On a μ-Kernel Based System Architecture Enabling Recovery from Rootkits

Julian B. Grizzard, Henry L. Owen
{*grizzard, owen*}*@ece.gatech.edu*
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia, 30332-0250, USA

Abstract

We present a system architecture called spine *that supports detection and recovery from many kernel-level and user-level rootkits. The architecture forms a reliable basis for an intrusion recovery system (IRS). The spine architecture is a multi-tiered approach, relying on the integrity of a small μ-kernel based hypervisor for correctness at the base level. Spine vertebrae are positioned at each level in the system in order to overcome the semantic gap in the understanding of system state. We discuss the design of the system, highlighting the main advantages and disadvantages from other approaches. A series of attacks are conducted against the prototype system in order to test for correctness and time to recover. Finally, some system performance benchmarks are presented that show that a small performance penalty is incurred from the increased reliability.*

Keywords: Operating Systems, Integrity, Rootkits, Recovery

1. Introduction

Society has become dependent on computer systems as another pillar of our critical infrastructure, and this dependence will continue to increase for the foreseeable future. With this dependence comes the need to protect and defend our computer systems. Various motives coupled with countless accessible and vulnerable computers have spurred numerous attacks. We have begun to address the security needs of our computers building intrusion detection systems (IDS), intrusion prevention systems (IPS), releasing system patches, and taking other evasive actions. In this paper, we argue that in addition to developing methods for prevention and detection of attacks, methods for verifying integrity and recovering from attacks should also be developed. We present a system architecture that is suitable for integrity verification and recovery techniques against one of the more difficult types of malware, which is rootkits. We term the system we envision an intrusion recovery system (IRS). However, we only begin to unveil the vision of an IRS in this work.

A *rootkit* is a set of tools used by an attacker to maintain access to a system and hide activities. A rootkit does not give an attacker the ability to break into an uncompromised system. Instead, the attacker uses a rootkit *after* he or she has gained access to a system. Typically, the attacker will need escalated privileges on the system in order to install a rootkit. Rootkits have been under development for years and have continued to increase in sophistication. We classify rootkits into one of two categories: *user-level* rootkits and *kernel-level* rootkits.

User-level rootkits can be considered the first generation of rootkits and typically replace system binaries such as *ls*, *ps*, and *netstat* with malicious binaries that appear as legitimate binaries. Careful examination of the replaced binaries reveals that they are in fact malicious in nature in that they hide files, processes, or network connections of the attacker's choosing. Some levels of sophistication have been achieved with user-level rootkits, such as matching timestamps, inode numbers, file sizes, and checksums with the original files or redirecting examination tools to hidden copies of the original files. However, *user-level* rootkits are now fairly easy to detect and the attacker's escalation in the arms race for such rootkits has little room for improvement.

Kernel-level rootkits can be considered second generation rootkits. These rootkits will insert malicious hooks into the running kernel code. For instance, they will redirect system calls, modify the virtual file system layer, or modify kernel data structures. What makes these rootkits more difficult to detect is that there is not a widespread availability of kernel integrity checkers. Tools and techniques exists but are not presently widely deployed. We have developed methods of manually detecting and recovering from some of these type of rootkits previously [1].

Rootkits are often installed on high end servers, on which an attacker has gained privileged access. End user systems are also targets, and some worms have included a rootkit in their payload. We believe that rootkits will be a serious threat for the future and will continue to target servers and end user systems. Presently, conventionally wisdom states that once a machine has been compromised, it should be completely wiped clean and reinstalled. The reasoning behind this argument is that there is no way to know for sure that a comprehensive cleanup of the system has been conducted. Complete reinstallation of systems may not be the most cost effective manner to recover from rootkit installations. Therefore, we suggest an alternative method to complete reinstallation, which is to repair the damage done by the rootkit and then verify the integrity of the system. We present a system architecture that is suitable for systems such as servers and end user systems.

1.1. Overview

In section 2, we present the reasoning behind our architectural design choices and suggest what we believe are the minimal requirements of a hypervisor for our system. Next, we discuss some of the details of our design in section 3. We provide an overview of our prototype in section 4. We present the rootkits we have used in forming an attack benchmark against our system in section 5. We show some performance results of our IRS as compared to a native system and a native μ-kernel based system in section 6. In section 7, we provide an overview of related work that has led to our presented architecture and highlight some of the advantages and disadvantages of our approach. Finally, we draw conclusions in section 8.

2. Architecture Reasoning

A rootkit is designed to hide the state α, state associated with the attacker's activities, and the state ρ, state associated with the rootkit itself. Further, in a system with state σ, the rootkit will conceivably modify any state in σ in order to hide the state of α and ρ. It is noteworthy that α and ρ are subsets of σ. Given this arrangement, it is important to design an architecture that supports a state λ, which is isolated from state σ and has the capability to verify the integrity of σ.

A *hypervisor* is a small layer that runs below the operating system, directly on the hardware, providing one method for obtaining the desired isolation. Litty discusses the use of a hypervisor for an IDS [2]. We extend this notion for our IRS system and further specify the requirements. First, we believe that the hypervisor should provide minimal mechanisms sufficient to guarantee isolation and not sacrifice significant performance. A hypervisor that meets these requirements is a μ-kernel. The performance of μ-kernels has been of debate in past literature [3–5]. Liedtke discusses how μ-kernels can achieve good performance and that beliefs to the contrary are not necessarily true [5]. Further, Liedtke suggest three minimal requirements for a μ-kernel in [5] as described below.

- *Address Space:* The μ-kernel is responsible for managing address spaces. Three operations *grant*, *map*, and *flush* are described so that memory can be managed with good flexibility. The μ-kernel must enforce this management so as to protect its own address space; however, the μ-kernel can grant or map memory to a user space memory manager and flush access rights if necessary.

- *Threads and Interprocess Communication:* Threads are tied to address spaces, and so basic thread support must be handled by the μ-kernel. Further, cross-address-space communication must also be handled by the μ-kernel.

- *Unique Identifiers:* Each task must have a unique identification for efficient communication.

Our recovery system is based on Liedtke's principals of a μ-kernel. Figure 1 shows an overview of the architecture. The μ-kernel is the only component that runs in the kernel space or the privileged execution mode. It runs directly on the hardware providing a thin interface to the guest kernel. The guest kernel and all processes it supports, P1 through PN, run in user space at the unprivileged execution mode. On the right side of the figure is the vertically integrated IRS. We term this architecture the *spine* architecture because the components of the IRS, called the vertebrae, seep throughout the host as noted by V0 through V4 in the figure.

One of our assumptions is that V1 will not be compromised by the attacker. It is difficult to prove V1 is immutable; however, we have designed the system toward the goal of achieving this immutability from the perspective of the guest system. The reasoning of how our approach can reach this goal is deduced from code size, simplicity, and limited interface. The code size of the μ-kernel is small, on the order of 20,000 lines of code. Further, we believe the μ-kernel is as simple as possible while achieving reasonable performance and strict isolation. Finally, there is a small interface that the μ-kernel provides to tasks, which is on the order of 10 system calls.

Although we base our architecture on Liedtke's suggested minimal requirements for a μ-kernel, we add one addition requirement as described below.

- *Task Control:* Task control includes the ability to inspect and modify another task's control block, which includes CPU registers. Specifically, it is important to be able to inspect another task's program counter. V1 should export this system call to V2 so that V2 can verify that the guest kernel is operating correctly.

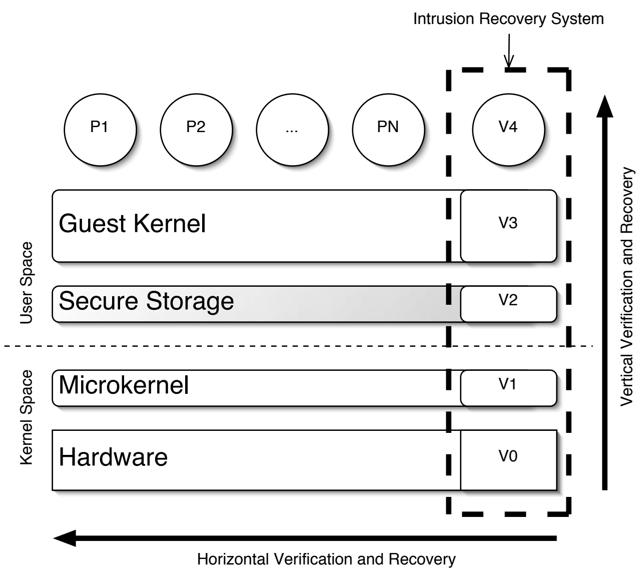
Intrusion Recovery System
P1
P2
...
PN
V4
Guest Kernel
V3
Secure Storage
V2
Microkernel
V1
Hardware
V0
User Space
Kernel Space
Vertical Verification and Recovery
Horizontal Verification and Recovery

Figure 1. Overview of architecture

We further specify three requirements for the V2 component. V2 should provide secure storage and protect the guest kernel/V2 component, which includes verification and repair if necessary.

2.1. Secure Storage

Secure storage must be provided by V2 for the IRS. This storage is used for storing a copy of the known good state, called γ, for the entire guest system. For reasoning, consider this mechanism was not provided. Then, the γ must be stored within the guest system as a subset of the state σ. Now, since γ is a subset of σ, a rootkit would be able to hide itself by modifying γ, what the IRS system considers to be known good state.

Each higher level in the IRS system requires read access to γ in order to verify the integrity of its realm. V2 can map γ to the address space of itself, V3, and V4 in order to achieve this goal. V2 must have access to an isolated storage disk in order to maintain persistent state and for large volumes of state such as a copy of the known good file system.

Real systems will not have stagnant state. Therefore, it is important to address the method for updating γ. We believe that there should not be a direct call for updating γ from within σ as this violates the isolation requirement. Instead, we propose the use of cryptographically signed hashes coupled with state upgrades in σ. Each segment of state in σ has a cryptographically signed hash associated with it. If a state change occurs, the IRS would check the state against the cryptographically signed hash, which should also be updated on a legitimate state change. The cryptographically signed hash would exists within σ, visible by the IRS, and the integrity would be guaranteed by the signature. For this approach, the IRS must rely on the authenticity of the signer. Further, the IRS must have previously shared a cryptographic key with the signing authority. This approach is not complete and additional means of updating γ will be explored in future work.

2.2. Guest Kernel/V3 Verification

The *guest kernel* runs directly above the μ-kernel. The V3 component of the IRS exists in the guest kernel. V2 must be able to verify the integrity of the guest kernel and the V3 component. Most of the hardware resources are given to the guest kernel; however, the V1 and V2 components maintain enough control over the hardware that recovery is feasible from most malicious actions taken by the guest kernel.

It is conceivable that V2 could verify the entire state σ. However, a *semantic gap* exists between V2 and σ. For example, it is difficult to interpret guest kernel data structures (e.g. process tables) from the perspective of V2, although possible. Instead of adding this complicated code to V2, we believe a layered IRS approach should be used.

Each vertebrae in the IRS system understands how to verify state at its level. However, with this architecture, portions of the IRS system itself, namely V3 and V4, are vulnerable to attack from within the guest system. Therefore, each layer of the IRS system must be verified for integrity. We assume that V0, V1, and V2 are intact. V2 then must verify that V3 is intact and V3 must verify that V4 is intact. V3 is also responsible for verifying state of the guest kernel, which is difficult to interpret from V2. For example, V2 can easily verify that the guest kernel text is correct as this is well defined, but data structures need access to kernel functions in the guest kernel text for interpretation. Thus, V2 must be able to verify the integrity of V3 in addition to the guest kernel.

2.3. Guest Kernel/V3 Repair

V2 must be able to repair the guest kernel and V3. Thus, in addition to being able to read and verify the state, V2 must be able to write and repair the state. Again, V3 will recursively repair and verify V4 and inspect data structures in the guest kernel.

3. Method Details

We have set forth our reasoning for the spine architecture and described minimal requirements as an extension of Liedtke's μ-kernel. In this section, we describe details of the architecture and present methods and details for coping with the threat of rootkits.

3.1. System Bootup

In this work, we focus on runtime requirements of a system architecture to support an IRS. However, it is important to discuss the bootup process briefly. The bootup process must be secure. The μ-kernel, V1, and V2 must be booted from a read-only medium or equivalent fashion. Further, the known good state of the system should be known at bootup. An example method for achieving a secure boot process is described by Arbaugh et al. in [6]. Other methods may also be suitable for the IRS.

3.2. IRS Levels

The spine architecture consists of four levels. The integrity of the system is verified by the matrix of bottom up verification and horizontal verification. Below, the requirements of each level are discussed.

- *V0:* This level provides hardware support necessary to meet the isolation requirements of the system. Some have suggested that the hardware itself needs additional support to build secure systems [7]. We believe that certain hardware enhancements would strengthen the design of our system, particularly from a performance perspective, and plan to explore such enhancements in future work.

- *V1:* The second level in the IRS is a modification to the μ-kernel to include process control. Specifically, V2 should have support to inspect and repair the guest kernel as needed.

- *V2:* The third level in the IRS is core to the reliability of the system. It resides just above the μ-kernel. V2 is responsible for (a) providing an interface to secure storage for higher levels, (b) verifying the code sections of the guest kernel/V3 for integrity and execution, and (c) repairing the guest kernel/V3 when the integrity or execution are not correct.

- *V3:* The fourth level in the IRS resides in the guest kernel. V3 is responsible for (a) verifying integrity of state for V4 and for the state in the guest kernel that V2 cannot easily interpret, denoted ϕ, and (b) repairing V4 and ϕ if necessary. Examples of state ϕ include page tables, process tables and the corresponding processes, and inserted modules.

- *V4:* The fifth level in the IRS resides as a user task under the guest kernel. V4 is responsible for (a) verifying the integrity of the state in the system that cannot be easily interpreted by V2 or V3, denoted ψ, (b) repairing ψ if necessary, and (c) providing an interface to the user reporting the activity observed by the IRS. The most important state in ψ that V4 is responsible for is the file system.

3.3. Sub-Level Verification and Repair

Integrity chaining consists of a root link that verifies the integrity of the next link which verifies the integrity of the next link and so forth. An important algorithm is the process for verifying the integrity of the next link in the chain. There are two pieces of the algorithm we describe. Given links C1 and C2, where C2 is above C1, C1 must have a copy of the known good state for the code for C2. Then, the first part of the algorithm is to verify the code for C2 is intact and repair if necessary. The second part of the algorithm is to verify that C2 is executing as expected. Our method for this verification is to periodically monitor the scheduler and the instruction pointer for C2.

3.4. Memory Mapping

In order to monitor and repair sublevels, each level must have visibility of the sublevel. Using Liedtke's model of granting, mapping, and flushing memory, we are able to achieve this visibility [5]. Figure 2 shows the memory hierarchy for the system. At the base, the μ-kernel owns all memory. It maps a large portion of the memory to the guest kernel, keeping some memory for internal structures and secure storage. It is important that the memory distributions are mappings in order to retain visibility. The guest kernel can in turn map portions of its memory, represented by the dotted lines pointing to P1 and PN. Note that the guest kernel maps a portion of its memory to the V4 process. Thus, there is a chain of mappings from V1 through V4 so that the IRS has visibility over the entire system.

3.5. Hashing and State Buckets

The intrusion detection portion of our system relies on the ability to verify state. We considered two approaches for this capability. We consider the best approach of the two is to hash the current state and compare against a hash of the known good state. A second method is to compare the working state against the known good state byte by byte. However, the way computer systems will be designed for the near future, memory accesses are much more costly than arithmetic instructions. Therefore, since the hashing method requires roughly half as many memory accesses, hashing is a much more efficient means to verify state. One risk is that an attacker could manipulate state such that the hash does not change; however, this is nearly impossible

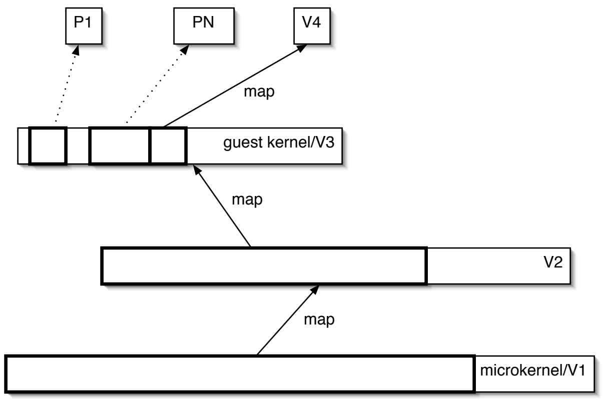

Figure 2. Memory Hierarchy

for good hashing schemes.

One approach to hash all of the state in the system is to divide the state into buckets. For a given state S, where any level L can be responsible for maintaining state S, L divides the state S up in to N buckets. L maintains an independent hash for each bucket. Then, for any bucket in N, L can verify the integrity of that bucket. One of the benefits of comparing state byte by byte is that the exact state that has changed will be detected. When hashing the entire state S, only a binary result specifying the validity of the entire state is computed. However, using the bucket approach, benefits of both hashing and byte by byte comparison can be used.

There are a number of possibilities that arise when using the bucket algorithm to verify state. First, consider that one of the important goals of the system is to maintain high performance or minimize the CPU cycles consumed by the IRS. To achieve this goal, not all of the state in the system can be verified in one sweep as performance concerns such as latency would be harshly affected. By using the bucket algorithm, small sections of the state can be verified quickly in one sweep. This would enables an algorithm in which all buckets are independently and periodically checked over time. Further, in order to thwart an adversary, a random sequence of bucket checking can be conducted.

3.6. Rootkit Repair

The IRS must not only detect an intrusion, but must also recover from the intrusion. We focus on the aspect of recovering from rootkit type intrusions, although there are numerous other factors involved in an intrusion that must be considered for a comprehensive IRS. Most importantly, if a rootkit is installed on a computer, an attacker must have gained access to that computer by some other means. The IRS must stop the intruder from using his previous method for reentry into the system. Below, we discuss methods for recovering from user-level and kernel-level type rootkits.

3.6.1. User-Level Rootkit

Current and future user-level rootkits replace system binaries, add malicious utilities, change configuration files, delete files, or launch uploaded processes. Repairing the damage done by a user-level rootkit is not difficult if the extent of the damage is understood. We believe that the extent of the damage can be understood if a copy of the known good state is available. The algorithm is then to compute the differences between the known good state and the current working state. For each difference, copy the known good state over the current working state. This yields parings of the form action/reaction: <file replaced>/<restore original>, <utility added>/<remove utility>, <config change>/<restore config>, <file deleted>/<restore original>, <malicious process>/<kill process>, and so on.

3.6.2. Kernel-Level Rootkit

Abstractly, kernel-level rootkits are similar to user-level rootkits. They replace presumably good state with malicious state. However, the details of kernel-level rootkits are much more complicated than user-level rootkits. Kernel-level rootkits modify running kernel code, which can drastically effect the stability of the system. Previously seen rootkits will modify the system call table, virtual file system, and kernel data structures. Future rootkits will likely attack these vectors but will also conceivably hide their presence using more sophisticated means. For example, the page table data structures could be modified to redirect the entire kernel such that static checks against the previous kernel addresses would remain valid. For simple redirections, the same algorithm for repairing user-level rootkits can be applied to kernel-level rootkits where example pairings would be: <system call redirected>/<restore original>, <vfs redirected>/<restore original>, <malicious module inserted>/<remove module>, and so on. However, other complications exists with kernel-level rootkits. For instance, memory allocated to a malicious redirection must be reclaimed. As for attacks such as page table redirections, these can be detected and repaired by the V2 component. For more sophisticated structures, the V3 component can periodically perform consistency checking of the data structures . The extent of methods for recovery from all kernel attacks will be the results of future work. One important complication is highly dynamic structures.

```
while(true)
{
   for(i = 0; i < num_buckets; i++)
   {
      delay(sleep_time);
      if(bad_hash(buckets[i])
      {
         repair_bucket(i);
         sleep_time = ADAPT_HIGH;
         count = 0;
      }
   }
   if(sleep_time > ADAPT_LOW)
   {
      if(count > BACKOFF_SCALE)
      {
         sleep_time--;
         count = 0;
      }
   }
   count++;
}
```

Figure 3. Adaptation Algorithm

3.7. Threat Adaptation Model

Under normal operations, the IRS should not tax the system very heavily. Most of the CPU cycles should go to the system. However, in the event of an attack, we believe that the IRS should receive more CPU cycles. While under attack, the most important objective is to recover from the attack. We also believe that the likelihood of detecting intrusions is significantly increased after the initial detection point. Based on these assumptions, we present the algorithm in Figure 3 for monitoring the state of the system.

For simplicity, the presented algorithm shows sequential checking of buckets and represents checking at each level in the system. Under normal operations, sleep_time is set to a reasonable rate so that performance is acceptable. However, in the event that an inconsistency is detected (bad_hash()), first the consistency is repaired, second the sleep time is set to a more adaptive level that is a smaller amount of time, and third the iteration count is reset. After increasing the alert level, the system will remain at that level through BACKOFF_SCALE iterations of checking the entire state. In the event that more inconsistencies occur, the count of iterations will be reset to zero again. If no more inconsistencies occur, then the system will slowly back down to the low adaptive level after (number of alert levels)*BACKOFF_SCALE iterations.

3.8. Limitations

The design and reasoning of our architecture may work for many situations and will add an element of protection. However, there are a number of limitations to our design that have not been fully addressed in this work. First, understanding the known good state of a system is not trivial. Large portions of the system work well for our approach. However, password file changes, legitimate configuration changes, log files, and other dynamic collections of state are problematic. Such files need special attention. Second, an attacker may damage a system so that it is beyond repair, or it may be difficult to repair the damage done without sacrificing the stability of the system. So, there may be circumstances in which the system must be rebuilt, but arguably the proposed system can recover correctly from many intrusion sequences. Finally, our system focuses on recovering from an intrusion after it has occurred. Therefore, it is possible that costly damage is done, such as stolen information, even if the system is able to repair itself.

4. Implementation

Based on our design and reason, we have implemented a large portion of our design and experimented with the performance and recovery capabilities of the system. We have leveraged the work of an implementation of Liedtke's μ-kernel specification, known as Fiasco [8]. This kernel serves as our μ-kernel. Furthermore, a port of the Linux kernel to the μ-kernel architecture has been done [9]. We use this kernel as our guest kernel. Our implementation has been done for the i386 architecture.

We have created basic V1, V2, V3, and V4 components. The V2 component verifies that the guest kernel text is not modified including data structures such as the system call table and the virtual file system structures. The V2 component also has secure storage that is completely isolated from the rest of the machine. The V3 component does some minimal consistency checking. Finally, the V4 component monitors the file system and has the capacity to undo illegitimate changes and some other consistency checks for the process listing verses file system listing. We have only partially implemented the vertical integrity checking, such as checking to make sure the V4 user space process is running.

We have used the sha1 implementation for our hashing algorithms. This hashing algorithm is used in the V2 and V4 layers. The sha1 algorithm may not be the most secure hashing algorithm, as current investigations have claimed, but another hashing algorithm could easily be replaced in our implementation.

Some things that we have not yet implemented include exporting the secure storage interface to the higher layers, implementing persistent storage, and enforcing a secure boot process. Also, some details in the system are more sticky in reality than in theory. For instance, memory management on i386 computers is complicated by various holes and legacy backwards compatible hardware. Although we have run into a number of complicated issues, we do think with reasonable amount of effort it is possible to build a mature and reliable system.

5. Attack Results

In order to test our system against rootkit attacks, we collected a representative assortment of rootkits that exists presently. Our set of rootkits may not cover all possible attacks made by rootkits, but we believe it does cover a majority of known attack techniques in use today. The listing of rootkits we used to develop our test can be seen in Table 1. We choose half kernel-level rootkits and half user-level rootkits with some rootkits having elements of both kernel-level rootkits and user-level rootkits. We attacked the system and observed the recovery capability. Some rootkits did not port to our architecture and so we were unable to test them. All rootkits were detected on the order of minutes after installation. The time to recovery was on the order of seconds.

6. Performance

We have done some performance analysis of our system comparing the following systems: native Linux, L4Linux, L4Linux with spine and low adaptation, L4Linux with spine and high adaptation. L4Linux is the guest kernel implementation that runs on top of the Fiasco μ-kernel. All tests were conducted on a Pentium IV 3 GHz machine with 1 GB of memory. For the spine architecture, we divided the memory evenly between the secure storage area and the guest kernel.

The first test we present shows the adaptive nature of the system. Figure 4 shows a series of impulses over time. Each impulse represents the initialization of an integrity check of the system. At the point where the impulses become solid, a rootkit was installed on the system. No other attacks were made against the system during the shown time frame. As time progresses after the attack, the system slowly adapts back to a normal level of integrity checking.

Figure 5 shows the number of bytes transferred per second using a TCP/IP connection. The system acted as a TCP sink and another identical Linux system served as a TCP source for each test. The TCP source sent packets of length 1500 with no delay for a period of 60 seconds. There was not much performance loss from the native system as compared to the adaptive L4Linux with spine. The difference is 3%. We expected to see a bigger performance loss. However, our current implementation does not schedule V1-V4 with any priority over other events. So, the networking code gets priority and this is why even the adaptive system performs well.

Figure 6 shows the amount of time required to compile a stock Linux kernel. There is a performance lost noticeable in this figure. A loss of 12% is incurred while the system is in its most adaptive state. This performance loss may be acceptable given the higher degree of assurance that the system is operating as expected. Also, the unmodified L4Linux system only incurs an 8% performance loss.

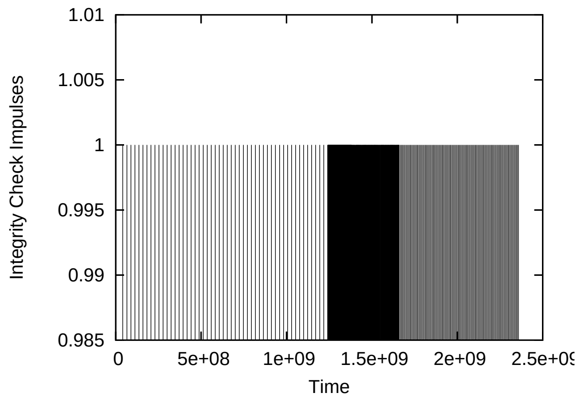

Figure 4. Adaptation after rootkit install

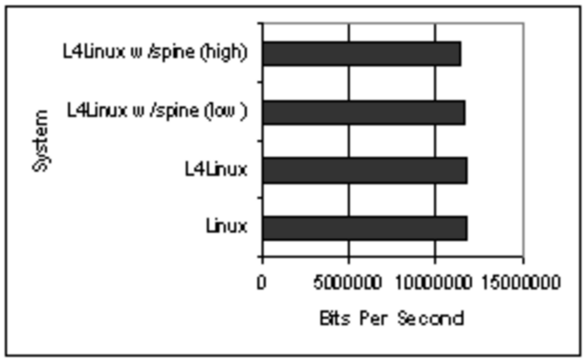

Figure 5. Number of bytes transmitted per second

7. Related Work

In past literature, much work on operating system architecture has focused on performance, flexibility, and extensibility. Security has also been discussed mostly focused on safety. The exokernel was an operating system designed around building a small programmable machine on top of the hardware so that mechanisms provided by the machine could be exported and not policy [4]. These goals are close to what we desire for our architecture. We want the μ-kernel to enforce a policy of integrity but nothing else. The SPIN architecture provides a monolithic approach for flexibility [3]. Although many have argued that a μ-kernel cannot compete with a monolithic kernel in terms of performance, Liedtke shows that these arguments may be based on results from improper implementations [5].

Rootkit	**Type**	**Description**
knark	Kernel-Level	System call table entry redirection.
adore-ng	Kernel-Level	Virtual File System Layer redirection.
sucKIT	Kernel-Level	System call table redirection.
zk	Kernel-Level	System call table redirection.
r.tgz	Kernel/User-Level	Blended rootkit captured on honeynet.
lrk4	User-Level	Replaces binaries; includes sniffer.
lrk5	User-Level	Later version of lrk4.
tOrn	User-Level	Replace binaries; mimics timestamps.
ark	User-Level	Replace binaries; no source code available.

Table 1. Rootkits used to design testing algorithms.

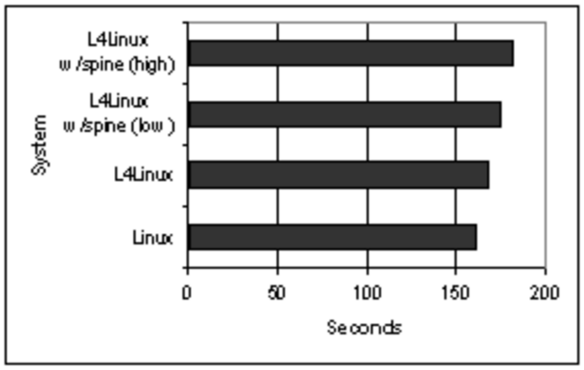

Figure 6. Linux kernel compile time in seconds

More recent literature has discussed virtual machine technologies. Recent virtual machine research focuses on running multiple operating systems on the same machine. Xen is an architecture that scales beyond previous virtual machine architectures by disregarding the need to fully emulate the underlying architecture [10]. A μ-kernel can be used as a virtual machine monitor. Our implementation is based on a port of the Linux kernel to a μ-kernel architecture [9]. The performance results show that the L4Linux kernel approaches the performance achievable by native Linux [11]. We choose a μ-kernel approach over the virtual machine approach because we wanted the simplest architecture that does not sacrifice performance and support for multiple operating systems increases the complexity. However, the virtual machine architecture is a reasonable architecture for our design criteria.

Other work has suggested using virtual machines for secure architectures. Terra is a virtual machine based architecture that suggests applications should be run in different compartments so that they cannot tamper with each others resources [12]. Litty suggest the use of a hypervisor for an intrusion detection system [2], and we build upon the suggestion for our architecture. Arbaugh et al. demonstrate how a system can be booted in a secure and reliable manner through the use of cryptography hash checks for each layer from the BIOS until the system is operational [6]. We believe that some form of a secure boot process such as Arbaugh's must be part of our system.

Candea et al. have discussed the possibility of building systems that are designed to recover rather than building systems designed to never fail [13]. Given that availability is related to mean-time-to-fail and mean-time-to-recover, they point out that the availability of systems could be increased if we reduce the mean-time-to-recover. Conventional recovery methods for recovering from rootkits is complete format and reinstallation. This may not be the quickest mean-time-to-recover; we have presented one possible alternative.

We rely on hashing algorithms for verifying the integrity of state. Tripwire is a notable work that presents the idea of known good state and hashing as a means to verify integrity [14]. Tripwire focuses on integrity of file systems; we extend this notion to the entire system.

Some other recent work has been conducted verifying the integrity of rootkits. Petroni et al. have designed Copilot, which consists of a PCI add-in card that is capable of scanning the hosts memory [15]. Their work focuses on detection of kernel-level rootkits, and reporting any events to a monitoring station via an interface on the PCI card. The advantages of their approach include that they can achieve hard isolation with hardware, do not need to modify the operating system, and do not sacrifice much performance. The disadvantages of their approach include the need for specialized hardware, the lack of visibility inside the OS, and the lack of visibility of CPU registers. We geared our architecture toward a software approach.

8. Conclusions

We have presented a system architecture that can support an IRS. We specifically focused on an architecture that is capable of recovering from rootkits, which we consider to be one of the most difficult types of malware to detect and repair. The specified system is one approach to recovering from a compromise in which a rootkit has been installed. The IRS we have specified and partially implemented is not comprehensive, but it does lay the framework for such a system. Some performance analysis and attack scenarios were carried out on a prototype of the system. It was found that for a small penalty in performance, the system is able to provide a higher level of assurance and reliability.

9. Acknowledgments

We would like to acknowledge the L4 Fiasco group at the Technische Universität Dresden for building the L4 Fiasco μ-kernel and porting the Linux kernel to the L4 architecture. They were also very helpful with answering our questions and debugging our problems. Additionally, we would like to thank Jessica Frame and Andrew Davenport for their thoughts and comments.

References

[1] J. B. Grizzard, J. G. Levine, and H. L. Owen, "Re-establishing trust in compromised systems: Recovering from rootkits that Trojan the system call table," in *Proceedings of 9th European Symposium on Research in Computer Security*, pp. 369–384, Springer, September 2004.

[2] L. Litty, *Hypervisor-Based Intrusion Detection*. PhD thesis, University of Toronto, 2005.

[3] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility, safety and performance in the spin operating system," in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, ACM, December 1995.

[4] D. R. Engler, F. Kaashoek, and J. O. Jr., "Exokernel: An operating system architecture for application-level resource management," in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, ACM, December 1995.

[5] J. Liedtke, "On μ-kernel construction," in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pp. 237–250, ACM, December 1995.

[6] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 65–71, May 1997.

[7] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Architectural Support for Programming Languages and Operating Systems*, pp. 168–177, 2000.

[8] "The fiasco microkernel." `http://os.inf.tu-dresden.de/fiasco/`, September 2004.

[9] "L4linux." `http://os.inf.tu-dresden.de/L4/LinuxOnL4`, November 2004.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pp. 164–177, ACM Press, 2003.

[11] H. Härtig, M. Hohmuth, J. Liedtke, S. Schōnberg, and J. Wolter, "The performance of μ-kernel-based systems," in *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, ACM, December 1997.

[12] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: a virtual machine-based platform for trusted computing," in *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pp. 193–206, ACM Press, 2003.

[13] G. Candea, A. B. Brown, A. Fox, and D. Patterson, "Recovery-oriented computing: Building multitier dependability," *Computer*, vol. 37, no. 11, 2004.

[14] G. H. Kim and E. H. Spafford, "The design and implementation of tripwire: a file system integrity checker," in *ACM Conference on Computer and Communications Security*, pp. 18–29, 1994.

[15] N. L. P. Jr., T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot - a coprocessor-based kernel runtime integrity monitor.," in *USENIX Security Symposium*, pp. 179–194, 2004.