a driver error is detected, the error information of the faulty driver is passed to the monitoring component in the VM. Because the driver is in faulted state, it should be removed as soon as possible. The error handling of the faulty driver is as follows:

1) The driver wrapper directly returns the error code without executing any driver code, for the reason of which the running stack of the driver may be faulted.
2) If there are still some processes using the faulty driver, the error handling will wait them to exit.
3) If the waiting time is expires, the error handling kills the processes by the Linux’s default error handling process.
4) When there is no process using the faulty driver, the monitoring component will release the driver’s resources, remove the driver, and reload the driver later if needed.

Thereby the VM kernel is protected and the fault impacting range is limited.
5 Driver Monitoring

Section 3 has described the form of wrappers. This section summarizes their types and functions. The relation between these types of wrappers will be described in section 6. The wrapper library is the core of Chariot’s monitoring component. According to the wrapped functions, the wrapper can be divided into the kernel wrapper and the driver wrapper. Their forms are similar, so they can be written by the auxiliary tool.

The form of the kernel wrapper is always the same as the form shown in Figure 3. The kernel functions called by the driver usually export their symbols. Among all these functions, only those related to the memory management and the driver register should be wrapped. Thus, the kernel wrapper can be divided into two types:

1) Memory wrapper: monitors the heap memory and the I/O memory occupied by the driver, which is the most common kernel wrapper and has been shown in Figure 3.

2) Register wrapper: is responsible for linking the kernel to the driver wrapper, which will be described in section 6.1.2; This type of wrapper is very important, though its number is few.

The form of the driver wrapper is also simple and regular, as shown in Figure 4. The driver wrapper is used to monitor the stack memory and the granted memory used by an isolated driver. Since each driver interface function may be called by a process, all of them should be wrapped in order to precisely capture the driver state. There are two main types of driver interfaces and both types should be wrapped:

1) One is function pointer fields of driver interface structures, such as the probe, remove field of the struct pci_driver structure and the open, close field of the struct net_device structure.

2) The other is the driver’s callback functions, such as an irq_handler type function used in request_irq, an ide_handler type and ide_expiry type function used in ide_set_handler.

Since the kernel calls the driver by function pointers in the unified interface, the number of driver interface functions which are needed to be wrapped is limited. And it is also a one time work.

In order to capture the driver’s write operations, when the driver is called, the messages that the driver starts or finishes to run have to be obtained timely to inform Chariot’s isolation component for setting the shadow page table in different strategies. Since the isolated driver is loaded by the interposition component, calls from the kernel to the driver via driver wrappers. Thus, these messages can be obtained at the beginning and the end of the driver wrapper, as shown in Figure 4 line 3, 9.

In addition, the isolation domain information still needs to be recorded by the monitoring component. The isolation domain information includes the basic information (i.e. the driver name and the address range), and wrapper information (i.e. the original driver functions’ addresses and the driver wrapper’s addresses).

6 Isolation Loading Mechanism

A driver is loaded by a pre-compiled module file as a medium. If a driver is directly loaded, the monitoring components can’t be inserted between the kernel and the driver. In order to achieve the transparency requirement, Chariot should link the undefined function symbols related to the driver monitoring in a driver’s module file to the corresponding wrappers without modifying the kernel and the driver in the VM. So Chariot has to modify this module file to achieve the requirement. The following works are included: 1) linking the undefined symbols in the driver module file to the corresponding kernel wrappers. 2) linking the driver interface functions to the corresponding driver wrappers.

6.1 Link to the Kernel Wrapper

The key idea of linking the driver to the kernel wrapper is to change the undefined symbols’ information of the memory management and I/O management functions in the driver module file into its kernel wrappers’ symbol information. Whenever the modified module file is loaded, these symbols will be directly linked to their kernel wrappers. As shown in Figure 6, the steps to modify the module file are as following:

1) Replace the symbol names of all undefined symbols, which should be replaced, with the symbol names of their corresponding wrappers’ symbols, in the symbol name section .strtab; (Step 1)

2) Update the offset fields in the symbol name table of the replaced symbols, in the symbol section .symtab. (Step 2)

3) Update the crc fields of the replaced symbols, in the module version section _version; (Step 3)

4) Generate a new driver module file; (Step 4)

5) Load the new driver module file by the original module loading mechanism of the VM kernel.

As a result of these, the undefined function symbols related to monitoring have been linked to their wrappers in the monitoring component. For example, before the driver is loaded, the symbol name of the slab allocation function kmalloc in the module file is changed to the symbol name of its wrapper chariot wc_kmalloc. Once the driver is loaded, it automatically links this undefined symbol to the address of kernel wrapper.
6.2 Link to the Driver Wrapper

Linking the kernel to the driver wrapper requires more efforts. Assigning addresses of driver functions to function pointer fields of interface structures is usually completed in driver register functions. If the wrapper of a register function changes the value of pointer fields from addresses of driver functions to addresses of driver wrappers, the kernel can be linked to driver wrappers, as shown in Figure 7.

In a driver module file, a driver register function is also in the form of undefined symbol. However, its wrapper doesn’t directly monitor the memory, and it is responsible for replacing function pointer fields in interface structures with the corresponding driver wrappers, as the dotted line in Figure 7. Thus, the process of linking the driver wrapper is as following steps:

1) Create kernel wrappers of driver register functions;
2) Link undefined symbols of driver register functions in the module file to their kernel wrappers;
3) After the driver is loaded, a driver register is executed, and the register wrapper automatically links the kernel to driver wrappers.

As shown in Figure 7, linking to the kernel wrapper just needs to replace the monitoring function’s symbol information with its kernel wrapper’s symbol information, and then the kernel wrapper is automatically linked when loading. However, linking the driver wrapper should first link to the register wrapper (Step 1, 2), then the driver’s register wrapper will complete the reset work (Step 3). Compare with the kernel wrapper, the driver wrapper is indirectly linked by the driver’s register wrapper.

7 New Driver Isolation

From the above analysis, if a new driver’s information such as memory resources and operation states can be automatically monitored, the driver can be isolated by Chariot. Thus, when a new driver needs to be isolated, the following steps should be taken:

1) Look up undefined symbols in the driver module file, and create the memory wrappers for the undefined symbols related to the monitoring. In these memory wrappers, the usage of the heap memory and the I/O memory should be monitored.
2) Analyze driver interfaces, and create driver wrappers for all function pointer fields in interfaces. In these driver wrappers, the stack memory, the grant memory and the driver state information should also be monitored.
3) Create register wrappers for the undefined symbol related to the driver register functions in the driver module file.
4) Add these kernel and driver wrappers to the wrapper library in Chariot’s monitoring component.
5) Notify Chariot’s interposition component to change
the symbol information of the undefined symbols which are related to the new added kernel wrappers.

When isolating a new driver, the programmer may be not sure which memory information should be monitored and injected to the VMM. Chariot provides a debugging mode to help the programmer to add appropriate permission injection statements in wrappers. By running a benchmark to test the isolated driver in the debugging mode, Chariot automatically grants the permission of the used memory to the ACT in KVM’s page fault handling. By analyzing the log of these automatic granting memory, the programmer can obtain the information, such as which objects’ memory are needed, where to add them, and even the functions which are needed to be wrapped but missed before. In the debugging process of Chariot, the system seldom crashes, and all important events are logged. Thus, using Chariot to isolate a new driver is much easier than existing solutions.

So far, the new driver can be loaded by Chariot’s interposition component and automatically monitored. Therefore, isolating a new driver just needs some extension of Chariot’s interposition component and monitoring component. A small engineering work can facilitate Chariot’s popularization and expansion.

### 8 Evaluation

This section evaluates the characteristics of Chariot with four aspects: the improvement of the reliability of the VM, the impaction on the VM performance, the expandability and portability of Chariot. Current implementation has isolated 10 different drivers, as shown in Table 1. We choose 6 of them (e1000, rtl8139, sd_mod, usb_storage, ens1370 and intel8x0) to illustrate the features of Chariot.

#### Table 1: Drivers isolated by Chariot. Drivers in bold font drivers are used in the following experiments.

<table>
<thead>
<tr>
<th>Driver</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1000</td>
<td>Intel PRO/1000 ethernet driver</td>
</tr>
<tr>
<td>pcnet32</td>
<td>AMD PCnet32 ethernet driver</td>
</tr>
<tr>
<td>rtl8139</td>
<td>RealTek 8139C+ chips’ ethernet driver</td>
</tr>
<tr>
<td>usb_storage</td>
<td>USB Mass Storage devices driver</td>
</tr>
<tr>
<td>ide_cd_mod</td>
<td>ATAPI CD-ROM driver</td>
</tr>
<tr>
<td>sd_mod</td>
<td>scsi disk driver</td>
</tr>
<tr>
<td>virtio_blk</td>
<td>VirtIO API driver for storage I/O devices</td>
</tr>
<tr>
<td>psmouse</td>
<td>PS/2 mouse driver</td>
</tr>
<tr>
<td>ens1370</td>
<td>Ensoniq ES1370 AudioPCI sound driver</td>
</tr>
<tr>
<td>intel8x0</td>
<td>ALSA driver for Intel ICH (i8x0) chipsets</td>
</tr>
</tbody>
</table>

Figure 8: The reduction in system crashes observed using Chariot.

#### 8.1 Reliability

An important goal of Chariot is to detect drivers’ write operation faults, and prevent the spreading of driver faults by memory isolation. In order to evaluate Chariot’s isolation efficiency on memory damage caused by driver faults, we use a fault injection tool to carry out a series of fault injection tests. The fault injection tool is the same as the automatic fault injection tool used in Rio File Cache [6], Nooks [38] and Mondrix [42]. We port the tool to the Linux 2.6.28.10 kernel for our test. In order to measure the efficiency of Chariot’s driver fault isolation, we define the following two test environments: VM-Native and VM-Chariot. 1) VM-Native: the test environment includes the KVM that Chariot’s isolation component is present but unused, and the unmodified VM kernel; 2) VM-Chariot: the test environment includes the KVM that Chariot’s isolation component is enabled, and the unmodified VM kernel with Chariot’s monitoring module loaded in.

In order to evaluate the isolation efficiency of driver faults, each injected fault is tested in both VM-Native and VM-Chariot environments. We tested 6 drivers, for each driver we run 300 fault injection trials. According to the serious level of the damage that the injected fault introduces to VM-Native environment, we divide the damages into three groups: system crash, non-fatal failure and silent failure. System crash means that when a fatal error occurs in the VM kernel, the system turns into crash like unresponsiveness or an automatic restart. Non-fatal failure means that the VM doesn’t crash, but some of its functions are in abnormal state. Silent failure means that the inject fault hasn’t been triggered, or the damaged data is temporarily not used, and the VM hasn’t shown abnormality. According to VM-Chariot’s iso-
ilation efficiency of the damaging results in VM-Native environment, we further divide the isolation results into three groups: isolated, detected and lost. Isolated means a driver fault can be detected by Chariot and the faulty driver is successfully removed. Detected means a driver fault is detected by Chariot but the faulty driver isn’t removed. Lost means a driver fault isn’t detected by Chariot.

8.1.1 System Crash

The main purpose of Chariot is to prevent the entire VM from crashing caused by driver faults as soon as possible, and avoid affecting other normal services of non-faulty drivers. Figure 8 shows VM-Chariot’s isolation efficiency of injected faults that crash VM-Native. Among all of 1800 trials, 469 cases cause VM crashes in VM-Native. Chariot can successfully prevent 388 cases of the VM kernel crashes and remove the faulty driver. The isolation rate is 82.7%. In addition, there are 76 cases of failures which are detected, but Chariot fails to remove the faulty driver. This is due to the faulty driver couldn’t be timely stopped, thus can’t be removed. Overall, Chariot is efficient to isolate driver faults.

In addition, as shown in the Figure 8, there is much fewer crashes caused by the usb_storage and sd_mod driver compared with those caused by the network driver and sound driver. This is consistent with the conclusion that the network driver is interrupt-oriented as mentioned in Nooks [38]. Linux considers the exception in the interrupt-oriented code as fatal and always causing a crash, which is different from that in process-oriented code. Since disk drivers are process-oriented, even an exception occurs, just the error process will stop and the kernel continues running. Compared with Linux 2.4, the sound card driver architecture has been significantly changed in Linux 2.6. Currently the sound card driver is interrupt-oriented, therefore errors in sound drivers cause a large number of crashes.

8.1.2 Non-fatal Failures

Because the locations of the injected fault are random, after triggering and spreading, these faults may cause a variety of abnormal driver behaviors. For example, the network driver may produce some non-fatal failure, such as non-existing route, a full transmitting ring, etc. Not all the abnormal behaviors are caused by the memory destruction of the VM kernel, and they are not belonging to the main error detection range of Chariot. Because most failures caused by the injected faults belonging to such failure, we need to analyze Chariot’s isolation efficiency of this kind of failure. Figure 9 shows VM-Chariot’s isolation efficiency of the injected faults that cause non-fatal failures in VM-Native. In all fault injection trials, there are 1002 cases resulting in non-fatal failures in VM-Native. VM-Chariot successfully isolates 743 cases and detects 145 cases. The isolation rate is 74.2%. Thus, VM-Chariot can reduce a large part of driver faults which cause non-fatal failures of VM-Native.

8.1.3 Silent Failures

If the fault injection tool injects faults to a seldom executed code path, these failures tend to be not triggered, or just cause non-significant failures after faults being triggered. However, there may still exist write memory operations. If the write address is beyond the driver’s ACT, although the user may not obviously feel the failures, Chariot may still detect it. To this end, we analyze...
Table 2: The performance overhead of drivers caused by Chariot; The performance measurement of sound drivers is running time, others is throughput;

<table>
<thead>
<tr>
<th>Driver</th>
<th>Benchmark</th>
<th>Driver Call</th>
<th>Throughput (M/s)</th>
<th>Relative Perf. (%)</th>
<th>CPU Util. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>VM-Nat.</td>
<td>VM-Char.</td>
<td>VM-Nat.</td>
</tr>
<tr>
<td>e1000</td>
<td>TCP Send</td>
<td>85236</td>
<td>93.57</td>
<td>86.65</td>
<td>92.60</td>
</tr>
<tr>
<td>e1000</td>
<td>TCP Rec.</td>
<td>243756</td>
<td>94.08</td>
<td>93.97</td>
<td>99.88</td>
</tr>
<tr>
<td>e1000</td>
<td>UDP Send</td>
<td>470334</td>
<td>402.8</td>
<td>330.4</td>
<td>82.02</td>
</tr>
<tr>
<td>e1000</td>
<td>UDP Rec.</td>
<td>163957</td>
<td>13060.3</td>
<td>11258.2</td>
<td>86.20</td>
</tr>
<tr>
<td>rtl8139</td>
<td>TCP Send</td>
<td>217829</td>
<td>93.95</td>
<td>78.92</td>
<td>84.00</td>
</tr>
<tr>
<td>rtl8139</td>
<td>TCP Rec.</td>
<td>189971</td>
<td>94.09</td>
<td>93.44</td>
<td>99.30</td>
</tr>
<tr>
<td>rtl8139</td>
<td>UDP Send</td>
<td>202038</td>
<td>270.3</td>
<td>225</td>
<td>83.24</td>
</tr>
<tr>
<td>rtl8139</td>
<td>UDP Rec.</td>
<td>123251</td>
<td>11341.3</td>
<td>10994.2</td>
<td>96.94</td>
</tr>
<tr>
<td>usb_storage</td>
<td>Untar</td>
<td>852</td>
<td>17.3</td>
<td>19.8</td>
<td>87.37</td>
</tr>
<tr>
<td>sd_mod</td>
<td>Untar</td>
<td>435</td>
<td>5.46</td>
<td>5.55</td>
<td>98.37</td>
</tr>
<tr>
<td>ens1370</td>
<td>mplayer</td>
<td>33943</td>
<td>286.32</td>
<td>286.35</td>
<td>99.99</td>
</tr>
<tr>
<td>intel8x0</td>
<td>mplayer</td>
<td>66803</td>
<td>286.47</td>
<td>286.49</td>
<td>99.99</td>
</tr>
</tbody>
</table>

Chariot’s efficiency of isolating such failures. Figure 10 shows VM-Chariot’s isolation efficiency of injected faults that cause VM-Native in silent failures. Chariot isolates 233 cases and detects 11 cases out of 329 cases, and the isolation rate is 70.8%. The isolation efficiency is acceptable.

Experiments show that Chariot can eliminate 82.7% of system crashes and satisfy the design requirements. Meanwhile, Chariot can handle a large part of non-fatal failures and silent failures. This makes the driver do pre-recovery before it causes serious failures. Therefore, Chariot is effective to prevent VM kernel failures caused by driver faults in the VM, avoid the VM kernel crashes, and improve the reliability of the VM kernel.

8.2 Performance

Chariot’s performance overhead is from driver memory usage monitoring, shadow page fault handling and write operation’s correctness checking, etc. We use several benchmarks to evaluate the performance loss regarding to drivers’ normal operations. All tests are completed within a VM. The test is performed on a Lenovo computer with Intel Core2 Duo CPU E8400 at 1.9G and 4GB of RAM. Both the host and guest OS are Centos 5.8, and the KVM runs with QEMU1.2. Network test is accomplished by two identical computers.

Network driver’s performance overhead is measured by the netperf tool to test the transmission and receive performance of TCP and UDP. In the TCP test, the transmission and receive buffer is 16384 and 87380 bytes respectively, and the message size is 16384 bytes. In the UDP send test, the transmit and receive buffer is 109568 and 110592 bytes respectively, and the message size is 65507. While, in the UDP receive test, the buffer sizes are just contrary. As shown in Table 2, in two network drivers, TCP throughput falls by 0.1-16% and the CPU utilization rate increases by about 2-10%. UDP throughput falls by 3-17% and CPU utilization rate increases by about 2-20%. From Table 2, we can observe that there are much more called times of network drivers than other drivers. It is consistent to the data processing method of the network driver, which frequently uses the interrupt to send data, receive data and query the network status. In the e1000 driver’s UDP send test, the called times of the driver is substantial more than other UDP tests, hence the shadow page table should be set and updated more frequently, which causes more performance loss. Although Chariot brings in some overhead, the performance loss of network drivers is still within an acceptable range.

We run the benchmark test on uncompressing a file in a U disk and a scsi disk to measure the performance loss of the usb_storage and sd_mod driver. As shown in Table 2, the throughput of the usb_storage driver falls by 12.6% and the CPU utilization rate increases by about 4.6%. While the throughput of the sd_mod driver falls by 1.6% and the CPU utilization rate increases by about 12.47%. The different performance loss is
related to the data processing methods of drivers. The usb_storage driver creates its own process to handle the block request, and the process sleeps when there is no request. When a new data request arrives, the kernel needs to schedule the process, which consumes some time. Since the shadow page table is set to read-only in VM-Chariot, the waking processing may produce some page faults which will delay the wake-up. Therefore, the throughput loss is relatively higher. As in the sd_mod driver, the kernel directly calls the driver interface function to handle the block request without process scheduling, so the time overhead is relatively lower. At the same time, due to its faster request processing, the CPU overhead is relatively higher.

The sound driver benchmark uses the mplayer to play an MP3 file at a rate of 128kbit per second. As shown in Table 2, the increasing time of both sound drivers is less than 0.1%, and the CPU utilization increases by less than 0.5%. Chariot nearly has no influence to the performance of the sound driver. Although both the sound driver and the network driver use the interrupt function, the sound driver doesn’t use the interrupt to handle data requests. Therefore, the called times of the sound driver is significantly fewer than those of the network driver, and the performance loss is at a lower level too. We don’t test the performance loss of other drivers, but we believe the performance loss should not be too large.

### 8.3 Extensibility and Portability

Chariot contains about 18K LOC. Except 244 LOC are added to KVM, there is no modification of the kernel and drivers in the VM. The major part of the code is for wrappers, which contains about 9.5K LOC. Currently, Chariot has isolated most of the common drivers in KVM, and their driver interfaces and kernel functions are already wrapped. If a new driver needs to be isolated, a small amount of engineering work is needed. As shown in Figure 11, the driver wrappers of all network device drivers are the same, and their kernel wrappers also have a great consistency. Thus, they can be directly reused. For example, the wrappers for the e1000 driver can be reused for isolating the pcnet32 and rtl8139 drivers, except one additional kernel wrapper for the pcnet32 driver. Similarly, there is a great reusability for the disk driver and the sound driver.

In addition, the convenience of driver isolation using Chariot is also reflected in the time advantage. When we isolated the first driver (usb_storage), we spent a few weeks on exploring, coding and debugging the architecture, while the isolation of usb_storage costed two weeks. Then, the isolation of e1000 just costed one week. The isolation of the other two network drivers...
only costed one day. The isolation of other drivers was all finished within 3 or 4 days. Thus, we believe that Chariot can be easily extended to isolate new drivers.

Because of the reusability of virtualization technology, VM can directly run several types and versions of operating systems. In order to make Chariot be widely accepted, its monitoring module which runs in the VM should be easily ported to different VM kernels. We choose several commonly used kernel versions of Linux as porting targets to demonstrate the portability of Chariot, and analyze the changes of wrappers. Figure 12 shows the degree of changes of the monitoring wrappers when porting Chariot’s monitoring module to Linux 2.6.18 and Linux 2.6.32.

Linux 2.6.18 is much different from our development environment Linux 2.6.28.10. The driver wrappers need to be modified at about 78 places, and the kernel wrapper needs to be modified at 19 places. Especially, the network device driver, usb_storage, ide_cd_mod and sd_mod have a lot of changes. But compared to the number of unchanged wrappers, this number is small, as shown in Figure 12(a).

There is a small number of modifications when porting Chariot to the Linux 2.6.32. The driver wrappers require modifications at 84 places and the kernel wrappers needs to be modified at 9 places. The modification number is larger than Linux 2.6.18. However, the task is quite easy because the main difference is between network drivers of two versions, that a large number of function fields (i.e. open, stop, do_ioctl, etc.) are moved to the next level structure (struct net_device_ops). And the number of this part modification is 77. At last, there are just 7 modifications left in all other drivers, as shown in Figure 12(b). Thus the effort for addressing the changes of wrappers is very small and the process is relatively easy.

9 Related Work

The virtualization technology has been widely used to enhance the utilization of the system resource and integrate services. It is also used as a solution to address the reliability problem of the driver, such as in Xen [15][2], L4Ka [27] and iKernel [39]. This technology is able to isolate drivers from the kernel into separated VM instances, and uses the isolation property of the VM to isolate the driver. This method just uses the VM as an alternative to the whole system to suffer the possible system crashes, and improves the reliability of the whole computer system. However, the reliability of the driver inside the VM isn’t improved. In order to avoid the interruption of services caused by driver faults inside the VM, they create many VM instances for every driver to provide services and introduce a higher performance penalty.

Some solutions use hardware-based protection mechanisms to isolate driver faults. Palladium [8] uses the hardware segmentation protection mechanism to isolate extensions in both kernel-mode and user-mode. The use of segmentation protection mechanism makes the method much difficult to program. FPD [36] uses the hardware page protection mechanism to isolate user-mode extensions, and provides determining strategies to judge whether a system call from an user-mode protection domain is legal or not. However, the method was not implemented to isolate kernel-mode extensions, such as the driver. Nooks [38] creates a private page table for each driver to limit its write permission by a complex synchronization and update mechanism. It has to modify the kernel to ensure that all interactions between the driver and the kernel switch the page table and the stack. This brings a great performance overhead. Furthermore it is hard to isolate a new driver because of the complex implementation. Mondrix [42] is a hybrid isolation method mixing hardware and software technologies. It provides 32-bit fine-grained access control and protects the inter-domain access switching. However, the method that requires the specific hardware will cause the problem of popularity, transparency and compatibility.

Though SFI [40], Pittsfie [29] and the XFI [14] isolate simple kernel extensions with a small overhead, they can’t be used for complex kernel extensions. BGI [5] extends XFI to the complex driver interfaces by manually interposing every possible interface between the kernel and the driver. It limits operations of the driver by an access control list. All its characters depend on the well-defined API of the Windows’ driver model. LXFI [28] extends the BGI to more complex driver interfaces in Linux. It divides the privilege of the sharing module by the principle. FGFT [24] isolates driver at the granularity of a single entry point and uses the existing power-management code to save and restore device state. However, most of these methods require the programmer to clearly add annotations to the interfaces between the kernel and the driver, and to modify the driver to add runtime checks. This destroys the transparency of the driver and affects its direct using in the VM environment.

CCured [11][32], Cyclone [23] and Deputy [10] strengthen the memory security of the C program language at the source code level by providing the run-time memory safety check. SafeDrive [44] guarantees type safety in the driver through type inference, programmer annotations and runtime checks which are automatically produced by Deputy. It needs to modify both the kernel and the driver, and affects its transparency. Its isolation is also poor. Dingo [33][34] uses a software protocol description language to define correct behaviors of
drivers. Although it is incompatible to existing drivers, it can effectively check synchronization faults and behavior faults of drivers. Checking these types of faults can be used as a complement to our architecture.

Static language checking tools [1][13][10] try to eliminate driver faults by checking drivers’ source code. These methods don’t bring the run-time checking overhead and can discover a part of type-safe faults. However, certain complex driver faults, such as synchronization faults, still exist. Some operating systems and kernel extensions like JavaOS [30], SPIN [3], Sigularity [20], Vino [35] are written in type-safe languages. They can ensure strong isolation, but need to rewrite the kernel and the extension. Thus, they are not compatible with the existing OS and large-scale extensions.

Many systems move drivers to a separate process address space, such as micro-kernels [41][19][31], and user-mode drivers [16][26][17]. In this way, driver faults just crash the corresponding user-mode process and don’t cause the system crash. However, these methods have low efficiency and large amounts of data need to be frequently transferred between user-mode and kernel-mode. Moreover they are non-transparent and incompatible with traditional monolithic kernels, requiring to rewrite all drivers.

10 Conclusion

The virtualization technology can increase transparency and exploit heterogeneous and application-customized hardware to deliver usability growth. The main purpose of Chariot is to effectively and transparently isolate drivers in virtualization environment, which prevents driver faults from crashing the VM kernel and improves the reliability of the VM. Currently, Chariot has effectively isolated 10 drivers in KVM environment on the x86 architecture with moderate performance loss. Studying the reduction of the performance loss of network drivers is a future work. Chariot can easily isolate new drivers and be ported to other versions of the VM kernel with little effort. Although Chariot is capable of dealing faults related to write operations of the driver, we will continue to consolidate its error detection mechanism allowing more driver faults to be isolated. Moreover, improving the recovery function of Chariot is an important part of our future work.

11 Acknowledgments

This work is supported by the State 863 project (Grant No. 2008AA01A202), the project of the National Key Technology R&D Program (Grant No. 2011BAH04B03), Guangdong NFC (Grant No.S201101006109) and Shenzhen Scientific plan (Grant No. JCYJ20120615101127404). We thank Inspur for their useful suggestions and supports. And we especially want to thank Yan Kang, Wu Shusen, Zhang Yizhi, Zhou Mosong and Wang Qiang for their great outstanding contributions to the experiments.

References


