

METHODS FOR THE DETECTION OF UNIX KERNEL ROOTKITS USING  
OUTLIER ANALYSIS

By

Douglas Ray Wampler  
B.S., Indiana State University  
M.S., Ball State University

A Doctoral Dissertation Proposal  
Submitted to the Dissertation Committee  
Computer Science and Engineering  
J.B. Speed School of Engineering

Dissertation committee:

Dr. James H. Graham, Chair  
Dr. Gail W. Depuy  
Dr. Adel S. Elmaghraby  
Dr. Mehmed M. Kantardzic

Computer Engineering and Computer Science Department  
J.B. Speed School of Engineering  
University of Louisville  
Louisville, Kentucky

October 26<sup>th</sup>, 2006

1. Introduction.....	3
2. Literature Review.....	4
2.1 Rootkits.....	4
2.2 Rootkit Classification.....	7
2.3 Rootkit Detection.....	8
3. Proposed Research.....	11
3.1 Scope of Research.....	11
3.2 Preliminary Model.....	12
3.3 Preliminary Work.....	13
3.3.1 Selection of Rootkits for Evaluation.....	13
3.3.2 Hardware Platforms.....	13
3.3.3 Kernel Modifications.....	15
3.3.4 Memory Analysis Toolset.....	15
3.3.5 Normality of Data.....	16
3.3.6 Statistical Methods.....	17
3.3.7 Preliminary Experimental Results.....	18
3.3.8 Validation of Experimental Results.....	20
4. Detailed Research Plan.....	21
4.1 Refinement of the Model.....	21
4.2 Final Evaluation and Testing.....	21
Conclusion.....	22
References.....	22
Appendix A: Unmodified SPARC System Call Address Values.....	26
Appendix B: Unmodified Intel System Call Address Values.....	32
Appendix C: Rkit System Call Table Results - Intel.....	38
Appendix D: Knark System Call Table Results - Intel.....	44

## 1. Introduction

Computer attacks happen every day, under every conceivable circumstance. Simply connect a computer to the internet, and it will be scanned, probed, and attacked tens, hundreds, or thousands of times a day. Should this computer be used for actual business, academic, or government purposes these attacks may increase to even higher levels. Society has become increasingly dependent on computer technology over time [1].

Nearly everyone has observed the seemingly unlimited flaws and vulnerabilities inherent in the protocols, operating systems, applications, and other software that makes up modern computing environments. By taking advantage of these flaws, attackers can assume control of systems, steal data, attack other systems, and general wreak havoc. Computing technology has simply advanced too quickly for security technology to keep up. The reality is that today's computing environments are inherently hackable [1].

Modern computer security efforts are primarily concerned with the *prevention* of attacks; the *detection* of attacks or attempted attacks when they occur; and *recovery* from successful attacks [2]. Prevention entails activities such as running secure versions of popular operating systems, disabling services with known vulnerabilities or weaknesses, and installing software or hardware designed to prevent successful attacks. Detection of successful or attempted attacks is covered in a broad field known as *intrusion detection*, which can further be divided into *network intrusion detection* and *host based intrusion detection*. Recovery from successful attacks includes those actions taken to restore the system to an operational state, and usually entails restoring data and applications from tape backups [3].

Network intrusion detection is typically conducted using a sniffing tool such as *Snort*. Network activity is typically saved and later analyzed for anomalous behavior and attack signatures. Host based intrusion detection is typically accomplished using host based security applications such as *Tripwire*. There are *many* applications for use in both network and host based intrusion detection. In the grand scheme of computer security, rootkit detection fits well into the area of *host based intrusion detection*. There also exist *many* programs for detecting rootkits on host systems.

A primary concern of attackers everywhere is not only how to gain privileged access to a system, but also how to *keep* it. In order to keep privileged access, the attacker must conceal his or her activities from the system administrator and other legitimate users of the system in question. Over time, concealment of illicit activities has evolved from the manual editing of log files, to the development of simple tools for this and similar purposes, culminating to the development of rootkits ranging from the simple to the Byzantine.

A rootkit is a method by which hackers maintain control of a compromised system, attack other systems, destroy evidence, and decrease the chance of being detected by system administrators [4]. The first rootkits were detected on SunOS machines in 1994. Since then, a "projectile/armor" race has erupted between those trying to develop/detect rootkits

[1;5]. A rootkit is essentially a set of software tools employed by an intruder after gaining unauthorized, privileged access to a system. Rootkit software has three primary functions: (1) to maintain access to the compromised system; (2) to attack other systems; and (3) to conceal evidence of the attacker's activities [5].

Rootkit detection is, in fact, a specialized form of intrusion detection. Effective intrusion detection includes the collection of information about intrusion techniques that can be used to improve methods of intrusion detection [2]. Why conduct further research into rootkit detection when there already exist many applications for this purpose? In all current techniques for detecting rootkits, some form of *a priori* knowledge about the specific system under observation is required. Either (a) some application must be installed when the system is deployed, as is typical with host based intrusion detection, or (b) some system metrics must be saved to a secure location when the system is deployed. In a perfect world, this would not prevent a problem. In reality, system administrators are busy people and the time, effort and expertise required for these activities is simply not available.

The purpose of this research is to detect rootkits using a more mathematically and statistically rigorous method, while not requiring any specific *a priori* knowledge of a specific system. However, it should be noted that it will still be necessary to have some *a priori* knowledge of general systems of the same type under observation. The research effort will be concentrated on one specific operating system using two different hardware platforms, specifically Linux kernel 2.4.27 on both an Intel 32 bit architecture and SPARC 64 bit architecture.

In it's more than twenty year history, UNIX has changed and evolved into many different flavors and releases. These changes include the introduction of UNIX into University environments, and the advent of BSD, System V, The Open Software Foundation, Posix, and several secure UNIX variants. During this time, many vulnerabilities and methods of attack have been discovered and utilized, but eighty percent (80%) of all security violations are permission based [6]. All of the dates presented herein are the dates upon which the information became publicly available. This software may have been available in the underground at a much earlier time [5].

## **2. Literature Review**

In the following section on literature review, general overview of rootkits will be presented including history and a discussion of the many backdoors techniques utilized by various rootkits in section 2.1. Section 2.2 covers rootkit classification, with special attention given to kernel rootkits. Section 2.3 includes a detailed discussion of existing rootkit prevention and detection techniques.

### **2.1 Rootkits**

The earliest rootkits have existed since approximately the early 1990s [1]. As early as 1989, some components (e.g., log file cleaners) of known rootkits were found on

compromised systems. The first early SunOS rootkits (for SunOS 4.x) were detected in 1994. In 1996, the first Linux rootkits publicly appeared. On April 9th, 1997, Linux Kernel Module (LKM) rootkits were proposed in the hacker magazine *Phrack* by *HalfLife* [5].

In 1998, Non-LKM kernel patching was proposed by Silvio Cesare in his landmark paper *Runtime Kernel Patching*[7]. He points out that it is possible to intrude into kernel memory without loadable kernel modules by directly modifying the kernel image (usually /dev/mem) [5]. In 1999, the first Adore LKM rootkit was released by TESO. This rootkit alters kernel memory via Loadable Kernel Modules. In 2000, the T0rnkit v8 libproc library Trojan was released. Library Trojans (usually libproc.a or glibc/libc [8]) can filter certain processes from being seen. Statically linked applications, or looking directly at /proc, will typically reveal the hidden process(es) [5].

In 2001, KIS Trojan and SucKit released. These rootkits alter kernel memory not by using Loadable Kernel Modules, but by directly modifying the kernel image (usually in /dev/mem). In 2002, Sniffer backdoors start to show up in rootkits. Maintaining access is typically accomplished using backdoors [5]. Rootkits came to public awareness in 2005, during the Sony CD copy protection scandal, wherein Sony placed rootkits on Microsoft Windows PCs when a CD was played. Sony did not mention this in the CD or packaging, mentioning only “security rights management measures” [9].

As mentioned above, maintaining access to a compromised system is typically accomplished by using one or several commonly known backdoor methods [1]. In the well known paper, *An Overview of Unix Rootkits* [5], Chuvavkin outlines the many backdoor techniques available to the rootkit developer. These backdoor techniques include:

Telnet/Shell – An attacker may simply connect to a compromised system using telnet or an inetd spawned shell on a high port. This is a very unsophisticated method.

Secure Shell – A Secure Shell connection on a high port is a common method employed by less sophisticated attackers. Custom Secure Shell daemons also may not even leave evidence in host log files. The netstat command, or an external scan by nmap, will reveal this technique.

CGI Shell - It is possible that a rootkit may deploy a hostile CGI script during installation. This is often considered a backdoor of “last resort”. The script may be able to run commands as “nobody” or “httpd” and display the results in the browser. Local exploits will need to be used to once again obtain root.

Reverse Telnet/Shell – In this case the compromised machine initiates an outbound connection to the attacker's machine. This technique has the advantage of possibly being able to circumvent firewalling efforts (i.e., outbound connections are typically allowed). Observant system administrators may find it odd that their servers are initiating unusual outbound connections.

ICMP Telnet – It has been said that everything can be tunneled over everything else. ICMP control messages can be made to carry payloads like command line sessions. It is not uncommon for ICMP traffic to be allowed through firewalls for network performance and monitoring reasons. Backdoors like these will not be discovered using commands like netstat and nmap. However, ICMP backdoor activities are visible to intrusion detection systems.

Reverse Tunneled Shell – In most environments, web browsing via port 80 TCP is allowed and typically unrestricted. In this case the command line session is carried across the HTTP protocol between the attacker and the compromised host.

Magic Packet Activated Backdoor - This backdoor will open a port, execute a single command, initiate a session, or perform some other action when it receives a single magic packet. The packet will possess a specific TCP sequence number or some other inconspicuous property.

Sniffer Based Backdoor - Instead of opening a port and listening, this backdoor sniffs network traffic instead. Upon receiving a specific packet (not necessarily directed to the compromised host, but instead observed on the network only), the Sniffer Based Backdoor performs an action and sends a response using a faked source IP address. This method is *extremely* stealthy and very difficult to detect [5;10].

Covert Channel Backdoor – If one were to create their own signal system and combine this with any known network protocol, it would probably never be detected using existing methods. The number of variables and large number of fields in existing network protocols and applications is very large. This method is provably undetectable.

It is worth re-emphasizing that some of these backdoor techniques (sniffer-based backdoor, covert channel backdoor) can be *extremely* stealthy or even *Undetectable* [5]. This suggests that even after discovering and removing a rootkit, a system administrator would be well advised to conduct a full system reinstall in order to be sure they have eradicated all existing backdoors on the suspect system. Fortunately, no known rootkits utilize the provably undetectable covert channel backdoor.

Tools for attacking other systems, both locally and remotely, began appearing in rootkits during the late 1990s. Local attack tools exist primarily for the purpose of recapturing root access from vigilant system administrators. Tools of this kind typically include local password sniffers or crackers.

Remote attack tools typically include a basic network sniffer to eavesdrop and obtain username/password pairs on the same local area network where clear text protocols are used. Also in this class of tools are various network scanners and automated exploit tools (autorooters). As an example, an attacker may scan a range of IP addresses for vulnerable web servers, and run an autorooter to gain root privileges on those vulnerable hosts

Most rootkits contain at least one or more denial of service tools. Some systems, in fact, contain system commands that may be used to flood other hosts (e.g., the *spray* command in Solaris). Attackers may use the DoS tools against their enemies or during their use of *Internet Relay Chat* [4].

The third and final area of rootkit functionality is the elimination of evidence. Ideally a rootkit strives to eliminate evidence generated during the initial attack, and prevent the generation of any new evidence. What this means, in reality, is the careful editing of various log files, audit records, shell histories, and application log files [10]. There are a large number of well known utilities that exist for this purpose. However, no known rootkits utilize any form of secure or reliable data removal – yet.

Preventing the generation of further evidence usually entails terminating or modifying the syslog daemon. Attackers also typically take action to ensure that shell history files and application log files are not generated [5].

## 2.2 Rootkit Classification

There are three known categories of rootkits. The first and simplest type are binary rootkits, composed of modified, malicious copies of system binaries that are placed on the host system. A logical second step in the evolution of the rootkit is the library rootkit, in which a modified and malicious copy of a system library is placed on the host system. These first two categories of rootkit are relatively easy to detect.

The third, and most insidious, category of rootkit is the kernel rootkit. There are two subcategories of kernel rootkits, loadable kernel module rootkits (*LKM rootkits*) and kernel rootkits that directly modify the memory image in `/dev/mem` (*kernel patched rootkits*) [11]. Kernel-level rootkits attack the system call table by three known mechanisms [12].

System Call Table Modification. The attacker modifies the addresses stored in the system call table. The attacker, having written custom system calls [13] to replace several system calls within the kernel, changes the addresses in the system call table to point to the new, malicious custom system calls.

System Call Target Modification. In this case, the attacker overwrites the legitimate targets of the addresses in the system call table with malicious code. The system call table does not need to be changed. The first few instructions of the system call function is overwritten with a jump instruction to the malicious code.

System Call Table Redirection. In this type of rootkit implementation, the attacker redirects references to the entire system call table to a new, malicious system call table in a new kernel address location. This method can pass many currently used detection techniques [12]. Upon further investigation, it appears that the system call table redirection attack is simply a special case of the system call target modification attack [14]. The attacker simply modifies the *system\_call* function, modifying the address of the system call table therein, which handles individual system calls.

## 2.3 Rootkit Detection

The first rootkits were simply tar archives of system binaries that were likely to be executed by suspicious system administrators of compromised systems. These binaries were typically, but not limited to, binaries such as `netsat`, `kill`, `killall`, `passwd`, `ps`, `pstree`, `sendmail`, `su`, `syslogd`, and `top`. These binaries would be replaced with modified copies created by the attacker in order to provide remote access, local access, process hiding, connection hiding, file hiding, and user activity hiding. These application rootkits are easily discovered by keeping secure copies of critical system binaries on read only removable media, checking binary file sizes, using checksums, looking at the `/proc` file system directly, and so forth [1].

Library rootkits, such as `T0rn`, replace the system library `libproc.a` with a special modified library in order to maintain stealth. System binaries such as `ps` and `top` rely upon this library to relay information from the kernel space. Using a modified library allows one to avoid changing system binaries but still selectively filter file and process lists. Once again, looking directly at the `/proc` file system will reveal this attack. It is also relatively straightforward to modify the `glibc/libc` main system library to filter data before it is sent to the kernel. Any application linked with this library (most applications) will report false information. This attack may be avoided by using statically linked applications. The UNIX commands `ltrace`, `strace`, and `truss` can be used to trace library and kernel calls [5].

The first kernel rootkits appeared as malicious loadable kernel modules (LKM). Processes under UNIX run either in user space or kernel space. Application programs typically run in user space and hardware access is typically handled in kernel space. If an application wants to read from a disk, it uses the `open()` system call and asks the kernel to open a file. Loadable kernel modules run in kernel space and have the ability to modify these system calls. If there is a malicious loadable kernel module in kernel space, the `open()` system call will open the file requested unless the name of the file is “rootkit” [1;5].

Many system administrators countered this threat by simply disabling the loading of kernel modules [1]. However, Silvio Cesare recently published a paper proposing a method for modifying system calls by directly accessing the kernel memory image in `/dev/mem` [7]. Several rootkits have since been discovered that successfully utilize this method.

Earlier rootkits such as binary and library rootkits may be detected using relatively simple countermeasures. Binary rootkits may be detected by simply checking the file size of system binaries or using checksums or hashes of the system binaries. Library rootkits may be detected by comparing file sizes, checksums, or hashes of the library files under suspicion as well as by using statically linked applications. Both binary and library rootkits may be easily detected by looking directly at the `/proc` file system [1;5].

Host based intrusion detection systems (*Tripwire* and *Samhain* being the most well known) are still a relatively straightforward and effective way of detecting known rootkits [1]. *Samhain* also includes functionality to monitor the system call table, the interrupt description table, and the first few instructions of every system call [5].

The Linux Intrusion Detection System (LIDS) is a kernel patch that must be applied to kernel source code, and requires a rebuild of the kernel. LIDS has the capability to offer protection against kernel rootkits through the following mechanisms: sealing the kernel from modification; prevent loading/unloading of kernel modules; immutable and read-only file attributes; locking of shared memory segments; process ID manipulation protection; protection of sensitive `/dev/` files; and port scan detection [10]. LIDS appears to be more rootkit *prevention* tool than rootkit *detection* tool. As with all other techniques discussed so far, LIDS requires either (a) some action be taken in advance to thwart rootkit activity, or (b) some *a priori* knowledge of the specific system under observation.

One detection method proposed by Sebastian Kraemer from SuSE in the past was to monitor and log any program execution when `execve()` calls were made. Combine this with remote logging, and one could maintain a record of program execution on a system. With a Perl script to monitor the log, one could perform actions such as sending alarms or killing processes in order to stop the intruder [15].

Applications do exist for the purpose of detecting rootkits (including kernel rootkits). These include several tools available for download including *chkrootkit*, *kstat*, *rkstat*, *St. Michael*, *scprint*, and *kern\_check* [9;16-22]. *Chkrootkit* is a user-space signature based rootkit detector, while several others (*kstat*, *rkstat*, and *St. Michael*) are kernel-space signature based detectors. These tools typically print the addresses of system calls directly from `/dev/kmem` and compare them to the entries in the `System.map` file [10]. This approach relies upon some trusted source of *a priori* knowledge of the specific system in question. *Chkroot*, *kstat*, *rkstat*, and *St. Michael*, as signature based detectors, suffer from the usual shortcomings of signature based detection.

*Scprint* and *kern\_check* are utilities for printing and/or checking the addresses of the entries in the system call table. Several of these utilities have proven quite useful in attempts to verify the results of detection attempts against various categories of kernel rootkits.

Other researchers have proposed to count the instructions used in system calls, comparing them to measurements taken from a “clean” system [23]. This approach seems very promising, but requires a kernel patch, installation of an application, and *a priori* knowledge of the instruction count of each system call on the specific system in question.

Further efforts in the field of rootkit detection include static analysis of loadable kernel module binaries [24]. The kernel exports a well-defined interface for use by kernel modules, and LKM rootkits typically violate this interface. By carefully analyzing this interface, one may extract an allowed set of kernel modifications. Using this set of

allowed kernel modifications, a researcher may statically analyze a loadable kernel module binary to determine whether it violates this allowable set of kernel modifications. This technique seems very promising for the detection of LKM rootkits, but the authors do not offer any alternatives for detecting kernel patched rootkits.

Until recently, efforts toward rootkit detection have been software based. College Park, Maryland based Komoku Inc. offers a low-cost, add-in PCI card that monitors a host system's memory and file system [25;26]. Copilot may be used in real time to detect the deployment of rootkits, and is effective. However, Copilot uses "known good" MD5 hashes of kernel memory and must be installed and configured on a "clean" system in order to detect the future deployment of a rootkit [27]. Spafford and Carrier have presented a technique in which binary rootkits were detected using an outlier analysis technique on the file system in an offline forensic analysis situation [28]. The research presented in this paper focuses on the real time detection of kernel rootkits through memory analysis.

By default, the Linux operating system may access up to 4 Gigabytes of virtual memory, with memory addresses between 0x00000000 and 0xFFFFFFFF in hexadecimal notation. An upper portion of this memory is allocated for use by the kernel. This upper memory area has addresses between 0xC0000000 and 0xFFFFFFFF. Typically, system calls will have addresses such as 0xC011D0E1, 0xC013A229, or 0xC010B4D0 [14].

The symbol `_text` indicates the first byte of kernel code. The end of kernel code is marked by the presence of the `_etext` symbol. The following kernel data is categorized as initialized and un-initialized. The initialized kernel data starts at the symbol `_etext` and ends at symbol `_edata`. The un-initialized portion of kernel data starts immediately after `_etext` and stops at symbol `_end` [29]. Preliminary experiments have shown that LKM rootkits create malicious system calls at address locations that exceed the memory value of symbol `_end` – at memory address 0xC041D8A9 or greater. This data suggests that malicious system calls may be detectable through the use of outlier analysis techniques.

Whenever a new loadable kernel module (LKM) is loaded, the kernel allocates a portion of memory for it usually starting at 0xC8800000. If there exists a system call, then, with an address such as 0xC8801A12 or higher, this implies that a system call has been replaced with a system call from a loadable kernel module. This is highly suspect, and strongly suggests the presence of an LKM kernel rootkit [14]. It may be possible to make mathematical or statistical observations about these memory addresses, and produce a more formal, reliable assessment of the presence of a rootkit without *a priori* knowledge about the specific system under scrutiny.

Whether this method will also succeed in detecting kernel patched rootkits that directly modify kernel memory in `/dev/mem` remains unknown. There are well known tools for analyzing memory and the system call table. This may be accomplished using common system tools such as the GNU debugger in conjunction with the `System.map` file or the `nm` system binary [14;29]

### **3. Proposed Research**

In the following section, several aspects of the proposed research will be discussed in detail. Section 3.1 includes a brief overview of the scope of this research. Section 3.2 will present a preliminary model for use in this research. Section 3.3 includes a detailed discussion of the preliminary work on this subject. Section 3.3.1 contains an explanation of rootkit selection criteria. Section 3.3.2 presents an explanation of which hardware platforms will be used during the research, and the importance of having as broad a selection of hardware available as possible. Section 3.3.3 covers kernel modifications necessary to conduct and analyze experiments in support of the research. Section 3.3.4 contains an explanation of the memory analysis tools used for analyzing the outcomes of the experiments. Section 3.3.5 examines the normality of the data used in the preliminary experiments. Section 3.3.6 includes a discussion on the specific statistical methods to be used in this research, and in Section 3.3.7 results from preliminary experimentation will be presented and discussed in detail.

#### **3.1 Scope of Research**

As mentioned in previous sections, there have been several attempts at preventing and detecting the deployment of rootkits. The simplest of these ranged from using checksums of various system binaries and system libraries, to recompiling the kernel to prevent the loading of loadable kernel modules [1] (thus hopefully thwarting the deployment of loadable kernel modules).

More advanced attempts included the use of host based intrusion detection systems such as Tripwire and Samhain. Other researchers have shown that the system administrator may detect even the most advanced rootkits through the skillful use of a debugger, such as the GNU debugger. Still others have attempted to detect loadable kernel module (LKM) kernel rootkits using binary analysis of modules and comparing their adherence to The well-known kernel interface exported for use by modules. LKM Rootkits generally violate this well known interface. This research will employ the GNU debugger and other memory analysis tools, and possibly other techniques, to detect rootkits, but with new more formal, more rigorous analysis of the data.

As a preliminary step it will be shown that there exists a statistical correlation between the values of memory addresses in the system call table and the presence of a LKM rootkit that uses the *system call table modification attack*. It will also be shown that there exists a statistical correlation between the values of memory addresses in the operands of disassembled instructions of system calls and the presence of a kernel patched rootkit that uses the *system call target modification attack*.

Having shown that kernel rootkits do impact the values of system call memory addresses in kernel memory, experiments will be conducted in order to investigate further. Specifically, malicious system call addresses (modified addresses) and operand addresses

within modified system call targets (modified instructions) will be isolated and identified without using any *a priori* knowledge about the specific system under observation, and without the advance installation of any rootkit detection software.

These techniques will be tested with as many different rootkits as possible. Furthermore, it will be shown that rootkits may be detected statistically by understanding the underlying distribution of system call addresses and operand addresses of disassembled system call instructions of the various major kernel releases of Linux, and recording this *general* information and comparing it to *specific* systems infected with rootkits. More specifically, outlier analysis techniques will be used to perform this analysis. These methods will not only indicate the presence of a rootkit, but also identify the number of individual attacks on the kernel and their locations. This information will aid other research efforts focusing on rootkit categorization and identification [12].

This research will also be constrained by the following:

The only operating system under consideration will be Linux Kernel version 2.4.27, Running on Intel 32 bit and SPARC 64 bit architectures. Only Kernel rootkits (Linux Kernel Module and Kernel Patched) will be investigated, but as many rootkits from each category as possible will be included in the research.

### 3.2 Preliminary Model

In this section, two new detection techniques for kernel rootkits will be presented. These techniques will not rely on either (a) advanced installation of any rootkit detection or other software, or (b) any other *a priori* knowledge about the specific system under observation. Additionally, these new techniques will be more rigorous than existing methods.

The first new technique is a method for detecting Linux Kernel Module rootkits. As mentioned previously, these rootkits modify memory addresses in the system call table. These memory addresses fit the *Largest Extreme Value distribution* very well; the Anderson-Darling goodness-of-fit test yields a score of approximately five (5). This seems to hold across multiple architectures; experiments on Intel 32 bit architectures and SPARC 64 bit architectures yield very similar results.

When a Linux Kernel Module rootkit is installed, several of the entries in the system call table are changed to unusually large values (indicative of the system call table modification attack discussed previously). This changes the goodness-of-fit score for the *Largest Extreme Value* distribution – the data is no longer such a good fit. Because of the Linux memory model and the method of attack, the outliers will be on the extreme right side of the distribution. If these outliers are eliminated one by one, the distribution slowly moves from a score approximately one hundred (100) back to its normal score of five (5). This result has been verified with preliminary experiments, and is discussed in more detail in Section 3.3.7.

The second new technique is a proposed method for detecting Kernel Patch Rootkits. As previously discussed, these rootkits operate by directly modifying the kernel image in `/dev/kmem`. System calls in the kernel are directly overwritten, and typically the first ten instructions are changed, and this includes a jump instruction to a location much higher in memory (i.e., an outlier).

In order to analyze this attack, all functions in the kernel must be disassembled, and those instructions with memory addresses as operands are of particular interest. These memory addresses may also fit a well known distribution. However it is not known if this data will contain outliers (unusually high memory addresses in a non-rootkit situation). If outliers do exist in this context, much more complex outlier analysis techniques will be necessary for rootkit detection. It will also be necessary to conduct experiments across multiple architectures, to determine whether this data fits the same distribution across several architectures. Fortunately, several outlier analysis techniques are available for this research that do not rely on any underlying distribution of data. There are no preliminary results for this approach, this is the planned model for the detection of this second category of more complex kernel rootkit.

### **3.3 Preliminary Work**

#### **3.3.1 Selection of Rootkits for Evaluation**

Several source code repositories containing rootkits are available on the internet [30-32]. Unfortunately, the selection of available rootkits is rather limited. Furthermore, experience has shown that individual rootkits are written for either (a) a particular Linux kernel version, or (b) a particular architecture, or (c) both. This fact underscores the need for this research to focus on multiple kernel versions and architectures. Rootkits available for download from the internet are not typically designed by their attack method – for example, LKM kernel rootkits or kernel patched rootkits.

Three known Linux Kernel Module rootkits are *Knark*, *Adore*, and *Rkit* [10]. *Knark* and *Rkit* were used to obtain the preliminary results presented in this paper. Two known Kernel Patched rootkits are *KIS Trojan* and *SucKit* [5], and these rootkits will be used in future work on this topic.

#### **3.3.2 Hardware Platforms**

One consideration that is critical to the success of this research is that the distribution of system call addresses for a specific kernel version must be very close across various architectures. This is an absolute necessity if analysis is to occur without any *a priori* knowledge of the specific system under study. Preliminary experiments were conducted on a 32-bit Intel machine and a 64-bit SPARC machine with different kernel compilation options in order to test this hypothesis.

Table 3.1: Distribution fits from 32-bit Intel machine, kernel 2.4.27

Distribution	AD
Largest Extreme Value	5.038
3-Parameter Gamma	6.617
3-Parameter Loglogistic	7.022
Logistic	7.026
Loglogistic	7.027
3-Parameter Lognormal	10.275
Lognormal	10.348
Normal	10.350
3-Parameter Weibull	49.346
Weibull	49.465
Smallest Extreme Value	49.471
2-Parameter Exponential	81.265
Exponential	116.956

Table 3.2: Distribution fits from 64-bit SPARC machine, kernel 2.4.27

Distribution	AD
Loglogistic	10.599
Largest Extreme Value	11.699
Logistic	11.745
Lognormal	19.147
Gamma	20.460
Normal	23.344
3-Parameter Gamma	26.456
3-Parameter Weibull	32.558
3-Parameter Loglogistic	34.591
Weibull	36.178
3-Parameter Lognormal	37.468
Smallest Extreme Value	41.015
2-Parameter Exponential	52.604
Exponential	102.787

While the *Largest Extreme Value distribution* best fits the system call addresses from the 32-bit Intel machine, it was not the best fit for the system call addresses for the 64-bit SPARC machine used in preliminary testing. However, *Largest Extreme Value* is still a *very good fit* (a close 2<sup>nd</sup>) for the SPARC. While *many* more observations are necessary to make claims of goodness-of-fit for the system call addresses for various categories of computers, this preliminary result suggests that this may be possible, especially for machines of different architectures but having the same kernel version.

Experience has shown that Linux seems to be developed for and works best with the Intel architecture. Installing, compiling, and loading custom modules with Linux on SPARC was problematic but was eventually successful [33-35]. Challenges such as this should be expected and planned for with the inclusion of additional architectures.

### 3.3.3 Kernel Modifications

In order to debug a running kernel (or any other process) it is necessary to have a *minimum* amount of debugging support compiled into the binary. Additional debugging symbols may be compiled into any binary simply by using a command such as “*gcc -g -o binary binary.c*”. In practice, additional debugging symbols do not need to be compiled into a kernel for the analysis necessary in this research. However, it is necessary that the kernel or binary in question has not been stripped with the *strip* command. During preliminary testing, it was discovered that Debian 3.1 Release 1 with kernel version 2.4.27 installs with a *stripped* kernel [36], presumably to save space. It was necessary to rebuild the kernel in order to have even basic debugging ability for this research.

As previously mentioned, the Linux operating system may access up to 4 Gigabytes of virtual memory in a default configuration, with memory addresses between 0x00000000 and 0xFFFFFFFF in hexadecimal notation. The kernel which will be used in this research has had support for 4 Gigabytes of memory removed, and now only supports up to one Gigabyte of memory. None of the hardware to be used in these experiments has four Gigabytes of memory, but if statistical outliers may be detected in a one Gigabyte (or less) memory space, detecting those same outliers in a four Gigabyte memory space should pose much less of a challenge.

Another tool available for kernel debugging is the Linux kernel debugger (*kdb*). Preliminary experiments have shown *kdb* to be unstable and problematic when used in conjunction with XWindows. Performance in terminal mode is much better, however many of the commands covered in the documentation do not appear to be implemented. It is mentioned here because it required two kernel patches and recompilation of the kernel to implement.

### 3.3.4 Memory Analysis Toolset

Two categories of memory analysis tools were selected for use in this research. The first category includes but one application, the GNU debugger, or *gdb*. *Gdb* is a source level debugger, and includes facilities for examining memory, disassembly, attaching to running processes, scripting support, and many other functions. *Gdb* does require a minimum set of debugging symbols to be compiled into the binary to be debugged, but in practice this simply requires that the debugging target must not have been *stripped* in order to save space. Debugging a running kernel with *gdb* requires the kernel binary (typically */boot/vmlinux*) and a core file for the running kernel (typically */proc/kcore*). *Gdb* has proven indispensable in kernel debugging for the purpose of rootkit detection, and will be a primary application used in this research [37-39]

The second category of memory analysis tool consists of the Linux kernel debugger, *kdb*. The kernel debugger consists of two kernel patches, and requires that the kernel be recompiled in order to use *kdb*. Preliminary experiments have shown *kdb* to be unstable, particularly when used in conjunction with XWindows, and a substantial portion of the

commands covered in the *kdb* documentation do not appear to be implemented. Further adding to these problems, *kdb* does not appear to support output redirection and other Unix command line conveniences, adding to the difficulty of utilizing it for anything other than a cursory examination of kernel structures and memory.

### 3.3.5 Normality of Data

All symbols in the Linux kernel version 2.4.27 - 18,948 of them – best fit the *Smallest Extreme Value* distribution (score of 757.471). The Anderson-Darling score is used to determine goodness-of-fit. Suspicious addresses seem to improve goodness-of-fit score for the Normal distribution, but greatly decrease goodness-of-fit score for Smallest Extreme Value distribution. This finding emphasizes the need to utilize best fitting distribution, even though differences in the original goodness-of-fit scores is small between the best fitting distributions. The *Smallest Extreme Value* distribution fits the data best (Anderson-Darling score of 757.471), and the *Normal* distribution has an Anderson-Darling goodness-of-fit score of 909.427, so this data is fairly normally distributed. This is summarized below in table 3.3.

Table 3.3: All Symbols in Linux Kernel 2.4.27

Goodness-of-fit Test	
Distribution	AD
Smallest Extreme Value	757.471
3-Parameter Weibull	757.476
Weibull	757.805
Logistic	849.898
3-Parameter Loglogistic	849.904
Loglogistic	850.159
Normal	909.427
Gamma	909.625
Lognormal	909.643
3-Parameter Lognormal	909.889
3-Parameter Gamma	926.644
Largest Extreme Value	1047.468
2-Parameter Exponential	2697.662
Exponential	8685.713

The symbol table in the Linux kernel version 2.4.27 contains 255 Symbols and best fits the *Largest Extreme Value* distribution. The Anderson-Darling goodness-of-fit metric is once again used to determine goodness-of-fit. Including suspicious addresses (obtained by applying a kernel LKM rootkit) *dramatically* decreases Anderson-Darling goodness-of-fit scores for all evaluated distributions. Goodness-of-fit scores range from 5.038 for the *Largest Extreme Value* distribution (best fit), to 116.956 for the *Exponential* distribution (worst fit). As seen in Table 3.2, this data has a goodness-of-fit score of 10.350 for the Normal distribution, so it is fairly normally distributed.

Table 3.4: System Call Table for Kernel 2.4.27

Goodness-of-fit Test	
Distribution	AD
Largest Extreme Value	5.038
3-Parameter Gamma	6.617
3-Parameter Loglogistic	7.022
Logistic	7.026
Loglogistic	7.027
Lognormal	10.348
3-Parameter Lognormal	10.275
Normal	10.350
3-Parameter Weibull	49.346
Smallest Extreme Value	49.471
2-Parameter Exponential	81.265
Weibull	49.465
Exponential	116.956

### 3.3.6 Statistical Methods

There exist *many* discordancy tests for detecting outliers in univariate data. These include tests for samples that fit many underlying distributions – Gamma, Exponential, Normal, Log-normal, Truncated Exponential, Uniform, Gumbel, Frechet, Weibull, Pareto, Poisson, and Binomial distributions [40]. Preliminary experiments show that the data analyzed in this research tends to fit the Largest Extreme Value or Smallest Extreme Value distributions best. Furthermore, most discordancy tests require at least an *estimate* of the number of outliers, and their locations. The purpose of this research is to identify outliers without *a priori* knowledge of this kind.

A general and early approach to identifying outliers is to identify the underlying distribution of the data and identify individuals that deviate from the distribution. This approach is common in statistics, but does not scale well [41]. Using this approach, two LKM rootkits were successfully detected. These results are discussed in more detail in the following section. Additional distance and density-based outlier analysis techniques may be used successfully in this research when the preliminary model begins to degrade due to scaling issues, and these techniques do not rely on some underlying distribution of the data.

The preliminary model in this research utilizes the method mentioned in the previous paragraph, in conjunction with the Anderson-Darling goodness-of-fit test to identify individuals that deviate from the underlying distribution. Hawkins suggests the possibility of using any goodness-of-fit test as the basis for an outlier test, and that any good candidate for an outlier test would emphasize the quality of fit in the tails – one such test is the Anderson-Darling goodness-of-fit test [42]. The possibility of using the Anderson-Darling test as an outlier test does not seem to have been investigated, but is promising since this statistic is completely general and can be used with *any* underlying distribution [42]. This fact alone makes the Anderson-Darling goodness-of-fit test preferable to any previously mentioned discordancy tests for univariate data.

Distance based approaches to outlier analysis have been investigated by Ramaswamy et al. [43] and Knorr & Eng [44;45]. These techniques typically explore some neighborhood and do not rely on the underlying distribution of the data [41]. Knorr & Eng identify outliers by counting neighbors within a specified radius, with the radius and threshold number of points as the only two parameters [41]. Ramaswamy et al. identify outliers by calculating the sum of the distances to their nearest neighbors [41].

Breunig et al. have investigated a density based technique to score data points using “local outlier factor”, a measure of outlyingness calculated for each data point [41;46;47]. Jin et al. introduced a method for more efficiently identifying top outliers using the local outlier factor [41;48].

### 3.3.7 Preliminary Experimental Results

As previously mentioned, there have been several attempts at preventing and detecting the deployment of rootkits, but they require some form of *a priori* knowledge about the specific system under observation. This technique will employ the GNU debugger and other memory analysis tools, and possibly other techniques, to detect rootkits, through formal, rigorous analysis of the data. As a final step it will be shown that there exists a statistical correlation between the values of memory addresses in the system call table and the presence of a LKM rootkit that uses the *system call table modification attack*.

When a Linux Kernel Module rootkit is installed, several of the entries in the system call table are changed to unusually large values (indicative of the system call table modification attack discussed previously). This changes the goodness of fit score for the *Largest Extreme Value* distribution – the data is no longer such a good fit. Because of the Linux memory model and the method of attack, the outliers will be on the extreme right side of the distribution [14]. If these outliers are eliminated one by one, the distribution slowly moves from a score of approximately one hundred (100) back to very close to the original score of approximately five (5).

This new technique is a method for detecting Linux Kernel Module (LKM) rootkits. These rootkits modify memory addresses in the system call table, which originally fit the *Largest Extreme Value distribution* very well; the Anderson-Darling goodness of fit test yields a score of approximately five (5). This seems to hold across multiple architectures; experiments on Intel 32 bit architectures and SPARC 64 bit architectures yield similar results.

In experiment one, the Rkit Linux Kernel Module rootkit version 1.01 was downloaded and installed on a 32-bit Intel computer running Linux kernel version 2.4.27. Rkit 1.01 only modifies one entry in the system call table – *sys\_setuid*. Rkit 1.01 was selected because (a) it is a LKM rootkit, and (b) it attacks only *one* entry in the system call table. If only one outlier can be detected using this method, rootkits that attack several system call table entries may be detected more easily.

From table 3.1, it is known that the test system – a 32-bit Intel computer running Linux kernel 2.4.27 – has a 255 entry system call table fitting the Largest Extreme Value distribution with an Anderson-Darling goodness of fit score of 5.038. When Rkit 1.01 is installed, the Anderson-Darling goodness of fit score changes to 99.210. Clearly, an outlier is present in the form of the *sys\_setuid* system call table entry with a greatly increased memory address. The *sys\_setuid* system call table entry address was changed from 0xC01201F0 (good value) to 0xD0878060. Converted to decimal, these values are 3,222,405,616 and 3,498,541,152 – a difference of 276,135,536 and approximately 8.5% larger than the original value.

When one system call table address is modified, the goodness of fit score changes from 5.038 to 99.210, a change of approximately 1970%. When the modified *sys\_setuid* memory address is removed from the data, the Anderson-Darling goodness of fit score for the *Largest Extreme Value distribution* returns to 4.968 – within 1.4% of the original score of 5.038.

Table 3.5: Results of Rkit 1.01 experiment

System	AD-Score
Clean	5.038
Modified	109.729
Modifications Removed	5.070

In experiment two, the Knark Linux Kernel Module rootkit version 2.4.3 was installed on the same test system – a 32-bit Intel computer running Linux kernel version 2.4.27. Knark is also a Linux Kernel Module rootkit, and attacks nine different memory addresses in the system call table. Experiment two yields similar results as experiment one – a 2178% decrease in goodness of fit, then a return to within 0.7% of the original score when the outlying modified addresses are removed.

Table 3.6: Results of Knark 2.4.3 experiment

System	AD-Score
Clean	5.038
Modified	99.210
Modifications Removed	4.968

Also in experiment two, as the modified system addresses are removed one by one, the Anderson-Darling goodness of fit score slowly improves, but does not show a dramatic or significant improvement until the final outlier is removed. The importance of this fact lies in the concept of *complete detection*. Through this method, a rootkit that attacks only *one* system call table address can be successfully detected. Figure 3.1, below, illustrates this finding. It is possible to not only detect *most* modified system call addresses, but *all* modified system call addresses.

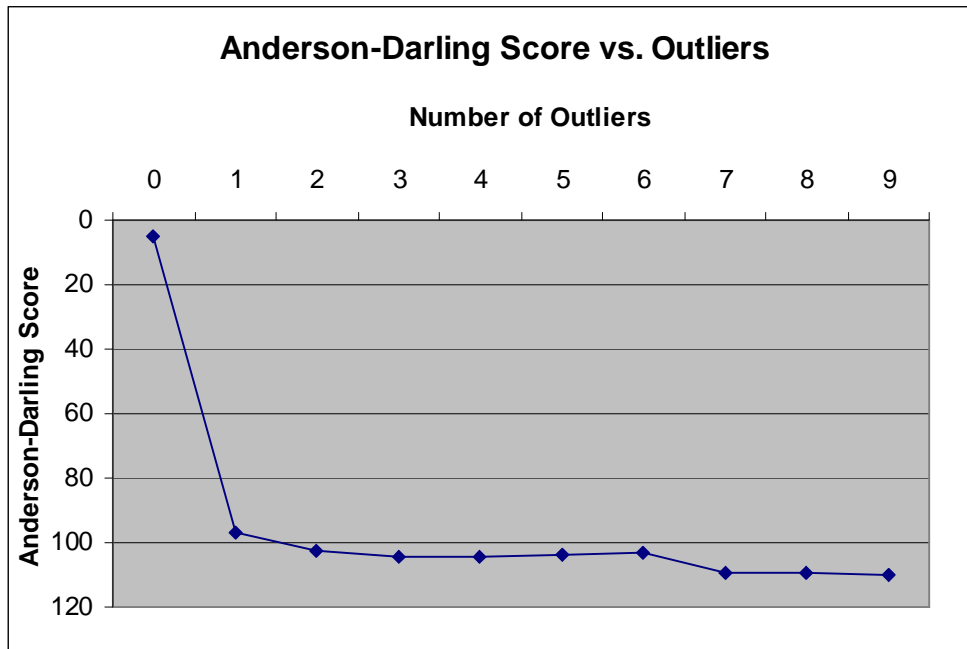


Figure 3.1

### 3.3.8 Validation of Experimental Results

Burdach has claimed that modified system call table entries have much higher than normal memory address. Does a modified system call table address really cause the address to be much higher than normal, as Burdach claims [14]? A 2 sample t test was used to test this hypothesis for experiment 2, since the samples are (1) those system call table entries that have been maliciously modified, and (2) those system call table entries that have not been modified. The null hypothesis,  $H_0$ , states that the presence of maliciously modified system call table entries has no bearing on the mean memory address value of the system call table entries. The alternative hypothesis,  $H_1$ , states that the presence of maliciously modified system call entries does affect the mean memory address value of the system call table entries. A confidence interval of 95% was used.

#### Two-Sample T-Test and CI: dec\_address, modified

Two-sample T for dec\_address

modified	N	Mean	StDev	SE Mean
0	246	3222441410	106233	6773
1	9	3498543362	874	291

Difference = mu (0) - mu (1)

Estimate for difference: -276101952

95% CI for difference: (-276115305, -276088598)

T-Test of difference = 0 (vs not =): T-Value = -40726.46 P-Value = 0.000 DF = 245

Since the P-value of  $0.000 < 0.050$ , the null hypothesis can be rejected. The presence of modified system call table entries does have a significant influence on the mean value of the system call table addresses.

#### **4. Detailed Research Plan**

In the final section of this document, Section 4.1 begins with a discussion of how the preliminary model presented earlier will be refined, plans for final evaluation and testing of this model are included in Section 4.2, and conclusions are offered in Section 4.3.

##### **4.1 Refinement of the Model**

The purpose of this research is to detect kernel rootkits (LKM and kernel patched) without any a priori knowledge of the system under study. With this in mind, the following refinements to the preliminary model presented in Section 3.2 are needed.

First, distance and density based techniques should be utilized in further attempts to detect LKM rootkits within the system call table only. This approach will be investigated on kernel versions 2.4 and 2.6. If successful, this approach will eliminate the need to identify the underlying distribution of data across kernel versions and architectures.

Second, distance and density based techniques will be applied to detect the system call target modification attack typically used by kernel patched rootkits. Initially this approach will only concentrate on those system calls present in the system call table. This approach will also be investigated on kernel versions 2.4 and 2.6.

Finally, analyzing addresses and assembler instructions from system calls outside the system call table is important for several reasons. Primarily, although the system calls within the system call table are the most popular targets of attack, it is *possible* to attack any system call within kernel. At least two malicious projects are known to attack the system calls associated with the virtual file system [49-51]. While VFS calls are not included in the system call table, they are certainly vulnerable to attack. Distance and density based techniques will be used in detections attempts using the disassembled instructions of all system calls within the kernel – not only those in the system call table. This approach also promises to yield important information about the scalability of the particular approaches used.

##### **4.2 Final Evaluation and Testing**

Rootkits of both categories (LKM and kernel patched) will be downloaded, compiled, and installed on test systems and subjected to the detection techniques discussed previously in this paper. However, experience has shown that many rootkits available for download either (a) do not successfully compile and require substantial debugging effort, or (b) exist for outdated kernel versions or some unusual combination of kernel version and hardware, or (c) both. Finding, compiling, and installing a rootkit to suit one's

particular purpose can be particularly daunting – particularly if a number of them are required for research purposes.

A particularly elegant solution to this problem is that relatively simple Linux Kernel Module rootkits and kernel patched rootkits may be written by the researcher in order to further test and refine the detection methodologies presented in this paper. These rootkits may be further refined to support various changes in attack vectors.

Rootkits found *in the wild* will be utilized in this research to the extent possible. Any deficiencies in the quantity or quality of rootkits available in the wild will be addressed by the development of researcher produced “test” rootkits.

### 4.3 Conclusion

These research will create a more rigorous, more formal framework for the detection of Linux kernel rootkits and the known methods of attack by which they operate. Future rootkits, which may operate by methods unknown at this time, are of particular interest. Hopefully this framework will assist in the detection of more advanced future rootkits.

### References

- [1] Ed Skoudis, *Counter Hack: A Step-by-Step Guide to Computer Attacks and Effective Defenses* Prentice-Hall, 2002, pp. 399-445.
- [2] William Stallings, *Network Security Essentials*, 2nd ed 2003.
- [3] Network Security: A Primer on Vulnerability, Prevention, Detection and Recovery, 9-1-2006, <http://www.integritycomputing.com/security1.html>
- [4] Know Your Enemy: Motives, 9-1-2006, <http://www.linuxvoodoo.org/resources/security/motives/>
- [5] A. Chuvakin, "An Overview of Unix Rootkits," *iALERT White Paper, iDefense Labs*, <http://www.megasecurity.org/papers/Rootkits.pdf>, February, 2003.
- [6] N.Derek Arnold, *Unix Security: A Practical Tutorial* McGraw-Hill, 1992.
- [7] Runtime Kernel Patching, 9-1-2006, <http://reactor-core.org/runtime-kernel-patching.html>
- [8] The GNU C Library, 9-1-2006, [http://www.delorie.com/gnu/docs/glibc/libc\\_toc.html](http://www.delorie.com/gnu/docs/glibc/libc_toc.html)
- [9] Wikipedia: Rootkit, 9-1-2006, <http://en.wikipedia.org/wiki/Rootkit>
- [10] Joel Scambray, Stuart McClure, and George Kurtz, *Hacking Exposed: Network Security Secrets & Solutions*, 2nd ed McGraw-Hill, 2001.

- [11] Linux Kernel Rootkits, 9-1-2006, [http://linuxcourse.rutgers.edu/documents/kernel\\_rootkits/index.html](http://linuxcourse.rutgers.edu/documents/kernel_rootkits/index.html)
- [12] J.Levine, B.Grizzard, and H.Owen, "Detecting and Categorizing Kernel-Level Rootkits to Aid Future Detection," *IEEE Security & Privacy*, no. January/February 2006, pp. 24-32, 2006.
- [13] Lab Exercise 2: Adding a Syscall, 9-1-2006, [http://www-static.cc.gatech.edu/classes/AY2001/cs3210\\_fall/labs/syscalls.html](http://www-static.cc.gatech.edu/classes/AY2001/cs3210_fall/labs/syscalls.html)
- [14] Detecting Rootkits and Kernel-level Compromises in Linux, 9-1-2006, <http://www.securityfocus.com/infocus/1811>
- [15] Root Kits and hiding files/directories/processes after a break-in, 9-1-2006, <http://staff.washington.edu/dittrich/misc/faqs/rootkits.faq>
- [16] Chkrootkit, 9-1-2006, <http://www.chkrootkit.org>
- [17] Detecting and Understanding rootkits, an Introduction and just a little-bit-more, 9-1-2006, [http://www.net-security.org/dl/articles/Detecting\\_and\\_Understanding\\_rootkits.txt](http://www.net-security.org/dl/articles/Detecting_and_Understanding_rootkits.txt)
- [18] kern\_check.c, 9-1-2006, [http://la-samhna.de/library/kern\\_check.c](http://la-samhna.de/library/kern_check.c)
- [19] Kernel Rootkits, 9-1-2006, [http://www.sans.org/reading\\_room/whitepapers/threats/449.php](http://www.sans.org/reading_room/whitepapers/threats/449.php)
- [20] Rootkit levels of infection and mitigation, 9-1-2006, [http://searchopensource.techtarget.com/tip/1,289483,sid39\\_gci1149598,00.html](http://searchopensource.techtarget.com/tip/1,289483,sid39_gci1149598,00.html)
- [21] Scprint.c, 9-1-2006, [http://jdoe.freeshell.org/howtos/ExitTheMatrix/misc/kernel\\_auditor/scprint.c](http://jdoe.freeshell.org/howtos/ExitTheMatrix/misc/kernel_auditor/scprint.c)
- [22] Wikipedia: Chkrootkit, 2006, <http://en.wikipedia.org/wiki/Chkrootkit>
- [23] Execution path analysis: finding kernel based rootkits, 9-1-2006, <http://doc.bughunter.net/rootkit-backdoor/execution-path.html>
- [24] C. Kruegel, W. Robertson, and G. Vigna, "Detecting kernel-level rootkits through binary analysis," *Proceedings of the 20th Annual Computer Security Applications Conference (ACSACÆ04)*, 2004.
- [25] Komoku Inc., 9-1-2006, <http://www.komoku.com/technology.shtml>
- [26] Government-funded Startup Blasts Rootkits, 9-1-2006, <http://www.eweek.com/article2/0,1759,1951941,00.asp>

- [27] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot-a coprocessor-based kernel runtime integrity monitor," *Proceedings of USENIX Security Symposium*, pp. 179-194, 2004.
- [28] B. Carrier and E. Spafford, "Automated Digital Evidence Target Definition Using Outlier Analysis and Existing Evidence," *Proceedings of the 2005 Digital Forensics Research Workshop*, 2005.
- [29] D. Bovet and M. Cesati, *Understanding the Linux Kernel* O'Reilly & Associates, 2003.
- [30] Open Source Security - Rootkits, 9-1-2006, <http://www.ossec.net/rootkits/>
- [31] Arati Baliga's Web Page @ Rutgers University, 9-1-0006, <http://www.research.rutgers.edu/~aratib/links.html>
- [32] Packetstorm Security, 9-1-2006, <http://packetstorm.linuxsecurity.com/groups/teso/>
- [33] SPARC Options - Using the GNU Compiler Collection, 9-1-2006, <http://www.eweek.com/article2/0,1759,1951941,00.asp>
- [34] Loading Modules on Sparc64, 9-1-2006, <http://www.ussg.iu.edu/hypermil/linux/kernel/0304.3/0479.html>
- [35] Ultra Linux Home Page, 9-1-2006, <http://www.ultralinux.org/>
- [36] Debian Kernel Compile Howto, 9-1-2006, [http://www.projektfarm.com/en/support/howto/debian\\_kernel\\_compile.html](http://www.projektfarm.com/en/support/howto/debian_kernel_compile.html)
- [37] Richard M. Stallman, Roland Pesch, and Stan Shebs, *Debugging with GDB: The GNU Source-Level Debugger*, 9th ed GNU Press, 2002.
- [38] Mike Loukides and Andy Oram, *Programming with GNU Software*, 2nd ed O'Reilly & Associates, 1997.
- [39] R. Love, *Linux kernel development* Sams Indianapolis, Ind, 2004.
- [40] Vic Barneet and Toby Lewis, *Outliers In Statistical Data*, 3rd ed John Wiley & Sons Ltd., 1994.
- [41] Data Mining Survivor - Building Models - Outlier Analysis, 10-4-2006, [http://www.togaware.com/datamining/survivor/Outlier\\_Analysis.html](http://www.togaware.com/datamining/survivor/Outlier_Analysis.html)
- [42] D.M. Hawkins, *Identification Of Outliers* Chapman and Hall, 1980.
- [43] S. Ramaswamy, R. Rastogi, and K. Shim, "Efficient algorithms for mining outliers from large data sets," *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pp. 427-438, 2000.

- [44] E. M. Knorr and R. T. Ng, "Algorithms for mining distance-based outliers in large datasets," *Proceedings of the 24th International Conference on Very Large Databases (VLDB)*, pp. 392-403, 1998.
- [45] E. M. Knorr and R. T. Ng, "Finding intensional knowledge of distance-based outliers," *The VLDB Journal*, pp. 211-222, 1999.
- [46] M. M. Breunig, H. P. Kriegel, R. T. Ng, and J. Sander, "LOF: identifying density-based local outliers," *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pp. 93-104, 2000.
- [47] M. M. Breunig, H. P. Kriegel, R. T. Ng, and J. Sander, "OPTICS-OF: Identifying Local Outliers," *Proceedings of the Third European Conference on Principles of Data Mining and Knowledge Discovery*, pp. 262-270, 1999.
- [48] W. Jin, A. K. H. Tung, and J. Han, "Mining top-n local outliers in large databases," *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 293-298, 2001.
- [49] LWN: Security, 9-1-2006, <http://lwn.net/Articles/75288/?format=printable>
- [50] Syscall Table AKA Hijacking Syscalls, 9-1-2006, <http://lkml.org/lkml/2004/1/3/63>
- [51] Design and Implementation of the Second Extended Filesystem, 9-1-2006, <http://web.mit.edu/tytso/www/linux/ext2intro.html>

**Appendix A**  
**Unmodified SPARC System Call Address Values**

<b>Decimal</b>	<b>Hex</b>
4263080	0x00410ca8
4263440	0x00410e10
4263308	0x00410d8c
4667392	0x00473800
4667744	0x00473960
4665600	0x00473100
4666112	0x00473300
4522112	0x00450080
4665856	0x00473200
4733536	0x00483a60
4732096	0x004834c0
4263028	0x00410c74
4662784	0x00472600
4664224	0x00472ba0
4729120	0x00482920
4663776	0x004729e0
4664320	0x00472c00
4326112	0x004202e0
4329504	0x00421020
4666976	0x00473660
4545888	0x00455d60
4537888	0x00453e20
4538560	0x004540c0
4558592	0x00458f00
4545984	0x00455dc0
4263080	0x00410ca8
4263248	0x00410d50
4545824	0x00455d20
4263096	0x00410cb8
4263080	0x00410ca8
4661920	0x004722a0
4263080	0x00410ca8
4263080	0x00410ca8
4662496	0x004724e0
4492672	0x00448d80
4263080	0x00410ca8
4675008	0x004755c0
4553152	0x004579c0
4706112	0x0047cf40
4595488	0x00461f20
4706272	0x0047cfe0
4742592	0x00485dc0

4263072	0x00410ca0
4560384	0x00459600
4263080	0x00410ca8
4790656	0x00491980
4557792	0x00458be0
4546048	0x00455e00
4554944	0x004580c0
4546016	0x00455de0
4546080	0x00455e20
4537856	0x00453e00
4263088	0x00410cb0
4263080	0x00410ca8
4745312	0x00486860
4556672	0x00458780
4263080	0x00410ca8
4732800	0x00483780
4706560	0x0047d100
4263016	0x00410c68
4563552	0x0045a260
4663328	0x00472820
4706432	0x0047d080
4263080	0x00410ca8
4325408	0x00420020
4598432	0x00462aa0
4263296	0x00410d80
4669024	0x00473e60
4669408	0x00473fe0
4263080	0x00410ca8
4263080	0x00410ca8
4327008	0x00420660
4263080	0x00410ca8
4327424	0x00420800
4607936	0x00464fc0
4600704	0x00463380
4666304	0x004733c0
4263080	0x00410ca8
4601472	0x00463680
4561344	0x004599c0
4561440	0x00459a20
4560992	0x00459860
4263080	0x00410ca8
4524128	0x00450860
4263080	0x00410ca8
4645088	0x0046e0e0
4523680	0x004506a0
4263080	0x00410ca8

4561888	0x00459be0
4263080	0x00410ca8
4742272	0x00485c80
4263080	0x00410ca8
4743808	0x00486280
4750048	0x00487ae0
4263080	0x00410ca8
4675296	0x004756e0
4556192	0x004585a0
6025472	0x005bf100
6026304	0x005bf440
6026048	0x005bf340
4556480	0x004586c0
4263216	0x00410d30
4329216	0x00420f00
4552128	0x004575c0
4552576	0x00457780
4552608	0x004577a0
4553440	0x00457ae0
4263152	0x00410cf0
4558976	0x00459080
4559456	0x00459260
4559552	0x004592c0
4559872	0x00459400
4263080	0x00410ca8
6027968	0x005bfac0
6027552	0x005bf920
4263080	0x00410ca8
4524864	0x00450b40
4563488	0x0045a220
6027328	0x005bf840
4770528	0x0048cae0
4668768	0x00473d60
4668896	0x00473de0
4525184	0x00450c80
4664416	0x00472c60
4663616	0x00472940
6026912	0x005bf6a0
4558080	0x00458d00
4557504	0x00458ac0
4736736	0x004846e0
4661184	0x00471fc0
4661600	0x00472160
4760704	0x0048a480
4263080	0x00410ca8
6026688	0x005bf5c0

6027456	0x005bf8c0
6025568	0x005bf160
4729888	0x00482c20
4731136	0x00483100
4662208	0x004723c0
4263080	0x00410ca8
4263080	0x00410ca8
6026560	0x005bf540
4263080	0x00410ca8
4546112	0x00455e40
4562592	0x00459ea0
4562784	0x00459f60
4796032	0x00492e80
4563616	0x0045a2a0
5801280	0x00588540
5801760	0x00588720
6026432	0x005bf4c0
4263080	0x00410ca8
4263080	0x00410ca8
4751360	0x00488000
4748320	0x00487420
4263080	0x00410ca8
4263080	0x00410ca8
4660640	0x00471da0
4660768	0x00471e20
4790816	0x00491a20
4263080	0x00410ca8
4263080	0x00410ca8
4328192	0x00420b00
4562336	0x00459da0
4328704	0x00420d00
4808512	0x00495f40
4263080	0x00410ca8
4795200	0x00492b40
4696192	0x0047a880
4802112	0x00494640
4802208	0x004946a0
4802304	0x00494700
4802816	0x00494900
4802912	0x00494960
4747968	0x004872c0
4561184	0x00459920
4663040	0x00472700
4803008	0x004949c0
4803424	0x00494b60
4803520	0x00494bc0

4803616	0x00494c20
4804000	0x00494da0
4804096	0x00494e00
4263080	0x00410ca8
4514944	0x0044e480
4560448	0x00459640
4804192	0x00494e60
4553216	0x00457a00
4263080	0x00410ca8
4326848	0x004205c0
4510656	0x0044d3c0
4326944	0x00420620
4263080	0x00410ca8
4263080	0x00410ca8
4263080	0x00410ca8
4263080	0x00410ca8
4263080	0x00410ca8
4545920	0x00455d80
4263080	0x00410ca8
4554816	0x00458040
4554848	0x00458060
4263080	0x00410ca8
4706272	0x0047cfe0
4707808	0x0047d5e0
4263080	0x00410ca8
4595872	0x004620a0
6028416	0x005bfc80
4506240	0x0044c280
4263080	0x00410ca8
4263080	0x00410ca8
4263080	0x00410ca8
4263080	0x00410ca8
4523008	0x00450400
4643296	0x0046d9e0
4524288	0x00450900
4326304	0x004203a0
4263080	0x00410ca8
4263316	0x00410d94
4263080	0x00410ca8
4526720	0x00451280
4263080	0x00410ca8
4510208	0x0044d200
4512704	0x0044dbc0
4515456	0x0044e680
4560832	0x004597c0
4690784	0x00479360

4693728	0x00479ee0
4263080	0x00410ca8
4559968	0x00459460
4560224	0x00459560
4750048	0x00487ae0
4263080	0x00410ca8
4263080	0x00410ca8
4524704	0x00450aa0
4263080	0x00410ca8
4263080	0x00410ca8
4667168	0x00473720
4610048	0x00465800
4610272	0x004658e0
4610624	0x00465a40
4610848	0x00465b20
4493280	0x00448fe0
4493504	0x004490c0
4493248	0x00448fc0
4493312	0x00449000
4493728	0x004491a0
4494080	0x00449300
4494144	0x00449340
4494208	0x00449380
4546144	0x00455e60
4327616	0x004208c0
4531168	0x004523e0
4561024	0x00459880
4675712	0x00475880
4786848	0x00490aa0

**Appendix B**  
**Unmodified Intel System Call Address Values**

<b>Decimal</b>	<b>Hex</b>
3222403552	0xc011f9e0
3222374480	0xc0118850
3222297088	0xc0105a00
3222493152	0xc01357e0
3222493472	0xc0135920
3222491648	0xc0135200
3222491984	0xc0135350
3222375456	0xc0118c20
3222491808	0xc01352a0
3222543024	0xc0141ab0
3222542032	0xc01416d0
3222297232	0xc0105a90
3222489152	0xc0134840
3222376848	0xc0119190
3222539984	0xc0140ed0
3222489968	0xc0134b70
3222413632	0xc0122140
3222403552	0xc011f9e0
3222522256	0xc013c990
3222492752	0xc0135650
3222394528	0xc011d6a0
3222591616	0xc014d880
3222588336	0xc014cbb0
3222413920	0xc0122260
3222414976	0xc0122680
3222376944	0xc01191f0
3222316784	0xc010a6f0
3222394448	0xc011d650
3222522768	0xc013cb90
3222325568	0xc010c940
3222488304	0xc01344f0
3222403552	0xc011f9e0
3222403552	0xc011f9e0
3222488864	0xc0134720
3222351360	0xc0112e00
3222403552	0xc011f9e0
3222498896	0xc0136e50
3222400816	0xc011ef30
3222545536	0xc0142480
3222540576	0xc0141120
3222541408	0xc0141460
3222548416	0xc0142fc0

3222323728	0xc010c210
3222407408	0xc01208f0
3222403552	0xc011f9e0
3222424176	0xc0124a70
3222413824	0xc0122200
3222415072	0xc01226e0
3222402816	0xc011f700
3222415024	0xc01226b0
3222415120	0xc0122710
3222388272	0xc011be30
3222588192	0xc014cb20
3222403552	0xc011f9e0
3222550752	0xc01438e0
3222549264	0xc0143310
3222403552	0xc011f9e0
3222407504	0xc0120950
3222403552	0xc011f9e0
3222325280	0xc010c820
3222410272	0xc0121420
3222489584	0xc01349f0
3222514624	0xc013abc0
3222548208	0xc0142ef0
3222394560	0xc011d6c0
3222407872	0xc0120ac0
3222408000	0xc0120b40
3222298608	0xc0105ff0
3222402704	0xc011f690
3222402736	0xc011f6b0
3222413856	0xc0122220
3222413760	0xc01221c0
3222298192	0xc0105e50
3222402032	0xc011f3f0
3222408640	0xc0120dc0
3222409408	0xc01210c0
3222409264	0xc0121030
3222410208	0xc01213e0
3222377072	0xc0119270
3222377392	0xc01193b0
3222414688	0xc0122560
3222414816	0xc01225e0
3222324304	0xc010c450
3222542528	0xc01418c0
3222522512	0xc013ca90
3222523024	0xc013cc90
3222524016	0xc013d070
3222473296	0xc0130a50

3222404096	0xc011fc00
3222552656	0xc0144050
3222324000	0xc010c320
3222428384	0xc0125ae0
3222486720	0xc0133ec0
3222487168	0xc0134080
3222489808	0xc0134ad0
3222413696	0xc0122180
3222403968	0xc011fb80
3222403728	0xc011fa90
3222403552	0xc011f9e0
3222486224	0xc0133cd0
3222486368	0xc0133d60
3222319440	0xc010b150
3223571920	0xc023cdd0
3222361280	0xc01154c0
3222376320	0xc0118f80
3222375840	0xc0118da0
3222522384	0xc013ca10
3222522640	0xc013cb10
3222522896	0xc013cc10
3222325136	0xc010c790
3222319712	0xc010b260
3222492080	0xc01353b0
3222403552	0xc011f9e0
3222311568	0xc0109290
3222374512	0xc0118870
3222472144	0xc01305d0
3222376560	0xc0119070
3222324448	0xc010c4e0
3222499056	0xc0136ef0
3222299200	0xc0106240
3222297136	0xc0105a30
3222408976	0xc0120f10
3222408512	0xc0120d40
3222321296	0xc010b890
3222378736	0xc01198f0
3222448160	0xc012a820
3222402064	0xc011f410
3222364176	0xc0116010
3222364544	0xc0116180
3222366320	0xc0116870
3222368992	0xc01172e0
3222602208	0xc01501e0
3222407776	0xc0120a60
3222489344	0xc0134900

3222511376	0xc0139f10
3222513040	0xc013a590
3222359872	0xc0114f40
3222403552	0xc011f9e0
3222414624	0xc0122520
3222414656	0xc0122540
3222492928	0xc0135700
3222552992	0xc01441a0
3222554816	0xc01448c0
3222564864	0xc0147000
3222440496	0xc0128a30
3222494480	0xc0135d10
3222494608	0xc0135d90
3222407904	0xc0120ae0
3222499408	0xc0137050
3222382032	0xc011a5d0
3222449792	0xc012ae80
3222449984	0xc012af40
3222450272	0xc012b060
3222450432	0xc012b100
3222351952	0xc0113050
3222352096	0xc01130e0
3222351904	0xc0113020
3222352000	0xc0113080
3222352272	0xc0113190
3222352432	0xc0113230
3222352480	0xc0113260
3222352528	0xc0113290
3222394736	0xc011d770
3222452832	0xc012ba60
3222413952	0xc0122280
3222414032	0xc01222d0
3222311856	0xc01093b0
3222368560	0xc0117130
3222556416	0xc0144f00
3222728448	0xc016ef00
3222414288	0xc01223d0
3222414368	0xc0122420
3222410304	0xc0121440
3222299456	0xc0106340
3222402448	0xc011f590
3222399376	0xc011e990
3222399984	0xc011ebf0
3222400016	0xc011ec10
3222401056	0xc011f020
3222298352	0xc0105ef0

3222494736	0xc0135e10
3222495056	0xc0135f50
3222413568	0xc0122100
3222573152	0xc0149060
3222388288	0xc011be40
3222388848	0xc011c070
3222298848	0xc01060e0
3222438080	0xc01280c0
3222403552	0xc011f9e0
3222403552	0xc011f9e0
3222297184	0xc0105a60
3222409168	0xc0120fd0
3222323824	0xc010c270
3222487568	0xc0134210
3222488000	0xc01343c0
3222523488	0xc013ce60
3222523616	0xc013cee0
3222523744	0xc013cf60
3222490528	0xc0134da0
3222394592	0xc011d6e0
3222394656	0xc011d720
3222394624	0xc011d700
3222394688	0xc011d740
3222405136	0xc0120010
3222404672	0xc011fe40
3222408112	0xc0120bb0
3222408224	0xc0120c20
3222490624	0xc0134e00
3222405984	0xc0120360
3222406496	0xc0120560
3222406672	0xc0120610
3222406928	0xc0120710
3222490432	0xc0134d40
3222405616	0xc01201f0
3222404896	0xc011ff20
3222407104	0xc01207c0
3222407296	0xc0120880
3222592160	0xc014daa0
3222443120	0xc0129470
3222442464	0xc01291e0
3222553456	0xc0144370
3222549360	0xc0143370
3222403552	0xc011f9e0
3222403552	0xc011f9e0
3222394720	0xc011d760
3222438560	0xc01282a0



**Appendix C**  
**Rkit System Call Table Results – Intel**

<b>Decimal</b>	<b>Hex</b>	<b>Modified</b>
3222403552	0xc011f9e0	0
3222374480	0xc0118850	0
3222297088	0xc0105a00	0
3222493152	0xc01357e0	0
3222493472	0xc0135920	0
3222491648	0xc0135200	0
3222491984	0xc0135350	0
3222375456	0xc0118c20	0
3222491808	0xc01352a0	0
3222543024	0xc0141ab0	0
3222542032	0xc01416d0	0
3222297232	0xc0105a90	0
3222489152	0xc0134840	0
3222376848	0xc0119190	0
3222539984	0xc0140ed0	0
3222489968	0xc0134b70	0
3222413632	0xc0122140	0
3222403552	0xc011f9e0	0
3222522256	0xc013c990	0
3222492752	0xc0135650	0
3222394528	0xc011d6a0	0
3222591616	0xc014d880	0
3222588336	0xc014cbb0	0
3498541152	0xd0878060	1
3222414976	0xc0122680	0
3222376944	0xc01191f0	0
3222316784	0xc010a6f0	0
3222394448	0xc011d650	0
3222522768	0xc013cb90	0
3222325568	0xc010c940	0
3222488304	0xc01344f0	0
3222403552	0xc011f9e0	0
3222403552	0xc011f9e0	0
3222488864	0xc0134720	0
3222351360	0xc0112e00	0
3222403552	0xc011f9e0	0
3222498896	0xc0136e50	0
3222400816	0xc011ef30	0
3222545536	0xc0142480	0
3222540576	0xc0141120	0
3222541408	0xc0141460	0
3222548416	0xc0142fc0	0

3222323728	0xc010c210	0
3222407408	0xc01208f0	0
3222403552	0xc011f9e0	0
3222424176	0xc0124a70	0
3222413824	0xc0122200	0
3222415072	0xc01226e0	0
3222402816	0xc011f700	0
3222415024	0xc01226b0	0
3222415120	0xc0122710	0
3222388272	0xc011be30	0
3222588192	0xc014cb20	0
3222403552	0xc011f9e0	0
3222550752	0xc01438e0	0
3222549264	0xc0143310	0
3222403552	0xc011f9e0	0
3222407504	0xc0120950	0
3222403552	0xc011f9e0	0
3222325280	0xc010c820	0
3222410272	0xc0121420	0
3222489584	0xc01349f0	0
3222514624	0xc013abc0	0
3222548208	0xc0142ef0	0
3222394560	0xc011d6c0	0
3222407872	0xc0120ac0	0
3222408000	0xc0120b40	0
3222298608	0xc0105ff0	0
3222402704	0xc011f690	0
3222402736	0xc011f6b0	0
3222413856	0xc0122220	0
3222413760	0xc01221c0	0
3222298192	0xc0105e50	0
3222402032	0xc011f3f0	0
3222408640	0xc0120dc0	0
3222409408	0xc01210c0	0
3222409264	0xc0121030	0
3222410208	0xc01213e0	0
3222377072	0xc0119270	0
3222377392	0xc01193b0	0
3222414688	0xc0122560	0
3222414816	0xc01225e0	0
3222324304	0xc010c450	0
3222542528	0xc01418c0	0
3222522512	0xc013ca90	0
3222523024	0xc013cc90	0
3222524016	0xc013d070	0
3222473296	0xc0130a50	0

3222404096	0xc011fc00	0
3222552656	0xc0144050	0
3222324000	0xc010c320	0
3222428384	0xc0125ae0	0
3222486720	0xc0133ec0	0
3222487168	0xc0134080	0
3222489808	0xc0134ad0	0
3222413696	0xc0122180	0
3222403968	0xc011fb80	0
3222403728	0xc011fa90	0
3222403552	0xc011f9e0	0
3222486224	0xc0133cd0	0
3222486368	0xc0133d60	0
3222319440	0xc010b150	0
3223571920	0xc023cdd0	0
3222361280	0xc01154c0	0
3222376320	0xc0118f80	0
3222375840	0xc0118da0	0
3222522384	0xc013ca10	0
3222522640	0xc013cb10	0
3222522896	0xc013cc10	0
3222325136	0xc010c790	0
3222319712	0xc010b260	0
3222492080	0xc01353b0	0
3222403552	0xc011f9e0	0
3222311568	0xc0109290	0
3222374512	0xc0118870	0
3222472144	0xc01305d0	0
3222376560	0xc0119070	0
3222324448	0xc010c4e0	0
3222499056	0xc0136ef0	0
3222299200	0xc0106240	0
3222297136	0xc0105a30	0
3222408976	0xc0120f10	0
3222408512	0xc0120d40	0
3222321296	0xc010b890	0
3222378736	0xc01198f0	0
3222448160	0xc012a820	0
3222402064	0xc011f410	0
3222364176	0xc0116010	0
3222364544	0xc0116180	0
3222366320	0xc0116870	0
3222368992	0xc01172e0	0
3222602208	0xc01501e0	0
3222407776	0xc0120a60	0
3222489344	0xc0134900	0

3222511376	0xc0139f10	0
3222513040	0xc013a590	0
3222359872	0xc0114f40	0
3222403552	0xc011f9e0	0
3222414624	0xc0122520	0
3222414656	0xc0122540	0
3222492928	0xc0135700	0
3222552992	0xc01441a0	0
3222554816	0xc01448c0	0
3222564864	0xc0147000	0
3222440496	0xc0128a30	0
3222494480	0xc0135d10	0
3222494608	0xc0135d90	0
3222407904	0xc0120ae0	0
3222499408	0xc0137050	0
3222382032	0xc011a5d0	0
3222449792	0xc012ae80	0
3222449984	0xc012af40	0
3222450272	0xc012b060	0
3222450432	0xc012b100	0
3222351952	0xc0113050	0
3222352096	0xc01130e0	0
3222351904	0xc0113020	0
3222352000	0xc0113080	0
3222352272	0xc0113190	0
3222352432	0xc0113230	0
3222352480	0xc0113260	0
3222352528	0xc0113290	0
3222394736	0xc011d770	0
3222452832	0xc012ba60	0
3222413952	0xc0122280	0
3222414032	0xc01222d0	0
3222311856	0xc01093b0	0
3222368560	0xc0117130	0
3222556416	0xc0144f00	0
3222728448	0xc016ef00	0
3222414288	0xc01223d0	0
3222414368	0xc0122420	0
3222410304	0xc0121440	0
3222299456	0xc0106340	0
3222402448	0xc011f590	0
3222399376	0xc011e990	0
3222399984	0xc011ebf0	0
3222400016	0xc011ec10	0
3222401056	0xc011f020	0
3222298352	0xc0105ef0	0

3222494736	0xc0135e10	0
3222495056	0xc0135f50	0
3222413568	0xc0122100	0
3222573152	0xc0149060	0
3222388288	0xc011be40	0
3222388848	0xc011c070	0
3222298848	0xc01060e0	0
3222438080	0xc01280c0	0
3222403552	0xc011f9e0	0
3222403552	0xc011f9e0	0
3222297184	0xc0105a60	0
3222409168	0xc0120fd0	0
3222323824	0xc010c270	0
3222487568	0xc0134210	0
3222488000	0xc01343c0	0
3222523488	0xc013ce60	0
3222523616	0xc013cee0	0
3222523744	0xc013cf60	0
3222490528	0xc0134da0	0
3222394592	0xc011d6e0	0
3222394656	0xc011d720	0
3222394624	0xc011d700	0
3222394688	0xc011d740	0
3222405136	0xc0120010	0
3222404672	0xc011fe40	0
3222408112	0xc0120bb0	0
3222408224	0xc0120c20	0
3222490624	0xc0134e00	0
3222405984	0xc0120360	0
3222406496	0xc0120560	0
3222406672	0xc0120610	0
3222406928	0xc0120710	0
3222490432	0xc0134d40	0
3222405616	0xc01201f0	0
3222404896	0xc011ff20	0
3222407104	0xc01207c0	0
3222407296	0xc0120880	0
3222592160	0xc014daa0	0
3222443120	0xc0129470	0
3222442464	0xc01291e0	0
3222553456	0xc0144370	0
3222549360	0xc0143370	0
3222403552	0xc011f9e0	0
3222403552	0xc011f9e0	0
3222394720	0xc011d760	0
3222438560	0xc01282a0	0



**Appendix D**  
**Knark System Call Table Results – Intel**

Decimal	Hex	Modified
3222403552	0xc011f9e0	0
3222374480	0xc0118850	0
3498542920	0xd0878748	1
3498543752	0xd0878a88	1
3222493472	0xc0135920	0
3222491648	0xc0135200	0
3222491984	0xc0135350	0
3222375456	0xc0118c20	0
3222491808	0xc01352a0	0
3222543024	0xc0141ab0	0
3222542032	0xc01416d0	0
3498544872	0xd0878ee8	1
3222489152	0xc0134840	0
3222376848	0xc0119190	0
3222539984	0xc0140ed0	0
3222489968	0xc0134b70	0
3222413632	0xc0122140	0
3222403552	0xc011f9e0	0
3222522256	0xc013c990	0
3222492752	0xc0135650	0
3222394528	0xc011d6a0	0
3222591616	0xc014d880	0
3222588336	0xc014cbb0	0
3222413920	0xc0122260	0
3222414976	0xc0122680	0
3222376944	0xc01191f0	0
3222316784	0xc010a6f0	0
3222394448	0xc011d650	0
3222522768	0xc013cb90	0
3222325568	0xc010c940	0
3222488304	0xc01344f0	0
3222403552	0xc011f9e0	0
3222403552	0xc011f9e0	0
3222488864	0xc0134720	0
3222351360	0xc0112e00	0
3222403552	0xc011f9e0	0
3222498896	0xc0136e50	0
3498543128	0xd0878818	1
3222545536	0xc0142480	0
3222540576	0xc0141120	0
3222541408	0xc0141460	0
3222548416	0xc0142fc0	0

3222323728	0xc010c210	0
3222407408	0xc01208f0	0
3222403552	0xc011f9e0	0
3222424176	0xc0124a70	0
3222413824	0xc0122200	0
3222415072	0xc01226e0	0
3222402816	0xc011f700	0
3222415024	0xc01226b0	0
3222415120	0xc0122710	0
3222388272	0xc011be30	0
3222588192	0xc014cb20	0
3222403552	0xc011f9e0	0
3498543252	0xd0878894	1
3222549264	0xc0143310	0
3222403552	0xc011f9e0	0
3222407504	0xc0120950	0
3222403552	0xc011f9e0	0
3222325280	0xc010c820	0
3222410272	0xc0121420	0
3222489584	0xc01349f0	0
3222514624	0xc013abc0	0
3222548208	0xc0142ef0	0
3222394560	0xc011d6c0	0
3222407872	0xc0120ac0	0
3222408000	0xc0120b40	0
3222298608	0xc0105ff0	0
3222402704	0xc011f690	0
3222402736	0xc011f6b0	0
3222413856	0xc0122220	0
3222413760	0xc01221c0	0
3222298192	0xc0105e50	0
3222402032	0xc011f3f0	0
3222408640	0xc0120dc0	0
3222409408	0xc01210c0	0
3222409264	0xc0121030	0
3222410208	0xc01213e0	0
3222377072	0xc0119270	0
3498544528	0xd0878d90	1
3222414688	0xc0122560	0
3222414816	0xc01225e0	0
3222324304	0xc010c450	0
3222542528	0xc01418c0	0
3222522512	0xc013ca90	0
3222523024	0xc013cc90	0
3222524016	0xc013d070	0
3222473296	0xc0130a50	0

3222404096	0xc011fc00	0
3222552656	0xc0144050	0
3222324000	0xc010c320	0
3222428384	0xc0125ae0	0
3222486720	0xc0133ec0	0
3222487168	0xc0134080	0
3222489808	0xc0134ad0	0
3222413696	0xc0122180	0
3222403968	0xc011fb80	0
3222403728	0xc011fa90	0
3222403552	0xc011f9e0	0
3222486224	0xc0133cd0	0
3222486368	0xc0133d60	0
3222319440	0xc010b150	0
3223571920	0xc023cdd0	0
3222361280	0xc01154c0	0
3222376320	0xc0118f80	0
3222375840	0xc0118da0	0
3222522384	0xc013ca10	0
3222522640	0xc013cb10	0
3222522896	0xc013cc10	0
3222325136	0xc010c790	0
3222319712	0xc010b260	0
3222492080	0xc01353b0	0
3222403552	0xc011f9e0	0
3222311568	0xc0109290	0
3222374512	0xc0118870	0
3222472144	0xc01305d0	0
3222376560	0xc0119070	0
3222324448	0xc010c4e0	0
3222499056	0xc0136ef0	0
3222299200	0xc0106240	0
3498543024	0xd08787b0	1
3222408976	0xc0120f10	0
3222408512	0xc0120d40	0
3222321296	0xc010b890	0
3222378736	0xc01198f0	0
3222448160	0xc012a820	0
3222402064	0xc011f410	0
3222364176	0xc0116010	0
3222364544	0xc0116180	0
3222366320	0xc0116870	0
3222368992	0xc01172e0	0
3222602208	0xc01501e0	0
3222407776	0xc0120a60	0
3222489344	0xc0134900	0

3222511376	0xc0139f10	0
3222513040	0xc013a590	0
3222359872	0xc0114f40	0
3222403552	0xc011f9e0	0
3222414624	0xc0122520	0
3222414656	0xc0122540	0
3222492928	0xc0135700	0
3498542232	0xd0878498	1
3222554816	0xc01448c0	0
3222564864	0xc0147000	0
3222440496	0xc0128a30	0
3222494480	0xc0135d10	0
3222494608	0xc0135d90	0
3222407904	0xc0120ae0	0
3222499408	0xc0137050	0
3222382032	0xc011a5d0	0
3222449792	0xc012ae80	0
3222449984	0xc012af40	0
3222450272	0xc012b060	0
3222450432	0xc012b100	0
3222351952	0xc0113050	0
3222352096	0xc01130e0	0
3222351904	0xc0113020	0
3222352000	0xc0113080	0
3222352272	0xc0113190	0
3222352432	0xc0113230	0
3222352480	0xc0113260	0
3222352528	0xc0113290	0
3222394736	0xc011d770	0
3222452832	0xc012ba60	0
3222413952	0xc0122280	0
3222414032	0xc01222d0	0
3222311856	0xc01093b0	0
3222368560	0xc0117130	0
3222556416	0xc0144f00	0
3222728448	0xc016ef00	0
3222414288	0xc01223d0	0
3222414368	0xc0122420	0
3222410304	0xc0121440	0
3222299456	0xc0106340	0
3222402448	0xc011f590	0
3222399376	0xc011e990	0
3222399984	0xc011ebf0	0
3222400016	0xc011ec10	0
3222401056	0xc011f020	0
3222298352	0xc0105ef0	0

3222494736	0xc0135e10	0
3222495056	0xc0135f50	0
3222413568	0xc0122100	0
3222573152	0xc0149060	0
3222388288	0xc011be40	0
3222388848	0xc011c070	0
3222298848	0xc01060e0	0
3222438080	0xc01280c0	0
3222403552	0xc011f9e0	0
3222403552	0xc011f9e0	0
3222297184	0xc0105a60	0
3222409168	0xc0120fd0	0
3222323824	0xc010c270	0
3222487568	0xc0134210	0
3222488000	0xc01343c0	0
3222523488	0xc013ce60	0
3222523616	0xc013cee0	0
3222523744	0xc013cf60	0
3222490528	0xc0134da0	0
3222394592	0xc011d6e0	0
3222394656	0xc011d720	0
3222394624	0xc011d700	0
3222394688	0xc011d740	0
3222405136	0xc0120010	0
3222404672	0xc011fe40	0
3222408112	0xc0120bb0	0
3222408224	0xc0120c20	0
3222490624	0xc0134e00	0
3222405984	0xc0120360	0
3222406496	0xc0120560	0
3222406672	0xc0120610	0
3222406928	0xc0120710	0
3222490432	0xc0134d40	0
3222405616	0xc01201f0	0
3222404896	0xc011ff20	0
3222407104	0xc01207c0	0
3222407296	0xc0120880	0
3222592160	0xc014daa0	0
3222443120	0xc0129470	0
3222442464	0xc01291e0	0
3498542548	0xd08785d4	1
3222549360	0xc0143370	0
3222403552	0xc011f9e0	0
3222403552	0xc011f9e0	0
3222394720	0xc011d760	0
3222438560	0xc01282a0	0

