Anomaly Detection Using Self/Nonself Discrimination for the Linux Kernel

Lars Olsson
Abstract Void Computing
12 Harrington Road
Brighton, BN1 6RE
England
lars@abstractvoid.net

Abstract

In this paper we discuss how computers can protect themselves from different forms of attacks, mis-configurations, and program errors. The work is inspired by the immune system and in a similar vein to the immune system our system learns how to distinguish self from nonself. The system is implemented as a couple of modules to the Linux kernel and analyses each system call of the monitored processes. To build and analyse profiles of the system calls we have implemented three different methods; a table lookup method, a feed-forward neural network, and an Elman recurrent neural network. Experiments show that this system can detect several methods of intrusion including buffer overflow attacks, format string exploits, and Trojan code.

1 Introduction

As computer systems become more and more interconnected they become more vulnerable to various forms of deliberate attacks, configuration mistakes, and program errors. There is also a growing need for more autonomous and self-controlling systems since the abundance of computers makes it impossible for all computers to be under close supervision by humans. If computers are to be left alone for extended periods of time without human intervention they also need to know how to protect themselves. Hence, the major question for computer science is not performance anymore, but *survival*.

How should a system be designed in order to survive? If one looks at animals, which are experts at survival, they all seem to share some common design ideas. One of the most important ideas is that of *homeostasis*. Homeostasis is basically a way of maintaining stability, for example keeping a constant body temperature, through sensors and actuators coupled in feedback loops. One system existing in all mammals is the immune system that can distinguish between cells belonging to the animal (self) and cells not belonging to the animal (nonself).

The work presented in this paper investigates how ideas taken from biology can be used to create computer systems that can protect themselves with as little human help as possible. More specifically, we look at how the Linux kernel can protect itself from processes that start to behave differently than the normal behaviour, where the abnormal behaviour can be caused by for example buffer overflow attacks, format-string exploits, or Trojan code. To detect abnormal behaviour profiles are built of the system calls that processes make to the operating system since these are (in modern operating systems) the only way a user space process can interact with the operating system and the hardware.

1.1 Background and Related Work

The seminal paper [6] by Forrest et. al. was the paper that established the analogy between the human immune system and intrusion detection. They did this by showing how correlations in fixed-length sequences of system calls executed by a process could be used as a signature, or profile, determining the "self" of each program. They analysed traces of system calls from several quite complex UNIX programs, including `sendmail` and `lpr`, and found that it was possible to build profiles of these programs using short sequences of system calls. The profiles could then be used to see if the process was behaving "normal" or not since different kinds of common intrusions changed the sequences of system calls. The system they developed was only used off-line using previously collected data and used a quite simple table-lookup algorithm to learn the profiles of programs. In [21], Forrest et. al. analyses alternatives to the table-lookup algorithm used in [6], including methods based on frequencies of different system calls, a data mining technique and Hidden Markov Models (HMMs). They show that HMMs have exceptional accuracy, but at a high computational cost. The simpler table-based methods performs almost as good and are much cheaper to execute.

Several other research groups then tried more complex machine learning algorithms. Ghosh et. al. [7] tried several different machine learning algorithms including another table-lookup algorithm and a back-propagation feed-forward neural network. Most of the early algorithms used fixed sequence lengths to analyse the sequences. In [22], the authors describe how they have used a variable length sequence matching algorithm known as the Teiresias pattern-discovery algorithm to identify anomalies in sequences of system calls. Sekar and Uppuluri show in [17] how a language they have designed called REE (regular expressions for events) can be used in conjunction with an extended finite state machine to generate an intrusion detection system. A similar approach is used by Kerschbaum et. al. in [11] where they show how it is possible to add code to the operating system and the user programs to detect attacks. This, of course, assumes that the source code to both the programs and the operating system is available to the user.

As far as we can tell, all examples discussed above except [11], were all implemented as proof of concept off-line systems and not as real-time intrusion detection systems embedded in the operating system. The first real anomaly detection system actually using the ideas developed in [6] was implemented by Anil Somayaji while he has writing his PhD thesis [19]. The system, *pH* (process Homeostasis), was implemented for the Linux 2.2 kernel and has been a great inspiration for the work presented in this work. In his thesis, Somayaji shows how the operating system can try to maintain a stable environment by slowing down processes that do not behave according to the learned profile. The user can then decide if the anomalies are part of the normal behaviour of the process or not. The system developed in this work is quite similar to [19], and we have even borrowed about 40 lines of code from Somayaji's system. One major difference is the design and another difference is some changes to Somayaji's profile training algorithm and that our system also can use different training algorithms.

The rest of the paper is structured as follows. In Section two we describe and discuss the biological inspiration for this work, namely homeostasis and immune systems. In Section three we describe how the behaviour of programs can be seen as sequences of system calls. We then describe the design and implementation of our system in Section four. In Section five we describe the experiments we have done and the results and Section six contains our conclusions.

2 Homeostasis and Immune Systems

All biological systems maintain a stable internal state by monitoring and responding to both internal and external changes [3]. This self-monitoring can be seen as one of the defining properties of life and is generally known as *homeostasis*. Although many homeostatic systems have been studied extensively, most are still not completely understood. One general feature seems to be the use of sensors that monitor some property of the body, for example body temperature. The sensors are then coupled to effectors [1], for example blood vessels in the extremities, that can contract when the body is too cold. This reserves a greater proportion of the body heat to the inner core of the body and hence makes the individual warmer.

Homeostatic systems generally have the following properties: a state that need to be maintained, a closed system where this state is to be maintained, a sensor that can detect the current state, and an effector that can change the state of the monitored state. Table 1 summarizes these properties and shows how they relate to one homeostatic system, namely temperature regulation.

Abstraction	Temperature Regulation
closed system	body of an individual
state	temperature
sensor	specialized nerve cells
effector	muscles, sweat, glands, etc

Table 1: The four defining properties of a homeostatic system.

To use biological metaphors in computing the right level of abstraction needs to be found. One should also think about what abstractions in the artificial system that correspond to what abstraction in biology. Following [19], Table 2 shows how the biological abstractions found in Table 1 can be used to find abstractions to build an intrusion detection system.

Abstraction	Anomaly detection
closed system	individual computer
state	normal program execution
sensor	sequences of system calls
effector	delayed or canceled calls

Table 2: The four defining properties of a homeostatic intrusion detection system based on analysing sequences of system calls.

2.1 Immune Systems

The immune system is a complex homeostatic system that is responsible for defending the body (self) from misbehaving and foreign cells (nonsself). A good overview of the immune system for computer scientists can be found in [16] and a more biological overview can be found in [3].

The immune system consists of a large number of cells and molecules which interact in a number of different ways to detect and eliminate dangerous agents (pathogens). Since the interactions depend on chemical bonding they are localised. Surfaces of the immune system cells are covered with receptors, some which bind other immune system cells or molecules to achieve communication and signaling to mediate the immune response. Other cells bind chemically with pathogens. Most of these cells travel around the body in the blood and lymph systems, without centralised control and very little, if any, hierarchical organisation [9]. Detection and elimination of pathogens is the result of trillions of cells interacting following simple, local rules.

As discussed in [5], immune systems have several properties that make them attractive from a computer science point of view. They are *distributed* across the whole body with no central controller and hence there is no single point of failure. Immune systems are also *diverse*, which enhances the robustness on both individual and population level, since different people may be vulnerable to different pathogens. The components of immune systems are also constantly created, destroyed, and moved around the body, leading to a *dynamic* system that increases temporal and spatial diversity. The last, and maybe most important feature of the immune system, together with its distributed nature, is that it can *adapt*, i.e. learn, to recognize and respond to new foreign pathogens. All of these features added together sounds like the ideal design for a distributed computer system.

What is needed, then, is a way to design and structure computer systems as homeostatic systems using immune systems and other forms of feedback loops for control. In this work we investigate how one of these systems, namely a security system based on analysing sequences of system calls, can be designed. In a sense, this can be seen as a low-level reflex system, similar to movements that are carried out without consultation of the brain, such as moving your hand from a hot stove. To build a really stable and secure system built on these principles, other feedback systems, designed around for example log analyzers, network usage, and other system properties, must be designed.

3 System Calls and Program Behaviour

When a program is used it is used because the user expects it to perform a certain task. If the program cannot perform the task, maybe because of faulty input or a broken network card, the user expects the program to halt the execution and inform the user. This can be defined as the legal program behaviour. If, on the other hand, the program does something that it is not intended to do, because of a programmer error, misconfiguration, or a deliberate attack by another user, this can be seen as illegal or abnormal behaviour. The way a program normally behaves on a particular computer with a particular configuration can then be seen as the normal behaviour. Figure 1 displays this relation between normal, legal, and abnormal program behaviour.

Since a user space program only interact with the kernel, and hence all hardware devices such as network interfaces, hard drives and filesystems, through system calls, the behaviour of a process can be seen (from the kernel's point of view) as a sequence of system calls and the arguments to the system calls [2]. If we define the sequences of system calls that a program executes during normal execution as self, the problem of detecting abnormal behaviour is then the problem of finding sequences of system calls that are not in the profile of normal behaviour, that is, nonself. To build these profiles a number of different machine learning algorithms can be used. A few of these are described in the next two sections.

3.1 Profiles of System Calls

Since most useful programs execute thousands or even millions of system calls during their lifetime, it is necessary to focus on sub-sequences, or windows of system calls. Also, since a system that analyses system calls should be able to detect abnormal behaviour while it is occurring, it also needs to execute fast since it will most probably be part of the kernel. The problem is then to build a profile of sequences of system calls that define the self of the program. More formally, given

$$\begin{aligned}
C &= \text{the alphabet of all system calls} \\
c &= |C| \text{ (225 in Linux 2.4.17)} \\
T &= t_1, t_2, \ldots, t_\tau | t_i \in C \text{ (The trace of calls)} \\
\tau &= \text{the length of T} \\
w &= \text{the window size where } 1 \leq w \leq \tau
\end{aligned}$$

Figure 1: The different classes of computer program behaviour.

the problem is then to build the profile

$$P = \text{patterns associated with } T \text{ and } w$$

and to detect whether a given sequence S of length w exists in P. An algorithm should also be able to say how abnormal S is, given P, maybe using Hamming distance or another way of measuring the difference between two sequences.

3.2 Lookahead-Pairs

In [19], Anil Somayaji uses an algorithm called lookahead pairs to build profiles. Here a window is slid over the sequence of system calls, recording for each system call what call(s) that came before it in the current window. More formally, given the definitions given above in Section 3.1 , we can define a profile P_{pair} as consisting of a number of pairs of system calls as:

$$\begin{aligned} P_{pair} = \{\langle s_i, s_j\rangle_l : \quad & s_i, s_j \in C, 2 \leq l \leq w \\ \exists p : \quad & 1 \leq p \leq \tau - l + 1, \\ & t_p = s_i, \\ & t_{p+l-1} = s_j\} \end{aligned}$$

How can we know when the training of the profile is finished? In [19] the author describes a two-step heuristic that depends on the total number of calls seen so far and how many calls that have been seen since the profile was last edited. Let *total_count* be the total number of seen calls and *last_mod_count* the number of calls seen since the profile was last edited. The profile is *frozen* when $\frac{total\_count}{(total\_count - last\_mod\_count)} > 4$. Then, if no changes changes has been made to the profile after *normal_wait* seconds, the profile is said to be normal and ready to be used. If the profile is changed when it is frozen it is thawed and the frozen flag set to 0.

3.3 Neural Networks

Another way to build profiles of traces of system calls is to use neural networks [7]. In this work we have used to different types of neural networks, the classic *back-propagation feed-forward* [8] and *Elman recurrent* neural networks [4].

The architecture of each type of network is shown in Figure 2. For both types of networks each input neuron represents one call in the window of system calls, Hence, if a window size of 8 is used the network has 8 input neurons. For the feed-forward network there is one output neuron whose value depends on whether the input sequence is abnormal or not. Elman networks have memory neurons and are used to predict sequences. Usually they have as many output neurons as input neurons where the values of the output neurons should be the next sequence. The sum of the absolute differences between the next input values and the values of the output neurons is used to calculate the error for the training.

The training is done using the back-propagation algorithm [8] for both the feed-forward network and the Elman network. Since the space of possible sequences is much larger than the space of normal sequences the training starts by training the networks with random data. This random data is classified as abnormal behaviour. Then the network is trained to detect the training (normal) data as normal. To get good performance a number of networks can be trained on the same data using the methods described in [7], such as varying the number of hidden nodes since the optimal number of hidden

Figure 2: The architecture of a feed-forward neural network (F) and an Elman recurrent neural network (E). The nodes labeled I are input nodes, the nodes labeled H are hidden neurons, and the nodes labeled O are the output nodes. In the Elman recurrent net the nodes labeled C are the context (memory) nodes.

nodes is not known before training. Because of this we have only been able to train the neural network profiles off-line. To compute how abnormal a sequence is we use the *leaky-bucket algorithm* described in [7]. The leaky-bucket algorithm keeps a memory of recent events by accumulating the networks output. This value is then is slowly leaking out. Thus, if there are several abnormal system calls after each other the abnormal count will quickly accumulate a large value. If, on the other hand, the calls are normal, the anomaly counter will "leak" away the anomaly value and eventually reach zero.

4 Implementation

To test our ideas we have implemented a system called *Homeos*. Homeos was implemented using the Linux 2.4.17 kernel but should compile and work on any 2.4 kernel. Conceptually the implemented system consists of six different parts. (1) First we need to access all system calls as they are executed. (2) Each executable that we wish to monitor should be represented by a profile. (3) Then we need to train the profiles of the monitored programs. (4) These profiles are then used to see if a process is behaving abnormal. (5) Abnormally behaving processes are then slowed down depending on how abnormal they behave. (6) What programs/processes to monitor and the administration of the system is done through a user interface. This interface also shows information about processes and executing processes. data. Figure 3 shows the relation between these parts and the rest of the operating system. In reality, the system is divided in two different modules. The first module is responsible for accessing all system calls as they are executed and the second module handles the profiles.

4.1 System Call Monitoring

The first problem we faced was how to monitor each system call. Since the system is implemented as modules it is impossible to add code to the actual system call dispatcher. Instead, the system call hijacker module implements its own version of each system call. When the module is initialised each function pointer in the global `sys_call_table` array is replaced with the corresponding function implemented by us and the original pointers are saved (these are the pointers that are used to call the original system calls)[1].

The reason to separate the system call module from the rest of the system was twofold. Firstly, by separating this functionality to a separate module it can also be used by other modules that need to monitor system calls. Secondly, and more important, is the possibility of race conditions [2] if the system is unloaded when no longer needed. Suppose that the whole system was built as one module. When the module is unloaded the original system call function pointers need to be put back in the `sys_call_table`. This in itself is no problem, since it is possible to turn off all interrupts and hence make the changes to the array atomic. The problem is that some system calls might take seconds, or maybe even days to execute, because they are waiting for some special event to occur. When these calls return, they return to

[1]Note that this will not work in Linux 2.5 since this array no longer is global

Figure 3: The system design. Our system consists of the changed syscalls, train, process_syscall and delay_process module, profiles, and interface subsystems.

the address of the corresponding function in the hijacking code. But, since this code is unloaded, this address can contain anything and a total system crash is likely to occur. The solution to this is to separate the hijacking code to a separate module that is never unloaded. Instead, when the user requests to unload the module it changes the pointers in `sys_call_table` back to the original system calls, but does not remove itself (in reality, refuses to be removed) from memory. Since this module only occupies around 13k of memory this is not a high price to pay for a stable system.

4.2 Profiles and Training

The other module is responsible for all functionality relating to the program profiles and the interface to the user. Each program that is monitored has one profile with a window size of 8, which means that each profile uses 130k of memory. Since each program can be executed as several independent processes at the same time, each process has an individual state structure but shares the actual profile with all processes executing the same program. Hence, several processes can train and hence update a single profile at the same time. This speeds up training if we assume that all instances of an executing program behaves similarly. Profiles are saved to file each time a process exits but they are also cached in memory for faster recollection. When the user adds a program to monitor via the user interface it is added to a list of monitored programs. Each time the `execve()` (the only way to execute a new process) system call is executed the list with monitored programs is traversed. If the program is found the new process is added to the monitoring system. If the profile (the profile for the program this process is executing) is not found in the cache or on disk a new empty profile is created.

4.3 Scheduling of Abnormal Processes

If a process is behaving abnormal, it is slowed down by putting the process sleep while it is executing system calls. The length of the sleep depends on how abnormal the process has behaved. In the current implementation each process has a anomaly window consisting of the 128 latest executed system calls. This window is originally filled only with 0's but for each abnormal call the value is changed to 1. Thus, it is possible to keep track on how many abnormal call this process has done in the last 128 calls since a normal call will change the value back to 0. This value is used to calculate the number of milliseconds to sleep. Let A_b be the number of abnormal calls the last 128 calls. Then the process should sleep for $delay\_factor * 2^{A_b}$ jiffies. Jiffies is the internal time measurement in Linux and is generally $10ms$ on the x86 platform, which means that the operating system switches between processes 100 times per second. Observe that the amount of time that abnormal processes sleep increases exponentially depending on the value A_b.

4.4 User Interface

The user communicates with the system module by writing and reading to/from files in the `/proc/homeos/` directory which resides in the *proc filesystem*. The proc filesystem is a virtual filesystem (VFS), which means that it does not represent a physical device [2]. It is designed to allow easy access to information about processes (hence the name) and it is nowadays used by every part of the Linux kernel which has something interesting to report.

The user can influence the running system in several ways. The user can add and delete programs to monitor. It is also possible to see the properties of currently monitored processes. The properties include if the profile of the process is frozen or normal, number of anomalies, if it is currently delayed, and, in that case, how long it will sleep.

The user can also influence running processes in six different ways:

- A monitored process can be killed.
- A monitored process can be stopped being monitored.
- The profile of a process can be reset.
- A process' profile can be set to normal so that normal monitoring starts.
- A process can be *sensitized*.
- A process can be *tolerized*.

The tolerize and sensitize commands have the same meaning as in [19] and are inspired by similar processes in real immune systems. Tolerize means that the user regards recent program behaviour as normal even if the system classified it as abnormal. Sensitize, on the other hand, tells the system that the recently learned behaviour for a process should be forgotten by the profile.

5 Experiments and Analysis

In this Section we described some experiments we have done with the implemented system. We also describe and discuss some information-theoretic measurements we have performed on live-data of system calls from several well known server programs including *sendmail* and the ftp-server *wu-ftp*.

5.1 Buffer Overflow Attacks

Buffer overflows all involve a program that uses a fixed amount of storage to save some kind of external data [14]. If the program (programmer) does not ensure that the external data fits in the storage, it is possible to overwrite neighbouring memory locations, for example return addresses from functions. To make this more concrete, consider the small program in Figure 4. If this program is given input that is longer than 1024 bytes, the data following the 1024th byte will be written over other data on the stack. On Intel processors it turns out that the return address (where the program should start executing when it returns from `parse(char *arg)`) is stored below the array `param[1024]` in the stack. Since the stack grows toward lower addresses [2], a string longer than 1024 bytes (in this case), will overwrite the return address. This usually results in a `Segmentation Fault` or `Illegal Instruction` since the return address is overwritten and usually points to a random position in memory. But, if a specially designed string is constructed where the return address points to an address where executable code is found, this code will be executed. To see if our system could detect typical buffer overflow attacks we deigned our own buffer overflow program similar to the program described in [14].

Before we used the program designed to induce a buffer overflow attack we used the program `strace` to look at the sequence of system calls generated by *breakme.c*, found in Figure 5.1. The first thing to note here is the argument string given to *breakme.c* that can be seen in the first `execve` system call. The A:s in the beginning are the NOP instructions that are needed to fill out the buffer. Then we can see the actual code that executes the system call (`\ùÄPSAÊ°Í`). Then follows the program to be executed, `/tmp/sh`, followed by the address that overwrites the return address in the stack. The next thing

```c
#include <stdio.h>
#include <string.h>

void parse(char *arg)
{
    char param[1024];
    strcpy(param,arg); /* potential buffer overflow */
}

main(int argc, char *argv[])
{
    parse(argv[1]);
}
```

Figure 4: *breakme.c*.A program that can be attacked by a buffer overflow. Note that the char array is 1024 characters long. *strcpy* does not check the length of the buffer it copies. Thus, if *arg* is longer than the buffer, the stack will be overwritten.

```
[lars@galadriel buffer>strace ./breakme ‘./overflow 1009‘
execve("./breakme", ["./breakme", "AAAAAAAAAA...AAAAAAAAAA
$ùÄPSAÊ°Í/tmp/sh‘øÿ>"],
...
execve("/tmp/sh", ["/tmp/sh"], [/* 1 var */]) = 0 /* the overflow */
...
```

Figure 5: The sequence of system calls executed by *breakme.c* if given the special overflow string written by *overflow.c*. Note that this trace has been edited in several places for space reasons.

to note is the second `execve` call. Here we can see that `/tmp/sh` is executed and hence the buffer overflow has succeeded in executing another program!

This is of course bad news that it is possible to execute arbitrary programs by overflowing buffers. The good news is that a buffer overflow probably will execute a sequence of system calls that is not found in the normal behaviour of the program. Hence, a system like ours that detects changes from normal behaviour by analysing sequences of system calls will always detect a buffer overflow if it is correctly trained. Consequently, when we tried to train our system with *breakme.c* above, it managed to detect the buffer overflow when we started *breakme.c* with input from *overflow.c*.

5.2 Format String Exploits

Another kind of commonly used intrusion technique is *format string exploits*. Here the idea is to exploit the fact that the `printf` family of C functions, for example `fprintf` and `snprintf`, all use format strings to control the output [10]. If the programmer forgets to add a format string when he prints out a string that the user has typed in using any of these functions, the user can supply his own format string in the input. By cleverly crafting your own format string it is for example possible to read or even change the value of any variable in the program, see for example [15]. In itself, this is something that can not be detected by our system, unless the changed value of a variable causes the program to execute a piece of code never executed before.

Format string exploits can also be used to gain root access to a system. In a fashion somewhat similar to the buffer overflow described above it is actually possible to design an input string that contains the machine code for starting a shell. To test this we used a program called *fmtbrute.c* downloaded from [15] to generate this string. We then trained our system on *vulnerable.c*. Then, when the profile was normal we used the string generated by *fmtbrute.c* as input to *vulnerable.c*. As expected, our system succeeded in detecting the exploit that gives the exploiter a root shell (if *vulnerable.c* is started as suid root).

5.3 Trojan Code

The aim of this experiment was to see that the system can detect changes in the way a program behaves because the executable file has been replaced or recompiled with another configuration. This can for example be the case in a Trojan Horse attack, where a commonly used program, for example `su` has been replaced or recompiled with new code. This code usually leaves a backdoor that the hacker can use even though he no longer knows the root password. To test this form of attack we used an *sshd* backdoor [20], a source code patch for the Secure Shell that allows users to connect to a foreign machine using an encrypted communication channel. When this patch is applied it is possible for a user to login on the machine where the patched version is installed by typing a special password that is compiled in to the compromised program. If this password is used *sshd* gives the user a root shell without using the normal authentication and logging procedures. We first used the normal version of *sshd* to get a normal profile and to analyse the sequences of system calls manually with `strace`. We then tried the patched version and found that the profiles were similar. But, when we tried to login with the special password an `execve` call materialised where there had not been a `execve` call before. Hence, the system was able to detect this intrusion as well.

5.4 Performance of Different Learning Algorithms

In the experiments described above we tried both the simple table-lookup method and the more complex neural network learning algorithms. Altough the Elman neural network performed well there are a couple of problems with using neural networks in real on-line systems. One problem is execution speed. Even though neural networks are considered to be fast machine learning algorithms they are much slower than simpler table lookup methods. We have not done enough experiments to say anything certain but it seems like the neural network based algorithms slows down the computer around 20-25% while the table lookup methods slows down the computer around 5% when system call intensive applications are executed. Also observe that the Elman recurrent neural net, which performs better than the feed-forward neural net, is even slower since it contains more nodes since it uses an extra set of hidden nodes (the context nodes). Another problem is the fact that neural networks usually use floating point representations, which can cause problems when they execute in kernel mode. The reason for this is that the Linux kernel does not save the floating point registers when it switches from user mode to kernel mode and back due to performance reasons [2]. This can be solved in two ways. Either by implementing an integer based neural network or by saving the floating point registers each time before the neural network is executed and then restoring them afterwards. This is quite costly and since the system probably will execute up to hundreds or even thousands of times per second, this can not be considered a feasible option.

5.5 Determinism of Sequences of System Calls

In *information theory* [18] *entropy* is used to measure the uncertainty, or impurity, of collections of data. Formally, the entropy of the data source X is defined as

$$H(X) = \sum_{i}^{N} p_i \log p_i$$

where N is the finite number of states that the data in X can be in and p_i the probability for a variable to be in state x_i. Table 3 shows the entropy for the system call traces for a number of programs. The first thing to note here is that the average program uses around 50 different system calls. It is also interesting to note that the two server programs, `sendmail` and `wu-ftp` are the two programs with the lowest entropy. This means that these traces are more regular (and hence easier to predict) than for example `xmms` and `emacs`. These results are not surprising since `emacs` is a program that interacts with the user while `sendmail` and `wu-ftp` both are server programs that basically execute a loop that waits for clients to connect to the server.

Program	Unique calls	*log*$_2$	Entropy
sendmail	48	5.58	2.66
wu-ftp	48	5.58	2.16
xmms	53	5.72	2.95
emacs	46	5.52	3.49

Table 3: The entropy for the traces of a number of different programs. Unique calls denotes the number of uniquely used system calls, *log*$_2$ the number of bits needed to represent one system call in trace if all calls have the same probability, and entropy the entropy of the trace.

Following work done in [12], we can measure the determinism and regularity in a trace of system calls by computing the *conditional entropy* of the trace. Let Y be a collection of sequences of system calls all of the length n (the window size). Each sequence is denoted as $(c_1, c_2, \ldots, c_{n-1}, c_n)$ where each c_i is a system call. Moreover, let X be a collection of subsequences where each subsequence is defined as $(c_1, c_2, \ldots, c_{n-1})$. Hence, X is the collection of all subsequences with a window one call shorter than Y. Then the conditional entropy $H(Y|X)$ measures how much uncertainty that remains in a sequence y after we have seen x. More formally, conditional entropy is defined as

$$H(Y|X) = -\sum_{i}^{N}\sum_{j}^{N} p(x_i, y_j) \log p(y_j|x_i)$$

This can be seen as a measure of how hard the data is to model.

To investigate our idea that the sequences of system calls are quite regular and deterministic, we computed the conditional entropy for a number of traces including data from `sendmail`, `wu-ftp`, and `ps`[2]. The conditional entropy in this case measures how the $n-1$ first system calls in a sequence determines the nth system call where a result of zero means that it is completely deterministic. The traces for `sendmail` and `wu-ftp` contained 48 different system calls each and the `ps` trace consisted of 22 different calls. Figure 6 shows the results for a window size between 2 and 20 for `sendmail`, `wu-ftp`, and `ps`. Here we can see that the conditional entropy for the `wu-ftp` program and `ps` is highly similar. With a window bigger than 4 the conditional entropy is less than 0.1, suggesting that the value of the nth call is highly deterministic when the $n-1$ first system calls have been seen. For `sendmail` the conditional entropy starts of high but gets down to the values of `ps` and `wu-ftp` with a window bigger than six calls. With a window bigger than 12 calls the value is almost 0.0, indicating a highly predictive behaviour. The con-

[2]downloaded from `http://www.cs.unm.edu/~immsec`

Figure 6: Conditional entropy for normal traces and exploit traces.

ditional entropy for the abnormal traces (exploits) are quite similar to the normal traces, the difference is that they are slightly less predictive with small window sizes.

6 Conclusions

In this paper we have studied how operating systems can be designed to detect and respond to various forms of anomalous behaviour and we have also built a system based on the studied principles. To detect anomalies we analyse the sequences of system calls that user space applications call in the operating system kernel. We argue that this is the best level to place an intrusion detection system since this is the only way for programs to actually access files or hardware on modern computers.

To detect abnormal behaviour profiles are built for each application that models the normal sequences of system calls that the application calls. Thus, the problem of detecting abnormal behaviour is to distinguish the normal behaviour (self), from abnormal behaviour (nonself). There is a number of ways to build the profiles and in this work we have tried three different methods; a lookup-table, feed-forward neural network, and an Elman recurrent neural network. Results from the experiments show that the Elman network and feed-forward network are too slow to use in an on-line system since they need to execute for each executed system call as a part of the Linux kernel. Hence, speed is extremely important and of the tested methods we argue that only the lookup-table method is fast enough.

There are also a number of problems with the suggested approach to anomaly detection. Maybe the most important is Denial of Service attacks that can exploit the fact that abnormally behaving programs are slowed down. Thus it is possible for an attacker to deny others to use an application by deliberately slowing it down. Another problem is to build good models of normal behaviour. If the model is not good enough it will alarm even though the program is behaving normal or not alarm when the program is behaving abnormal.

To conclude we definitely believe that system call based anomaly detection has a future. As computers become more abundant and autonomous they need to use every method they can to survive. Discrimination between self and nonself will most probably be one of them.

References

[1] W. Ross Ashby, *An Introduction to Cybernetics*, Chapman & Hill, London, 1956, Internet (1999): `http://pcp.vub.ac.be/books/IntroCyb.pdf`

[2] Daniel P. Bovet and Marco Cesati, *Understanding the Linux Kernel*, O'Reilly, 2001

[3] Neil A. Campbell, *Biology* 4th edition, The Benjamin Cummings Publishing Company, 1996

[4] Jeffrey L. Elman, Finding Structure in Time, in *Cognitive Science 14*, 1990

[5] Stephanie Forrest and Stephen Hofmeyr, Information Processing in the Immune System, in Lee A. Segel and Irun R. Cohen editors, *Design Principles for the Immune System and Other Distributed Autonomous Systems*, Oxford University Press, 2001

[6] Stephanie Forrest, Steven Hofmeyr, Anil Somayaji, and Thomas A. Longstaff, A Sense of Self for Unix Processes, in *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*, IEEE Computer Society Press, 1996

[7] Anup K. Ghosh, Aaron Schwartzbard, and Michael Schatz, Learning Program Behaviour for Intrusion Detection, in *USENIX Proceedings of the Workshop on Intrusion Detection and Network Monitoring*, 1999

[8] Simon Haykin, *Neural Networks* second edition, Prentice-Hall, 1999

[9] Stephen Hofmeyr and Stephanie Forrest, Architecture for an Artificial Immune System, in *Journal of Evolut ionary Computation*, 2000

[10] Brian W. Kernigan, Dennis M. Ritchie, *The C Programming Language* second edition, Prentice-Hall, 1988

[11] Florian Kerschbaum, Eugene H. Spafford, and Diego Zamboni, Using Embedded Sensors for Detecting Network Attacks, in *Proceedings of the First ACM Workshop on Intrusion Detection Systems*,

[12] Wenke Lee and Dong Xiang, Information-Theoretic Measures for Anomaly Detection, In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001

[13] David Newman, Joel Snyder, and Rodney Thayer, Crying Wolf: False Alarms Hide Attacks, in *Network World*, `http://www.networkworld.com`, June 2002

[14] Aleph One, Smashing the Stack for Fun and Profit, in *Phrack* issue 49, `http://www.phrack.org`, 1998

[15] scut, Exploiting Format String Vulnerability, `http://www.team-teso.net/articles/formatstring`, March 2001

[16] Lee A. Segel and Irun R. Cohen editors, *Design Principles for the Immune System and Other Distributed Autonomous Systems*, Oxford University Press, 2001

[17] R. Sekar and P. Uppuluri, Synthesizing Fast Intrusion Prevention/Detection Systems from High Level Specificiations, in *Proceedings 8th Usenix Security Symposium*, August 1999

[18] Claude E. Shannon and Warren Weaver, *The Mathematical Theory of Communication*, University of Illinois Press, 1949

[19] Anil Somayaji, *Operating System Stability and Security through Process Homeostasis*, PhD. thesis, The University of New Mexico, July 2002

[20] timecop, Root kit SSH 5.0, `http://www.ne.jp/asahi/linux/timecop`, January 2000

[21] Christina Warrender, Stephanie Forrest, Barak Pearlmutter, Detecting Intrusions Using System Calls: Alternative Data Models, in *Proceedings of the IEEE Symposium on Security and Privacy*, 1999

[22] Andreas Wespi, Marc Dacier, and Herve Debar, An Intrusion-Detection System Based on the Teiresias Pattern-Discovery Algorithm, in *EICAR Proceedings*, 1999