Using Independent Auditors as Intrusion Detection Systems *

Jesus Molina and William Arbaugh

Department of Computer Science
University of Maryland
20742 College Park, MD
{chus,waa}@cs.umd.edu

Abstract. A basic method in computer security is to perform integrity checks on the file system to detect the installation of malicious programs, or the modification of sensitive files. Integrity tools to date rely on the operating system to function correctly, so once the operating system is compromised even a novice attacker can easily defeat these tools. A novel way to overcome this problem is the use of an independent auditor, which uses an out-of-band verification process that does not depend on the underlying operating system. In this paper we present a definition of independent auditors and a specific implementation of an independent auditor using an embedded system attached to the PCI bus.

1 Introduction

Computer systems have been made increasingly secure over the past decades. However, new attacks and the spread of harmful viruses have shown that better methods must be used. One approach gaining increasing popularity in the computer community is to use Intrusion Detection Systems (IDSs).

Intrusion Detection Systems identify attacks against a system or users performing illegitimate actions. Using a common analogy, having an Intrusion Detection System is like having a "burglar alarm" in your house. The alarm will not prevent the burglar from breaking into your house, but it will detect and warn you of the problem. Following the publication of the first research in Intrusion Detection Systems, a large number of diverse applications have been developed. One method of accomplishing this type of detection is the use of file system integrity tools. When a system is compromised, an attacker will often alter certain key files to provide continued access and to prevent detection. The changes could target any portion of the system software, e.g. the kernel, libraries, log files, or other sensitive files. File system integrity checkers detect those changes and trigger a corresponding alert. To guarantee the integrity of the file system, two approaches can be followed.

The first approach is to create a secure database, which is usually composed of hashes. The stored hash will be periodically checked against a newly computed hash. This method is used with tools such as Tripwire [1], Aide [2], and others.

* This work funded in part by DARPA grant #F306020120535

The second, more recent approach is to create digital signatures of sensitive data, such as executable files using asymmetric cryptography, and use these signatures to check the integrity of the signed file ([3], [4]).

Both approaches have advantages and drawbacks, but they share a common flaw: the auditing relies on the validity of the operating system. All the previous applications have made the assumption that the OS itself is not corrupted. Once the operating system is compromised the intruder can easily defeat integrity tools. As an example, in the Linux operating system, redirecting system calls using kernel modules can potentially compromise the system.

Also, since the binary of the Integrity Tool resides in the machine to be audited, the attacker may be able to corrupt the binary or the configuration files of the tool.

This work develops a novel way to overcome the problems of traditional Integrity Tools. Our approach is to use an independent auditor, i.e. a completely standalone and independent device, potentially tamper resistant, to perform the integrity detection checks.

2 Motivation

An Integrity Verification Tool that relies on the operating system of a penetrated machine can be easily deceived by corrupting the kernel. In fact, this problem is well known. In an article by "Halflife" [5], a loadable kernel module was used to bypass the Tripwire integrity checking System. Since then, several tools for corrupting the operating system have been developed including Knark, famous for being used in the Ramen worm [6]. This section will explain the mechanics of these attacks. Although the attacks discussed here occurred on Unix-like operating systems, all operating systems are vulnerable to these kinds of attacks.

2.1 System Calls

User processes and the kernel run in different modes. The CPU itself enforces this policy. Every modern processor has at least two modes of operations, and in some cases, as in the x86, more than two. Every mode of operation allows some actions and does not permit others. In the case of the Unix operating system, only two levels are used: the lower, called user space or protected mode and the higher mode, called kernel space or supervisor mode. In this mode the process has unrestricted access to memory and devices. User-space applications are run in protected mode, while the kernel is executed in the supervisor mode. The only way an application will be able to access the sources restricted by the protected mode is through the kernel. If an application requests a service from the kernel, such as asking for more memory or accessing a hardware device, system calls are used to access the second mode of operation. These system calls, along with an interrupt reaching to the system, are the only ways to access kernel space.

In order to use system calls, the process will fill certain registers with appropriate values, including the type of system call to access, and then call a defined

interrupt, dependent on the operating system and architecture. For example, in the Intel architecture the user process will call interrupt 0x80 if the operating system is Linux or interrupt 0x21 if the operating system is Windows. Then, depending on the system call used, the process will jump to a certain location of the kernel. The location in Linux is stored in a table `(sys_call_table)`, where the addresses of the functions in the kernel are stored. The kernel will look at this table and jump to the corresponding address. After it returns from the call the kernel will do some system checks and continue in the address of the user space calling process.

2.2 Attacks in kernel space

In this section, we explore kernel attacks specific to the Linux operating system. Similar attacks could be launched in other Unix-like operating systems. The most straightforward way of changing the kernel is to replace the kernel binary itself. The kernel binary is usually placed in the /boot partition, so an attacker could compile his/her own version of the kernel and replace the binary. Some operating systems make this more difficult now, but the attack remains feasible on several current operating systems. Another possibility an attacker has is to use Loadable kernel Modules (LKMs). LKMs are a feature of Unix-like operating systems which allow dynamic changes to components of the kernel. An attacker will not have to recompile the complete kernel, but rather just code a LKM which can be loaded at any time and become part of the kernel.

Once the intruder has gained access to the kernel space, several attacks could be launched against the system so as to remain undetected. The most obvious attack is to redirect the system calls. Any program in user space such as Integrity Tools will use system calls to access kernel space, even for very simple operation such as reading a file. By redirecting the system call to a "rogue" routine system call the attacker can hide the existence of any file in the system even from integrity checkers. Redirecting a system call using kernel modules is simple. As we have seen, the addresses where systems call will jump to when loaded are stored in a table. When the module is initialized, the kernel module will use code similar to the following:

```
ori_syscall=sys_call_table[SYS_sycall]
sys_call_table[SYS_syscall]=hacked_syscall
```

where hacked_syscall is a pointer to the function used to replace the system call. In the function hacked_syscall the attacker will call the original syscall and then change the results. For example, in Figure 1 the IDS never sees the file /foo/evilbinary because the system call filter eliminates it from the results.

```
res=(*ori_syscall)(parameters)
//change res to mislead the system
return(res)
```

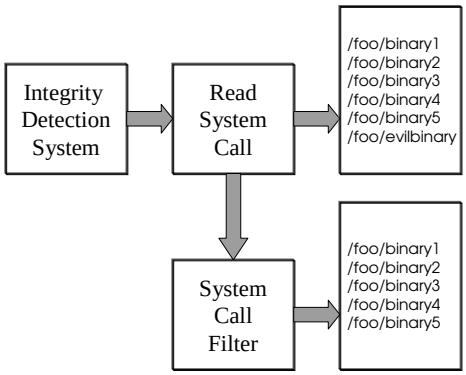

Fig. 1. Redirecting System Calls

Several Rootkits (a set of tools that an attacker uses to mask an intrusion and/or regain access later) take advantage of kernel modules. Other operating systems, such as Windows NT or 2000, may also be targets for these attacks, by using malicious system patches to the system or corrupted drivers. Some efforts have been made to counter the loading of kernel modules. Most of these techniques, as [7] in Windows NT, operate by restricting modules and drivers to be loaded or by using vigilant modules. However, the former creates a lack of flexibility that is usually not reasonable for most systems and the later tends to be a "chicken and egg" solution, as the attacker could modify his/her module to be loaded before the checking module, i.e. during the bootstrap.

3 Definition of Independent auditors

In the present and following sections the terms "host processor" or "host system" will be used to define the machine or set of machines to be verified for file system integrity. The term "host" is slightly inaccurate. However, as in this work, the out-of-band verification system is implemented as an embedded coprocessor, and the term host processor will be used in order to avoid confusion. Other terms, such as host operating system, will be used throughout the text to refer to components of the system to be verified.

3.1 Properties

Machine A is an out-of-band auditor or independent auditor of Machine B if it accomplishes the following set of properties

1. *Unrestricted access*: Machine A must have unrestricted access to the internal devices of machine B to be verified or needed for the verification, including

peripherals, hard disks and interrupts. Notice, however, that unrestricted write access, i.e. without mutual exclusion, to the internal components of the host system could lead to an unstable system.
2. *Secure transactions*: The channel used by the independent auditor to retrieve the data should be a secure channel, meaning a channel which cannot be eavesdropped or intercepted, nor modified.
3. *Inaccessibility*: Machine B must not have access in any way to the internal components of machine A, including memory and internal interrupts.
4. *Continuity*: Machine A must run immediately after machine B has setup the internal devices and is in a known trusted state. After this moment, Machine A must run continuously, independently of the behavior of machine B. Notice that power failures or hardware reboots should be the only way to restart machine A and must be labeled as high risk level alerts.
5. *Transparency*: The access to the internal devices should be transparent to the host system. However, concurrent access to the devices will probably occur unless mutual exclusion is provided. In these situations, the consequences to the host system should be minimized.
6. *Verifiable software*: All the code running in machine A must be trusted and verifiable. This, at least, implies that all running software in machine A must have the source code available. This includes the firmware, operating system and user space programs in machine A.
7. *Non-volatile Memory*: Machine A must be capable of retaining a record of the alerts even in the event of a power failure or reboot. Hence, machine A should have some non-volatile storage to record sensitive data.
8. *Physically secure*: Machine A should be physically secure.

3.2 Modes of Operation

The independent auditor has three different possible states. The running state is the normal mode of operation and is dependent on the method used for the integrity verification. The second state is the alarm state which is reached if an alarm is triggered in the running state. A final mode, which can be accessed only at boot time, is the management mode. This mode is only accessible through a set of secure mechanisms, and allows the administrator to change parameters in the secure coprocessor. An independent auditor is hence not vulnerable to API level attacks as described by Bond & Anderson in [8], as there is no interface to the independent auditor from the protected host. The running mode of operation could follow different methods to ensure the integrity of the data in the host machine, but the implementation described in section 4 uses a database to perform the integrity checking. All of these modes of operation assume that the host operating system is in a trusted state when the first checking takes place.

3.3 Audit logs

Information pertaining to the alarm should be stored in a non-volatile storage device in the event of an alert. Using an independent auditor for integrity also

creates an opportunity to not only store the information of the attack but also information before the attack. This is useful, as the auditor is not checking the system in real time. The auditor could log processes, measurements or events. The auditor stores these sensitive logs in a trusted state. Every check without an alarm will ensure that the system remains in this trusted state. Hence, if the system audit occurs without an alarm, the auditor will update the data. In the event of an alarm, the data before the system compromise took place will be preserved, allowing the supervisor to retrieve the logs before the attack took place, in a trusted state. The recorded file could be compared to the files in the compromised machine to discover if the attacker has tampered with the logs and possibly uncover information about the type of attack and identity of the intruder. A discussion of the importance of secure audit logs can be found in [9].

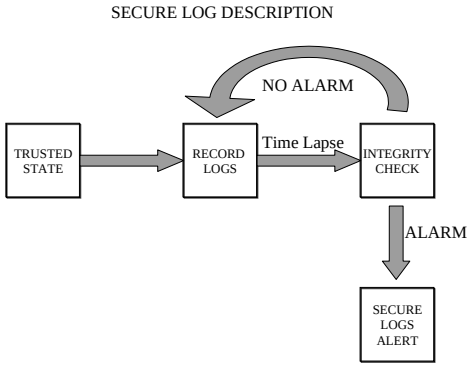

Fig. 2. Secure Logs Mechanism

3.4 Independent auditor using a database mode

This method does not differ much from its software counterpart, but as we see in Figure 3, all sensitive actions are moved inside the auditor and are therefore protected by the strong separation between the protected host and the auditor. The independent auditor will have a policy file, which is uploaded into the system in management mode, where the files to be checked will be declared along with the parameters to be verified. The files will be accessed periodically, and the set of actions stated by the policy file will occur. The independent auditor will retrieve the file's information, possibly computing its hash function. This information will then be checked against the locally stored information inside the auditor. If the information matches, the file has not been compromised. If it does not match, the alarm state will be triggered.

Using an independent auditor in database mode has several advantages as compared to its counterpart which is managed by the host operating system.

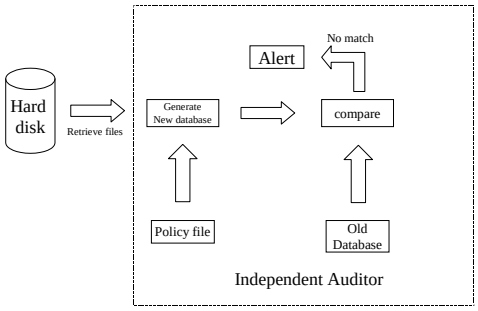

Fig. 3. Database Mode using an independent auditor

- The auditor handles the computational work. Hence, the performance of the host system is not degraded.
- The system is not vulnerable to subversion errors. Following directly from the inaccessibility property, an attacker gaining access to the host machine will not be able to corrupt the auditor.
- Secure logs can be stored, which could greatly help in the forensics of the attack.
- In a system with more than one administrator a central administrator could make sure that the local administrators have not changed sensitive files.

4 Implementing an Independent Auditor

As an example of implementing an independent auditor, we used an embedded co-processor plugged directly into the PCI bus. The co-processor used was the SA-100 ARM processor with the 21285 core logic [10]. This coprocessor comes packaged as a EBSA-285 [11]. The EBSA-285 includes support for both volatile and non-volatile storage. The non-volatile storage is 4MB of flash ROM memory, and the volatile memory is upgradeable up to 256 MB, but for our system only 16 MB were necessary. The EBSA-285 is shaped as PCI card, and runs a romized Linux kernel. The Linux OS was chosen for the system as it supports a wide range of file-systems and it had good support for the SA-110 processor. AIDE was used as the application for performing the Integrity checks.

Our objective was to create a reliable integrity tool, which could be portable to any OS. We met the first requirement. We were able to mount the partition of a Linux box and a Windows NT box, and check the integrity of the file-systems using AIDE. However, since the access to the IDE controller through the PCI bus is non-atomic, when both the host OS and the EBSA-285 write to the registers on the IDE controller at the same time, a race condition occurs. To avoid these problems, patches for the host OS must be supplied to create a mutual exclusion mechanism.

4.1 The EBSA-285 as an independent auditor

In this section the suitability of the EBSA-285 to act as an independent auditor will be discussed. An assumption of this work is that the host machine is physically secure. The term physically secure is used in this section to describe a machine whose internals can not be tampered with by an attacker. The attacker, however, could have access to the peripherals attached to the host, including keyboard and monitor.

The EBSA-285 must have open access to the hard disk data to meet the property of unrestricted access. The EBSA-285 has access to the entire PCI bus, hence is capable of reading the registers and data from all PCI devices plugged into the same bus. If the IDE controller is plugged into the same bus as the EBSA-285, the EBSA-285 will be able to read the hard disk without the intervention of the host OS. Notice that if the EBSA-285 is plugged into a slot using a different bus than the IDE controller, the access will not be possible and the EBSA-285 will not be capable of acting as an independent auditor of the host processor. The EBSA-285 is not able to "listen" to the interrupts raised by the different devices in the PCI bus. These interrupts, however are not imperative to the auditing, as a polled method can be used to read and write data to the peripherals.

The channel used to retrieve the information from the peripherals is the PCI bus. This channel is secure as it is an internal part of the computer and we have assumed that the internal parts of the host are secure. Hence, the EBSA-285 accomplishes the property of secure transactions. The EBSA-285 does not map its memory (either ROM or RAM) via the PCI bus, which allows the host processor to access only mailbox registers and doorbell registers. The EBSA-285 uses these register as information, and does not interfere with its operation, hence not breaking the inaccessibility property.

Once the EBSA-285 starts auditing only a power failure or reset of the host machine will stop it from functioning, and these events will be labeled as alarms. The EBSA-285 will begin functioning after the host machine has set up all the internal peripherals. In our case, the EBSA-285 will begin functioning before this happens, so the EBSA-285 has a mechanism to stall its booting until all the peripherals have been configured. Therefore the property of continuity is satisfied. Because the EBSA-285 has direct access to the registers of the PCI devices, it is able to access the data by polling without the supervision of the host OS, and therefore satisfying the requirement of the property of transparency. Notice, however, that some mechanism should be implemented to avoid concurrent writes to the IDE registers, which would lead to an unstable system.

The software running in the EBSA-285 is open source. It is composed of a minimum bootloader and the ARM-port of the Linux operating system, with some changes to support the EBSA-285 and the polling method.

The EBSA-285 can use the flash ROM to store the alarms and logs. The ROM normally cannot be reprogrammed if the program is executed from the flash ROM. To avoid this problem, the bootloader copies the root file-system

and the operating system to the RAM memory before executing it, freeing the flash ROM.

4.2 Solving race conditions

The EBSA-285 was able to access the IDE hard disk using a polled IDE driver instead of the usual Linux driver and was able to mount the hard disk independently of the host operating system used, as long as it was supported by the Linux operating system[12]. However, if the host operating system tried to access the hard disk at the same time as the EBSA-285, race conditions occurred, as requests to the IDE controller through the PCI bus are not atomic. The PCI specification [13] contains a method to achieve atomic transactions using the PCI bus with the LOCK# signal. This signal, however, is not used in common drivers and few motherboards support its use. While we hoped to avoid changes to the host operating system, we could not due to the lack of an atomic lock on the PCI bus.

To avoid the race conditions we used a communication path between the host processor and the independent auditor, named mailbox registers and doorbell interrupts. Using these tools will not break the inaccessibility property as the independent auditor will only use this signal as information to prevent the race conditions. The host processor is only able to modify these specific registers.

If the host accesses the hard disk, it will write a doorbell register in the EBSA-285, which will raise a sleep interrupt in the EBSA-285. If the EBSA-285 is engaged in any hard disk transaction, it will finish the current block transaction, and then send an ACK to the host using a mailbox register. After receiving the ACK the host will resume normal operation. Once the host has finished using the hard disk, it will raise a wakeup doorbell interrupt in the EBSA-285. The EBSA-285 will be then free to begin a new transaction. The transaction size depends of the mode of operation.

The behavior of both the EBSA-285 and the host processor can be summarized in the state machines shown in figures 4 and 5.

It could be argued that an attacker could easily halt the EBSA-285 by changing the driver so that the host will never send a wakeup interrupt to the EBSA-285. To combat this attack we use a counter, which will be set to the maximum transaction value and will wakeup the EBSA-285 after that time, even if the interrupt never arrives. Additionally, an alarm will be raised.

Another possible attack would be to prevent the host from sending the sleep interrupt to the EBSA-285. In this case, the EBSA-285 would proceed to read even if the host is accessing the hard disk. This could cause a corruption of the file-system. However, this attack is not only a problem with the EBSA-285, for if an intruder is able to change the drivers, obviously he/she would be able to corrupt the file-system anyway.

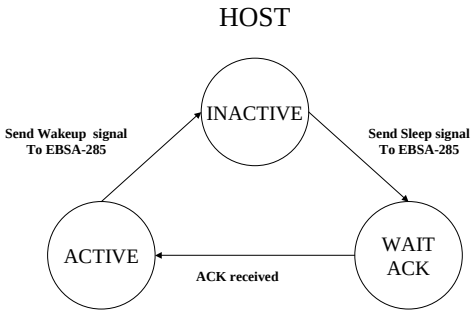

Fig. 4. Host Hard disk Access State Machine

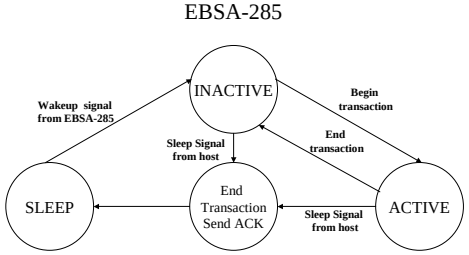

Fig. 5. EBSA-285 Hard disk Access State Machine

Table 1. Different Timings of executing AIDE on the EBSA-285 . The first column states if the host machine is accessing the hard disk concurrently with the EBSA-285. the second column the number of files and the total size of all of them. The third and fourth columns the amount of time spent in User Space and in kernel Space, respectively. The tests performed by AIDE were: permissions, inode, user, group, size and checksum using SHA1 checksum

Concurrent	File Amount and Size	User Space	System Space	Total
No	36 (7644k)	1.18s	3.69s	4.86s
Yes	36 (7644k)	1.26s	4.02s	5.03s
No	2302 (47552k)	7.12s	36.55s	43.68s
Yes	2302 (47552k)	7.06s	55.00s	62.71s
No	3484 (128736k)	17.53	75.51	93.29
Yes	3484 (128736k)	17.67	101.37	119.25

4.3 Implementation Results

To collect the timing information, the host used was an Intel Pentium III Coppermine. As stated before, the EBSA-285 uses a polled driver. The average throughput of the EBSA-285 to the hard disk using this polled driver is 1.40 Mb/s.

The performance varies greatly from one system to another, or even in the same machine: at different times the data to be checked could be stored in cache or not, the CPU could be burdened by a huge amount of processes or just a few, and so on.

In the EBSA-285, the numbers are deterministic. The EBSA-285 always bypasses the internal cache and retrieves the data directly from the hard disk at a constant rate. The CPU is always running a minimal number of process, so the CPU is 99% devoted to computing the hashes and perform the different tests. In Table 1 we can see the comparison between the two unique states we can have, the EBSA-285 executing AIDE alone and the EBSA-285 executing AIDE at the same time as the host machine is performing a hard disk access, for three different amount of files. This hard disk access is performed using a "worst scenario" approach: we read a file from the host machine bigger than the RAM memory while AIDE is running on the independent auditor, therefore the data will never be cached.

The time spent in kernel space is roughly the time the EBSA-285 spent retrieving the data. This time is directly proportional to the total size of the data to be retrieved, while the amount spent in user space is proportional to the complexity of the hash function.

The overhead of the locking mechanism to the host machine when there isn't a concurrent access with the EBSA-285 is negligible, as the only addition to the driver is a new register write through the PCI bus (writing to a Mailbox Register in the EBSA-285).

The locking mechanism was created to give priority to the host machine. When the host machine begins a hard disk access, if the EBSA-285 is accessing the hard disk at the same time, the EBSA-285 will stall until the host machine has ended the request (the size of a request depends of the mode used to access the hard disk). As a result the impact in the performance of the host machine is very low. In fact, most of the time the host machine does not access the hard disk, but reads the data stored in the internal cache. Even in the worst case scenario, where the hard disk is reading a file bigger that its memory while the EBSA-285 is performing several hard disk access, only supposes a maximum of a 5% overhead in the timing results of the host machine.

5 Conclusions

Current computer systems can not fully protect themselves against motivated attackers because of the attackers ability to change the underlying operating system once gaining system privilege. The attacker can simply change the operating system to "lie" to any security system– not only integrity detection systems.

In this paper, we proposed the notion of an independent auditor whose role is to serve as an unimpeachable reviewer of the state of the protected host. We defined the properties required for such an auditor, and we implemented such a device for measuring the integrity of a protected host. The results we achieved were excellent in that at most only five percent of overhead was added to the protected host– in the worst case.

While such a system is not required in all computer systems within an organization, the capabilities provided by an independent auditor are tremendously important to server systems. Given the low cost of these boards, $200, and the vast increase in protection they provide– organizations serious about protecting their resources should consider such protection.

6 Acknowledgments

The authors wish to thanks (Alphabetically) to Adam Agnew, Michael Barr, Jeffrey Chung, Jim Fischer, Virgil Gligor, Russel King, Arunesh Mishra and Charles Silio for the helpful feedback and comments during the course of this work.

References

[1] G. H. Kim and E. H. Spafford: The design and Implementation of TRIPWIRE: A File System integrity checker. Technical Report, TR-93-071.

[2] R. Lehti, P. Virolainen: AIDE (Advanced Intrusion Detection Environment), `Web ref: http://www.cs.tut.fi/ rammer/aide.html`

[3] S. M Beattie, Andrew P. Black, Cristing Cowan, Calton Pu, Lateef P. Yang: Cryptomark: Locking the Stable door ahead of the Trojan Horse. Technical review, Jul 2000.

[4] L. Van Doorn, G. Ballintijn and W. Arbaugh: Signed Executables for Linux. Technical review, 2000.

[5] "Halflife": Bypassing Integrity Checking Systems. Phrack, volume 7, Issue 51 , September 1997.

[6] CERT(r): em Incident Note IN-2001-01, Widespread Compromises via "ramen" Toolkit. January 18, 2001

[7] Pedestal software, Integrity Protection Driver (IPD), `http://pedestalsoftware.com/intact/ipd/`

[8] Mike Bond, Ross Anderson: API-Level Attacks on Embedded Systems. In IEEE Computer Vol. 34, No. 10 pp. 67-75, October 2001.

[9] Bruce Schneier and John Kelsey: Secure Audit Logs to Support Computer Forensics. ACM transactions on Information and System Security, Vol.2, No 2, May 1999, Pages 159-176.

[10] Intel Corporation: Datasheet, 21285 Core Logicfor the SA-110 Microprocessor. September 1998.

[11] Intel Corporation: Reference Manual, StrongARM EBSA-285 Evaluation Board. October 1998.

[12] M. Tanuan: An Introduction to the Linux operating system Architecture Web Reference `http://www.grad.math.uwaterloo.ca/ mctanuan/cs746g/LinuxCA.html`

[13] PCI Special Interest Group: PCI Local Bus Specification, Revision 2.2. December 1998.
[14] Intel Corporation: User Manual, 21555 Non-Transparent PCI-to-PCI Bridge. July 2001.
[15] Jim Fischer: CiNIC - Calpoly Intelligent NIC. EE MS Thesis, June 2001.
[16] Xilinx: Virtex-II Pro Platform FPGA Handbook. January 31 2002.