1.0 Abstract
In recent times, there is a big explosion of embedded devices like PDAs and digital cameras in the consumer market arena. In order to incorporate these new devices into the computer systems a large number of device drivers have been written by different vendors. One way to define device drivers is to view them as an extension to the operating system kernel. Today the preferred way to add features to the operating system kernel is extending the kernel by loading unsafe object code and linking it directly to the kernel. In a recent study of FreeBSD operating system, new device drivers account for 38% of the growth [4]. According to recent studies the largest problem for OS reliability are the device drivers. According to a study at Stanford University, Linux drivers have 3 to 7 times as many bugs as the rest of the kernel [1]. The purpose of this research is to compare and contrast two new techniques named as Nook’s Architecture and Mondrian Memory Protection system and assess their strengths and weakness in solving the device driver problem.

2.0 Introduction
This paper describes the two new memory protection schemes, Nooks Architecture and Mondrian Memory Protection scheme that were recently envisioned and perform a comparison between them to find their relative efficacy in providing reliability to the modern operating system against the explosion of device drivers. Computer Memory is an integral part of the modern operating system that must be carefully managed and protected. In recent times, there is an enormous growth in processor performance and it is becoming more and clearer that reliability rather than performance is now the greatest challenge for modern computer operating system.

Recently two new approaches have been introduced to deal with the device driver reliability concerns and obtaining high reliability for the modern operating systems. [4, 5, 7] These are “Nooks: architecture for Reliable Device Drivers” and “Mondrian Memory Protection”. [7] Nooks architecture basically allows device drivers to run in an isolated and protected environment where a faulty driver can not crash the whole operating system and prevents it from functioning. It is mostly software oriented and uses the existing hardware features, i.e. no hardware change is required.[4,5] Mondrian Memory Protection is a hardware memory protection scheme that is fine provides permission granularity down to single 32-bit words. MMP can enforce the existing boundaries between device drivers and the core of kernel prohibiting the device drivers from corrupting the memory.

3.0 Techniques
This section will point out different software and hardware memory protection techniques.

4.0 Hardware Techniques
4.1 Page Based Implementation
In 1961, Fotheringham [10] devised a way to divide large programs into smaller ones through a virtual memory process. The main idea behind virtual memory is that the whole program with its data structure and stack can exceed the physical limit of the memory. Operating system will only keep the used parts in the physical memory at any arbitrary time. Virtual memory is implemented through paging in most modern operating systems. Linear addressing is used in modern operating systems where every user process has its own linear virtual addressed space. All the threads running under the single address space possessed a single protection domain.

4.2 Capabilities Based Systems
Capabilities systems [6] came into prominence because of the need for better ways for information sharing, communication, protection and sharing between different processes. The demands of modern application really cause the existing hardware architecture to look at these problems. Capabilities based addressing is one of the architectural designs in the direction to solve the above problems.

One of the disadvantages of the capabilities systems is they would require additional space than conventional pointers because, other than carrying the permission, they carry other addressing information.

4.3 Segmentation
Segmentation is the solution to solve the one dimensional nature of the virtual memory. It consists of a linear sequence of addresses and each segment can possess a different length. Its length can change dynamically during execution. Since each segment constitutes of a separate address space, they can grow or shrink independently. Segments provide a very useful approach for sharing the data between different procedures.

1 Domain is identified as a set of pairs consisting of objects and rights. Each object is combined with a right (permission) list to perform different operations.
One of the drawbacks of segmented addressing is that hardware details get exposed to the programmers. A classic case can be the use of a near pointer (within the segment) or a far pointer (outside the segment) in 16 bit code segments while making a procedure call. Also, if the segments grow extremely large it is very inconvenient to keep them in memory. In that scenario the idea of paging comes into existence, i.e. only keep those segments that are needed. MULTICS\textsuperscript{2} \cite{10} was the first system to actually implement this technique of paging and segmentation.

5.0 Software Techniques
This section will point out some of the software techniques like Software Fault Isolation, and Light Weight Remote Procedure Call.

5.1 Software Fault Isolation
In order to provide software isolation between the cooperating software modules or numerous OS extensions, they can be placed in their own address space with different protection access. One of the drawbacks of the approach is that it can incur prohibitive context switching that can be very expensive.

One of the drawbacks of this approach is that it requires system services like files that are not available for use in OS kernel or in some embedded system environments making it unsuitable to use.

5.2 Lightweight Remote Procedure Call
The lightweight remote procedure call \cite{2} is designed to provide a sage communication facility between protection domains on the same machine. It is based on software to enforce protected calling between different protection domains. One of the major drawbacks of LRPC is that it relies heavily on data copying during the protected calling domain that can be of very high cost depending on different work loads.

6.0 Nook’s Architecture
Nook’s Architecture design relies on two core principles: It should prevent and recover from most but not all OS extension. It assumes that all OS extensions are trusted against misbehavior but not in terms of their design.

From the first principle it is clear that Nook’s Architecture is not designed for all possible OS extensions and in the second principle its design makes a distinction between “unprotected” and “safe extensions” i.e., it is trusting OS extensions not to be malicious.

Three major goals of Nook’s Architecture:

1) Provide Isolation to kernel against failed OS extensions

2) Provide Recovery in case of failure

3) Provide backward compatibility so that the architecture should support existing systems and extension without compromising the performance of the system.

\textsuperscript{2} MULTICS first ran on the Honeywell 6000 machines providing each program with a virtual memory of up to $2^{18}$ segments with potential length of 65,536 words long.

In order to achieve the above mentioned three major goals, Nook’s architecture design has a reliability layer inserted between the kernel and the OS extensions. The main feature of the reliability layer is to provide a transparent and a reliable way for the OS extensions to work with the kernel. The reliability layer is also termed as the Nook’s Isolation Manager which is further divided into four different categories:

- Isolation
- Interposition
- Object Tracking
- Recovery

Figure 1 provides a pictorial view of the communication between the kernel and the OS extensions.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{OS_Communications.png}
\caption{OS Communications}
\end{figure}

6.1 Isolation
The main feature of the Isolation layer is to prevent kernel damage from OS extensions. Under Nook’s Architecture each extension runs under its own “Light weight Kernel Protection domain” which has the same processor privilege as the kernel but a limited write access to the kernel memory. The complex communication system between the kernel and its extension is handled through XPC (extension procedure call) which is similar to lightweight remote procedure call.

6.2 Interposition
The main function of Nook’s Interposition layer is to make the interaction between the different extensions and kernel transparent. It will also ensure that the communication channel between the different extensions and the kernel is through XPC.

6.3 Object Tracking
The main function of this layer is to manage the use of kernel objects by extensions. It keeps tracks of the resources an extension is using during its execution. This functionality comes in handy during recovery operation or when the extension fails so that Nook’s can free up the resources the failed extension is using.

6.4 Recovery
This design function will recover from the effects of various faulty extensions. Nook’s detects the faulty extension either by examining the invocation of a kernel
service (i.e., invalid arguments) or by examining the resource consumption by the extension (i.e., too many resources consumed).

7.0 Mondrian Memory Protection
Mondrian Memory Architecture is based on providing a fine grained memory protection scheme (granularity to the word instead to page, as most modern operating systems provide) with the ability to allow multiple protection domains that can communicate with each other. It also has the ability to easily export the protected service across different protection domains. The core principle of MMP (Mondrian Memory Protection Scheme) is to implement the following three requirements

Different: Each protection domain residing in the same memory region can possess different permissions.

Small: Most of the modern operating systems implement page based [10] memory protection schemes which can have significant disadvantages when used for protected sharing. In order to share the pointer based data structures they have to reside in the same memory region and must have the same virtual addresses for all the processes.

Revoke: A memory protection scheme should have the ability to revoke the permission of any other domain that it has issued for that memory.

The overall structure of an MMP system contains of three main features that includes a Permission Table which resides in main memory, a PLB (permission look aside buffer) which caches permission table entries and a Sidecar register that caches the last table segment to further improve performance.

Permission Table: It is very much synonymous to the permission part of the page table [10], but instead of keeping the permission for the pages, permissions are kept for words in an MMP system. Each protected domain has the permission table associated with it. It is stored in privilege memory and it specifies the permission that domain possess for each address.

Permission Look Aside Buffer (PLB): During a memory access in the MMP system, it should be checked in order to find out the permission access for that domain. In order to increase the performance of the MMP system, a permission look aside buffer very similar to the Transition look aside buffer [10] is also provided.

Sidecar Register: An additional Sidecar register is also included in order to further improve the performance. It will cache the last table segment cached through this address register.

8.0 Methodology
Given below is the methodology which was used to conduct this research.

8.1 Metrics
The goal of this research is to present a case study in comparing and contrasting Nook’s Architecture and the Mondrian Memory Protection system for solving the device driver problem. A framework of three different criteria’s has been created keeping in mind the problem these two approaches are trying to solve and also the practicality of their solution. Both Nook’s Architecture and Mondrian Memory Protection systems are being compared against the background of the device driver problem, therefore one of the metrics is going to compare their ability to prevent OS kernel failure against faulty device drivers. Also, since these two new techniques are going to be implemented in current operating systems, implementation is one of the other criteria’s against which these two techniques will be compared. Finally, these two new techniques are going to introduce new functionality, therefore it is important to compare these two techniques against the performance background, i.e., the additional overhead while executing these two techniques.

8.2 Protection Mechanism
Protection mechanism refers the ability of both these techniques to safeguard the kernel against the device drivers. In order to evaluate this metric three different criteria are taken into consideration.

Criterion 1
This criterion will compare these two techniques against the faulty device drivers i.e. whether these two techniques provide protection against device driver extension failure. One of the important factors that will be taken into consideration while evaluating this criteria is whether they just prevent extension failure or they also recover from them.

Criterion 2
This criterion will compare these two techniques against the malicious device i.e. the drivers whose intent is to corrupt the kernel rather than there being any fault in their design or implementation

Criterion 3
There are many other extensions other than only device drivers that can be plugged into OS kernel. These criteria will take into consideration those miscellaneous extensions and how these two techniques come into play while preventing the OS kernel from their failures.

8.3 Implementation
Implementation refers to the ability of these two techniques to be implemented into different operating systems. Since these techniques are only a small part of the whole system architecture, therefore for their practical success it is essential that these two techniques can be easily incorporated into some of the most popular system architectures

Criterion 1
In this criterion, both Nooks and Mondrian Memory Protection system will be evaluated against the software changes that are needed in the existing architecture to implement their technique.

Criterion 2
In this criterion, both Nooks and Mondrian Memory Protection system will be evaluated against the hardware changes that are needed in the existing architectures that are needed to implement their technique.

8.4 Performance
Both Nook’s Architecture and Mondrian Memory Protection scheme are adding additional code and complexity into the system, therefore performance is one
of the other metrics against which these two techniques are going to be evaluated. One of the other things that will be compared against the performance metrics is their efficiency to catch the device driver failures before and after the implementation of their architecture. Two different criteria’s that are going to be used to evaluate the performance metrics are given below:

**Criterion 1**
In this criterion, CPU over head will be taken into consideration for both Nook’s and Mondrian Architecture i.e. how much additional CPU time will be taken to complete the given device driver operation. In other words, execution time for the device drivers will be compared for both these techniques.

**Criterion 2**
In this criterion, the efficiency of both the techniques in catching the number of device driver failures that would not be caught before the implementation of their architecture will be compared, i.e., what is the success ratio of both the techniques in catching the device driver failures in a given system?

**8.5 Measurement**
Each criterion from all the three different metrics is either given a “POSITIVE”, “NEUTRAL” or “NEGATIVE” rating. For example, for criteria 1 under the protection mechanism if either Nook’s or Mondrian Architecture demonstrates that they are not a viable solution for the faulty device drivers they will be given a Negative rating, on the flip side of the same situation if they do demonstrate they are efficient enough in detecting the faulty device drivers and protecting the kernel from extension then both of these techniques will be give a positive rating.

**9.0 Data Gathering**
Given below are the results gathered for all the three metrics.

**9.1 Protection Mechanism**
Nooks Architecture is able to detect and automatically recover from over 99% of extension faults that would otherwise crash the Linux Operating system [7]. One of the other nice features of the Nooks Architecture is the recoverability aspect of its architecture, i.e. it supports automatic recovery from a failure extension so that the application can keep executing. [5] MMP hardware is implemented in a modified version of 2.4.19 operating system and it was able to enforce the module boundaries, only letting device drivers to access the memory they need to function. Isolation provided by MMP system helped in increasing Linux resistance to programmatic errors and also exposed two kernel bugs in heavily tested common drivers.

[7] Nooks Architecture can be easily circumvented by malicious code working within the kernel. It is intended to recover from device drivers that are faulty in design but not malicious in nature. [5] MMP is very similar in nature as it is designed to find bugs in other services and not malicious code. It is not capable of providing any safeguard against denial of service attack or resource exhaustion. It is designed more in the mould of a permission system rather than a security system. [7] Nooks solution is not intended to cover all types of extensions whereas the MMP system is generic enough to extend to any types of extension, and also to a variety of safe user extensions and other web applications like data watch points and garbage collectors.

**9.2 Implementation**
[7] One of the main goals of Nook’s Architecture is to support backward compatibility i.e. the architecture must apply to existing systems and extensions, with minimal changes to either. It relies heavily on the conventional processor architecture, uses C as its programming language and is built upon a conventional operating system and existing extensions. Nook’s architecture basically composed of 22,000 lines of code in contrast to the Linux kernel that has 2.4 million lines of code. [7] Table 1 shows the code distribution

<table>
<thead>
<tr>
<th>Source Components</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Management</td>
<td>1,882</td>
</tr>
<tr>
<td>Object tracking</td>
<td>1,454</td>
</tr>
<tr>
<td>Extension Procedure Call</td>
<td>770</td>
</tr>
<tr>
<td>Wrappers</td>
<td>14,396</td>
</tr>
<tr>
<td>Recovery</td>
<td>1,136</td>
</tr>
<tr>
<td>Linux Kernel Changes</td>
<td>924</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>2,074</td>
</tr>
<tr>
<td><strong>Total number of lines of code</strong></td>
<td><strong>22,266</strong></td>
</tr>
</tbody>
</table>

Nook’s Architecture does not rely too heavily on hardware support. It is built upon existing hardware support whereas the MMP system is the exact opposite. It is more of a hardware technique. It introduces the concept of Protection Page Table, Protection Look Aside buffer and Side Car register. Since MMP support is more hardware oriented it will be faster in execution whereas Nook’s Architecture is more software oriented it might be easy to implement. MMP requires modification to a processor pipeline that can be difficult to implement in current systems.

**9.3 Performance**
For Nook’s Architecture, Linux was the main operating system where they have tested their performance benchmarks. Since Nook’s Architecture adds more code to the existing system architecture compared to native Linux, there is more code to run but one of the interesting revelations is that the existing code runs slowly when Nook’s Architecture is implemented. It has been noticed that under the Linux environment “other kernel” components slow down with Nook’s. The reason is attributed to the TLB being flushed during each XPC (extension procedure call). Using the Pentium 4 performance counters it has been noticed that the 167 kernel-mode seconds spent in Nook’s, 127 are spend handling the TLB while under native Linux only 5.5 seconds were spent during the TLB miss-handling. This is a big performance drawback under the Nook’s Architecture. Also it has been noticed that a given system such as the kHTTPd web server that traditionally perform many XPCs will be a poor Nooks application because of the XPC workload and its interaction with TLB.

In the case of the MMP system, performance hits are not bad because of its hardware implementation. It doesn’t
rely on software to carry out its protection architecture. It is largely language independent and brings increased memory safety to large, legacy systems. In extensive tests over a large number of OS extensive workloads it has added less than 12% to the execution cycle. [5] Under the MMP system, if an application uses coarse-grained protection, MMP’s consumed less space (<1% of the application memory usage) and only adds about <1% of the application memory references. When MMP is used for fine-grained protection scheme this number can increase up to 8% of the application Memory usage and reference.

[7] With the help of automatic fault injection Nook’s architecture was able capture more than 99% of the errors that cause Linux to crash but it is designed to capture all types of user extensions. MMP system on the other hand is generic enough to capture all types of extension failures.

10.0 Data Analysis
Given below are the tables that are formed during the data gathering portion

Table 2. Protection Mechanism Metric

<table>
<thead>
<tr>
<th>Protection Mechanism</th>
<th>Criteria 1</th>
<th>Criteria 2</th>
<th>Criteria 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nook’s Architecture</td>
<td>Positive</td>
<td>Negative</td>
<td>Neutral</td>
</tr>
<tr>
<td>MMP System</td>
<td>Positive</td>
<td>Negative</td>
<td>Positive</td>
</tr>
</tbody>
</table>

Table 3. Implementation Metric

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Criteria 1</th>
<th>Criteria 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nook’s Architecture</td>
<td>Positive</td>
<td>Positive</td>
</tr>
<tr>
<td>MMP System</td>
<td>Positive</td>
<td>Positive</td>
</tr>
</tbody>
</table>

Table 4. Performance Metric

<table>
<thead>
<tr>
<th>Performance</th>
<th>Criteria 1</th>
<th>Criteria 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nook’s Architecture</td>
<td>Neutral</td>
<td>Positive</td>
</tr>
<tr>
<td>MMP System</td>
<td>Positive</td>
<td>Positive</td>
</tr>
</tbody>
</table>

Table 2 represents the data gathered for the protection mechanism metric. From the table it is clear both these approaches are novel for device driver failures. MMP is more general in approach, i.e. it can be applied to all sorts of user extensions whereas Nook’s architecture is specially designed for OS extension failures. Also, both these approaches are not intended to protect against malicious drivers. These approaches are more geared as a protection and permission mechanism approach rather than a security system.

Table 3 represents the data gathered for the implementation metric. The key factor where these approaches differ is that Nook’s Architecture is a software driven approach relying heavily on the principle of backward compatibility whereas the MMP system is a hardware approach introducing new concepts in the existing hardware architecture like PLB (protection look aside buffer) similar to TLB (transition look aside buffer) and protection table similar to page table in the modern operating system.

On the performance front Nook’s Architecture is a little less efficient (Table 4). It has fared badly against the applications that are XPC (extension procedure call) intensive and also under the Linux environment other kernel components take more time executing under Nook’s Architecture. Also, in case of Nook’s Architecture if the CPU is already running under heavy load then the performance hit will be greater, if not, the penalty may not be important. In case of the MMP system, the performance hit is not that severe. Even if the application is using fine-grained memory protection scheme only 8% of additional CPU time cost will be incurred.

11.0 Conclusion and Future Work

Nook’s Architecture and MMP system are both novel ways of solving the device driver problem. The former is more geared towards just solving the device driver problem whereas the latter is more generic and can work with most of the OS extensions. One of the nice features of both the techniques is that they are backward compatible and can work with existing hardware and software infrastructure. MMP can work with most language interfaces whereas Nook’s Architecture is more language specific (mostly C). One of the nice features that Nook’s architecture possesses is that the ability to recover kernel objects and reclaim kernel resources. The MMP system lacks this feature and therefore one of the future works might be to implement a recoverability feature at the coarse granularity level.

One of the future works that can be performed is to extend these techniques towards protecting the OS kernel against malicious drivers. In recent times, there has been an upsurge in writing malicious code to attack vulnerable computer systems. In modern architecture, plug and play architecture is becoming such a common thing it won’t be far fetched that somebody will write a malicious driver that can disrupt the computer system. Therefore, in the future these techniques can be advanced to incorporate a security feature for malicious drivers like the security policies for the drivers and their interaction with the OS kernel.

8.0 References


