

DefensiveDriving: Guarding Operating System from Device Driver Bugs and Crashes

Ganesh Bikshandi, Jia Guo, Jusin Trobec

University of Illinois at Urbana-Champaign
{bikshand, jiaguo, jtrobec2}@uiuc.edu

ABSTRACT

This paper presents a Nooks-like reliability subsystem, *DefensiveDriving System*, in Choices, an OO operating system. Our system successfully isolates the OS from driver failures and prevents many types of driver-caused crashes with little or no change to existing driver and system codes.

Implementing such a system in Choices is an interesting experience. Choices being written using Object Oriented paradigm, has a well organized structure and good encapsulation of different functionality of the kernel code. Our interface in Choices is much cleaner and more effective than that of Nooks for Linux. We show the benefit of building a system on an OO operating system in our paper.

1. INTRODUCTION

In modern operating systems, kernel extensions such as device drivers are the major source of failures. For example, a study at Stanford University found that device drivers have 3 to 7 times the bug frequency as the rest of the OS[6]. There are several reasons that contribute to the high rate of device driver failures[16]. First, drivers are typically written by device manufacturers rather than by operating system developers with extensive kernel programming experiences. Second, drivers are frequently created by copying and editing code templates from existing drivers, often without complete understanding, leading to subtle bugs. Third, the kernel programming environment has many unenforced or poorly-documented conventions about synchronization and memory access, making kernel-mode programming and debugging challenging. Finally, driver programming often requires understanding the operation of complex asynchronous devices, their control protocols, and their failure modes. In the future, it is clear that improving operating system reliability depends on improving device driver reliability, because the kernel is no longer the primary source of bugs.

On the other hand, device drivers, as part of kernel exten-

sions have become increasingly prevalent in commodity systems like Linux or Windows. For example, extensions now account for over 70% of Linux kernel code, while over 35,000 different drivers with over 120,000 versions exist on Windows XP desktops. With the enormous amount of drivers, we need to find a smart way to tackle the problem with minimal modifications to the drivers.

In the literature, there are many research topics on improving extensibility and reliability through (1) the use of new kernel architectures[7], (2) new driver architectures[18], (3) user-level extensions[10], (4) new hardware[9], (5) type-safe languages[2]. However, those approaches have limitations. For example, although the use of new kernel wrapper can isolate and protect the usage of driver devices, it does not support an easy way to recover. Type-safe languages require rewriting the driver. Some of the approaches require major modifications to both the kernel and the driver architecture.

For the above reasons, we propose to adopt the approach that Nooks[16] uses. Rather than guaranteeing complete fault tolerance through a new (and incompatible) OS or driver architecture, it aims to prevent the vast majority of driver-caused crashes with little or no change to existing driver and system code. In Nooks, for eight kernel extensions they isolated, seven required no code changes, while only 13 lines changed in the eighth. Nooks isolates drivers within lightweight protection domains inside the kernel address space, where hardware and software prevent them from corrupting the kernel. Nooks also tracks a driver's use of kernel resources to facilitate automatic clean-up during recovery[17]. Our approach is similar as the Nooks work because we want the extension to execute on the current platform without changes if possible. Also, we will use conventional language such as C or C++ so that the developers need not change the programming language, environment, or their perspective.

We are interested in implementing a scheme similar to Nooks in Choices[4], an object-oriented operating system. Choices comprises a hierarchy of frameworks. Frameworks not only allow the design of layers, but also permit the construction of more complex structures. The object-oriented operating system approach builds system software that models system resources and resource management as an organized collection of objects that encapsulate mechanisms, policies, algorithms, and data representations. It ensures the encapsulation of kernel data structure. The marriage of Nooks and

Choices would solve the problem of direct access to kernel data by drivers addressed in section 4 in [16].

Our system implementation is different from Nooks. The design is more concise and portable than Nooks thanks to the object oriented design of the operating system. The encapsulation of drivers can be easily done by inheriting the driver class and adding guarding functions in the inherited class. The creation of protection domain is also straightforward since in Choices, domain, stack, heap are already defined as classes. We will show statistics for our concise design in Section 5.

In summary, We seek to address the following questions in this research. What is the minimum requirement (in choices) for achieving the intra-address space protection? How much change is required for the kernel and the driver code? How much performance is lost due to the scheme? How feasible is to generate the wrapper automatically? The rest of the paper is organized as follows. Section 2 discusses the works related to this paper. Section 3 discusses the overall design of the system and section 4 answers several issues related to protection. In section 5, we present several experimental results. In section 6, we compare our approach with that of nooks. Section 7 presents several future directions and finally, we conclude in section 8.

2. RELATED WORK

Our project is an extension of Nooks [16] to Choices. Nooks strives to enhance OS reliability by isolating the OS from driver failures. This is achieved by implementing a lightweight protection domain inside the kernel address space, where hardware and software prevent the drivers from corrupting the kernel. Nooks was implemented for Linux operating system, while we plan to improve the reliability of Choices. Choices is different from Linux in that it is a fully object oriented operating system and hence amenable to additions of new functionality (like our driver protection scheme). Also, our design is based on object oriented programming concepts. We use subclass of drivers to achieve the wrapper functions that Nooks uses.

Microkernel [11] based operating systems achieve reliability by minimizing the kernel functionality and executing the kernel extensions in user space, like normal programs. These programs communicate with other programs and the kernel via the message passing interface, provided by the kernel. The biggest drawback of this approach is the cost incurred by inter-process communication and extra software complexity in employing fast IPC techniques. Our approach is rather practical. It applies to existing operating systems, with not much of complexity being introduced.

Microsoft Driver verifier [13] is another work related to our approach. Here, all the calls between kernel and the drivers are wrapped by wrapper functions. These functions check the calls for any violation. This is a small subset of our scheme. Our scheme, apart from differentiating the kernel-driver boundary, also restricts the driver part by enforcing a lesser privilege. Also, Privilege Level Change [8] based schemes execute the drivers in lesser privilege mode. Execution of privileged instructions by the driver will essentially trap the operating system. One can think of our scheme as

the combination of these schemes.

Hardware memory protection[15] schemes use memory protection to restrict the device drivers from writing to kernel memory. These schemes do not address recovery or try to make an OS recoverable. Software Fault Isolation [14] based mechanisms inject codes, in to the drivers, that check the critical instructions and addresses. Such mechanisms have huge run-time overhead. Safe Language [3] methods rely on type safe languages that detect bugs at compile time. This is not applicable to existing commodity operating systems and device drivers.

Virtual machine technologies [5] have been proposed as a solution to the reliability problem. They can reduce the amount of code that can crash the whole machine. Virtualization techniques typically run several entire operating systems on top of a virtual machine, so faulty extensions in one operating system cause only a few applications to fail. However, the guest operating system that had a faulty extension will still crash. To achieve reliability, multiple instances of guest operating systems should be running on the virtual machine and applications should be partitioned among them. This restricts the efficient utilization of some features provided by an OS, like fast IPC. Also, if a driver executes directly in a virtual machine monitor, a fault in the driver could cause all the virtual machines to halt.

3. SYSTEM DESIGN

We propose that operating systems should support executing drivers in a fault-isolating environment so that a faulty driver cannot prevent the rest of the OS from functioning. In addition, we hope to provide this isolation with minimal changes to the kernel and drivers, to maximize compatibility with the existing code base. Also, our scheme involves as little overhead as possible.

3.1 Major components of our system

There are five major components in our system: protection domains, domain manager, XPC, proxy and exception handling. We will explain their features one by one.

1. Protection domains

Protection domains control the memory access rights according to different protection levels. For example, the kernel has *read-write* access to the entire address space, while each driver is restricted to *read-only* kernel access and *read-write* access to its local domain.

When the protection domain is created inside the kernel, all the kernel space should be mapped to the new domain. However, the access to the kernel should be changed to *read-only* for the drivers. Only object table (such as page table) and I/O buffer should be *read-write*. Then the protection domain creates its own stack and heap. For those local data structures, the accesses are marked *read-write*.

One example of protection domain is shown in Figure 1. There are two protection domains created. Each one contains the mapped kernel memory with *read-only* access and its own extension area with *read-write* access. The drivers in one protection domain only have

read-only access to the other. This ensures the driver only writes data in the protected area which will be monitored by *Domain manager* explained below.

In Choices[4], we identified that the domain class is very similar as the protection domain we described. It has the supports for mapping the kernel domain, adding new memory objects such as stack and heap, and changing the access level. We have been using it to realize our protection domain concept.

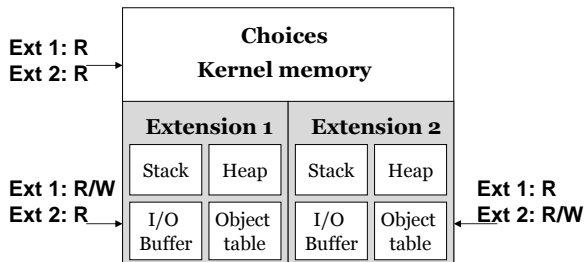


Figure 1: Two protection domains in the kernel

2. Domain manager

Domain manager is designed to manage the protection domain. It involves the creation, manipulation, and maintenance of lightweight protection domains. It assigns the same type of drivers to the same protection domain. In this way, the same type of drivers can share the resources in the domain and the creation overhead can be amortized. The domain manager is placed inside the kernel.

3. XPC

Extension procedure call(XPC) is a kernel service that transfers mechanism to isolate extensions within the kernel. It should support the control flow in both directions between extensions and the kernel.

4. Proxy

Proxy are required to ensure backward compatibility with the device drivers. No changes to both the extensions and kernel should be done, except some minimal one time changes to the kernel. This is achieved by using proxy functions that have the same interface as the kernel API and the extension API.

Initially, we need two types of proxy functions to interface with the kernel and extension functions as what Nooks does[17]. One is kernel proxy called by extensions to execute kernel-supplied functions. The other is extension proxy called by kernel to execute driver-supplied functions.

Thanks again to the nice design of Choices, the drivers are implemented in an object-oriented way. Both functions are provided in the driver class as member functions. Our proxy can inherit the original driver class to naturally achieve the proxy functionality. There are

several advantages of such design for proxy. First, it keeps the driver code untouched. Second, on the kernel side, the modification is minimal; we only need to change the type of driver to the proxy type. Third, the kernel proxy and the extension proxy are kept together in a natural way. It helps the programmer understand the code better.

5. Exception handling

Inside the proxy, we put the major function call into a *try-catch* statement. Thus, even the driver fails, the proxy can catch the exception and try to deal with it. In the future, a recovery manager can be added here to recover the devices from failure.

3.2 Overall system design

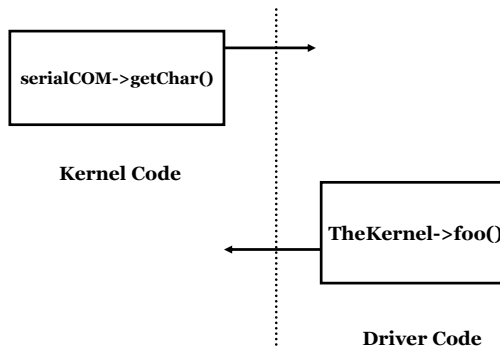


Figure 2: Normal kernel-driver calling sequence

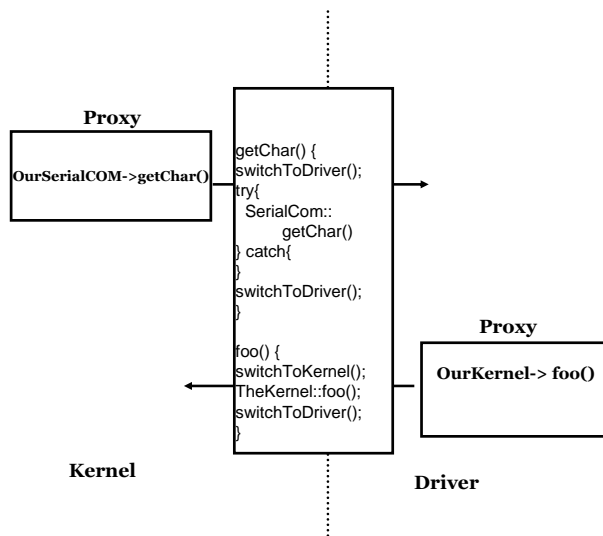


Figure 3: Overall view of our scheme

The key idea of our project is to achieve isolation of kernel region from the driver region. We achieve this by subclassing the kernel and device driver classes which overload

the original methods in those classes. The overloaded methods perform the domain switching and delegate the call to the original call. This is illustrated in the Figure 3. Figure 2 shows the normal kernel-driver calling sequence.

In Figure 3, OurKernel and OurSerialCOM are instances of the subclass of SerialCOM and TheKernel, the original serial device driver and the original Choices Kernel. As can be seen in the figure, exception raised by driver code is caught by the proxy functions. Proxy functions also perform the following tasks - 1) Checking and verifying the parameters. 2) Copying certain kernel data structures inside the extension domain. 3) Calling the appropriate function call in kernel (locally) or extension (via XPC).

When the kernel calls a function in a driver, the call is mapped to a function of the proxy driver class instead of the actual function. The proxy function switches the domain to that of the driver. While switching domain, we make sure that the kernel regions are included in the new domain, but with *read-only* access to it. We also change the stack pointer to that of the driver's private stack. Finally, a change to user mode instruction is executed.

Then the original driver function is invoked. The driver function now operates under restricted environment. Any writes to kernel region will result in an exception. Also illegal instructions cannot be executed by the driver. However, we make sure that the driver has *read-write* access to those specific memory regions it requires to modify (for e.g. to the memory region UART_BASE for the IntegratorSerial driver).

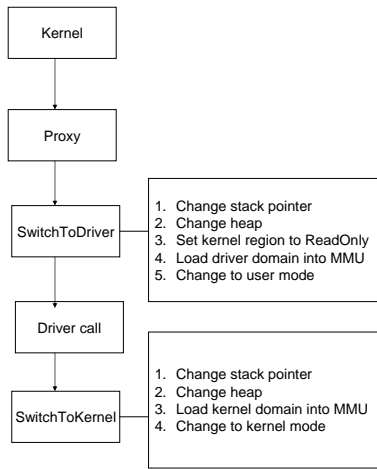


Figure 4: Flow of control in our scheme

Upon return from the original driver function, the stack pointer is reset to its original value and the mode is changed to supervisor mode. Also, the domain is changed back to that of the kernel. During the driver-to-kernel call, exact reverse process occurs. Figure 4 graphically illustrates the flow of control during kernel-to-driver call. Also show in the

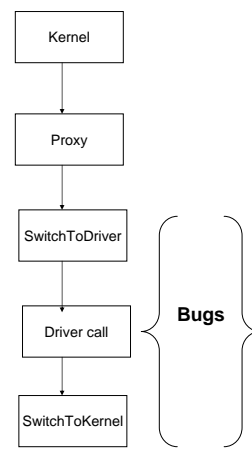


Figure 5: Fault containment using our scheme

figure are the major steps performed during `switchToDriverDomain` and `switchToKernelDomain` function calls. Figure 5, illustrates how a bug in a device driver is contained within the boundaries of the protection domain.

3.3 Design Details

This section discusses the details of our design, including structures and processes implemented to achieve isolation and reliability. First we describe the entities that are present in our system, including the data and capabilities of which they are composed. Second, we discuss the way that these entities interact with the Choices OS, drivers, and other entities within our system.

3.3.1 Entities

The following section describes the entities that comprise the defensive driving system. A high-level UML block diagram of the system is shown in Figure 6.

- DDomainManager

The DDomainManager class is a singleton class responsible for the creation and management of protection domains. When a DDomain is needed, a method can be called on the DDomainManager instance to either retrieve an existing domain, or return a new one. When a domain is requested, the DDomainManager queries a hash table to determine if the domain already exists. If so, it will return a reference to that domain; if not, it will create a new domain, store a reference to the domain in the hash table, and return a copy of the reference. This gives our system the ability to reuse existing domains for multiple drivers.

- DDomain

Instances of the DDomain class represent an abstract view of our protection domains. Each DDomain includes a Choices Domain object – an object that Choices

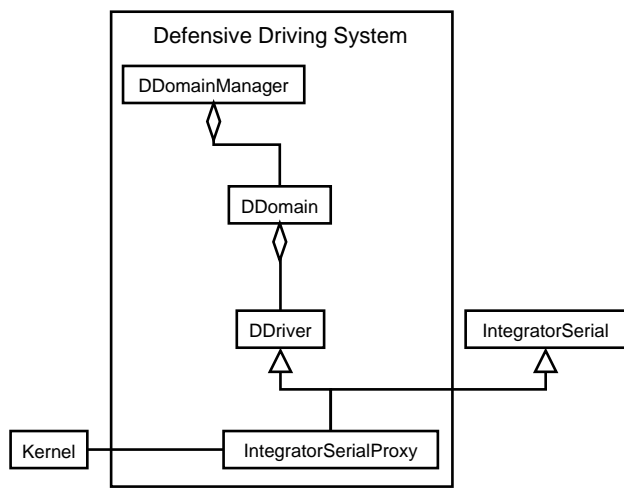


Figure 6: UML Block Diagram of the Defensive Driving System

uses to define a set of memory maps integrated into one abstract memory area. In addition, `DDomain` objects include a set of references to `DDriver` objects that utilize the domain. The `DDomain` provides a central point for managing all the drivers within a domain, similar to the way the `DDomainManager` provides a way to manage all domains within a system.

- **DDriver**

The `DDriver` abstract class encapsulates the basic data and methods necessary for drivers to use the Defensive Driving system. In addition, it provides an interface for the other classes in our system to handle drivers in a uniform manner – an essential detail since drivers in Choices previously did not implement a common interface, and consequently could not easily be treated in a uniform manner. Classes deriving from `DDriver` – the Driver Proxy Classes – have access to a `DDomain`, and functionality that allows them to switch to that domain so that they can safely execute existing, potentially unsafe driver methods.

- **Driver Proxy Classes**

Driver Proxy Classes are classes that are used to adapt an existing Choices driver to work inside of the Defensive Driving system. These classes use multiple inheritance and derive both the `DDriver` class, and the actual driver that’s being adapted for use within our system. This allows us to change the Choices kernel only minimally – simply instantiate an instance of the proxy class instead of the original driver class, and all other interaction with the object is the same since it implements the same interface that its parent class does. Extending the `DDriver` class gives the new proxy access to a protection domain, the methods required to switch to it, and the ability to manage it along with the other drivers that use our system.

In order to reduce the amount of time necessary to add an existing driver to our system, we have begun work on a utility to generate the Defensive Driving

proxy classes that act as intermediaries between the actual Choices drivers and the Choices Kernel. Having a reliable and functional proxy class generator would make preparing existing drivers for use in our system a very simple matter. After the proxy class is generated for a given driver, we have only to change the lines of code in the kernel where the driver is instantiated to use our proxy, and the driver will run in the context of our system.

We are building our utility utilizing Elsa, a C++ parser developed by a student at the University of California-Berkeley [12]. Elsa generates an Abstract Syntax Tree (AST) that we can examine to take a more intelligent approach to developing our proxies. This will provide us with a more robust and functional proxy-generator than a simple text substitution approach. For example, being able to understand when we are examining public versus private methods in a driver would allow us to skip generation of methods that would not be exposed to the kernel, and therefore not be necessary to create in the proxy.

3.3.2 Interactions

The basic flow of our system is as follows:

1. The Choices Kernel first creates a domain to contain the memory accessible to a driver using the `DDriverManager` instance.
2. In order to load a driver, the kernel instantiates the driver’s proxy class, passing in a reference to the domain in which it will operate.
3. The kernel interacts with the driver as it would have previously. This time calls made to the driver are first passed through the proxy.
4. The proxy classes methods switch into our protection domains, and execute the appropriate method on the original driver.
5. After the original driver call returns (successfully or otherwise), the proxy switches back to the original domain, handles any exceptions, and returns control to the caller.

4. PROTECTION ISSUES

In this section, we answer several questions to justify our design.

- - *What is the need for private stack?* - Stack overflow is one of the biggest source of bugs in a software. Stack overflow may lead to silent data corruption, without generating any signal or exception. In order to prevent a buggy device driver from corrupting a kernel stack, we insert a private stack within each protection domain. Every other region in the kernel (except the heap region of the domain) is made read-only. This ensures that an access to a kernel region via a stack overflow will raise an exception.

- *What is the need for private heap?*. Like stack overflow, heap overflow is another biggest source of concern. By having a private heap, we restrict all the dynamic allocations performed by the driver to this region. This way, a illegal write-access to kernel heap can be prevented.
- *What is the need for user mode switch?* - In ARM processor, virtual memory protection does not apply to privileged mode. That is, in privileged mode, even if the memories are marked read-only, the code still can write to it. In order to achieve full protection, we need to switch to user mode.
- *How do you limit the access of application to the driver domain?*. The stack and the heap of the driver domain are added to the kernel region of the memory. This way, after the driver call returns and the domain is switched backed to that of the kernel, the application will have no access to the driver domain. This also prevents the domain memory from being *swapped out*.
- *How do you prevent preemption (since the driver code runs in user mode)?*. Though we change to user mode, we do not enable interrupts.
- *Are the driver and kernel code separated?*. No. Though, this will give us better protection, we did not attempt this at this point of time.

5. EXPERIMENT EVALUATION

The main purpose of our evaluation is to establish that a) Our project can successfully catch several common device driver bugs and isolate the crashes due to them. b) The performance degradation due to domain switching and other procedures is minimal. c) The fraction of changes applied to both the kernel and the driver code base is as little as possible.

We evaluated our design on the Choices[4] OS which runs on ARM integrator. To make the development and demonstration easier, we choose to use QEMU ARM emulator[1].

5.1 Reliability experiments

First, we studied the driver mechanism in Choices, using IntegratorSerialDD.cc, the serial device driver, as our base. Then we wrote dummy device drivers with bugs injected into them. The four test cases we designed are:

- stack out of bounds access
- null pointer dereference
- corrupting the kernel memory
- read access to kernel data structure.

The first test case creates an array with 10 elements. But it tries to assign a value to the 2000th element. Without the Defensive driving system, the test case is passed successfully. However, running within our system, the buggy code is successfully caught because it is in user mode and it tries to assign values to the place out of the protection domain. We encountered compiler challenges when designing

	Unprotected Driver	Desired Result	DefensiveDriving
Writing Beyond the Boundaries of an Array	Successful (bad)	Write prevented.	Write is prevented, exception is generated, handled by proxy.
Dereferencing an Invalid Pointer	Creates an exception, caught by the test function (bad)..	Exception caught and handled before returning to the test method.	Creates an exception, handled by proxy.
Reading from Kernel Memory	Successful.	Successful.	Successful.
Writing to Kernel Memory Areas	Successful (bad)	Write prevented.	Write is prevented, exception is generated, handled by proxy.

Figure 7: Summary of test results.

the test case. The smart compiler tries to remove the out-of-bound assignment because it is not used anywhere else. We solved this problem by creating random numbers in the array assignments. As a result, it complicates the code and the compiler does not remove our out-of-bound assignment anymore.

In the second test case we create a class pointer but do not instantiate it. Then we attempt to access the member function on this pointer. Without the DD system, the buggy code resulted in an infinite loop which crashes the kernel. With our system, the raised exception is successfully handled inside our proxy and exception messages are printed out.

The third test case is to set a value in the kernel heap space. The original system passed the test without any warning messages. However, our system successfully catches this bug and generates exceptions.

The last one tries to read some value from kernel heap. It is not a bug, because driver has read-only access to kernel. This case succeeds for both the original system and our DD system. We design this case to show that we do not over protect the kernel.

Figure 7 shows a summary of the results.

Currently, we injected all the bugs manually. Due to the unavailability of fault injection tools, we were unable to inject transient bugs. This is our future work.

5.2 Performance experiment

We have tested our implementation on a real machine OMAP 1610 processor from Texas Instruments with 48 MHz CPU. One important experiment is to measure the domain switch overhead. We measured the overhead of switching from kernel and then switching back in 1000 iterations. The overhead is 0.828 second. Therefore, in one iteration, the overhead is 0.828 ms.

We consider the overhead large in the current implementation. However, it is partially because the implementation of our system on QEMU emulator got *data handler abort* exceptions on the real ARM machine due to the difference in the Choices for Integrator and OMAP. If this problem can be fixed, the overhead can be reduced greatly. Also we describe the future improvement for performance in our future work.

5.3 Lines of code measurement

Our goal is to improve the existing operating systems. We would like to show that the changes applied to the driver and the kernel codes are minimal. Figure 8 shows the ratio of lines of codes for four categories: the changes in the kernel, the code size of driver proxies, the code size of the domain manager and the code size of the XPC. Notice that we do not change the driver at all. That is why it is not shown in this figure. The change to the kernel is also very small: only one line of change.

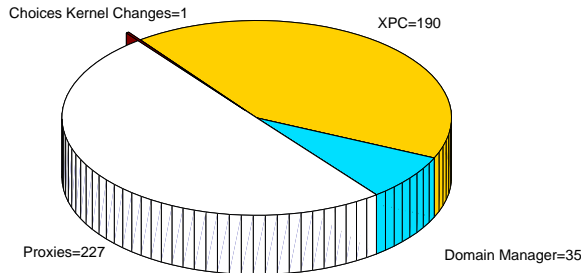


Figure 8: Defensive Driving system code distribution

Figure 8 also shows the lines of codes for our key components. It is a very small system. The codes in total are less than 1000 lines. Although the codes for driver proxies may increase when we introduce more drivers, these type of code can be automatically generated in the future. The domain manager and XPC parts remain small so that they are easy to understand and maintain.

6. COMPARISON WITH NOOKS

We implemented our Defensive Driving system with Nooks[17] as a prototype. However, we believe our system has several differences from Nooks.

First, our target architectures are different. Nooks runs on the Intel x86 architecture. Our target architecture is ARM, lying at the heart of advanced digital products, from mobile phones and digital camera to games consoles and automotive systems. Embedded systems, based on processors like ARM, are facing high reliability demands that have never been required before. The ARM architecture requires us to also change the privilege mode in addition to the memory access mode. As a result, when switching from kernel to driver domain, the program runs in user mode and it cannot modify the kernel regions which are already marked *Readonly*.

Second, Nooks is in Linux kernel, while our system is built on top of Choices [4]. The object oriented nature of Choices

allows for the establishment of well-defined boundaries between components of Kernel. The concept of *Domain* in Choices made it easy for us to create the protection domain. Moreover, The proxy functions are written as the subclass of drivers and are naturally invoked when we replaced the driver object with its proxy object. It leads to a clean and easy-to-maintain design for our system. Differently, Nooks modified the standard module loader to bind extensions to wrappers instead of kernel functions when the extensions are loaded.

Third, Nooks uses Linux style of signal handling which only provides error number. Our system utilizes C++ style exceptions in OS which allow for more robust error handling. For example, we can incorporate the information of the stack pointer, PC pointer into the exception object. These data are useful in the recovery process.

7. FUTURE WORKS

We have identified several additions to the base architecture that we have designed. Specifically, these changes are in the domain of recovery and compiler-assisted domain switch.

7.1 Recovery

Recovery is the eventual goal of our project. We plan to achieve this in several ways, based on the type of errors.

- *Function re-try* - If a crash occurs in the driver function, we can simply try to re-execute the function, for a threshold number of times. *Transient errors* may be eliminated during the second run.
- *Process restart* - A more sophisticated scheme is to restart the process from the last *checkpoint*.
- *Process kill* - If the function re-try or process re-start fails even after the threshold, we can conclude that the process has some permanent errors, kill the process and schedule another one.

While recovery, care must be taken to ensure the release of resources held by the process.

7.2 Compiler assisted domain switch

The domain switch is indeed a costly process. A scheme to generate customized proxies, proxies whose domain switch procedure depends on the nature of the real driver call, will be another worthy addition. For example if a driver function operates only on its local variables or only reads kernel region, it might be unnecessary to perform the entire domain switch sequence. Using compiler analysis it should be possible to classify the driver function as *read-only* or *read write* or *only local* based on whether it only reads kernel data structures or reads and writes to kernel data structures or never accesses a kernel data structure. The compiler analysis will involve *inter-procedural alias analysis*, which is a costly proposition. However, we can adopt an approximate in-accurate strategy (eg. *intra-procedural analysis*), to cover at-least some cases, if not all.

8. CONCLUSION

Device drivers have high bug frequency and lead to frequent OS crashes. The reliability of an OS is largely dependent

on its immunity to device driver induced crashes. In this project, we strive to contain a device driver bug within an isolated environment that resides inside the kernel address. These environments, named protection domains, are associated with each driver. During a kernel to driver call, the current domain is switched to that of the driver domain. In the driver domain, the kernel memory is given only read-only access. Also, each domain has its own stack and heap. Additionally, the processor is switched to user-level. The original calls to device drivers are replaced by proxy calls and the bug in a driver code is caught by exceptions handled within the proxy. Such a scheme can be implemented with minimal changes to the existing kernel and driver code. Using the scheme several bugs, like stack overflow, null pointer access etc, were caught and handled successfully. We also have identified several future directions.

9. ACKNOWLEDGMENT

We would like to thank Francis David, Ellick Chan, and Jeffrey Carlyle in Choices group for their valuable help and insightful suggestions. We also would like to thank our TAs Jacob Biehl and Alexander Sorokin for their hardworking and timely response.

10. REFERENCES

- [1] Qemu technical documentation.
- [2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, 1995.
- [3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, 1995.
- [4] Roy H. Campbell, Nayeem Islam, Ralph Johnson, Panos Kougiouris, and Peter Madany. Choices, frameworks and refinement. In *1991 International Workshop on Object Orientation in Operating Systems*, pages 9–15, 1991.
- [5] J. Chapin, M. Rosenblum, and T. Lahiri S. Devine. Hive: Fault containment for shared-memory multiprocessors. In *In Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 12–25, December 1995.
- [6] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An empirical study of operating system errors. In *Symposium on Operating Systems Principles*, pages 73–88, 2001.
- [7] Dawson R. Engler, M. Frans Kaashoek, and James O’Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.
- [8] Dawson R. Engler, M. Frans Kaashoek, and James O’Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.
- [9] R. S. Fabry. Capability-based addressing. *Commun. ACM*, 17(7):403–412, 1974.
- [10] A. Forin, D. Golub, and B. Bershad. An i/o system for mach. In *Usenix Mach Symposium*, pages 163–176, 1991.
- [11] J. Liedtke. On M-kernel construction. In *In Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, December 1995.
- [12] Scott McPeak. Elkhound and elsa.
- [13] Microsoft. Windows xp device driver development kit. Technical report, Microsoft Corporation, October 2001.
- [14] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, 1996.
- [15] Takahiro Shinagawa, Kenji Kono, and Takashi Masuda. Exploiting segmentation mechanism for protecting against malicious mobile code. Technical Report 00-02, Department of Information Science, Faculty of Science, University of Tokyo, 2000.
- [16] M. Swift, B. Bershad, and H. Levy. Nooks: an architecture for reliable device drivers., 2002.
- [17] M. Swift, B. Bershad, and H. Levy. Improving the reliability of commodity operating systems, 2003.
- [18] Project UDI. Introduction to udi version 1.0. technical report, project udi, Aug. 1999.