Towards a Homeostatic Operating System

Dissertation Proposal

Anil Somayaji
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131
soma@cs.unm.edu

May 10, 1999

1 The Vision

> You come back home from a long day at work. After shuffling through your regular mail, you check the logs on your home's main server to see if there are any anomalies. It appears your server has had a busy day. Someone attempted to break into your system via *sendmail*; however, the system noticed that *sendmail* was acting strangely, and restarted the daemon. The intruder tried three times before giving up. Also, an agent that claimed to be from a search engine came to your machine, presumably to index your web pages. It tried to delete a few of your files; fortunately, the system noticed that the agent wasn't acting like an indexing bot, and suspended it. Your system asks whether it should report these activities to the authorities; after examining the suspended agent and the source of the *sendmail* attacks, you realize a friend's daughter was the source of both incidents. She must have been playing around with her friends. You say no, and resolve to have a talk with the little prankster and her parents. Luckily no harm was done.

The above story, unfortunately, is a work of fiction. Indeed, life on the Internet today is not nearly so easy. Most current systems cannot notice when they are under attack, and even when they can, they require the assistance of a system administrator to repel intruders. Even equipped with the latest security software, experienced administrators must maintain constant vigilance to maintain the integrity of their networks. Home users are easy targets, especially when connected via high bandwidth connections.

The primary goal of my dissertation is to develop autonomous mechanisms that would enable a computer to respond to anomalies in a way similar to the system described above. My previous work has shown that sequences of system calls can be used to distinguish between the normal and abnormal behavior of privileged processes [17]; my dissertation will build on this work by creating mechanisms that adjust the behavior of a system in near real-time in response to system call anomalies. These adjustments should prevent an intruder from gaining access, and should increase a system's stability in high-stress environments. Specifically, my thesis is the following:

> System security and stability are improved by delaying anomalous system calls, both at the process and the system level.

Although my proposed research is strongly influenced by the human immune system and by mechanisms used by the human body to maintain homeostasis, my primary goal is to make a system that is useful from a computer science, not biological, perspective.

My proposal proceeds as follows. First, I will discuss the motivation for my project, focusing on the limitations of current systems. Next, I will describe some biology that has inspired this project, followed by a description of the proposed research. I will then discuss relevant past work, with comparisons made to the work proposed here. Finally, I will present the outline of my dissertation.

2 Motivation

Computer systems today are remarkably fragile. Consider the following problems:

- Buffer Overflows: Many programs use static buffers for storing and processing user input. Size constraints are frequently not checked, and user input can overflow these buffers. When such a buffer is allocated on the stack, an overflow can result in foreign code being executed. If the program is running with special privileges, such an attack can result in a serious breach of system security.

- Fork Bombs: In UNIX, a process (running program) may duplicate itself by using the fork system call. If a program is buggy or malicious, it may continue to fork recursively until the resources of the system are exhausted. Often, it is possible to limit the resources used by a given user, minimizing the effects of such an attack; however, if such a program is run by the superuser, there is no hope for the system.

  Simple fork bombs are not common in practice; however, many denial-of-service attacks are identical to fork bombs in their effect on a system. Several common server programs fork in response to connections: a new child services each incoming request, while the parent awaits further connections. A flood of incoming connections thus causes a large number of forks, leading to the same symptoms as a a simple fork bomb. Although some servers (such as *sendmail*) try to limit the number of active connections, others do not.

Although these and other vulnerabilities are a problem, they are not the focus of my dissertation. Instead, I am concerned with two aspects of these vulnerabilities: that they can be exploited without warning, and that they can cause damage without resistance. Computers are oblivious to most perturbations of their internal state, even when these perturbations can cause catastrophic failure. For example, current operating systems cannot detect when a privileged program behaves abnormally, even when completely foreign code is being run (such as with a buffer overflow attack). They also cannot notice when there is a dangerous change in resource utilization, even when it is clear that the change indicates a denial-of-service attack is taking place.

Such obliviousness can cause reliability and security problems; however, it does have one advantage: It means that the computer is devoting almost all of its resources to providing services for user programs. Careful design, extensive testing, and an attentive operator all allow an operating system to devote minimal resources to maintaining system integrity. Memory is instantly allocated, CPU cycles are divided fairly, and if the machine starts to behave badly, a human is present to hit the reset button.

This model made sense when computers were weak, isolated, expensive, and rare. Today, however, computers are powerful, interconnected, cheap, and ubiquitous. Now, human attention and skill are limited resources, not processor time and memory. Consider two populations: home users and large business users. Home users now have machines more powerful than workstations of a decade ago. These machines spend most of their time awaiting user input, generally to do simple tasks such as word processing or finance management. Such users do not have the skill to monitor their computers and are often at a loss when something goes wrong. Business users, even though they may have much greater technical and financial resources, are in a similar position. System administrators may be able fix any given machine; however, in general there are too many computers for them to monitor closely. It can even be a struggle to maintain servers, especially when they are under constant attack from the Internet.

To escape this administrative nightmare, we must make survival, not performance, our key design criterion. As we lose the ability to take care of our machines, we must give them the means to take care of themselves. To develop such mechanisms, I believe we should look to biology. Living systems survive in diverse, changing, and often hostile environments without outside assistance; our computers must do the same.

My dissertation is based on the idea that computer operating systems can be augmented with mechanisms similar in spirit to those of biological systems, without overly compromising system performance. These additions should allow a system to monitor and adjust its internal state as circumstances change. To be practical, these mechanisms must be lightweight (on average consuming less than 10% of a system's resources), and should require little, if any, hand-tuning. They should make a computer more reliable, and when failure is unavoidable, should cause a system to degrade gracefully. It is clear when an organism is in pain; similarly, it should be evident when our computers are in distress, alerting a user or administrator that actions need to be taken to avoid a loss of data or service.

3 Biological Inspirations

Before describing the technical basis of my dissertation, it is helpful to examine some biological patterns of organization. Two patterns of interest are the mechanisms of protein regulation within a cell, and the maintenance of homeostasis in multicellular organisms.

3.1 Protein Regulation

Proteins are complex chemicals present in all living cells. They play multiple roles: some serve as structural elements, providing the scaffolding that supports the various subsystems (organelles) of a cell. Others serve as catalysts: Because most chemical reactions inside a cell occur at very low concentrations, most would not occur at all without protein catalysts (enzymes). Thus, events inside a cell are controlled by the presence or absence of different proteins.

Proteins are constructed out of one or more polypeptides. Polypeptides are chains of peptides, and peptides are small chains of amino acids. Amino acids are analogous to Lego bricks, in that they have a uniform mechanism for attachment — the peptide bond. However, each "brick" has a side group that gives each amino acid a distinct chemical personality. There are only twenty amino acids generally used in the human body; these are enough, however, to provide most of the chemical functionality required for cellular metabolism.

Proteins are a resource that must be carefully managed by a cell. While each cell in the human body contains a complete copy of the human genome, only a small subset of this DNA is translated into proteins at any time. This restriction is partly due to functionality constraints; for example, a liver cell should not express proteins specific to nerve cells. However, proteins are selectively produced also because of simple resource limitations; maintaining unneeded proteins would be expensive both in terms of energy and materials.

"Protein allocation" is an extremely dynamic process, with proteins being created and destroyed continuously. This process is managed through a complex set of regulatory relationships. The concentration of many substances (including proteins) affects the concentrations of others. Protein A and sugar X may together stimulate the production of protein B, while sugar Y may suppress the production of protein B. Proteins can even regulate their own production: a protein C can inhibit its own synthesis, resulting in a consistent concentration even if C is being created or destroyed by other processes. These positive and negative feedback interactions are generally nonlinear and are connected through complex, interlocking cycles, known as genetic regulatory networks. Over the course of evolution, these networks have become extremely robust, with relative imbalances in any substance affecting the network so as to correct the imbalance. Furthermore, in the cells of the human body, external signals are integrated with these cycles, allowing relatively few signal molecules to cause complete changes in a cell's metabolism, even causing a cell to commit suicide.

Complex, interconnected regulatory networks allow a cell to monitor its metabolism with extreme precision. Current computer systems, however, are designed to be modular, with interactions between components minimized. By carefully adding regulatory interconnections, it should be possible for a computer to control its internal state with similar precision. It has taken billions of years for biology to produce the networks inside modern cells; I believe that it will take many years of incremental work before our computers are similarly robust. However, I think of my dissertation as an attempt to start computer evolution on this path.

3.2 Homeostatic Mechanisms

A textbook definition of homeostasis is "the maintenance of a relatively stable internal physiological environment or internal equilibrium in an organism" [14, p. G-11]. Loosely, almost any mechanism in biology can be said to contribute to homeostasis; however, homeostasis generally refers to the automatic mechanisms that monitor and adjust an organism's internal environment, irrespective of direct measures of the outside world. Most parts of our body depend on a consistent internal environment for proper functioning; homeostatic mechanisms are there to ensure that these conditions are maintained, even as external conditions and inputs change.

What follows are descriptions of three homeostatic systems: temperature control, fluid balance, and the immune system. Although far from exhaustive, they help illustrate the basic character of biological homeostatic mechanisms.

3.2.1 Temperature Control

The mechanism most commonly associated with homeostasis is the maintenance of a constant internal temperature in warm-blooded animals. Organisms that live in water generally have internal temperatures similar to that of their environment. As water has a high heat capacity, water temperatures tend to be relatively constant; thus, this coupling is generally not a problem. On land, however, temperatures vary wildly, even within a given day. Cold-blooded animals regulate their internal temperature by moving to warmer or colder areas as needed. Although this strategy is quite energy efficient, it has its disadvantages. The rate of all chemical reactions is directly proportional to temperature; thus, as an organism's internal temperature decreases, its metabolic rate also decreases. We can easily observe this relationship in nature: reptiles are slow in the morning, and must wait until later in the day to be active.

Warm-blooded animals do not have this limitation, because they metabolize food to maintain their internal temperature. Birds and mammals are both able to be active at all times of the day in an extremely wide variety of climates. Although we humans now use a variety of technologies to protect our bodies from temperature extremes, our evolutionary heritage has left us with a wide variety of mechanisms for temperature control.

Our bodies detect temperature changes through sensors (specialized nerve cells) in our skin and inside our body. As humans get cold, blood vessels in our extremities constrict, reserving a greater proportion of the body's warmth to our inner core. Shivering is induced, causing muscles to produce additional heat. In some animals, feathers or hairs are made to stand up, increasing the amount of air trapped near the skin, enhancing the insulating properties of the outer layers. In humans, however, this same mechanism simply produces goose bumps. Extended exposure to the cold causes us to eat more food, in response to the increased metabolic demands of heat generation.

Analogous mechanisms happen in response to heat: we sweat, blood vessels in our extremities dilate, and over time we tend to eat less. Note that at the onset of a fever, we have reactions (such as shivering) that are associated with cold. These symptoms occur because our body is attempting to reach a new, higher temperature equilibrium. While very high temperatures (above 40°C) can cause dementia and convulsions, moderately higher temperatures stimulate the immune system and disrupt the functioning of invaders [31, pp. 588–597]. When a fever breaks, we tend to sweat: the danger has passed, and it is time to move back to a normal temperature equilibrium.

The negative effects of a fever help illustrate the need for precise temperature control: the cells of our bodies have evolved to function best at approximately 37°C, and deviations from this temperature tend to cause problems. Temperature regulatory mechanisms help ensure the rest of the body functions as it should.

Although sweating can be effective in removing heat from the body, it is not cost free: sweat is a mixture of water, salts, and body wastes. The wastes need to be removed from the body; however, the loss of water and salts is a more serious issue.

3.2.2 Fluid Balance

Unlike our ancient one-celled ancestors, the cells of our bodies do not have an ocean as a source of water and salts and as a sink for metabolic waste products. Our cells, however, still need an environment similar to that of an ocean; thus, our bodies create and regulate their own in the form of extracellular fluids. These fluids are a significant proportion of our body mass; we are roughly 70% water, and roughly one-third of this water is in the extracellular fluid [14, p. 765]. The most important organs for the regulation of our body's overall fluid composition are the kidneys. They filter the blood continuously, removing metabolic wastes, and balancing water and ion levels. Although our blood is only about 7% of our fluids, it reaches every part of our body, and circulates rapidly; thus, its composition directly regulates the composition of other fluids in the body. Wastes are sent to the bladder, and water and other ions are absorbed or excreted as necessary. Like our lungs, our kidneys are delicate organs, and without them we will quickly die; it makes sense, then, that there are two of them!

While the kidneys are central to maintaining fluid balance, other parts of the body also play key roles. For example, there are pressure sensors in the heart that send signals to the hypothalamus in the brain. If blood pressure decreases, the hypothalamus increases its production of ADH (antidiuretic hormone), causing the kidneys to increase retention of water and salts [31, pp. 493–494]. Thus, overall fluid volume is increased, indirectly increasing blood pressure. Many factors can cause a decrease in blood pressure, but a few interesting ones are hemorrhaging (bleeding),

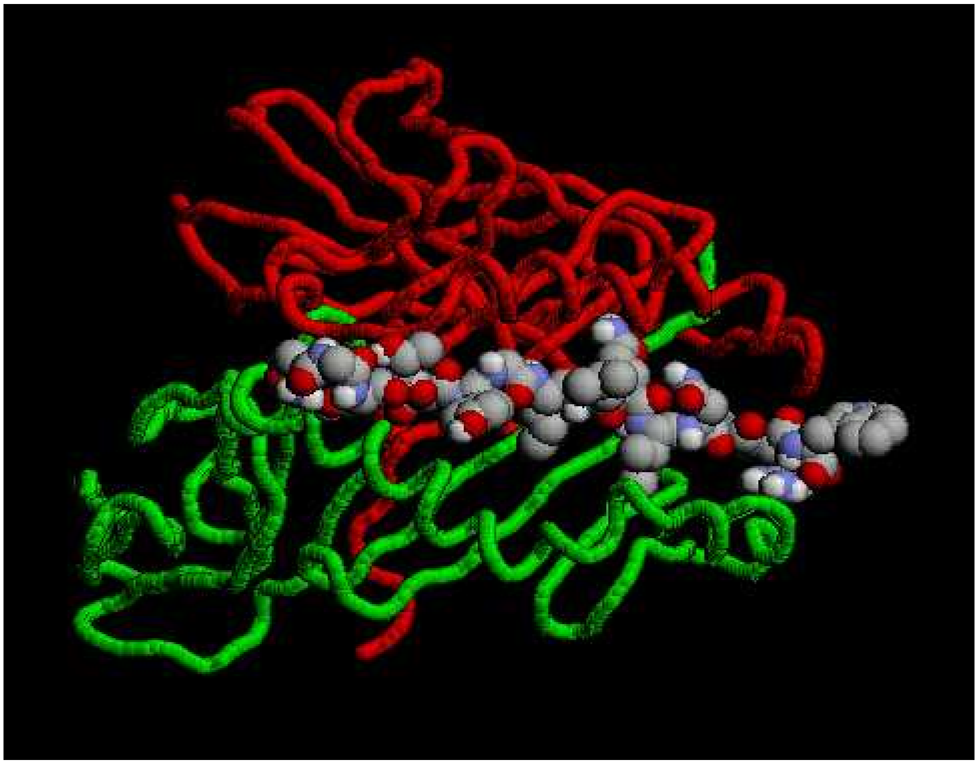

Figure 1: A rendering of a MHC molecule with a bound peptide [34].

dehydration, and congestive heart failure.

3.2.3 Immune System

One system for maintaining homeostasis that has received a great deal of attention in both biological and computer science circles is the immune system. The immune system has been explained extensively elsewhere; in particular, Steve Hofmeyr has written an overview oriented towards the computer scientist [20]. Although the immune system can be seen as analogous to a military defense system, in many ways it is closer in spirit to the homeostatic mechanisms discussed above. Millions of cells roam our bodies, each attempting to detect specific perturbations in the body's internal environment. Cellular damage, unusual cellular behavior, or just unusual chemical compounds can all cause an immune response, much as variations in blood pressure cause changes in urine composition. This immune response must be proportional to the change detected, much as the violence of our shivering depends on how cold we are. Further, the type and severity of an immune response must be balanced against other factors. For example, in a high stress fight-or-flight situation, not only is urinary production reduced through decreased ADH levels; in addition, the immune system is suppressed, freeing energy for our muscles [31, pp. 698–699].

As each cell recycles its proteins, some peptide fragments are not reused immediately. Instead, they are used to inform the immune system about the internal state of the cell. This communication is facilitated by a protein called the major histocompatibility complex, or MHC. MHC has a cleft in its middle, large enough for an 8–10 amino acid peptide fragment to fit in. (See Figure 1.) This cleft must be filled with a peptide fragment in order for MHC to be stable; otherwise, it breaks down into its component polypeptides [11, p. 4:8]. Fragments are transported into areas that are topologically outside of the cell (the lumen of the endoplasmic reticulum or intracellular vesicles), where unbound MHC constituents are stored. When a suitable peptide fragment comes into contact with MHC components, a complete MHC is formed. This MHC is then transported to the cell's surface, to be presented to passing immune system cells called T-cells [11, pp. 4:14–15].

Each T-cell has on its surface receptors that are designed to bind to MHC present on another cell's surface. Different T-cells have different receptors, each specialized to recognize a different MHC/peptide fragment combination. During T-cell development, each T-cell constructs its receptors using a random combinatorial process that can produce 10^{16} different receptors [11, 4:35]. Note that these receptors can bind to almost any peptide fragment. However, during T-cell development T-cells that bind to MHCs containing "normal" peptide fragments are killed. Thus, when roaming the body, if a T-cell is able to bind to a cell's MHCs, that cell must be producing an abnormal protein. As proteins directly control the behavior of a cell, it can be assumed that the target cell is behaving abnormally, either because of a genetic defect, or because it has been invaded by a pathogen such as a virus. Thus, a T-cell, upon finding a target cell with MHC it recognizes, can initiate an immune response against that cell and its neighbors, generally resulting in cell death.

To summarize, organisms maintain homeostasis by using interconnected feedback loops, with different loops monitoring different aspects of an organism's physiological state. These loops are quite complicated, and interconnect many outwardly distinct systems. However, through their actions, living systems are able to maintain relatively consistent internal environments, helping ensure the functioning of the whole organism.

4 Proposed Research

Before the biology described above can help us design better computer systems, we must make an analogy. Although connections can be made at various levels, I would like to start by comparing the relative roles of amino acids and system calls.

On UNIX-like systems, each process executes its code within a private address space. On its own, a process cannot interact with other processes, and it cannot access outside resources, such as disks and network connections. System calls are a mechanism provided by the operating system to allow restricted access to those resources. For example, when a process wants to open a file for reading, it creates an open request in memory, and then executes a special processor instruction that by convention is used for system calls. Control passes to a program called the kernel. The kernel has access to all computer resources. It is responsible for providing the services, such as memory protection and virtual memory, that allow multiple processes to run on a computer. The kernel notes the process that invoked it, checks to see whether it is allowed to perform the operation requested, and if so proceeds to execute it. If this request will take a significant amount of time, CPU control is passed to another process that is ready to run. In this fashion, multiple programs appear to run concurrently on a computer with only one microprocessor.[1]

Amino acids and system calls have somewhat analogous roles in their respective systems: They both provide mechanisms for communication and resource manipulation. System calls are the primary interface allowing processes to access powerful resources. Through patterns of system calls, processes are able to modify files, establish network connections, and interact with users. Amino acids, in the form of proteins, mediate interactions within and between cells. Proteins catalyze chemical reactions within a cell, and are the primary constituent of cell-surface channels and receptors, allowing components and signals to flow into and out of a cell.

[1] The kernel also switches between processes when the currently running process has run for more than a fixed amount of time. This way, even if a process never makes a system call, it cannot monopolize the CPU.

Through the MHC mechanism, the human immune system monitors cellular behavior by observing patterns of expressed peptides (short chains of amino acids). The fundamental insight of this research has been to ask whether we may monitor the behavior of a process in a similar fashion, by examining the short sequences of system calls emitted by it. There was no reason for us to necessarily believe that this somewhat fanciful connection would be useful; however, as our past work has shown, it has proven to be useful indeed [17].

For my dissertation, I am continuing to work with system calls. Now my focus is on connecting patterns of system calls with feedback mechanisms. In the human body, the immune system can kill thousands of cells without causing any outward sign of illness; it seems that execution delays are the simplest mechanisms that can be easily implemented in current computers that are similarly benign in small amounts, but which are potent at higher doses. Thus, I am proposing to have anomalous sequences trigger delays in the execution of system calls. By having the delays be proportional to the number of recent anomalous sequences, I believe that a strangely-behaving process can be stopped in its tracks.

The second part of my dissertation will involve monitoring the overall pattern of system calls. Here the biological connection is somewhat different: each system call is analogous to a resource, such as glucose or potassium ions. Just as a cell must balance its use of different resources, a computer system should balance its use of different system calls. In particular, when a certain system call, such as fork, is being invoked more often than normal, further invocations of that system call should be delayed. By favoring processes that are not executing the overused system calls, the system's overall system call usage pattern should return to normal, maintaining the "homeostasis" of the system.

Although both process and system level system call monitoring require similar implementation strategies, fundamentally they address different problems. Process-level monitoring allows exploits aimed at server processes, such as *sendmail* or *apache*, to be detected. Such attacks often leave a system functional, but allow an intruder to access and modify protected data on a system. In contrast, even though denial-of-service attacks may make use of specific server programs, they are actually attacks on the resources of the entire system. By monitoring and adjusting overall resource usage, a system should be better able to maintain services in the face of an attack.

In the following sections, I explain in more detail the different parts of my dissertation, outlining what has already been completed, and which parts remain.

4.1 Offline Analysis

The first part of my dissertation will be a review of our past work with system calls. My work with online monitoring and response is a direct outgrowth of this work, and so it is necessary to describe the methods and results of our previous experiments if one is to understand the technical choices made for my project. Below is an overview of that work, emphasizing the different methods used to gather and analyze data.

Our work with system calls began in earnest in the summer of 1995, when Stephanie Forrest, Steve Hofmeyr, and I decided to explore whether sequences of system calls could detect real intrusions. We did our experiments on SunOS 4.1.1 and 4.1.4 systems, using a program called *strace* to record the system calls made by a process. We obtained exploit scripts (scripts that automate the actual break-in procedure, taking advantage of flaws in privileged programs) from the Internet, and through our collaborator Tom Longstaff at the Computer Emergency Response Team (CERT) [17].

The experiments consisted of two phases. First, the flawed privileged program was run in an artificial environment and was supplied with synthetic inputs meant to simulate a normal workload. *strace* was used to record the systems calls of the program, with *strace's* output being saved to a file. Then, an exploit script was run against the same program in the same environment, again with *strace* recording its system calls to a file. The "synthetic normal" traces were then fed into our analysis program, called *tide* (for time-delay embedding), which created a profile (or database) of the program. The same program was then used to compare the abnormal trace against the stored profile. If *tide* found any discrepancies, we detected the exploit.

To understand how *tide* works, it is useful to go through an example. Here is a trace of five system calls, with the first system call executed on the left:

fstat, mmap, close, open, mmap

To build a profile of the program producing these system calls, first we must choose a sequence length n — the length of our "peptide." For this example, let $n = 3$. Having made this choice, the system call trace breaks down into three sequences:

fstat, mmap, close
mmap, close, open
close, open, mmap

Our original idea was that a program's profile would consist of these sequences. Thus, to compare a new trace to a profile, you would split it up into overlapping sequences, and then would check to see if those sequences are present in the profile. Any sequence not present would be considered an anomaly. Although this is in fact what we have done for our later work, this is not what we did in 1995.

Instead, *tide* recorded a generalization of these sequences, using a technique we call "lookahead pairs." Continuing the example, instead of recording the three sequences above, *tide* would record:

mmap, close
fstat, *, close
close, open
mmap, *, open
open, mmap
close, *, mmap

Here the "*" is a wildcard value, matching any system call. Note that each line is a pair of system calls, with zero or more wildcards between the calls. To generalize, let the most recent system call be in position 0, and number them in increasing order from right to left. Then, for a sequence of length n, *tide* records in its profile the following pairs:

$1, 0$
$2, 0$
$\ldots$
$n - 1, 0$

To see whether a sequence matches an existing profile, *tide* splits it into lookahead pairs, and then checks whether each pair is present in the profile. Each missing pair is reported as an anomaly.

Although this representation may seem baroque, actually it is much simpler to implement than profiles recording full sequences. A tree or other dynamic data structure is needed to record overlapping sequences within reasonable space bounds. Lookahead pairs, however, can be stored using $n - 1$ bit arrays, each $k \times k$ in size, where k is the number of system call types. A UNIX system typically has around 180 different system calls, so each bit array can be stored in less than 4K. Each array represents the values of a different lookahead pair distance, with a one indicating that the pair is allowed. In our example, then, we would have two bit arrays, representing the $1, 0$ and $2, 0$ lookahead pairs. Assuming a suitable system call numbering, the arrays would consist of the following:

$1, 0$: 1 in (mmap, close), (close, open), and (open, mmap); 0 elsewhere
$2, 0$: 1 in (fstat, close), (mmap, open), and (close, mmap); 0 elsewhere

tide was able to detect anomalies with respect to synthetic normal traces [17]. While this result was extremely promising, we had three major questions outstanding: would our method detect other intrusions, would it do so under live conditions, and how often would we get false alarms in a production environment? We addressed the first by continuing to test programs with new exploit scripts as they became available. To address the other two issues, however, we needed to gather data on production systems.

One approach is to use a wrapper script to cause *strace* to be invoked every time a given executable is called. We did this with *lpr* at UNM and the MIT Artificial Intelligence Lab, on hosts running SunOS 4.1.4. We got a significant amount of data from these experiments, and they allowed us to show that our method (using full-sequence profiles rather than lookahead pairs) produced a low rate of false positives (approximately 2 per day for *lpr*)[21]. Unfortunately, our wrapped *lpr* sometimes failed to work the same as the unwrapped *lpr*; complaints from users and the need for constant vigilance forced us to suspend the MIT *lpr* experiments after less than two weeks.

We had initially chosen *strace* because it was available, easy to use, and provided the data we needed; unfortunately, it also has certain limitations. Not only can *strace* interfere with normal execution; it also causes the traced process to run much more slowly than normal. One of our goals was to gather data on a busy electronic mail server running *sendmail*, because we felt this would be a stringent test of our method; thus, we needed another mechanism for gathering data. With the aid of Andrew Gross from the San Diego Supercomputing Center, in the spring of 1997 I attempted to create a modified SunOS 4.1.4 kernel which would log the system calls of certain programs. Unfortunately, I did not have access to the kernel's source code; thus, debugging was an extremely painful process. In the end I was unable to create a kernel that was stable and that could reliably record system calls.

As we still wanted to gather data on an e-mail server, I then modified a SunOS C library, changing the system call function wrappers so that they would log the system call to a file before executing it. Again, this was done without source, via disassembly of the selected functions. (Luckily, they were short, and mostly similar.) These efforts allowed me to trace the behavior of *sendmail* on the MIT AI Lab's central e-mail server in September 1997.

Although modifying a C library allowed me to gather samples of normal behavior, this technique cannot easily be used to trace buffer overflow attacks, one of the most common types of intrusions: standard buffer overflow scripts do not call our functions for logging system calls, and thus would not be traced. Although a modified exploit script combined with a shell linked against the modified C library might together allow one to capture the behavior of an attack, such modifications are inherently intrusive and fragile, and cannot lead to a system that can securely and reliably detect intrusions.

Our recent work has focused on using a Linux 2.0 kernel modified to record system calls. In June 1998 I used such a modified kernel to record the normal and abnormal behavior of a DNS server *(named)* running on a Red Hat Linux 4.2 system. The availability of source code and documentation, combined with an active developer community, together have made kernel development in Linux much easier than in SunOS. My tracing kernels have proved to be stable, and have been able to collect data reliably.

One thing I have learned from these experiments is that keeping a log of a process's system calls is expensive, no matter how you do it. It is also dangerous, because I/O operations can fail, and failures at the logging level can interfere with the normal behavior of the traced process. Through online monitoring, I plan to sidestep both of these problems by analyzing system calls as they occur, removing the need for an intermediary log file.

In addition to reporting our past results, in this section I will also compare the relative performance of *tide* (our lookahead-pair analysis program) and *stide* (our sequence analysis program). Although early experiments indicated that *stide* was better at detecting anomalies than *tide*, it is not clear which method is preferable when efficiency and false-positive rates are also considered. Thus, I will use both analysis techniques on all of our old data, comparing their true-positive and false-positive rates. Also, I will compare the performance of both methods through experiments and algorithmic analysis. Using these results, I will then choose a method of anomaly detection to be used in the subsequent chapters.

4.2 Online Monitoring

For the next part of my dissertation, I will extend my past work with system call tracing by creating a system that will monitor processes online. Instead of recording the system calls to disk for later analysis, my prototype will analyze them on-the-fly, signalling anomalies as they occur. Not only will this work help show the practicality of our technique; it will also form the basis for my work on automated response.

My implementation testbed will be Intel-based PCs running Debian Linux 2.1. Modifications will be made to a Linux 2.2 kernel. By working at the kernel level, this system will avoid the overhead of communicating with a monitoring process, and should provide a computational environment nearly indistinguishable from that of the unmodified system. The system call dispatcher will be modified first to invoke my routines before actually executing the system call. Certain system calls, such as fork and execve, also will be modified to invoke housekeeping routines needed to account for new processes (fork) and programs (execve).

Currently, I have a kernel which can do basic online monitoring using the lookahead pair method of analysis. I chose to go back to this method because it does not require variable-size data structures, and thus is simple to implement in kernel space.[2] This kernel cannot yet load and save profiles reliably; that will be the first problem I need to solve. However, the performance of this minimal version is outstanding: in preliminary experiments, I can barely measure the performance impact of the monitoring code.

I have four goals for this part of my dissertation. I want to show:

- that my modified kernel produces the same system call profiles as the offline analysis code,

[2] In the Linux kernel, memory is generally allocated in page-size (or multiple page-size) chunks. On Intel machines, the Linux page size is 4K; thus, to allocate smaller data structures, you must create your own mini-allocator. Such an allocator is not too difficult to write; however, the lookahead-pair method can be efficiently implemented using a fixed-size data structure. This, combined with its inherent simplicity, made the lookahead-pair method the obvious choice for a proof-of-concept implementation.

- that my modified kernel is robust,
- that my modified kernel is efficient, and
- that I can achieve a "low" false positive rate in terms of alarms per day (instead of per system call).

To test the first part, I will do two things. First, I will create a standalone version of the kernel analysis code, and I will compare its functioning with that of the old analysis code using at least three old data sets. Then, I will modify my monitoring kernel so it will also log system calls to disk. I will run the old analysis code on the recorded calls, and check to make sure that it and my kernel code produce identical profiles.

To see if my monitoring kernel is robust, I will run it under stressful conditions, probably using the standard stress-testing program *crashme*. I will check to make sure my modified kernel does not crash if an unmodified kernel was able to survive. This should even be the case if every process on a system is being monitored (modulo the memory overhead of monitoring).

To measure the overhead of my modifications, I will test the modified and unmodified kernels using at least one standard benchmark. I will need to choose a benchmark that measures kernel performance, rather than CPU or memory performance. I will test the modified kernel under three conditions: no monitoring, monitoring, and profile creation.

To test the rate of false positives, I will test at least one program (for which I have an exploit) in a production environment, preferably on a busy server. To a large degree, I am flexible about what false-positive rate I can accept, a-priori. However, whatever the rate, I must make sure that the automated response mechanisms of the next section will not cause the system to be more than 10% slower than an unmodified system under normal conditions.

Note that profiles will be stored on a per-executable basis. Because buffer overflows often cause a new program to be run, I may need to record what executables a given program spawns. (For example, I may record that *named* calls *named-xfer* to perform DNS zone transfers. Thus, a normal exec sequence which causes a shell to be run will trigger an anomaly.) Also, for this part, I will manually switch from training to testing mode on a per-executable basis.

4.3 Process-level response

The goal for this part is to implement mechanisms which freeze or kill processes that are being attacked. The previous chapters will have demonstrated that anomalies at the system call level can be used to detect attacks; in this chapter, I will demonstrate that these anomalies can be used to trigger responses which interfere with these attacks.

Although I hope to be able to stop many different types of attacks, I will focus on trying to interfere with buffer overflow exploits. I am using buffer overflows as my prototypical attack method because they are common[3], are easy (for us) to detect, and are extremely dangerous.

My working hypothesis is that buffer overflow exploits can be defeated by delaying suspicious system calls. I have chosen to focus on delays because short delays have a minimal effect on a process's behavior, while longer delays can in effect suspend a process. Thus, a low level of false positives can be tolerated, even if each causes a short delay.

There are two main issues I will need to address: What is the impact on normal system operation, and can attacks be stopped before they do harm? To address the first, I must perform

[3] In 1998, CERT [10] issued thirteen security advisories. Of these, seven (5, 6, 8, 9, 10, 11, and 12) refer to buffer overflow vulnerabilities.

experiments on production systems to see how often processes are delayed under normal conditions. On average, the modified system should be at most 10% slower, including delays due to false positives. Also, I must see whether these delays cause any other problems with system behavior, such as a clustering of delays which cause the system to freeze for a short period of time.

To see whether the attacks can be stopped in time, I will use three kinds of buffer overflow exploit scripts: ones that do nothing except infinitely loop, ones that run a shell, and ones that attempt to shut down the system. The infinite loop attacks will cause the process to stop emitting system calls after the attack has completed; thus, they will be the hardest to detect. The shell attacks are what most exploit scripts do by default, so it serves as the "real world" example. The shutdown attacks provide a worst-case scenario for the system: If a successful response is not mounted, the system dies. If my system can detect the infinite loop attacks, and can stop the shutdown attacks before the system dies (all without overly compromising system performance), I will have succeeded.

Time permitting, I will also look at other types of attacks, particularly trojan horses. Trojan horse programs are modified versions of standard system programs such as *login* or *ps*. Attackers use them to permit easy repeat access, or to conceal their activities. Because trojan horse programs do not behave like the original programs, my system should be able to interfere with them when their behavior deviates from that of the normal program.

For my initial experiments I will modify the online monitoring prototype as follows: every time an anomaly occurs in a traced process, the current system call (and hence the process) will be delayed for a time $t^{f(a_w)}$, where t is some fixed amount of time, a_w is the number of anomalies that have occurred during the past w system calls, and f is a linear function. This delay will occur even if there are no other processes to run.

Although this approach may work, it is likely that I will have to analyze the course of several buffer overflows, noting the proportion of system calls in the different stages of an attack. Ideally, significant delays should occur after an attack has begun, but before any damage can occur — all while avoiding significant delays for programs incurring a low rate of false positives. I will need to match the delay schedule to these constraints.

I will also implement user-space programs which will allow the delay parameters to be set, and which will detect and kill processes that have been delayed for more than a fixed period of time. This reaping mechanism will allow the system to clean up after an attacker, preventing a lingering attack from causing damage at some later date. Also, these user-space programs will automatically switch the system from a training to a monitoring state. Initially, this transition will be abrupt; however, I will also explore ways to make this transition more gradual.

Because I will be interfering with the behavior of programs without user intervention, I will need to spend some time examining whether an attacker could use these mechanisms to mount an attack. In addition to thought experiments, I will attempt to devise ways to defeat my methods. Although I will certainly not think of everything, the exercise will certainly help me improve my system.

4.4 System-level response

Process-level responses can help stop attacks on individual programs; however, denial-of-service attacks directed against a system require a more global approach. What is needed is a method for monitoring and modifying resource allocation.

My goal for this chapter is to develop mechanisms that allow a system to function when aberrant processes attempt to consume all available resources. I will focus on addressing fork bomb attacks because they are extremely simple, very dangerous, and UNIX web servers under high load behave similarly to a machine undergoing a fork bomb attack. I may also examine other denial-of-service resource-based attacks; however, I will have succeeded if I can stop a fork bomb.

My working hypothesis is that overall resource allocation can be influenced in useful ways by monitoring and adjusting the overall frequency pattern of system calls. To be more concrete, consider a fork bomb. When one is executed, forks become much more frequent relative to other system calls. My basic idea is to have the kernel respond to such an attack by delaying all fork requests. These delays will slow down the bomb's rate of resource allocation, preventing the instant death of the system.

I have not looked at any data on the overall frequency distribution of system calls in a system, under normal or abnormal conditions; thus, I am sure that as my understanding of the problem evolves, so will my techniques. However, I do know that there are at least three scenarios I must address: a system under light load, a system under continuous, heavy load, and a system undergoing an attack. To succeed, my system must behave similarly to an unmodified system under both constant light and heavy loads, while at the same time preventing a fork bomb from killing the system.

I see the response to an attack as consisting of two phases. First, processes making overused system calls are delayed (e.g. all processes making fork requests are delayed). This response prevents the attack from instantly saturating the system, but does not prevent the attack from eventually succeeding. Then an analysis module, probably running as a user-space daemon, is activated in response to the delays, and determines which programs are responsible for the change in system behavior. Those processes are then preferentially delayed, and eventually killed off. The general form of this blame-assignment problem is probably impossible to address; however, there is one heuristic which I believe will be effective: If most of the children of a process are being delayed, then all of its children should get extra delays.

One significant challenge will be the online comparison and modification of the frequency profiles. For a first cut, I will have the system remember the past n system calls, where n is a large number ($n \approx 100,000$). As system calls are added and removed from this fixed window, counters recording the frequency of each system call will be updated. In parallel, frequency counters will also be updated for the most recent m system calls, where $m < n$ ($m \approx 1000$). Let $C_n(x)$ be the count of system call x in the last n system calls, and let the relative frequency of x over the past n calls be

$$F_n(x) = C_n(x)/n.$$

The delay for system call x, D_x, will then be

$$D_x = f(F_m - F_n),$$

where f is still to be determined (but which will at least involve a threshold below which $D_x = 0$).

If this method fails, I will look to the computer science literature for work on fast online comparisons and updates of frequency profiles; however, before investigating alternative algorithms, I will need to examine the frequency data carefully. It may be possible that this problem requires an expensive algorithm; if so, part of the work will be shifted to a helper process.

To validate the method, I will perform tests to answer the following questions:

- What is the overhead for a "normal" system?
- Can attacks can be stopped?
- What are the failure modes?

To answer the first, I will need to measure the overhead of the basic mechanisms and the frequency and magnitude of delays during normal usage patterns. I will perform experiments on both servers and workstations. I suspect that variations in user behavior on a workstation may cause a high rate of false positives (unnecessary delays), but I will need to test this hypothesis. A server under a relatively constant load should suffer no more than a 5% decrease in performance.

To see whether attacks can be stopped, I will run attacks on non-production servers and regular workstations. I will perform experiments using a regular fork bomb, a net-based fork bomb (e.g. a web server with too many connections), and a parallel make of a large program (such as the Linux kernel). Although a parallel make is a reasonable thing for a user to run, I have seen parallel makes of the Linux kernel cause a Linux machine to go into an unresponsive state for minutes at a time; thus, it is another "attack" which I would like to address!

As with the process-level response, I will need to put on my "black hat" and investigate ways an attacker could use these automated response mechanisms against us. Also, I will need to check whether unusual patterns of behavior could result in a system spontaneously freezing or otherwise failing. I am attempting to give a great deal of power to a computer system; thus, I must at least consider how this power could be turned against us.

This is the most immature, and most ambitious part of my dissertation. I will probably have to experiment with several methods before I find one that works reliably. However, as the basic kernel code will have been developed by this stage, it should be easy to implement and test new methods.

5 Related Work

Although it is difficult to make direct comparisons between this project and past work, there are many systems that are related to different aspects of my dissertation. Some mechanisms in current systems can be thought of as homeostatic; they deserve examination. My proposed system will probably be most useful as an intrusion-detection system; thus, it is important to consider other work in this field. Also, connections can be made with work in automated system administration, software fault-tolerance, artificial life, and robotics.

5.1 Homeostasis in Current Systems

Homeostasis has not generally been recognized as a useful computer design principle; however, like cells, our programs need to be provided with a consistent environment for them to function properly. The mechanisms that help maintain that environment can be thought of as homeostatic mechanisms. Some, such as temperature and power regulation, mostly operate at the hardware level; others, such as virtual memory management and process scheduling, exist mostly at the operating systems level. Although they are far from perfect, they are essential to the proper functioning of almost any application program.

Consider the requirements placed upon hardware. Dynamic RAM must retain a perfect record of bits for as long as a computer is on; hard disks must provide a persistent, perfect store of information; CPUs must execute every instruction without error. We take these requirements for granted, even though they are not easy to achieve. To maintain these properties, our computers have a number of dynamic processes. Fans, heat sinks, and indoor climate control systems all work together to maintain a consistent temperature environment. Large-scale power and distribution systems, combined with internal power management circuits, ensure that just the right amount of electricity reaches every part of a computer. Dynamic RAM is frequently refreshed, ensuring that the stored bits remain constant. Hard disks have circuits to regulate disk rotation, and digital signal processors to correct for variations in the storage medium.

Most of these mechanisms normally operate far from the view of a programmer or user; however, mobile computing has provided an impetus for increasing the regulatory connections between hardware and operating systems. On laptops, an OS can adjust the properties of the display, hard disk, and CPU to maximize either performance or power conservation. In addition, when battery reserves are low, an OS can even cause the computer to suspend operation, preserving system state until additional power is available. These homeostatic interconnections are only going to become stronger with the increasing presence of personal digital assistants and other small, mobile computing devices; however, these interconnections are not limited to small systems. In fact, many desktop and server machines are able to monitor case temperature and hard disk status, providing warnings of fan and disk failures.

Although hardware-level homeostatic mechanisms are essential, others are needed to manage the demands placed on a system by external inputs and multiple application programs. For example, consider the challenge of virtual memory. In order for a program to run, its code and data must be present in physical RAM. If a program is particularly large, or if multiple programs are sharing the resources of a computer, often these programs will require more RAM than available. Most PC and workstation operating systems mitigate this problem through the use of slower auxiliary storage, generally on a hard disk. Code and data that are not being used are written to disk, freeing RAM for actively executing programs. Eventually the information on disk is needed again, necessitating the "swapping" of information between RAM and disk. Because swapping is very time-consuming (because of the speed difference between RAM and hard disks), a virtual-memory system must be designed so as to minimize the amount of actual swapping that occurs. To facilitate this, bit flags are often used to indicate whether a given part of RAM has been recently accessed; if it has, the swapping algorithm will try keep that page in RAM.

Process scheduling is also a homeostatic mechanism. As the kernel switches from running one process (active program) to another, it keeps track of how much CPU time each has consumed. This information is used to make sure that each process is given a chance to execute. Some systems also adjust scheduling policy based upon differing priorities assigned to each process; even so, in a preemptively scheduled system, timers are used to ensure that no process runs for too long. By ensuring that every process gets some minimal amount of CPU time, process scheduling helps maintain the health of a computer system.

Both virtual memory management and process scheduling help maintain the homeostasis of a computer; however, neither is able to help a machine survive when it is truly stressed. This limitation stems from the policy that a kernel should be fair. Except for static priorities, in most systems the kernel allocates resources such that no process is considered more important than another. Processor time is shared amongst all processes, and memory requests are immediately granted if they can be satisfied. Such policies work as long as the demands on the system remain moderate; however, there are fixed costs associated with resource sharing: there is always a minimum size to every memory request, and every context switch has a minimum execution cost. Thus, there is a limit to the amount of sharing that can be done, and thus a fundamental limitation to fairness.

When resource demands become extreme, fairness necessitates poor performance for all processes, either through "thrashing," when the virtual memory system becomes stressed, or through unresponsiveness, failed program invocations, and entire system crashes, when there are too many processes to be scheduled. One fair response to extreme resource contention is through the random killing of processes contending for the resource; this is the strategy employed by AIX for processes actively allocating memory [12]. Such a strategy, though, generally does not result in acceptable behavior: it is too easy to kill processes that are critical to the functioning of the system.

Experimental extensible kernel systems, such as SPIN [5] and VINO [29], are related to my work in a couple of ways. First, most employ novel OS protection mechanisms, such as safe languages or a transaction system, to help ensure the safety of grafts inserted into the kernel. I am also interested in novel safety mechanisms; however, instead of trying to ensure the safety of a new kernel interface, I am interested in adding more protection to an existing one: the system call interface. Also, designers of extensible kernels are interested in gathering data on system behavior, and modifying the operation of the system based on this information. In particular, at the 1997 HotOS workshop, Seltzer & Small [30] presented a paper discussing techniques for having a system monitor and adjust its behavior autonomously. However, like most work in extensible operating systems, their focus is on performance, not stability.

There is some evidence that others, particularly Microsoft, are seeing the usefulness of homeostatic mechanisms, and are incorporating new ones into mainstream operating systems. Windows 95 maintains a directory, called "sysbckup," which contains copies of some essential executables and libraries. If a program installer replaces one of these files with an incompatible version (such as one designed for Windows 3.1), Windows 95 uses these backups to gracefully recover. It appears that Windows 2000 will take this concept one step further. According to press reports, Windows 2000 will have some self-repair capabilities, for example allowing applications such as Word to replace essential files if they have been deleted [33]. Such additions could be extremely useful; however, it remains to be seen whether these changes will make Windows more reliable.

5.2 Intrusion Detection

The field of intrusion detection originally developed in response to the needs of the military, which wanted a way to detect unauthorized computer usage, whether by insiders or by unauthorized outsiders. With the rise of the Internet, though, the field has shifted towards the needs of businesses. Although preventing unauthorized access is important, businesses are generally more interested in preventing crackers from damaging their ability to engage in e-commerce. Because most intrusion detection systems have limited automated response capabilities, in their current form they are not particularly homeostatic. However, they are worth examining, because my system should address some of the same needs.

Most research into intrusion detection and response has focused on methods for processing various input streams, ignoring the question of what are appropriate inputs and outputs. We can, in fact, classify most systems on the basis of their choice of inputs (the observables they choose to monitor), and obtain a relatively clear overview of the field of intrusion detection. What follows is an overview of approaches to intrusion detection, classified by the choice of primary input source. It includes work that is not generally labeled as "intrusion detection;" however, they all aid in the detection of and response to security compromises.

- **Files:** Many systems simply look at the state of the filesystem to determine whether a system is vulnerable to attack, or has actually been compromised. One technique is to calculate and record a checksum, or fingerprint, of important files. By periodically recalculating and comparing these fingerprints, files that have changed may be detected. Because many crackers modify system files after gaining access to a system, such checks can help detect intrusions [23]. Another technique is to examine the versions of binaries and the state of configuration files, looking for versions with known vulnerabilities or insecure settings [16].

  To detect foreign code, files can be scanned for short identifying strings present in such code — this technique is used by most virus scanners. While signature scanning is quite effective at catching viruses, it can only detect viruses for which the scanner has a signature — new, novel viruses cannot be detected. Because of this, many commercial virus detectors also try to detect suspicious activity, such as the modification of executable files. Jeffrey Kephart at IBM has led an effort to create a computer immune system focused on preventing the spread of computer viruses. By automating detection, signature extraction, and signature propagation, they hope to be able to stop fast-spreading epidemics that are possible in a networked world [22].

The state of the filesystem can identify problem areas, or can reveal that a compromise has taken place; however, many intrusions involve perturbations of code in memory, not disk. Such intrusions cannot be detected unless the attacker changes a file — an event which may not happen. Thus, filesystem examinations cannot, by themselves, provide complete online security protection.

- **Logs and Audit Trails:** Most work in host-based intrusion detection has focused on analyzing log files produced by various programs. Many programs on UNIX systems use the syslog facility to store a record of their activities. Programs create a syslog record by calling special library functions, providing event categories, importance ratings, and text messages. The *syslogd* daemon receives these messages, adds the machine's hostname and a time stamp to them, and writes these records to a local log file or a remote log host based upon the information in `/etc/syslog.conf`.

  The amount of information contained in these records varies greatly depending upon the program producing them; however, there are packages, such as TCP Wrappers, which attempt to make this information more uniform and useful. The TCP Wrappers package provides a uniform mechanism for controlling access to TCP network services based on hostnames and IP addresses; in addition, it produces syslog messages for every attempted connection, recording for each the source host, the service requested, and whether it was successful or not [32]. If every network service on a system is "TCP wrapped," then the syslog log files will contain a complete record of TCP connections. There are numerous programs for analyzing syslog log files; for example, the Freshmeat software repository (www.freshmeat.org) has an entire sub-category called "Console/Log Analyzers."

  A source of data that has been used by many intrusion-detection systems are audit packages. Audit packages have been designed to provide a detailed record of system activity, as required by government guidelines for secure systems. They record events such as authorization requests, program invocations, and even some system calls. This voluminous data is generally written to a special-purpose binary log file. Care is taken to record this data in a secure fashion, so that unauthorized users cannot easily hide their tracks. Audit trails are designed to provide forensic evidence for the consumption of human analyzers; however, they can also provide a basis for an automated intrusion detection system. Some of the most sophisticated uses of audit trails were the IDES and NIDES projects, which used SunOS Basic Security Module (BSM) audit data to look for unusual patterns of user behavior [26]. Audit packages tend to be rather costly in terms of system performance and storage requirements, and packages such as NIDES only added to that burden. Thus, neither audit packages nor intrusion-detection systems based on them have generally been fielded outside of government agencies.

  I spent some time considering whether I wanted to use an audit package as my data source; unfortunately, I found that they were cumbersome to use, and tended to record data that I didn't care about (authorization events, network activity) while missing data that I wanted (the complete record of system calls).

- **Network Traffic Analysis:** Since intruders often attack a system through its connections to the outside world, a natural approach to intrusion detection is to monitor network traffic and services. Because of the complexity of network traffic, more attention has been devoted to the problem of what parts of the stream should be examined. One technique, used by SATAN [15] and by many commercial systems, is to be proactive, and to scan a network for insecure services and configurations. Most methods, however, are more passive, snooping traffic on a broadcast local area network. One interesting approach is taken by the Network Security Monitor — it focuses on characterizing the normal behavior of a network by examining the source, destination, and service of TCP packets [19]. However, most commercial systems, such as Cisco NetRanger [2] and ISS RealSecure [1] look at packet headers and payloads, searching for customized attack patterns defined by hand-crafted matching rules. Although it is difficult to update the attack signatures of some products, some new systems, such as Axent's NetProwler, allow semi-automatic updates via e-mail [28, 3].[4]

  One benefit of looking at the network level is that there is a natural response to any detected problem: Terminate the anomalous communication stream. Unfortunately, this response can quickly degrade network services in the presence of false alarms, and so in practice such mechanisms are rarely used, even when they are present in a package. Instead, automated responses are mostly limited to increasing the detail level of logging and sending messages to the system administrator.

Although these categories cover most approaches, there are also systems that examine other data streams. For example, StackGuard instrument binaries to allow for the detection and termination of programs which incur buffer overflows [13].

For my work, instead of using the data streams described above, I have chosen to monitor the patterns of system calls made at a process and system level. System call monitoring cannot be a complete substitute for the methods described above; however, my hope is that measures and responses at this level will prove to be lightweight, efficient, and effective enough to merit inclusion in future intrusion detection systems.

Other groups have chosen to use system calls as the key observable for detecting security violations. At UC Davis, Calvin Ko and others formally specified allowable system calls and parameters for privileged programs [24]. Another project of note is Janus, a program that provides a secure environment for untrusted helper applications (e.g. graphical viewers like *ghostscript* or *xv*) by monitoring the execution of certain dangerous system calls, such as exec and fork [18]. One nice feature of Janus is that it also has real-time response: system calls violating policy are aborted. Unfortunately, these policies must be hand-crafted, making Janus most suited for programs with relatively simple needs.

Interestingly enough, most UNIX-like systems already use delays to enhance system security at a common interface: the login prompt. On many machines (including Debian Linux), there is a few second delay after each failed login attempt, and after a minute the login process times out, forcing a reconnect on remote logins. These delays can be mildly irritating to a clumsy user; however, they also make it much more difficult for a remote attacker to repeatedly try many different username/password combinations. Though simple, this mechanism is quite analogous to my work with system calls. In this domain, an anomaly is an unauthorized username or an incorrect password; by delaying (and logging) anomalous patterns, system security is enhanced.

[4] Many commercial intrusion-detection products, such as ISS RealSecure [1], actually combine features from all three categories. However, the mechanisms they use for file and log monitoring are quite similar to those described here.

5.3 Other Related Work

There hasn't been much work on automated response in the intrusion detection community; however, there is work from other fields that relates to this problem. One interesting alternative approach to building a computer immune system from the perspective of system administration is outlined in Mark Burgess's paper, "Computer Immunology" [9]. Although much of the language describing the human immune system is similar to that outlined here, his thoughts on incorporating immune system concepts into computer science are quite different. The primary focus of Mark Burgess's immune system is a program called *cfengine* [8], a tool for building expert systems for maintaining the configuration information of a network of UNIX machines. This program allows system administrators to specify configuration constraints and modification methods using a declarative language. The most innovative feature of *cfengine* is its ability to monitor system state, and to trigger actions to modify that state when it deviates from pre-specified criteria. For example, if a system's disk becomes more than 98% full, *cfengine* can run a script that removes temporary files. These simple feedback loops can help a system maintain homeostasis. Other than this, however, the architecture of *cfengine* and the other envisioned components of Burgess's immune system are grounded in standard computer science, not immunology: his envisioned system examines things such as file permissions and load averages, evaluates them using a rule-based expert system, and modifies system behavior by running arbitrary shell scripts. He does not appear to believe that knowledge of the specifics of natural immune systems can help us design better computer systems; in particular, in a section of [9], Burgess states that "Self/non-self is not a very useful paradigm in computing" — quite a contrast to our views.

Although my work's primary application is to computer security, its goals are similar to those of the field of software fault tolerance. The field of fault tolerance is focused on methods for ensuring reliable computer operation in the face of flawed components, both hardware and software. Most work on fault tolerance has focused on hardware fault tolerance, primarily through the use of redundant components. Software fault tolerance, however, focuses on the problem of flawed software. By assuming that the hardware is reliable, flaws in software must come from flawed specifications, designs, or implementations. These flaws cannot be mitigated through simple replication; instead, software fault tolerant systems use two basic techniques: fault detection and recovery, and design diversity. When there is only one version of the application available, faults are detected through special error-detection code, and are mitigated through exception handlers. Numerous programs employ these techniques in an ad-hoc fashion. In addition, several methodologies for structuring error detection and recovery have been developed, the most prominent of which is the "recovery block" method. A recovery block consists of an acceptance test, followed by several alternatives. Execution of the block proceeds as follows. First, the program is checkpointed. Then, the first alternative is executed, and the acceptance test is run. If the test returns true, the other alternatives are skipped, extra state information is discarded, and the program proceeds. If the test fails, the program is restored to the checkpointed state, the second alternative is run, and the acceptance test is run again. This process continues until the acceptance test returns true, or until the program runs out of alternatives. Recovery blocks can be nested, allowing for elaborate recovery schemes [27, pp. 1–21].

Design diversity is based on the hope that independent solutions to a given problem will have different errors, allowing them to be used to check each other. There are two major approaches to design diversity: N-version programming (NVP) and N self-checking programs (NSCP). NVP systems contain N complete implementations of a given specification, each written by different teams of programmers. These versions are run concurrently (generally on different processors), with the output of the system being the output of the majority of the implementations, assuming that a majority agree on a single action. NSCP systems also consist of multiple implementations of a given program specification. However, in NSCP, only one of these versions is active at any time, with the others serving as "hot spares." These programs have error-checking code that attempts to detect incorrect behavior. If the active version detects a problem with its behavior, it activates a spare and becomes inactive. The spare implementations may not offer full system functionality; however, like a spare tire, they should be enough to keep the system limping along until the problem can be fixed [27, pp. 49–51].

While I am also interested in detecting and correcting problematic behavior, I have a much more flexible notion of faults. Instead of defining a-priori what types of behavior are acceptable, my system measures a program's behavior by monitoring its pattern of system calls. These patterns cannot tell us for sure that a fault has occurred; they can only indicate whether current behavior is similar to that of the past. However, if we assume that the behavior of the past was acceptable, then deviations from that "normal" can be thought of as a kind of generic fault. The main hypothesis of my dissertation is that responses that reduce the occurrences of those faults will improve system stability and security. While this sort of fault tolerance is not enough to ensure that a flight computer won't steer a plane into the ground, hopefully it should be enough to prevent a web server from crashing at 3 AM in response to a flood of connections from a high-school cracker.

Although my project is aimed at advancing the state of computer security research, its inspiration has come from the field of artificial life, and in particular the work of David Ackley. Through the ccr project [4], Ackley has advocated the view that existing computational systems are in fact living systems, and if we are to tap the possibilities of our computers, we must follow the design principles of biological systems. Although ccr can be described as a peer-to-peer, distributed multi-user dungeon (MUD), it can also be described as an experiment in making computer artifacts that treat communication as a risky endeavor. Communication is inherently dangerous, and this danger is reflected in the structure of all biological systems. Whether you examine a cell membrane, the human immune system, or predator-prey interactions in ecosystems, it is clear that there is a need to manage resources that are devoted and exposed to interactions with others. Most programs have very primitive notions of resource and access control; either an interaction is permitted, or not. And, if that interaction is permitted, all requested resources are granted. In contrast, ccr limits all interactions. Movement within ccr is strictly regulated. Communication between worlds is controlled, both in terms of what kinds of information, and how much may pass between worlds. There are bandwidth regulators on all channels, and if one world attempts to flood another with more data than allowed, the connection is terminated. Also, certain information is considered inherently private (such as a world's private key) — such information is carefully managed, and is not allowed to flow over any outside connection, no matter how much trust is attributed to that connection. These and other features have framed my views on how a computer should behave.

While the ccr project has influenced my aesthetics, the field of robotics has informed my views of implementation. In particular, Rodney Brooks's work on subsumption architecture has been inspirational. In a 1985 MIT AI Lab memo, Brooks described a new approach to mobile robot control [6]. Through the use of loosely coupled feedback loops, each connecting a limited set of sensor inputs to a small set of actuators, he was able to have a robot interact with real environments in a rapid and robust fashion. One particular difference with traditional robotics was that there was no centralized representation of the outside world; different components might in fact have contradictory models. However, by arranging these modules in a hierarchy (with higher modules "subsuming" lower ones), and using this hierarchy to arbitrate between conflicting actions, these different (mostly implicit) representations are able collectively to provide robust control for a robot. This approach has been quite fruitful; [7] describes a number of successful robots. Other researchers have adopted the subsumption architecture, using more traditional AI algorithms in the higher levels (such as planners) to produce more sophisticated behaviors [25, pp. 6–12].

Although the field of robotics might appear to be quite distant from that of operating systems, with the rise of the Internet both face a common challenge: they must interact with a rapidly changing, potentially threatening outside world on very short time scales. Although a networked computer does not have to deal with faulty sensors and imprecise motor control, it does have to deal with a barrage of network packets and a multitude of independent programs and users, some of which may be malicious. In both cases, getting precise information about the state of the world is difficult and expensive, whether it be a room full of furniture and people, or a busy web server communicating with machines around the world. Consequently, heuristics must be used to couple simplified inputs to useful actions. I already have a candidate "sensor": sequences of system calls. Now the question is what sort of "motors" should be used in response to this and other sensors, with the goal of helping a machine survive on the Internet.

6 Outline of Dissertation

The proposed dissertation will consist of the following chapters:

- **Introduction:** This chapter will describe the motivation for this work, discuss connections with biology, and describe related projects.
- **Offline Analysis:** Here, the offline analysis experiments will be described, along with their results. The major results of this section have been published [17, 21]; however, this work provides essential background material which motivates the later chapters.
- **Online Monitoring:** Here, the online prototype of the process system-call monitoring system will be explored, with design, implementation, and results being covered.
- **Process-level Automated Response:** This chapter will test the hypothesis that delays in system-call execution can be an effective countermeasure to various intrusions.
- **System-level Automated Response:** This chapter will describe the design, implementation, and performance of a system-call frequency monitor and responder.
- **Discussion and Summary:** Here there will be an examination of the significance of the results, with a focus on what role such mechanisms could play in commonly-used systems. Related work will be revisited, and directions for future research will be discussed.

References

[1] ISS RealSecure. http://www.iss.net/prod/rs.html December 1998. Product Summary.

[2] Netranger intrusion detection system technical overview. http://www.cisco.com/warp/public/778/security/netranger/ntran_tc.htm, 1998.

[3] Netprowler (formerly idtrak). http://www.axent.com/iti/netprowler/idtk_ds_word_1.html, 1999.

[4] David H. Ackley. ccr: A network of worlds for research. In C.G. Langton and K. Shimohara, editors, *Artificial Life V*, pages 116–123. MIT Press, 1997.

[5] Brian Bershad, Craig Chambers Susan Eggers, and Others. The spin operating system. http://www.cs.washington.edu/research/projects/spin/www/.

[6] Rodney A. Brooks. A robust layered control system for a mobile robot. A.I. Memo 864, Massachusetts Institute of Technology, September 1985.

[7] Rodney A. Brooks and Anita M. Flynn. Fast, cheap, and out of control: a robot invasion of the solar system. *Journal of The British Interplanetary Society*, 42:478–485, 1989.

[8] Mark Burgess. cfengine home page. http://www.iu.hioslo.no/~mark/cfengine/index.html.

[9] Mark Burgess. Computer immunology. http://www.iu.hioslo.no/~mark/research/AIdrift/, October 28 1998.

[10] CERT Coordination Center. Cert advisories. http://www.cert.org/advisories/.

[11] Jr. Charles A. Janeway and Paul Travers. *Immunobiology: the Immune System in Health and Disease*. Garland Publishing Inc., New York, second edition edition, 1996.

[12] International Business Machines Corporation. *AIX Version 4.3 System Management Guide: Operating System and Devices*, chapter Understanding Paging Space Allocation Policies. IBM, 1997. http://www.rs6000.ibm.com/doc_link/en_US/a_doc_lib/aixbman/baseadmn/pag_space_under.htm.

[13] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, January 1998.

[14] Helena Curtis and N. Sue Barnes. *Biology*. Worth Publishers, Inc., New York, 5th edition, 1989.

[15] Dan Farmer and Wietse Venema. Satan home page. http://www.trouble.org/~zen/satan/satan.html, 1995.

[16] Daniel Farmer and Eugene H. Spafford. The COPS security checker system. In *Proceedings of the Summer Usenix Conference*, pages 165–170, June 1990.

[17] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for UNIX processes. In *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*. IEEE Press, 1996.

[18] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 1996 USENIX Security Symposium*, 1996.

[19] L.T. Heberlein, G.V. Dias, K.N. Levitt, and B. Mukherjee. A network security monitor. In *Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy*, pages 296–304, 1990.

[20] Steven A. Hofmeyr. An overview of the immune system. Technical report, University of New Mexico, 1997. http://www.cs.unm.edu/~immsec/html-imm/immune-system.html.

[21] Steven A. Hofmeyr, Anil Somayaji, and Stephanie Forrest. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 1998. In Press.

[22] Jeffrey O. Kephart, Gregory B. Sorkin, David M. Chess, and Steve R. White. Fighting computer viruses. *Scientific American*, November 1997. http://www.sciam.com/1197issue/1197kephart.html.

[23] Gene H. Kim and Eugene H. Spafford. Experiences with tripwire: Using integrity checkers for intrusion detection. Technical Report CSD-TR-94-012, Department of Computer Sciences, Purdue University, February 21 1994.

[24] Calvin Ko, George Fink, and Karl Levitt. Automated detection of vulnerabilities in priviledged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, December 5–9 1994.

[25] David Kortenkamp, R. Peter Bonasso, and Robin Murphy, editors. *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*. AAAI Press/The MIT Press, 1998.

[26] T.F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P.G. Neumann, H.S. Javitz, A. Valdes, and T.D. Garvey. A real-time intrusion detection expert system (IDES) — final technical report. Computer Science Laboratory, SRI International, Menlo Park, California, February 1992.

[27] Michael R. Lyu, editor. *Software Fault Tolerance*, volume 3 of *Trends in Software*. John Wiley & Sons, New York, 1995.

[28] Ken Phillips. One if by net, two if by OS: RealSecure halts host intrusion but can't cover the enterprise. *PC Week*, February 14 1999. http://www.zdnet.com/products/stories/reviews/0,4161,389071,00.html.

[29] Margo Seltzer et al. The vino operating system. http://www.eecs.harvard.edu/~vino/vino/.

[30] Margo Seltzer and Christopher Small. Self-monitoring and self-adapting systems. In *Proceedings of the 1997 Workshop on Hot Topics on Operating Systems*, Chatham, MA, May 1997. http://www.eecs.harvard.edu/~vino/vino/papers/monitor.html/.

[31] Arthur J. Vander, James H. Sherman, and Dorothy S. Luciano. *Human Physiology: the Mechanisms of Body Function*. McGraw-Hill Publishing Co., New York, 1990.

[32] Wietse Venema. TCP WRAPPER: network monitoring, access control, and booby traps. In *Proceedings of the 3rd UNIX Security Symposium*, 1992. ftp://ftp.win.tue.nl/pub/security/tcp_wrapper.ps.Z.

[33] Network World and IDG News Service staff. Is anyone listening to bill gates' theory on the PC's future? *CNN Interactive*, November 18 1998. http://cnn.com/TECH/computing/9811/18/listen.bill.cdx.idg/index.html.

[34] Stephen Young. Dr. Stephen Young's home page. http://rheumb.bham.ac.uk/youngsp.html, February 1995.