ON-LINE INTRUSION DETECTION USING SEQUENCES OF SYSTEM CALLS

Name: Damon Snyder
Department: Department of Computer Science
Major Professor: Robert van Engelen
Major Professor: Kyle Gallivan
Degree: Master of Science
Term Degree Awarded: Spring, 2001

This thesis investigates the use of anomaly detection techniques for on-line intrusion detection. A detailed analysis of methods introduced by Forrest et al. is presented and an improved characterization of intrusions is derived. A novel approach to integrating this characterization into an on-line intrusion detection system is developed and implemented. Finally, experiments examining the efficiency, flexibility and efficacy of the system are presented.

THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

ON-LINE INTRUSION DETECTION USING SEQUENCES OF SYSTEM CALLS

By

DAMON SYNDER

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Spring Semester, 2001

The members of the Committee approve the thesis of Damon Snyder defended on MARCH 28, 2001.

Robert van Engelen
Professor Directing Thesis

Kyle Gallivan
Professor Co-Directing Thesis

Theodore P. Baker
Committee Member

Approved:

Theodore P. Baker, Chair
Department of Computer Science

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This thesis investigates the use of anomaly detection techniques for on-line intrusion detection. A detailed analysis of methods introduced by Forrest et al. is presented and an improved characterization of intrusions is derived. A novel approach to integrating this characterization into an on-line intrusion detection system is developed and implemented. Finally, experiments examining the efficiency, flexibility and efficacy of the system are presented.

CHAPTER 1

INTRODUCTION

Intrusion detection systems based on examining sequences of system calls often define normal behavior of an application by sliding a window across a system call trace of an application and recording the positional relationships between system calls in the window. System call traces are normally produced with programs like `strace` on Linux systems and `truss` on Solaris systems. Several methods have been proposed for storing this information and using it to detect anomalies in an intrusion detection system. The first of these methods was proposed by Forrest et al., [7], where normal behavior was stored by sliding a window of size $k + 1$ across a system call trace and recording which system calls followed the system call in position 0 at offsets 1 through k. This work showed promise in detecting intrusions by representing the positional relationships between system calls within a short sliding window.

Later work, [11], attempted to quantify the signal of an intrusion so that it could be detected efficiently and effectively in real time. This technique captured normal behavior by storing the sequences of system calls explicitly in a database. The Hamming distance between an input window and the sequences in the database was used to determine the strength of an anomalous signal.

A variation on the method proposed in [7] records the system calls in the k positions preceding the last system call in the window [12]. An implementation was presented where every system call for every running process could be monitored using this sliding window approach in real time.

This thesis provides a detailed analysis of these sliding window implementations and the metrics used to measure the strength of an anomalous signal. A new metric is introduced that amplifies the signal of an intrusion and allows the definition of a threshold to be more rigorously established using techniques from queuing theory. Finally, a novel approach for integrating these sliding window implementations into an on-line intrusion detection system is presented.

1.1 Research Motivation

The coarse-grained access control mechanisms provided by the popular UNIX operating systems require the use of trusted helper applications with higher levels of security to perform needed functions on behalf of users. These trusted helper applications are frequently exploited for their heightened security level because of vulnerabilities in their design or the underlying operating system. The number of operating system vulnerabilities discovered each year is on the rise [4]. The number of new vulnerabilities reported across the most popular operating systems has doubled between January 1999 and January 2001. Although many of these vulnerabilities are fixed within days or weeks of their discovery, effective intrusion detection systems (IDS) are a powerful supplemental preventative security measure.

Although progress has been made in the area of intrusion detection, all of the commercial IDS have been implemented independently of the operating system they attempt to protect. It has been argued that this approach is flawed [18], and that an IDS should be integrated into the operating system. Such an integrated IDS could provide real time detection and response to intrusions.

The constraints placed on an on-line IDS are different from those placed on an IDS in user-space. Integrating an IDS into the operating system requires special attention to speed, overhead, and and flexibility (accuracy is implicit). If the resulting IDS is slow and takes up a large amount of unswappable memory, the performance of the system is compromised. Furthermore, in order to be accepted and used, the IDS must maximize the distance between the signal observed during normal behavior and the signal of an intrusion. This distance allows the security administrator the flexibility in setting the sensitivity of the IDS to fit the needs of the environment[1]. These properties are a function of the characterization and representation of the definition of normal behavior, and the metric used to measure the anomalous signal.

1.2 Thesis Structure

Chapter 1 is a brief introduction to this area of research. Chapter 2 discusses the related work. Chapter 3 is a detailed analysis of the different sliding window implementations. Chapter 4 discusses the different detection metrics used with the sliding window implementations. A presentation of the experiments appears in Chapter 5. Chapter 6 introduces a novel approach for integrating an IDS into the operating system. The concluding remarks and future work are presented in Chapter 7.

[1] It is quite common for administrators to toggle the anomaly threshold according to their tolerance for false positives. Without the flexibility to make this adjustment, the IDS will either miss intrusions (threshold set to high) or be turned off as a result of annoyance (threshold set too low).

CHAPTER 2

RELATED WORK

Much of the work related to this thesis falls in to one of three categories; augmenting coarse-grained access control with user-space sandboxing, retrofitting enhanced security features to an existing operating system, or intrusion detection using statistical modeling of normal behavior. This chapter presents an overview of each.

2.1 Augmenting Coarse-grained Access Control

All modern Unix variants offer very coarse-grained access control and little or no mechanism for implementing the principal of least privilege. This is one of the fundamental problems with using UNIX in a secure environment– there is no mandatory access control [14]. However, there have been several projects that attempt to provide this functionality in user-space. This approach has several benefits. The implementation is simple, nonintrusive– there are no kernel modifications needed– and it can be ported to several popular UNIX variants (Solaris, Linux, IRIX, FreeBSD, etc).

Though simple, this approach is only effective for non-privileged processes such as text editors, compilers, and web browsers. For example, if a normal user was allowed to trace a setuid program, a trivial method of obtaining root access would be created. Given that most intrusions involve setuid helper applications, this approach is only useful in protecting the user from malicious applications and data.

2.1.1 Janus

The Janus project attempted to apply the principal of least privilege to a user-space implementation [8]. Today, the distinction between data and application is increasingly blurred. The document format language Postscript, for example, has some interesting and potentially harmful features that can allow files to be read and written without the knowledge of the user. The same is true for macros in Microsoft Office. It is conceivable a user's files could be deleted, replaced, or sent to someone else when a user downloads a Postscript file and automatically launches `ghostview`. This is an undesirable feature that, aside from trusting the application that displayed the document, the user can do little to prevent.

The implementation of Janus assumes the worst case scenario. The security of the system and all of the user's files must be maintained even if an intruder has complete control of the application. Clearly, without a mechanism to enforce the principal of least privilege, this is very difficult.

To solve this problem, Janus creates a limited execution domain (or sandbox) around the untrusted application [8]. This sandbox is created using the `ptrace` system call. `ptrace` gives an attached process the ability to inspect and modify the user and data areas of the traced process. When a system call is made in the traced process, it is suspended, and control is given to the tracing process. The tracing process can then inspect the traced process' save area to inspect the system call invoked and the parameters passed to that system call. Given this information, the tracing process can make a decision to allow the traced process to continue or not, or simply add an entry to an audit log.

This functionality is used by Janus, along with features provided by the `proc` file system under Solaris 2.4, to implement an access control policy. If the combination of system call and parameters is allowed by the current policy, execution is allowed to continue. If it is not, Janus aborts the system call immediately, and returns control to the traced process.

Janus has several limitations. Strictly speaking, this is a problem with all such systems, it must be invoked to be of any use. The primary use of Janus is that of a wrapper. If users are given the option of using the wrapper, and living with limited privileges, it could work well. However, when given the opportunity to live with greater convenience or less security, most users choose a higher level of convenience. If they are repeatedly denied access because the security policy is global and static, they tire of requesting more flexibility from the system administrator. As a result, the global security policy is likely to be diluted, i.e., the global security policy often devolves to the security of the application that requires the least restriction in order to run.

Janus also adds a non-trivial amount of overhead to the execution of the traced process due to context switching. Tracing a process introduces at least two additional context switches per system call. If this overhead is applied to every process on a system, significantly lower performance on a heavily loaded multi-user system can result.

2.1.2 Safe-Tcl

Safe-Tcl is another approach to augmenting coarse-grained access control [16]. It uses an mechanism similar to the kernel space/user space interface to add a layer between the operating system and the user space application. Safe-Tcl attempts to sandbox the address space of the application and limit the set of system calls available within the sandbox. The sandbox is built within the Tcl scripting language. Policy and mechanism are carefully distinguished.

The major drawback of Safe-Tcl is the assumption that all C and C++ code associated with a Tcl application can be trusted. This is a faulty assumption.

Without some form of verification of the embedded C code, there is no way to assert anything about the security of the application.

2.1.3 Specification Languages

Ko et al. explored using a program policy specification language. This technique attempts to track anomalies in system call traces by flagging violations to a policy defined for the application being traced [13]. For example, the following is a sample specification for the finger daemon:

```
PROGRAM fingerd(U)
        read(x) :- worldreadable(x);
        bind(79);
        write ("/etc/log");
        exec("/usr/ucb/finger");
END
```

This would define as "normal" the ability to read anything that is world readable, bind to port 79, write to the file `/etc/log`, and execute `/usr/ucb/finger`. This approach is elegant in that it provides a formal specification that can be rigorously analyzed, it is simple, and it allows the application of the principal of least privilege. Although this approach has promise for real-time implementation, the main application appears to be audit trail generation and analysis.

Bowen et al. have implemented the approach above in the Linux kernel [3]. Observations regarding the system calls invoked and assertions on the parameters to those system calls are used to manually create a specification of normal behavior. This specification is flexible in that it can be configured for anomaly detection and misuse detection. In addition to specifying (ab)normal behavior, this implementation allows the administrator to specify actions associated with each assertion.

The following specification was taken from [3]:

```
execve(f) | f != "/usr/ucb/finger"
        -> exit(-1)
```

This specification is for the setuid root `finger` daemon. The specification simply restricts `fingerd` to exec'ing only one other executable, namely `finger`. If any other executable is exec'ed from within `fingerd`, it is treated as a misuse, and `exit`() is called before `execve`().

The power of such a specification language is easily recognized within the context of buffer overflow exploits. A buffer overflow exploit typically involves the replacement of the instruction pointer with a pointer to a specially crafted sequence of assembly instructions that exec's a UNIX shell while the application still has root privileges. With such a restrictive specification, it would be extremely difficult for a cracker to craft shell code capable of exec'ing an interactive shell.

Such restrictive specifications are implemented effectively by intercepting system calls within the Linux kernel. The mechanism that performs the interception is implemented in the same way as that used by `ptrace`. The detection engine can thereby evaluate sequences of system calls and their environment for any deviations from the specification and take action to prevent damage.

Some of the actions described in [3] are creative in preventing further damage and deceiving a potential intruder into thinking they were successful. Their responses include killing the offending process and isolating it in a non critical file system.

2.1.4 MAPbox

This approach is similar to Janus but more general. MAPbox, groups applications into classes based upon their expected behavior [1]. The result is a "specification" of the expected behavior of the class that is built by observing system call traces.

For example, elements of the class of text editors that includes, `emacs`, `vi`, `vim`, and `pico`. All exhibit similar behavior in terms of the system calls used and the resources requested. To build the sandbox for the specific class, samples of the applications that compose the class are traced. Based upon the behavior observed, normal behavior for the class is defined. This definition is used to sandbox the applications in the class.

2.2 Retrofitting Enhanced Security Features

The retrofitting approach takes an existing operating system with coarse-grained access control and retrofit enhanced security features. Given the size and complexity of modern operating systems, this is a difficult task of questionable effectiveness. In creating a secure operating system, security must be considered in every element of the design and implementation [14]. By retrofitting an existing operating system, enhanced security is added to modules and procedures that may have been designed ignoring security all together.

2.2.1 LIDS

The LIDS project attempts to retrofit the Linux operating system with enhanced security features [22]. These features include: a reference monitor, kernel capabilities (more refined access control), process hiding, raw disk protection, port protection, and enhanced file protection (e.g. append only). The motivation for LIDS is the inability of the classic UNIX model to provide fine grained access control, and the existence of a "superuser". LIDS uses access control lists (ACL) to control access to hardware and resources. It also uses a combination of ACL's and capabilities (see [20] for more on capabilities) to grant access based on the program executing, the uid of the user, and the inheritance of the parent process.

The LIDS project also attempts to eliminate the need for a superuser– the source of many problems in the UNIX environment. Its aim is to provide enough fine-grained access control so normal users can be given the privileges needed to run the system without the need of superuser. Using this fine-grained access control, LIDS attempts to compartmentalize the multi-user system so that compromising one service (the web server) does not effect any other (e.g. the database server).

2.2.2 Security Enhanced Linux

This is an effort headed by the National Security Agency (NSA) to add mandatory access control to Linux [15]. It is based on the argument that by not providing mandatory security, modern operating systems are missing the crucial security feature required to provide the separation of information based upon confidentiality and integrity. Linux was chosen in the hope that adding these features to a mainstream, open operating system would encourage further research while increasing the overall security of a system with significant market share.

2.3 Statistical Modeling of Normal Behavior

Statistical modeling of normal behavior has also been used as a method for detecting intrusion. The research presented in this thesis is based on the work of this type done by Forrest et al. to define normal behavior as sequences of system calls [7, 11, 12].

2.3.1 System Call Tracing: pH

The `pH` system is a kernel-based implementation that uses sequences of system calls to define normal behavior [12]. Building upon the earlier work, [7, 11], a backward tabular implementation (see Section 3.3) in combination with the locality frame metric (see Section 4.3) is used to detect intrusions in real time.

The focus of `pH` is the automated response problem– what is done when an observed measure for anomalous behavior exceeds a threshold? The solution in `pH` is the introduction of system call delays based upon the number of anomalies observed within the locality frame. Little analysis is presented for the choice of parameters used.

The mechanism used to capture the sliding window and introduce the delays is similar to that proposed in [3]. The system call interface is monitored with a simple addition to the Linux kernel `system_call` function (the system call dispatcher). A procedure is inserted that is called before the invocation of the system call and after the system call completes.

The methods described in Section 2.1 attempt to augment the security of the system by confining applications based upon their expected behavior. `pH`, and the implementation presented in this thesis, detect intrusions by statistically modeling normal behavior. The implementations presented in Section 2.2 approach the problem of coarse-grained access control by adding features to the operating system that are necessary to prevent intrusions, i.e. mandatory access control, and the principle of least privilege.

CHAPTER 3

SYSTEM CALL TRACING

In order for a process to request resources from the operating system, it must use the system call interface. By monitoring this interface, much can be learned about the behavior of the process being monitored. This is why tools like `strace`, `truss`, and the `proc` tool suite have become so popular with system administrators working in the UNIX environment. A process behaving mysteriously may be monitored via a window into the system call interface. Patterns in the types of system calls used and their frequency also may be observed at this interface. Forrest et al. concluded that a behavior-based intrusion detection system could be built by creating a definition of normal based on these patterns [7].

Several methods have been proposed for characterizing this behavior. One example is the frequency based method. This method models the frequency distribution of patterns of system calls in a trace. With this method, the definition of normal is defined by a set of sequences of length n and a corresponding frequency with which this sequence occurs in the system call traces for the application. This approach is not applicable to on-line implementation because the frequency distributions cannot be determined until after the process terminates [21].

Another approach for characterizing this behavior is data mining. This approach attempts to determine the most important features from a large collection of data, in this case, system call traces. Applications using data mining approaches are applied to system call traces to characterize the sequences of system calls occurring during normal behavior as small sets of rules that capture the common elements of the system call traces. These rules are then used to detect anomalies. Warrender et al. suggest indicate that this approach is not well suited for an on-line intrusion detection system because it requires multiple passes over the training data [21].

Hidden Markov models (HMM) have also been proposed for characterizing this behavior. A HMM is a powerful type of finite state machine that describes a doubly stochastic process. In this model, each state carries a certain probability of producing any of the system calls and a separate probability indicating which states are likely to follow. This approach is different than the other approaches in that anomaly decision are made regarding each system call and not sequences of system calls. Though powerful, HMM's are computationally expensive, and therefore not conducive to on-line intrusion detection [21].

Finally, the characterization techniques discussed in this thesis are based upon the sliding window implementations presented by Forrest et al [7, 11, 12]. This approach slides a window of size k across the system call trace and records the positional relationship between system calls in the window. This method is easy to implement, efficient, and only requires one pass over the training data. These properties make this approach the most suitable for on-line intrusion detection. Furthermore, Forrest et al. have shown that characterizing normal behavior using this approach is equally as powerful as the stronger analysis methods such as HMM's [21].

3.1 Data Collection

In order to collect data for their definition of normal, Forrest et al. used the `strace` package to record the system calls used by `sendmail`, and other UNIX applications (`sendmail` is a popular mail delivery agent for UNIX systems). The output is roughly of the form (the following is a sample trace taken from `sendmail` on our production mail server);

```
768   select(5, [4], NULL, NULL, {36, 140000}) = ? ERESTARTNOHAND ...
768   wait4(-1, [WIFEXITED(s) && WEXITSTATUS(s) == 0], WNOHANG, ...
768   wait4(-1, 0xbfffde08, WNOHANG, NULL) = 0
768   sigreturn()                        = ? (mask now [])
768   rt_sigprocmask(SIG_BLOCK, [ALRM], [], 8) = 0
768   time([970255240])                  = 970255240
768   time([970255240])                  = 970255240
768   open("/proc/loadavg", O_RDONLY)    = 5
768   fstat(5, {st_mode=S_IFREG|0444, st_size=0, ...}) = 0
768   read(5, "0.00 0.00 0.00 1/268 17114\n", 1024) = 27
```

Along with the system call, the output of `strace` produces additional context information including the parameters passed to the system call and the return value. The trace above produces the following ordering of system calls:

```
select,wait4,wait4,sigreturn,rt_sigprocmask,time,time,open,fstat,read
```

Forrest et al. use a sliding window algorithm to populate a table or forest of trees with the positional relationships between system calls. For example, they use a sliding window of size $k + 1$ to record which system calls succeed/precede each other at offsets 1 through k.

3.2 Table Representation: Forward (Lookahead)

The forward lookahead version of this approach builds a table of normal behavior by sliding a window of size $k + 1$ across the system call trace and recording which system calls follow position 0 (current system call) within the window. Consider the following trace with window size $k = 3$:

```
select,wait4,wait4,sigreturn,rt_sigprocmask,time,time,open,fstat,read
```

Sliding the window across this trace, yields the following sequences of length $k+1$:

```
select,wait4,wait4,sigreturn
wait4,wait4,sigreturn,rt_sigprocmask
wait4,sigreturn,rt_sigprocmask,time
sigreturn,rt_sigprocmask,time,time
rt_sigprocmask,time,time,open
time,time,open,fstat
time,open,fstat,read
```

The calls that follow the current system call (the first in each sequence) at offsets 1 ... k are recorded in a table like Table 3.1. Note the rows beginning with `wait4` and `time`. Explicit sequences that have the same current system call, are collapsed into a single row in the tabular representation.

Unless otherwise noted, the term table refers to a compressed representation of normal for an application where the current system call is used to index the table and there are at most N rows, where N is the number of unique system calls used by this application. If during the creation of the table, two explicit windows with the same current system call are encountered, the rows are collapsed into one (the `wait4` and `time` rows in Table 3.1 are examples of this compression). If this trace is considered normal, Table 3.1 contains the unique lookahead pairs that comprise a definition of normal [7].

Sample traces can be compared with the entries in Table 3.1. To do so, the current system call is used as an index to the table, and the system calls at each offset are compared. If the trace has a matching system call in the table at each offset, then it can be considered normal (assuming your table has been "trained" sufficiently). Otherwise, it is anomalous.

Table 3.1. Table Representing Definition of Normal: Forward

current	offset 1	offset 2	offset 3
select	wait4	wait4	sigreturn
wait4	wait4 sigreturn	sigreturn rt_sigprocmask	rt_sigprocmask time
sigreturn	rt_sigprocmask	time	time
rt_sigprocmask	time	time	open
time	time open	open fstat	fstat read

Take the following trace as an example:

```
select,wait4,wait4,sigreturn,open,time
```

Sliding a window of length $k = 3$ over this trace, yields the following 3 windows:

```
select,wait4,wait4,sigreturn
wait4,wait4,sigreturn,open
wait4,sigreturn,open,time
```

The second and the third sequences are rejected by comparing to Table 3.1. The second sequence has a mismatch at offset 3 with the open system call, and likewise, the third sequence has a mismatch at offset 2 with the open system call.

This implementation is said to have a "forward" lookahead because the current system call is used as the index to the table and anomalies are found by doing a pairwise comparison between the current system call and each system call that follows at offsets 1 through k.

Intuitively, one would expect this implementation to detect anomalies further in the sliding window because the index into the table is the *first* system call (the system call at the head of the window), and not the system call that just entered the window. This suspicion is explored further in Chapter 5.

3.3 Table Representation: Backward

This method is a variation of the method presented in the previous section. Instead of looking "forward" by using the first system call as the current system call, this method looks "backward" by using the last system call as the current system call. At each offset in the table, the system call that preceded the current call in the window is stored. This implementation was first introduced in [12] as a simple and efficient way to recognize anomalies.

Consider the same trace from the description of the "forward" implementation:

```
select,wait4,wait4,sigreturn,rt_sigprocmask,time,time,open,fstat,read
```

Sliding a window with length k = 3 over this trace, yields the following unique sequences:

```
sigreturn,wait4,wait4,select
rt_sigprocmask,sigreturn,wait4,wait4
time,rt_sigprocmask,sigreturn,wait4
time,time,rt_sigprocmask,sigreturn
open,time,time,rt_sigprocmask
fstat,open,time,time
read,fstat,open,time
```

These sequences comprise the definition of normal in terms of "lookahead" pairs represented in Table 3.2. As with the forward implementation, any two explicit windows with the same current system call are collapsed into one in the table (in this case, the windows with current system call `time` are collapsed into one).

Again, this implementation is termed "backward" lookahead[1] because the last system call in the window is used as the current system call when indexing the table.

[1] The term "backward lookahead" is contradictory. However, it is reproduced here for consistency with [12].

Table 3.2. Table Representing Definition of Normal: Backward

current	offset 1	offset 2	offset 3
sigreturn	wait4	wait4	select
rt_sigprocmask	sigreturn	wait4	wait4
time	rt_sigprocmask time	sigreturn rt_sigprocmask	wait4 sigreturn
open	time	time	rt_sigprocmask
fstat	open	time	time
read	fstat	open	time

Hofmeyr, Forrest, and Somayaji [12] do not present any rationale for switching from the original "forward" implementation to the "backward". It is offered it as a simpler, more efficient representation than using explicit sequences (see Section 3.4). Chapter 5 provides some justification for preferring the backward over the forward implementation.

3.4 Tree Representation

This sliding window implementation does not collapse explicit windows with the same current system call. In effect, there is no notion of a current system call or lookahead pairs because each input window is explicitly compared with the database. Continuing with the trace example from the previous two sections:

```
select,wait4,wait4,sigreturn,rt_sigprocmask,time,time,open,fstat,read
```

Sliding a window of length $k = 4$ yields the following exact sequences:

```
select,wait4,wait4,sigreturn
wait4,wait4,sigreturn,rt_sigprocmask
wait4,sigreturn,rt_sigprocmask,time
sigreturn,rt_sigprocmask,time,time
rt_sigprocmask,time,time,open
```

```
time,time,open,fstat
time,open,fstat,read
```

There are several ways to represent this information. Hofmeyr et al. proposed the use of trees to represent the unique sequences that define normal behavior [11]. Taking as the root of each tree a particular system call, the unique sequences above create the forest show in Figure 3.1.

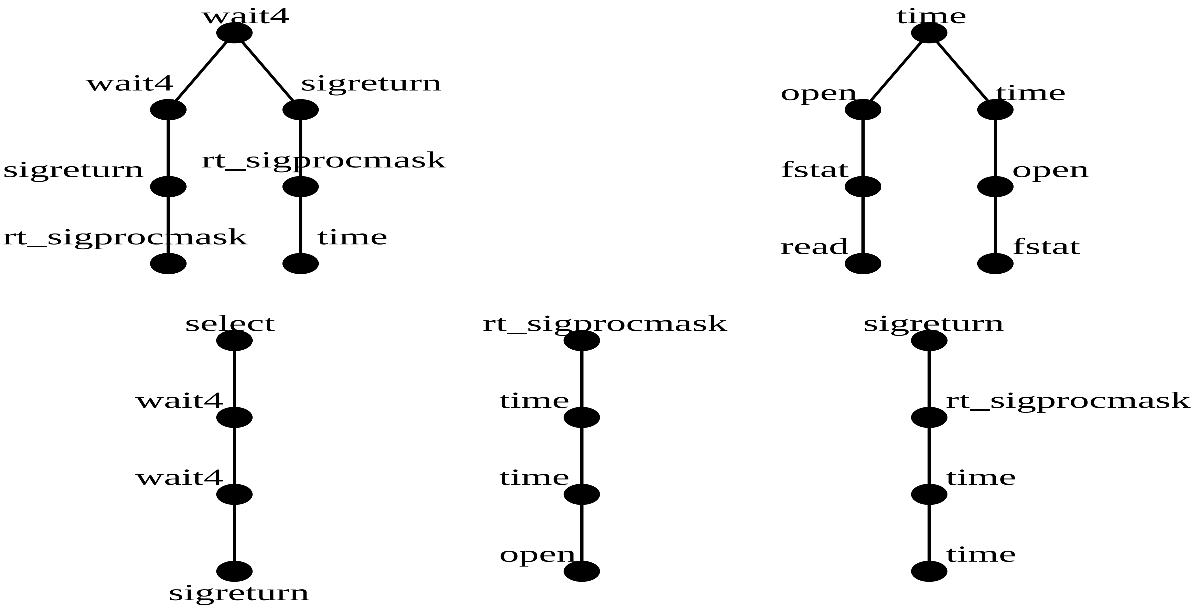


Figure 3.1. Forest representation of sequences of system calls.

Storing the exact sequences as a forest of trees provides the same lookup time as the tabular representation while maintaining more information about the windows accepted by the definition of normal. The search time for this representation is the same as the tabular method because each tree is rooted at a different system call. Likewise, determining if a sequence is accepted or rejected can be determined in $O(k)$.

3.5 Lexical Analyzer Representation

The unique set of explicit windows contained in the database can also be used to create a lexical analyzer (deterministic finite automaton) capable of recognizing the "language" of program traces that describes normal behavior for a given application. This is done by defining the *alphabet* as the set of system calls implemented in the operating system observed (about 220 in Linux 2.4.0). The database of explicit windows representing normal behavior is used as the set of *strings* that describes a *language* over this alphabet. The actual alphabet that describes this language is typically smaller than the set of all the system calls implemented in a given operating system. Empirical results from this thesis indicate that the size of this set typically ranges between 20 and 40 for small applications.

Consider the same set of windows used to create the forest from the Section 3.4 (see Figure 3.1):

```
select,wait4,wait4,sigreturn
wait4,wait4,sigreturn,rt_sigprocmask
wait4,sigreturn,rt_sigprocmask,time
sigreturn,rt_sigprocmask,time,time
rt_sigprocmask,time,time,open
time,time,open,fstat
time,open,fstat,read
```

Converting each system call to its corresponding system call number (under Linux, this mapping can be found in <asm/unistd.h>), yields the following strings:

```
82 114 114 119
114 114 119 126
114 119 126 13
119 126 13 13
```

```
126 13 13 5
13 13 5 108
13 5 108 3
```

If converted to hex, these strings can be used to create "rules" via *Lex*[2] that recognize these unique windows. The resulting lexical analyzer is capable of recognizing an input string in $O(k)$ time where k is the window size chosen for the database of sequences. This implementation is equivalent to the tree representation from the previous section.

[2] *Lex*, is a lexical analyzer generator commonly found on UNIX systems.

CHAPTER 4

DETECTION METRICS

When developing an intrusion detection system, it is important to formulate a metric to determine whether an event is deemed normal or anomalous. In this section, some metrics that have been proposed to gauge the strength of an anomalous signal are discussed.

Two very important definitions must be presented before discussing any anomaly-based intrusion detection system; false positives and false negatives. To measure an anomalous signal for its strength, a quantitative measure of anomalies is observed at discrete intervals and this measure, say χ, is compared against a predefined threshold T. In each interval i, the measure is considered normal if and only if $\chi_i \leq T$, otherwise, χ_i represents an intrusion [5]. Based on this, false positives and false negatives are defined:

True Positive A true positive occurs when $\chi_i > T$ is observed during an intrusion.

False Positive This occurs when $\chi_i > T$ signals an intrusion during legitimate behavior. T is a function of the amount of noise (the height of χ_i) expected during normal behavior. A false positive can be the result of inadequate training or a miscalculation of T.

False Negative A false negative occurs when the IDS fails to detect an intrusion because $\chi_i \leq T$ during the intrusion. The goal of any IDS should be to eliminate false negatives.

True Negative This occurs when $\chi_i \leq T$ for all i during legitimate behavior.

In the sliding window implementation, the interval over which χ_i is sampled includes every system call i where $k \leq i \leq N$ where k is the window size and N is the length of the trace.

For this discussion to be complete, anomaly detection and exploit detection also need to be defined. They have been defined as follows:

Exploit Detection : The threshold T is a function of the signal observed during an intrusion.

Anomaly Detection : The threshold T is a function of the noise expected during normal behavior.

Exploit detection may be valuable in detection of know exploits. However, anomaly detection is simpler and more general as it attempts to detect known and unknown exploits.

4.1 Counting Pairwise Mismatches

The pairwise mismatch metric quantifies anomalies by counting the number of pairwise mismatches between the input window and the table. Forrest et al. propose that the number of mismatches be recorded as a percentage of the total possible number of mismatches [7]. The following formula is provided for this calculation where L is the length of the trace and k is the window size:

$$k(L-k)+(k-1)+(k-2)+...+1 = k(L-(k+1)/2).$$

For example, with a trace length of 6, and a window size of 3, the total possible number of mismatches is 12. Therefore, given two windows with 1 pairwise mismatch each, the percentage mismatch for this trace would be 16%.

Care must be taken when combining this metric with a tabular representation of normal behavior. If the tabular representation of normal collapses rows with the same indexing system call (as in Table 3.1) then calculating the number of pairwise mismatches on an input window without an index into the table is difficult without incurring significant overhead. Furthermore, if the number of pairwise mismatches defaults to something other than one, the strength of the signal generated may be exaggerated.

Assuming that the input trace contains only system calls that index into the table, this algorithm can be run in $O(N)$ where N is the number of system calls in the input trace [7]. This algorithm is a function of the trace length, so it is only applicable to the post processing of system call traces.

4.2 Hamming Distance

Hofmeyr et al. proposed a metric for measuring anomalies independently of trace length [11]. This metric is based on the Hamming distances between sequences. Given an input window of length $k+1$, the Hamming distance is found by determining the number of pairwise mismatches between the input window and a database with the explicit window representation (no collapsing of rows with the same indexing system call).

Comparing the following input window to Table 4.1 will produce a mismatch at position 2 (there is no `open` entry in column 2, preceded by `sigreturn`, and followed by 2 `time` system calls):

```
sigreturn,open,time,time
```

This single mismatch represents a Hamming distance of one from the representation of normal. Likewise, the following window:

```
time,open,time,time
```

has no entry for `time` at position 1 or `open` at position 2. The Hamming distance for this trace is two.

Table 4.1. Sample database of normal behavior. No collapsing of rows.

position 1	position 2	position 3	position 4
select	wait4	wait4	sigreturn
wait4	wait4	sigreturn	rt_sigprocmask
wait4	sigreturn	rt_sigprocmask	time
sigreturn	rt_sigprocmask	time	time
rt_sigprocmask	time	time	open
time	time	open	fstat
time	open	fstat	read

Computing the Hamming distance between an input trace and the database is extremely inefficient. If the database is stored as a forest of trees as in Figure 3.1, then determining if a sequence is a mismatch requires at most $(k-1)$ comparisons. The Hamming distance is determined by traversing the entire database and comparing each explicit sequence with the input sequence and determining the number of mismatches. This traversal will require at least $O(Nk)$ time, where k is the window size and N is the number of unique sequences in the database.

Hofmeyr et al., [11], uses this notion of Hamming distance to gauge the strength of an anomalous signal by determining the *minimum* Hamming distance between the input trace and the sequences in the database. More formally, let $d(i,j)$ denote the Hamming distance between two sequences i, and j, then the minimum Hamming distance $d_{min}(i)$ is $\min\{d(i,j)$ for all normal sequences $j\}$.

Hofmeyr et al. [11], suggest using this metric to determine the strength of an anomalous signal observed in a trace. This signal, S_A, can be computed by $\max\{d_{min}(i)$ for all input sequences $i\}$. This can be normalized over different window sizes, k, by taking $\hat{S_A} = S_A/k$.

Although this metric is capable of determining how different an *individual* sequence is from the database, it is not clear how $\hat{S_A}$ can be used to effectively quantify an anomalous signal because $0 \leq \hat{S_A} \leq 1$. Assuming perfect training (the definition of normal only produces a measureable χ for an intrusion) and a maximun $\hat{S}_A$, the distance between χ and T is at most 1. This distance may not always provide sufficient distinction between a false positive and a true positive. Chapter 5 will address this issue. Furthermore, because the theshold for $\hat{S}_A$ is set by analyzing exploits, Forrest et al, by the definition, have implemented exploit detection and not anomaly detection.

4.3 Locality Frame

In another paper, Hofmeyr et al. concluded that intrusions produce anomalies that are temporally clustered [12, 21]. In order to accurately measure this phenomenon, a potentially infinite trace needs to be partitioned into local subsets that can be observed individually. This can be done by using a fixed circular buffer, called a *locality frame*, to record recent anomalies. This metric is similar to the Hamming distance approach in that the locality frame allows you to measure mismatches independent of trace length.

More formally, let n be the size of the locality frame and let A_i be the i-th entry where $0 \leq i < n$ and $A_i \in \{0,1\}$. Then for each system call s, where $(1 \leq s \leq N)$ with N being the length of the trace, let $A_{s\ mod\ n} = 1$ when,$M_s > 0$, where M_s is the number of pairwise mismatches, recorded for s is > 0. Otherwise, let $A_{s\ mod\ n} = 0$. The sum of A_i for all i in $0 \leq i < n$ then records the number of past n system calls which were rejected by the definition of normal. This summation is termed the locality frame count or LFC [12, 21].

The LFC can be applied to any of the sliding window implementations discussed. Furthermore, because it is not dependent upon the absolute trace length, it is feasible to implement this metric in an "on line" intrusion detection system. Hofmeyr et al. uses this metric in a , "delay-based", hybrid IDS implemented in the Linux kernel. The kernel based implementation presented by Hofmeyr et al., [12], uses an LFC to introduce delays in the Linux scheduling algorithm. For example, a process with recent anomalies wanting to use the open system call would have to wait ($2^{LFC} * delay\_factor$) where $delay\_factor$ is some multiple of jiffies (the Linux kernel epoch). This implementation proved to be effective at slowing and or delaying intrusions, however, it is not clear how effective it would be at preventing intrusions or discriminating between the noise found in normal behavior and the signal of an intrusion.

4.4 Accept/Reject

The simplest metric for measuring anomalous behavior over a trace is to count the number of sequences rejected and report that as a percentage of the total number of sequences observed. More specifically, if a tabular implementation is used, a sequence will be rejected if any pairwise mismatches are obsevered or if the current system call cannot be indexed into the table. Furthermore, if a tree implementation is used, any sequence not found in the database will be rejected. This metric only works well for traces of finite length. It does not work well for online tracing of continuous processes (e.g. system daemons like `sendmail`).

This metric can be used to compare the relative effectiveness of the different sliding window implementations at rejecting anomalous sequences. Chapter 5 will use this metric to make this comparison.

4.5 Event Counter

The event counter metric[1] was designed with the intention of quantifying the temporal clustering of anomalies observed during intrusions[2]. The premise that by amplifying the signal produced by large clusters of anomalies during an intrusion, an IDS will be more capable of distinguishing between this signal and the occasional

[1]The term *event counter* was first introduced in [5] as a metric for recording some number of events ocurring during a period (e.g. the number of failed `su - root` attempts in an hour).

[2]This behavior was first observed in [11] and later confirmed by the experiments in Chapter 5.

noise of legitimate behavior. The result is an increased signal to noise ratio which in turn leads to increased flexibility in choosing a threshold T that distinguishes legitimate behavior from an intrusion.

This metric quantifies the temporal clustering of anomalies by maintaining a simple counter during the execution of an application. The counter is modified by an increment I when a sequence is rejected and a decrement D when a sequence is accepted. The increment is a function of the noise expected during normal behavior. Let R be the ratio of sequences likely to be rejected during legitimate behavior, then the increment $I = 1 \ / \ (R * k)$ where k is the window size. The rejection rate is multiplied by the window size k to prevent the exaggeration of the signal as k windows slide over a single anomalous system call. The decrement is set to one. To simplify the implementation of this metric, the window size has been moved from the denominator in the calculation of I to the decrement D. Therefore, $I = 1 \ / \ R$ and $D = k$. For example, if one percent of the sequences are likely to be rejected during normal behavior, then $R = 0.01$ and $I = 1/0.01 = 100$ and $D = 10$. The value for R just presented is not unreasonable and can be determined empirically by observing the percentage of sequences rejected from traces not included in the construction of normal behavior. Furthermore, in an attempt to ensure detection, assume $k \leq I \leq 100$, this prevents the decrement from dominating the counter. The event counter is assumed to have a floor of 0.

4.6 Tuning the Event Counter

In this section the use of probability theory to improve the choice of threshold to minimize false positives while maximizing true positives will be discussed. To improve the event-counter-based intrusion detection approach, the anomaly threshold for a certain application needs to be optimized to minimize occurrences of false positives while maximizing the detection of true positives. This is a classical optimization problem. However, the optimization problem can only be solved under the assumption that the signal and noise measures are well defined and do not significantly change between observations. The complexity of most applications prohibits measuring the complete spectrum of noise of the application. This would otherwise require extensive testing of the application under normal circumstances to obtain the maximum level of noise to define a threshold that is the upper bound on the observed noise. Note that if extensive testing is performed anyway, it would be better to collect the system call sequences for training to improve the DFA instead of validating the DFA after limited training.

4.6.1 Random Walks

There is a correspondence between the event counter's noisy behavior and the probabilistic theory of random walks. An example random walk is the price of a stock over time. Predicting stock prices is near impossible. However, accurately predicting the noise generated by an application is feasible.

This section will attempt to use queueing theory to establish a "safe" anomaly threshold. That is, the threshold should be large enough in theory to limit false positives to a certain low percentage of all observed system calls.

The underlying assumption for applying queueing theory is that the anomalies generated by an application are distributed according to a Poisson distribution. That is, it is expected that longer periods of low noise are likely to be followed by short busy periods of high noise.

Queueing theory states necessary properties of a queue such as it's customer service time and customer inter-arrival times for the random process describing the size of the queue to become stationary. Note that the detection of an exploit by an increasing event counter that eventually reached the threshold corresponds to a situation in which a queue grew out of it's bounds. That is, assuming an underlying

Poisson distribution for the inter-arrival times of anomalies the event counter models a the number of customers (anomalies) in a queue requiring service.

The M/M/1 queue is a simple model of the event counter's random process. For the M/M/1 queue, the probability distribution of the queue size (event counter) $Q(t)$ (time t) is a continuous-time Markov chain. Let $\rho = \lambda/\mu = 1/dec < 1$ be the traffic density, where dec is the event counter's decrement [9]. Then,

$$\mathbf{P}(Q(t) = n) \to (1 - \rho)\rho^n = \pi_n$$

where π is the unique stationary distribution. Hence, the probability that the event counter will reach a value of n is $(1 - \rho)\rho^n$ with $\rho = 1/dec$.

To define a threshold value given an acceptable rate of false positives, e.g. 0.01%, the maximum number of anomalies n that occupy the queue at any given time is determined. This gives $(1 - \rho)\rho^n \leq 0.0001$. Solving for n gives $n = \log(0.0001/(1 - \rho))/\log(\rho)$. For example, using a window size of 11 ($dec = 11$), $\rho = 1/11$ yields $n \approx 4$. The upper bound to the threshold is set to $n * inc * dec \approx 4400$, and the lower bound to the threshold is set to $inc * (dec + n) \approx 1500$. Normalization using the inc and dec values is necessary because one anomaly in a system call trace will likely result in a gradual increase of the event counter by $inc * dec$, since dec equals the window size. Furthermore, the upper bound attempts to capture the signal produced by anomalies that are spread across multiple windows. In this case, $(dec * n)$ reflects the expected maximum number of anomalies being serviced by the queue across dec windows. The lower bound attempts to capture the signal produced by anomalies that occur within the same sliding window. The multiplier $(dec + n)$ thus reflects the number of anomalies expected within the same sliding window.

Experimental validation of this model is necessary. In addition, other more general queues such as G/M/1 might be more accurate in describing the random process and therefore should be used to define a threshold for anomaly detection.

The metric just proposed can be implemented using any of the sliding window implementations discussed. Furthermore, the value of the counter is independent of the trace length, which makes this metric suitable for use in an on-line IDS. Finally, based on the definitions presented in Chapter 4 this approach is based upon anomaly detection.

CHAPTER 5

EVALUATION OF CURRENT APPROACHES

This Chapter presents an evaluation of the current approaches to intrusion detection using sequences of system calls. The design of the experimental setup will be discussed followed by an analysis of the complexity of the forward tabular, backward tabular and tree representations. The behavior of the event counter approach is compared to the Hamming distance approach in Section 5.3. This section will compare the two approaches and attempt to justify the use of the event counter metric. Section 4.6 will present methods for tuning the event counter metric that optimize its detection capabilities and maximize the flexibility in determining the tolerances for false positives and false negatives.

5.1 Design of Experimental Setup

The trace data used in this evaluation came from both synthetic environments and "live" production environments. For clarity, the following definitions are provided (these definitions have been reproduced from [11] for consistency).

Synthetic Environment A synthetic environment is one that has been created for the sole purpose of the experiment. In this case, the [virtual] system within which the data is being gathered is isolated from the network and is *assumed* to be free from Trojans or any other form of intrusion.

Production Environment This type of environment is one that is attached to a network and is used in some capacity by one or more people on a daily basis.

It is *assumed* to be free from Trojans or any other form of intrusion but to a lesser degree than a synthetic environment.

Two different production environments were used for the evaluation, and one synthetic environment. The two production environments were a RedHat 6.1 machine used as a workstation and a RedHat 6.2 machine also used as a workstation. The synthetic environment is unique in that it is contained within the RedHat 6.1 production environment using a experimental fork of the Linux kernel called User-mode Linux. User-mode Linux provides the ability to run the Linux kernel as a user-mode process. This "virtual machine" allows the specification of resources to which the user-mode kernel has access. These features are especially useful in kernel development and security related research. For more information about User-mode Linux, see `http://user-mode-linux.sourceforge.net/`.

There are two additional definitions important to this evaluation. The trace data is being relied upon to build the definition of normal, therefore, it is important to outline the techniques used to gather data and their possible effect on the definition.

Live Traces Live traces are produced when an application is traced during *normal* use in a *production* environment. For example, live traces can be obtained by tracing `/bin/login` on a UNIX machine providing shell access to students at a university.

Fabricated Traces This type of trace is collected when the sole purpose of executing the application is to gather its trace output. Much of the research done in this area is based on fabricated trace data. For example, to produce an accurate definition of normal from `/bin/ps`, one might trace `ps` multiple times while attempting to combine as many of the most frequently used command line options as possible.

The distinction between live traces and fabricated traces is subtle but important. Although the implications of using fabricated vs. live traces is beyond the scope of this thesis, it should be noted that the majority of the data used has been collected from fabricated traces.

Four different applications were used for the evaluation. Three of them are known to have exploits that result in root shells. A root shell is a term used to describe the resulting execution of a shell (such as `bash`) as the superuser or root user. On UNIX machines, there are no restrictions on what root can do. The fourth application used in the evaluation is `ls`. `ls` is a fairly simple application that is frequently used. Since `ls` is not setuid root, it can not be exploited for a root shell. However, since it is frequently used, it is a perfect target for a Trojan horse.

The three exploits affect `dump`, `traceroute`, and `su`. All three exploits are different. The exploit for `dump` is a programming error which allows someone to execute an arbitrary script as the root user [6]. The exploit for `traceroute` is a buffer overflow attack [17]. Finally, the `su` exploit takes advantage of a locale subsystem format string vulnerability [10]. Table 5.1 summarizes the applications used in the evaluation, and how extensively each application was traced to build the definition of normal.

The majority of the work done in this area has focused on window sizes between 5 and 30 [7, 11, 12]. The most recent has focused on window sizes between 6 and 11. In an effort to reproduce some of the results, training for these applications was done over window sizes 2 through 11.

The reasoning for stopping the training at 10,000 system calls is based upon the conclusions reached in [7]. In this paper, a distinctly asymptotic curve was presented that leveled off at about 2800 system calls and remained relatively flat to 10,000 system calls where the training was stopped. Similar results are presented in Figure 5.1. Also included are results where the database was trained using 1,000 system calls. For the evaluation involving known exploits, both training sizes will be used to

Table 5.1. Data Set Summary

App	Env	Training	Exploit Type	# seq used
dump	Production	Fabricated	Programming Error	1000, 10,000
traceroute	Production	Both	Buffer Overflow	1000, 10,000
ls	Synthetic	Both	Trojan Horse	1000, 10,000
su	Production	Fabricated	Format String	1000, 10,000

obtain a better understanding of how sensitive the different metrics are to training. This issue will be explored in Section 5.3. See Figure 5.2 for the database growth curve for training based on 1,000 system calls.

All of the observations, except for those involving the use of the Hamming distance metric were run using the lexical analyzer described in Section 3.5. This implementation was chosen for its ability to create and optimize the database automatically. Since Lex is capable of recognizing regular expressions, transforming the tabular or explicit representation of system calls is simple. The following example will illustrate this process.

To build a specification in Lex capable of recognizing sequences of system calls, the sequences are transformed from their literal representation to their numerical representation. In Linux, this representation is provided by `<asm/unistd.h>`. To transform a tabular representation into a Lex specification, the system calls from Table 3.1 are replaced with their corresponding system call numbers. The result is presented in Table 5.2.

To create the Lex specification, the decimal numbers are converted into hex "strings" that can be used as rules to recognize this language. Converting Table 5.2 to hex strings gives the following:

```
\x52\x72\x72\x77                          { return 1; }
\x52(\x72|\x77)(\x77|\xaf)(\xaf|\xd)      { return 1; }
\x77\xaf\xd\xd                            { return 1; }
```

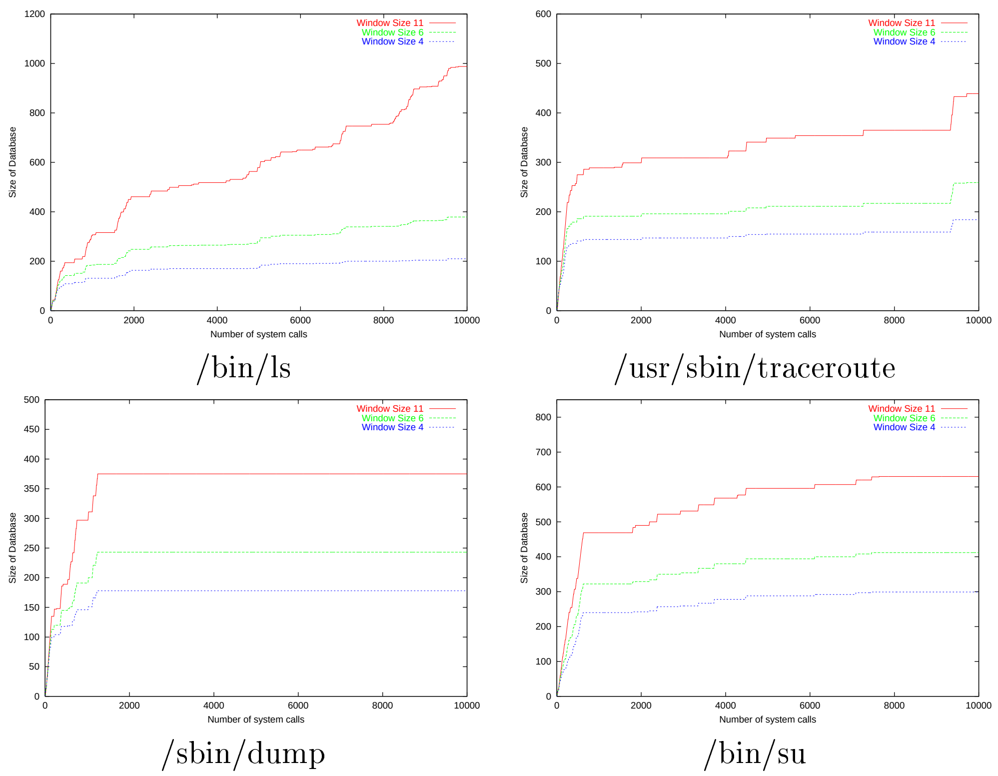


Figure 5.1. Growth in the number of explicit sequences in the database of normal behavior, 10,000 system calls.

```
\xaf\xd\xd\x5                          { return 1; }
\xd(\xd|\x5)(\x5|\x6c)(\x6c|\x3)       { return 1; }
.                                      { return -1; }
```

The "{ return 1; }" statements inserted at the end will be used by the program calling the lexical analyzer to determine if the sequence was accepted or not. The '|' symbol is used by Lex to indicate that either of the hexadecimal numbers is to be accepted by this string. The last line is the default rule that is matched when the input string is rejected by the analyzer. The "." is a wild card that will accept anything not already represented by a hex string. Likewise, explicit sequences can be accepted by simply converting their literal representation to hex. The third and fourth rows above represent hex strings stored literally. Representing strings

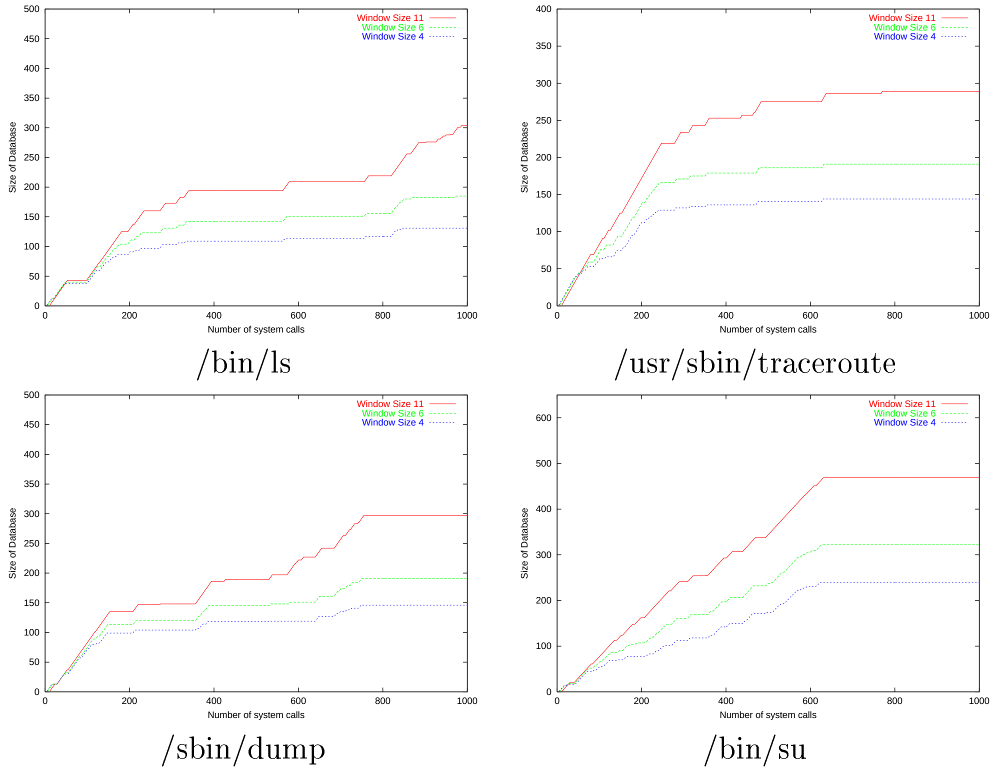


Figure 5.2. Growth in the number of explicit sequences in the database of normal behavior, 1,000 system calls.

explicitly in Lex is analogous to the tree representation representation presented in Section 3.4. As stated above, the Lex implementation was chosen for its simplicity and automatic optimization of the underlying DFA.

5.2 Complexity of Forward, Backward, and Tree Representations

In the literature behind Chapters 3 and 4 there is little rationale given for the different sliding window implementations. This section will attempt to provide the rationale by addressing the practical implications for using the forward lookahead vs. the backward lookahead and compare them to the tree approach.

By addressing these issues, this thesis will attempt to fill the gaps in the current research while providing some insights and improvements towards the goal of creating

Table 5.2. Table Representing Definition of Normal in Decimal

current	offset 1	offset 2	offset 3
82	114	114	119
114	114 119	119 175	175 13
119	175	13	13
175	13	13	5
13	13 5	5 108	108 3

a powerful, lightweight, intrusion detection system suitable for implementation in the Linux kernel.

5.2.1 Average Offset of Sequentially Replaced System Call

This experiment was created to determine the ability of each sliding window implementation to recognize the random singular replacement of a system call in an otherwise anomaly-free trace. This experiment was also designed to reveal the position in the sliding window where the anomaly is detected. The results should create a better understanding about the differences between using the forward vs. backward lookahead.

For each of the applications in Table 5.1, a sample trace was chosen that produced zero anomalous sequences when scanned by the lexical analyzer using the forward tabular, backward tabular, and tree representations over window sizes two through eleven. For each system call s_i, where $11 < i \leq N$, and N is the length of the initial trace, a new trace was created with s_i replaced by a system call from the set of system calls known to be used by the specific application. For example, let R be the set of system calls used by an application. Each system call s_i is replaced by a system call $x \in R$. The newly created trace is then analyzed by the lexical analyzer. The number of rejections are counted, and the offset where x was detected is calculated.

Figure 5.3, shows the results for `dump`, `traceroute`, `ls`, and `su` respectively. In these graphs, the y axis represents the position in the window where the replaced system call was detected. A position of one would indicate that the replaced system call was recognized as anomalous as soon as it entered the input window. It is no coincidence that all four applications exhibit the same behavior. The difference in the slope of the lines between the backward lookahead, the tree, and the forward lookahead can be attributed to the variability in the last column of the tabular implementation. Figure 5.4 helps explain this observation. As the window size is increased, the variability of system calls in the last column also increases. If the objective is to recognize an anomaly as soon as it is presented, this observation favors the use of the backward lookahead or the tree implementation.

Considering the forward implementation in operation strengthens these observations. As the "sliding window" traverses the input trace in the forward implementation, each new system call encountered is placed at the end of the window, and the current system call, the system call used to index the table, is at the head of the window. Therefore, if the number of system calls in the last column is greater, on average, than any other column, there is a higher probability that a randomly replaced system call will be accepted.

The results from Figure 5.3 give some indication as to why the lookahead was switched from forward to backward between the writing of [7] where the forward lookahead was presented and [12] where the backward lookahead was described. Chronologically positioned between the these two papers, the tree implementation was introduced in [11].

5.2.2 Anomaly Rate with Sequentially Replaced System Call

This analysis should expose the effect collapsing rows has on the tabular implementation as compared to the tree implementation.

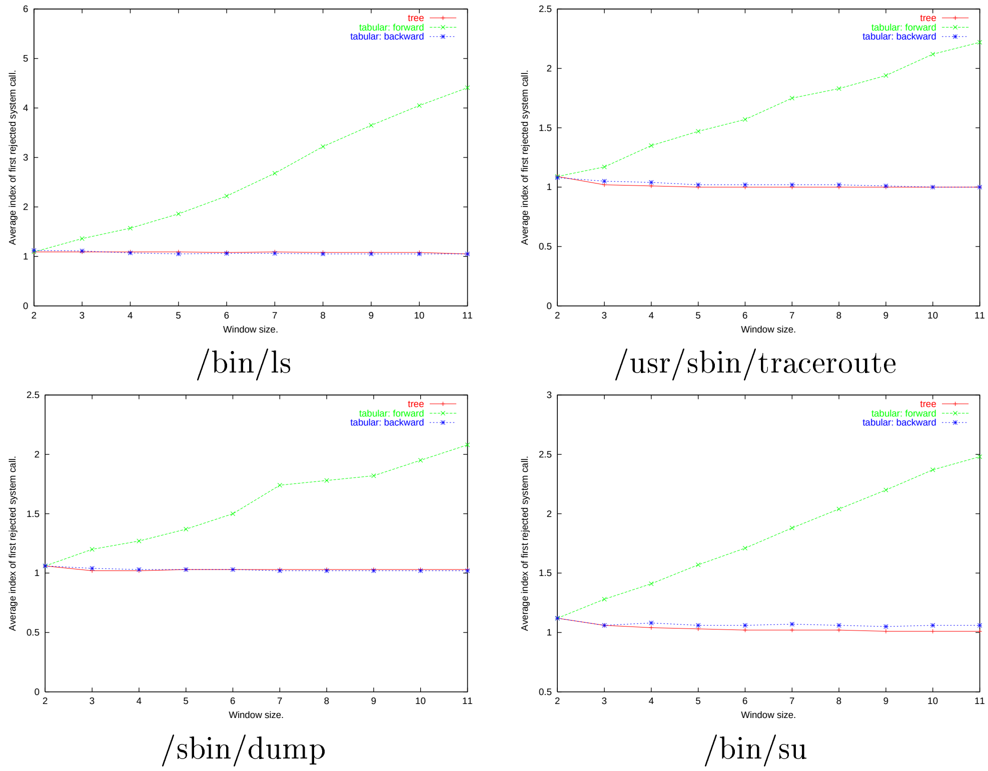


Figure 5.3. Average index using sequential replacement.

If the sequences are stored explicitly, as in the tree implementation, only the explicit sequences of system calls that are found in the database will be accepted. On the other hand, the tabular representation has the capacity to accept a superset of the sequences that were used to create the definition of normal. This is best illustrated with an example.

Looking at the explicit sequences used to generate Table 3.1 from Section 3.2 it is easy to find an explicit sequence that was not used to build Table 3.1 but is accepted by it. For example, the following window

```
wait4,wait4,rt_sigprocmask,rt_sigprocmask
```

is accepted by Table 3.1 though it is not in the set of explicit sequences used to build the definition of normal. Accordingly, the window above would be rejected by the

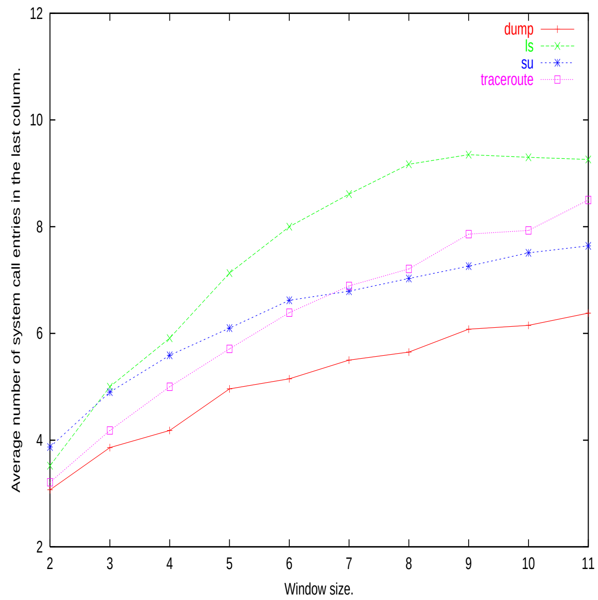


Figure 5.4. Average number of system calls in the last column of the tabular implementation.

tree implementation represented in Figure 3.1. This discrepancy can be attributed to the compression of the explicit sequences when rows are collapsed.

5.2.3 Discussion of Kernel-space Requirements

Integrating an IDS into kernel-space requires more attention than a similar implementation in user-space. An IDS suitable for integration into kernel-space must be small, fast and accurate. In the following sections, each of the sliding window implementations will be compared by these qualities.

5.2.3.1 Kernel-space: Overhead

Memory is not cheap. For a kernel-space implementation to be accepted, it must be small. Integrating an IDS into the kernel means taking more of main memory from user-space and allocating it for the kernel. This motivates careful consideration of space requirements when choosing which sliding window implementation to use.

In Section 5.2.2 the difference in rejection rates between the tabular implementations and the tree implementation were attributed to the compression of the explicit sequences that takes place when the rows are collapsed (see Figure 5.3). The collapsing of rows also has the effect of reducing the storage requirements of the DFA in the lexical analyzer. Figure 5.5 shows the storage requirements for the transition table used by lex to implement the DFA. This graph suggests that the number of transition table entries needed for storing sequences explicitly increases exponentially with window size while the tabular implementation requires a linear increase [1]. This is a high price to pay for the modest increase in detection capabilities as suggested by Section 5.2.2.

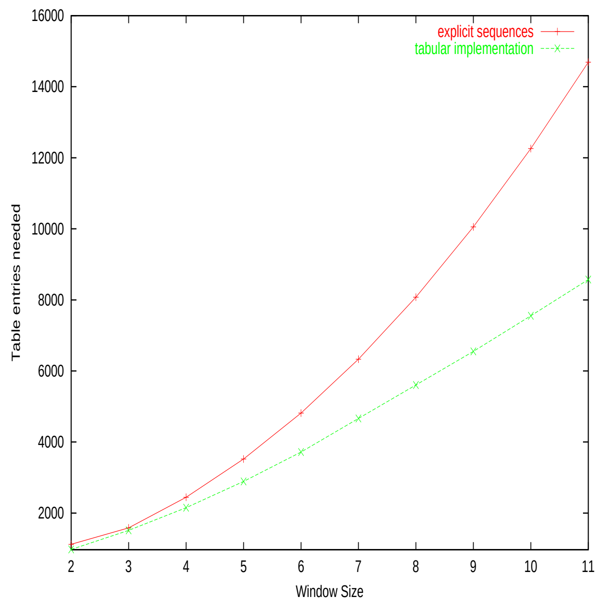


Figure 5.5. Number of transition table entries needed for the finite automaton for tree implementation or table implementation.

[1] Lex uses a DFA to perform pattern matching on the input string. The implementation of this DFA uses a transition table to map the set of states that can reached from any state i on input a. For more information about the implementation of Lex see [2].

5.2.3.2 Kernel-space: Speed

In all of the experiments, both the tabular and tree implementations are implemented using a lexical analyzer (see Section 3.5. Therefore, the time requirements for both are the same; $O(k)$ where k is the window size.

Although the three implementations are able to determine if a sequences is anomalous in the same amount of time, they differ slightly in their ability to detect these anomalies. The rejection rates observed in Section 5.2.2 suggest a very modest improvement in detection capabilities by using the tree representation. Also note that the backward implementation exhibits equivalent results to the tree implementation in the sequential replacement experiments from Section 5.2.1.

To conclude, Figure 5.5 suggest a disproportionate storage penalty for the modest increase in detection capabilities offered by the tree implementation. Although more detailed analysis is required, these observations suggest the use of the tabular implementation over the tree implementation for integration into kernel-space. Furthermore, of the two tabular implementations, the backward implementation appears to maximize the detection capabilities while minimizing the space requirements.

5.3 Behavior of Event Counter and Hamming Distance on Known Exploit

Recall from Chapter 4, that the $\hat{S_A}$ metric ($\hat{S_A}$ is the normalized Hamming distance, see Section 4.2) attempts to quantify the signal of an intrusion by determining how different an *individual* sequence is from the database of normal behavior. On the other hand, the event counter metric (see Section 4.5) attempts to detect intrusions by quantifying the cumulative effect of temporally distributed anomalies. Arguably, the Hamming distance metric and the event counter metric measure two different but related phenomena. Furthermore, because the $\hat{S_A}$ metric is based on the Hamming distance between the input window and the database. Reporting the Hamming distance requires the database to store sequences of system calls explicitly, as in the tree implementation. Conversely, as stated in Section 4.5, the event counter metric can be implemented independently of the underlying sliding window representation (forward, backward, or tree). This section will address these issues in greater detail and present the culmination of the research done by Forrest et al., [7, 11, 12, 21], in intrusion detection using sequences of system calls. A new approach building upon this research will be introduced and the effectiveness of this approach will be compared to the Hamming distance approach on a known exploit.

5.3.1 Unification of Forrest et al. Approach

A recurring theme throughout the research done by Forrest et al. is the notion that intrusions produce temporal clusters of anomalies [7, 11, 21, 12]. By representing normal as sequences of system calls, this behavior can be observed at multiple levels. By sliding a window of size $k + 1$ across a system call trace, the temporal clustering of anomalies can be observed within each individual window. Hofmeyr et al. propose the Hamming distance between the input trace and the database be used to quantify this behavior at the sliding window level [11]. Furthermore, Warrender et al., [21], propose using an LFC (see Section 4.3) to quantify this behavior at the next level higher by counting the number of anomalies in a sequence of sliding windows. These two approaches are the same. More precisely, calculating the Hamming distance between an input window and the database of normal (when represented as explicit sequences) is really calculating the LFC with a frame size of one (independent of the window size k).

The parameterization of the LFC is described vaguely in the research using it to quantify the signal of an intrusion [12, 21]. Little or no empirical analysis is presented for the choice of frame size or threshold. The only detailed analysis of this approach comes from the evaluations presented by Hofmeyr et al., [11], regarding the use of a threshold based on the minimum Hamming distance between an input window and the definition of normal. Furthermore, the use of the LFC in `pH` (see Section 2.3.1) is *not* intrusion detection but a form of temporal sandboxing based on system call delays [12]. `pH` institutes delays into the Linux kernel scheduling algorithm based on the number of recent anomalous sequences. The result is an eventual program termination due to impatience or operating system timeouts. The general idea is right i.e. the notion that the temporal clustering of anomalies should be quantified.

In order to define a threshold T for use with the Hamming distance, the definitions of false positive and false negative as defined by Hofmeyr et al. [11] need to be addressed. These terms are described as follows:

False Positive "A false positive occurs when a single sequence generated by legitimate behavior is classified as anomalous."

False Negative "A false negative occurs when none of the sequences generated by an intrusion are classified as anomalous, i.e. when all of the sequences generated by an intrusion appear in the normal database."

These definitions are confusing. The confusion lies in the definition of false negative, in particular, the overloading of the term anomalous. It is highly unlikely that any of the training done in [11], is complete. Complete meaning that the database of normal behavior contains all sequences that could possibly be created by a given application. The result of incomplete training is *noise*. More specifically, the observations of anomalous behavior will produce some $\chi_i > 0$ (in this case, $\chi_i =$ Hamming distance). Therefore, in order to limit false positives, a threshold T must be chosen such that any $\chi_i > T$ will be considered anomalous.

Hofmeyr et al. confirm this definition of anomalous by describing thresholds, and therefore anomalies as a function of $d_{min}(i)$, and not as a function of the observed sequences containment in the database of normal behavior as suggested by the use of the term anomalous in their definition of false negative. Hofmeyr et al. suggests that the threshold T should be set such that any sequence with $d_{min}(i) > T$, where $1 \leq T \leq k$, be classified as anomalous[2].

Hofmeyr et al. states that they are more willing to tolerate false negatives than false positives. Furthermore, because of the way they constructed normal, they assume zero false positives, i.e. any Hamming distance > 0 indicates an anomaly [11]. What this really means is that the threshold value, T, for the Hamming distance, is a function of the signal observed from an intrusion. By observing the Hamming distance during an intrusion, they set the threshold T such that the intrusion will always be detected and noise will be eliminated.

After making this clarification, their definitions of false positive and false negative then become:

False Positive A false positive occurs when $\chi_i > T$ is observed during legitimate behavior.

False Negative A false negative occurs when none of the sequences generated during an intrusion produce $\chi_i > T$.

These definitions are now consistent with the definitions from Chapter 4. The only difference now becomes the definition of T. By defining T (and the definitions of false positive and false negative), as a function of the Hamming distance during an intrusion leads to the conclusion that this metric is based upon exploit detection and not anomaly detection. This approach is much more complex than anomaly detection and requires more than one exploit to make any conclusions about setting T. This approach is limited to known exploits. Exploits are much harder to come by than normal behavior so evaluating T empirically would be difficult. Furthermore, if the

[2]Note that the setting of a threshold is not described consistently in [11]. On the one hand, they claim that thresholds should be set using $d_{min}(i)$. On the other hand, they show intrusion data reporting $\hat{S_A}$ while claiming that certain values of $\hat{S_A}$ would detect intrusions– as if $\hat{S_A}$ was the metric. In order to be consistent with their analysis, $\hat{S_A}$ will be used in the analysis.

anomalies of an unknown exploit have a wide temporal distribution, this approach may not work.

5.3.2 Event Counter Approach

The event counter will now be described as an improvement to the research done by Forrest et al. in detecting intrusions using sequences of system calls [7, 11, 21, 12]. Although this method is being presented in conjunction with the use of explicit sequences of system calls to detect anomalies in system call traces, it has applications independent of the characterization of normal behavior. Unlike the Hamming distance approach, the event counter metric is generic enough that it can be implemented independently of the underlying representation. For example, the event counter approach could be applied to the Hamming distance metric to quantify the temporal distribution of sequences of system calls reaching some minimum Hamming distance. More generally, the event counter metric serves as a time integration of the anomaly signal in real time.

As described in Section 4.5, the event counter metric was designed with the intention of quantifying the temporal clustering of anomalies observed during an intrusion while maximizing the signal to noise ratio. Maximizing the signal to noise ratio also increases the flexibility in choosing a threshold T that distinguishes legitimate behavior from an intrusion. These properties can be achieved by modeling the temporal distribution of anomalies as a queue where the value of the event counter represents the height of the queue at any given time. The increment I is then set to reflect the expected arrival of anomalies so that the value of the event counter increases towards infinity during an intrusion and quickly settles to the floor when an anomaly is encountered during normal behavior.

Unlike the Hamming distance metric, where T is based upon an intrusion, the event counter metric sets the value for T based upon the noise expected during normal behavior. For clarity and consistency with the description of T for the Hamming distance metric from Section 5.3.1, the terms false positive and false negative will be reproduced from Chapter 4.

In Chapter 4, a false positive was described as the observation of $\chi_i > T$ during legitimate behavior. Furthermore, false negatives were defined as the absence of any $\chi_i > T$ observed during an intrusion. Acknowledging the improbability of *perfect* training, it is reasonable to concluded that some measure of noise is inevitable. Therefore, in order to take this noise into account, T is set based upon the amount of noise expected from normal behavior.

The event counter metric is primarily defined by three parameters, the threshold T, the *inc* and the *dec*. The threshold T and the *dec* are statically defined with T being a function of the *dec* (which is the same as the window size, see Section 4.5). The increment, *inc*, is a dynamic parameter tunable by the user. The *inc* can be set to respond to the expected noise and is therefore more flexible. Furthermore, the tuning of *inc* can be done in real time based on changes in the environment.

5.3.3 New Approach Based on Noise Analysis

This section will present the empirical motivation of the new approach by comparing it to the Hamming distance approach (using the $\hat{S_A}$ metric, see Section 4.2) introduced by Forrest et al. The comparison presented will be done using a known exploit. Furthermore, the two approaches will also be compared based on their suitability for an on-line intrusion detection system. Finally, an analysis of their individual signal to noise ratios will be presented.

5.3.3.1 Empirical Motivation

This experiment was devised to compare the event counter metric introduced in this thesis (Section 4.5) to the Hamming distance approach (Section 4.2) introduced by Forrest et al [11]. Anomaly detection should be based on noise analysis. The event counter metric introduced in this thesis defines a threshold as a function of the noise expected. The evaluation made in this section will attempt to show that the event counter approach is at least as effective as the Hamming distance approach introduced by Forrest et al.

This experiment is different from those above in that the lexical analyzer implementation was used to apply the event counter metric, and the `stide` tool, [19], was used to apply the $\hat{S_A}$ metric. This tool is made available by Forrest et al. for the post-processing of system call traces using the tree approach described in Section 3.4. Note that the lexical analyzer approach (Section 3.5) and the tree approach (Section 3.4) are equivalent in function.

This comparison was made using window sizes 4, 6, and 11. To create a definition of normal, the same set of sequences were used for both the lexical analyzer and `stide`. (See Table 5.1 for a summary of the type and amount of training used).

To get an idea of how these two metrics compared, two traces were done for each application. The first trace is a sample trace taken from a data collection that produced some anomalous sequences. The second trace was taken during a well known intrusion.

The results for each of the applications in Table 5.1 are presented in Figures 5.6–5.17. Note the scale of the graphs. For example, in Figure 5.12 the y scale for the event counter metric ranges from $0 \leq y \leq 25{,}000$, while all of the $\hat{S_A}$ graphs range between $0 \leq y \leq 1.2$.

The increment for the event counter was chosen based upon the expected amount of noise. This expectation was derived from the percentage of sequences rejected over the sample trace. For example, if `traceroute` rejects 10.11% of the sequences with a window size of 4, the percentage is rounded to the nearest whole number. In this case 10%. The increment is then set to 1 / 0.1, or 10. If the percentage of sequences rejected is below one, a floor of one is assumed. The decrement is set to the window size k as described in Section 4.5. The increment parameter is not fixed. It can be determined empirically.

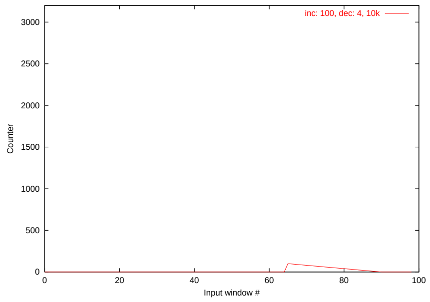
The $\hat{S_A}$ graphs in Figures 5.6–5.17 suggest that intrusions exhibit clumps of anomalous sequences. Furthermore, these graphs also suggest that legitimate behavior exhibits an occasional spike in the quantitative measure as indicated by both metrics. Without measuring the cumulative effect of these temporally distributed anomalous sequences, it is unclear how effectively $\hat{S_A}$ could be used to distinguish anomalies occurring during legitimate behavior and anomalies occurring during an intrusion.

Note the increase in the amount of noise measured during the sample traces when the size of the training set is reduced to 1,000 system calls. For the event counter metric, this increase appears to have little effect on the exploit graph, the exploit graph still appears to be easily distinguishable from normal behavior. On the other hand, some of the sample trace graphs using the $\hat{S_A}$ metric exhibit $\hat{S_A}$ values at nearly the same height as the exploit.

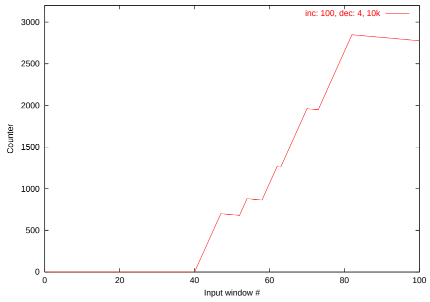
The window size also appears to have a negative impact on the $\hat{S_A}$ graphs. In particular, the `su` graphs in Figures 5.12–5.14 suggest that the $\hat{S_A}$ metric is more sensative the window size when the training set is reduced to 1,000 sequences. This is particularly evident in Figure 5.12, graph (7). In this graph, the noise from the sample trace is indistinguishable from the signal produced during the intrusion (graph (8)). On the other hand, the event counter metric clearly detects the intrusion (graphs (5) and (6)).

The remainder of this section will evaluate the event counter approach and the Hamming distance approach to determine if the event counter approach is at least as effective as the Hamming distance approach. This evaluation will be be made using three criteria. First, each metric will be evaluated on its ability to determine if χ_i represents an intrusion.

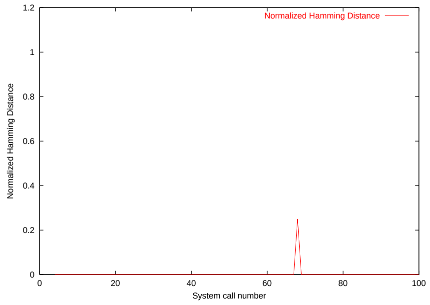
Next, given the motivation presented in the introduction, for an IDS to be effective and difficult to undermine, it must be incorporated into the operating system and be capable of detecting intrusions in real time. Since the constraints on implementing



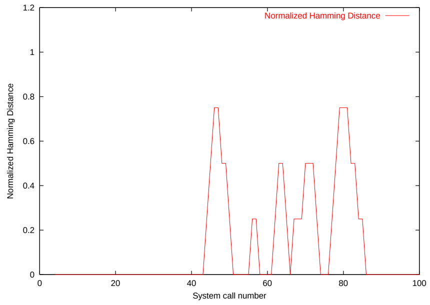
(1) Event Counter: dump Sample



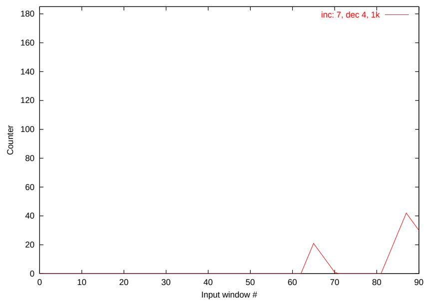
(2) Event Counter: dump Exploit



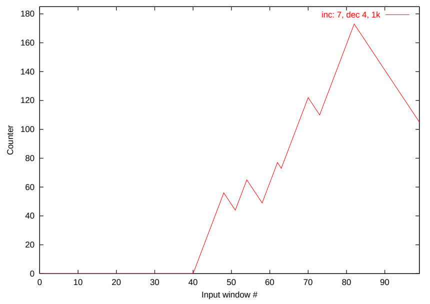
(3) $\hat{S_A}$: dump Sample



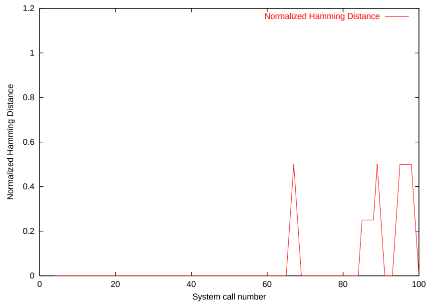
(4) $\hat{S_A}$: dump Exploit



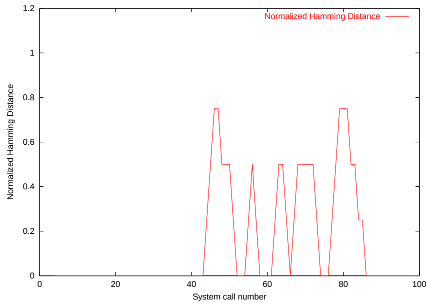
(5) Event Counter: dump Sample



(6) Event Counter: dump Exploit

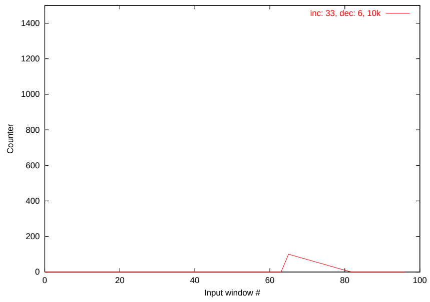


(7) $\hat{S_A}$: dump Sample

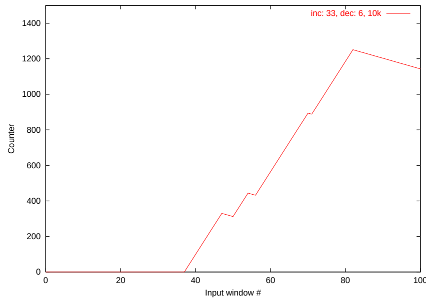


(8) $\hat{S_A}$: dump Exploit

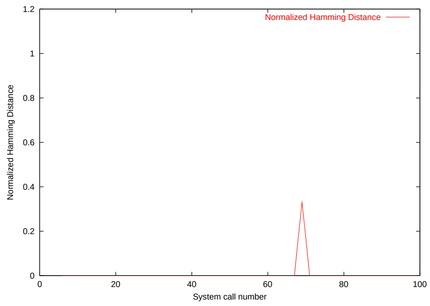
Figure 5.6. /sbin/dump window size 4. Graphs (1)–(4) were compiled using a training size of 10,000 sequences. Graphs (5)–(8) were compiled using a training size of 1000 sequences.



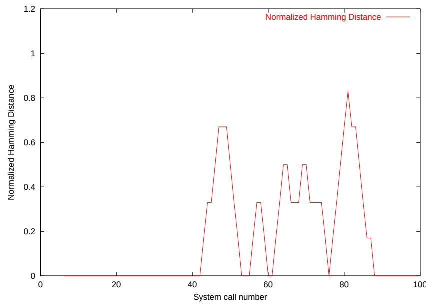
(1) Event Counter: dump Sample



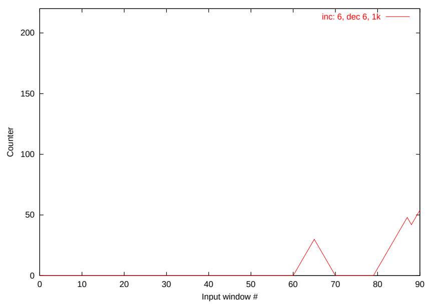
(2) Event Counter: dump Exploit



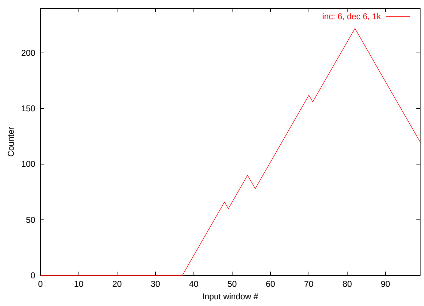
(3) $\hat{S_A}$: dump Sample



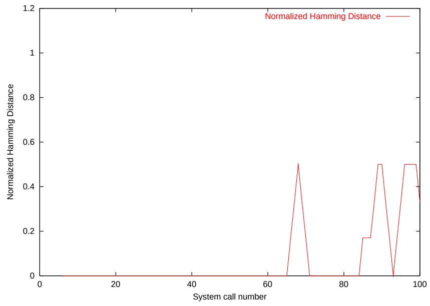
(4) $\hat{S_A}$: dump Exploit



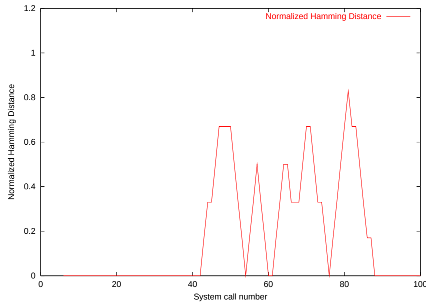
(5) Event Counter: dump Sample



(6) Event Counter: dump Exploit

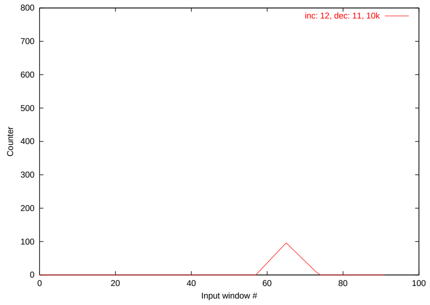


(7) $\hat{S_A}$: dump Sample

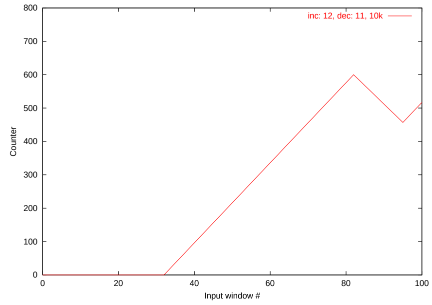


(8) $\hat{S_A}$: dump Exploit

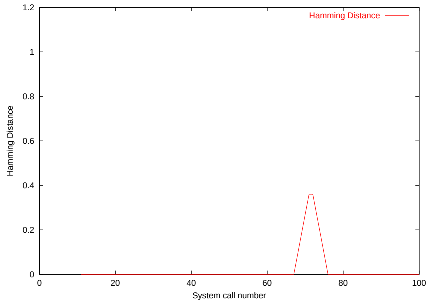
Figure 5.7. /sbin/dump window size 6. Graphs (1)–(4) were compiled using a training size of 10,000 sequences. Graphs (5)–(8) were compiled using a training size of 1000 sequences.



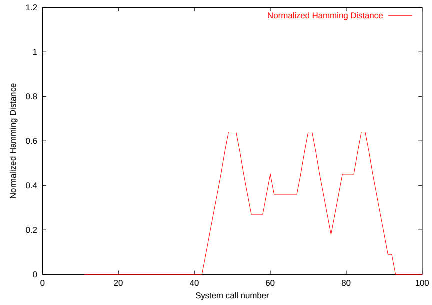
(1) Event Counter: dump Sample



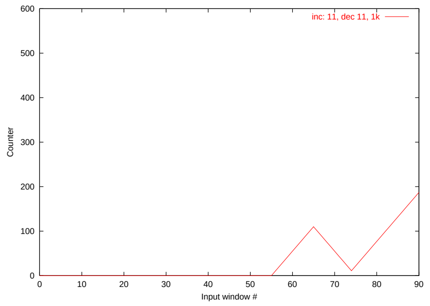
(2) Event Counter: dump Exploit



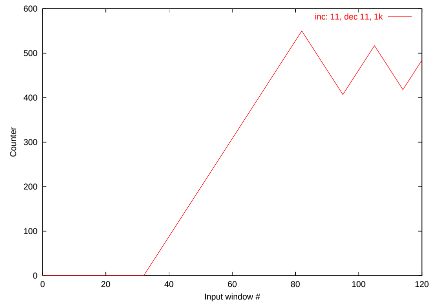
(3) $\hat{S_A}$: dump Sample



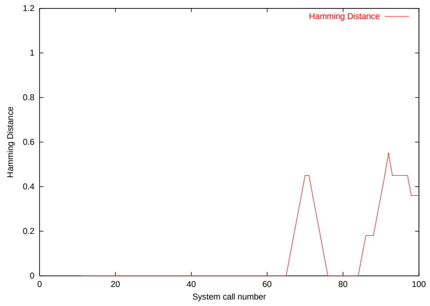
(4) $\hat{S_A}$: dump Exploit



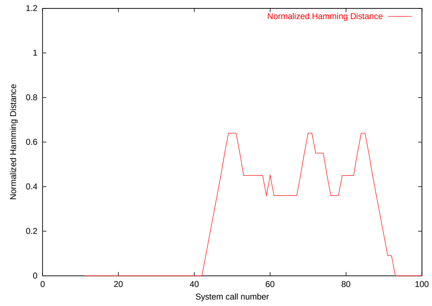
(5) Event Counter: dump Sample



(6) Event Counter: dump Exploit

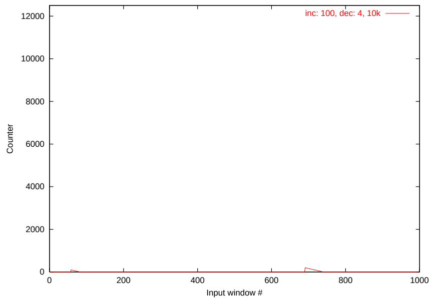


(7) $\hat{S_A}$: dump Sample

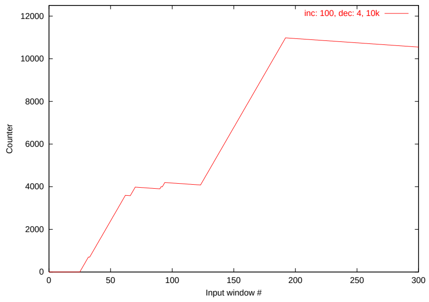


(8) $\hat{S_A}$: dump Exploit

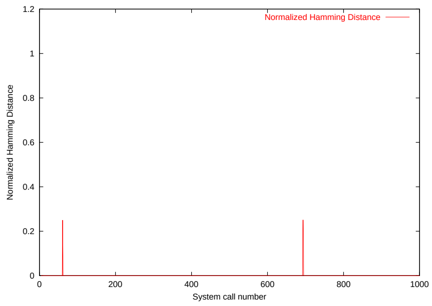
Figure 5.8. /sbin/dump using window size 11. Graphs (1)–(4) were compiled using a training size of 10,000 sequences. Graphs (5)–(8) were compiled using a training size of 1000 sequences.



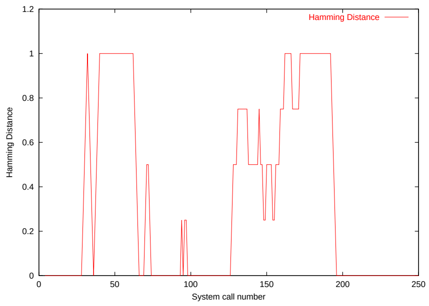
(1) Event Counter: ls Sample



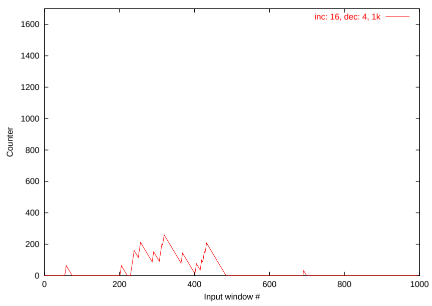
(2) Event Counter: ls Exploit



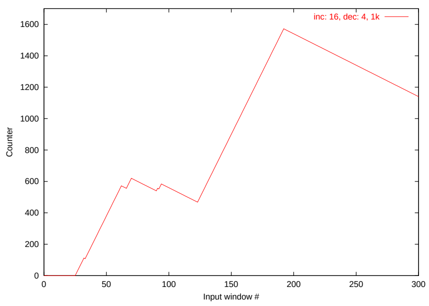
(3) $\hat{S_A}$: ls Sample



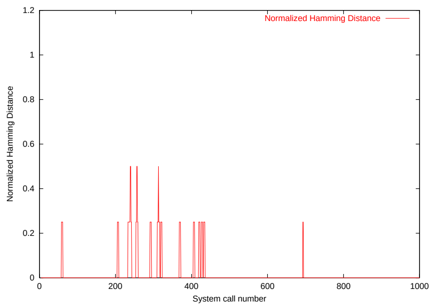
(4) $\hat{S_A}$: ls Exploit



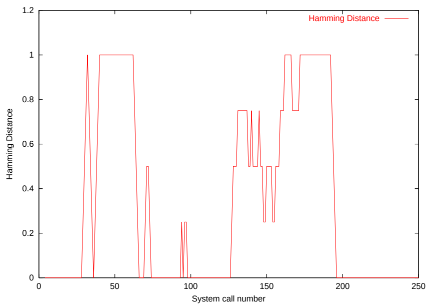
(5) Event Counter: ls Sample



(6) Event Counter: ls Exploit

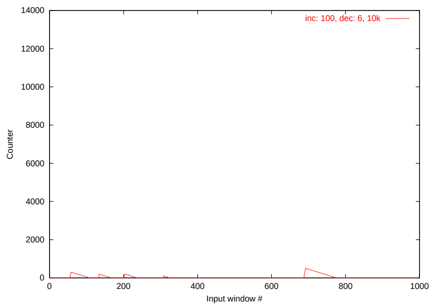


(7) $\hat{S_A}$: ls Sample

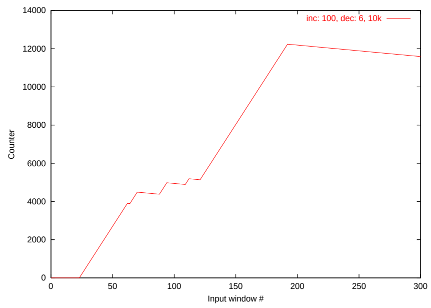


(8) $\hat{S_A}$: ls Exploit

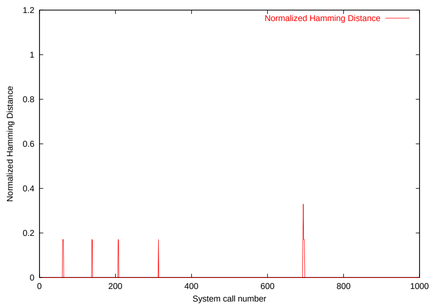
Figure 5.9. /bin/ls using window size 4. Graphs (1)–(4) were compiled using a training size of 10,000 sequences. Graphs (5)–(8) were compiled using a training size of 1000 sequences.



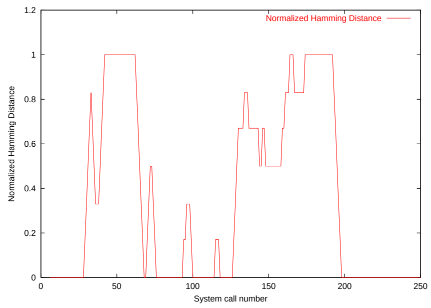
(1) Event Counter: ls Sample



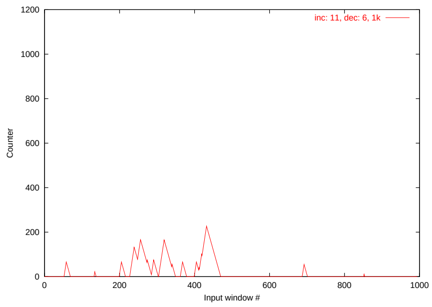
(2) Event Counter: ls Exploit



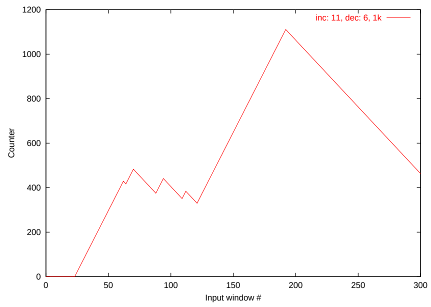
(3) $\hat{S_A}$: ls Sample



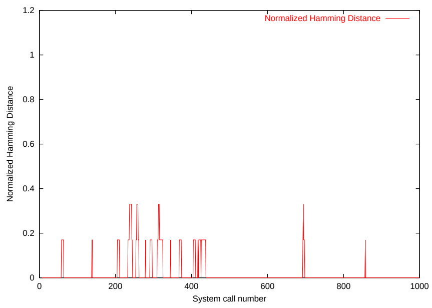
(4) $\hat{S_A}$: ls Exploit



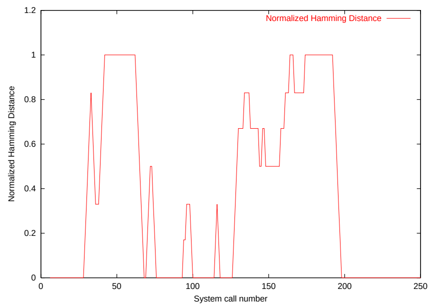
(5) Event Counter: ls Sample



(6) Event Counter: ls Exploit

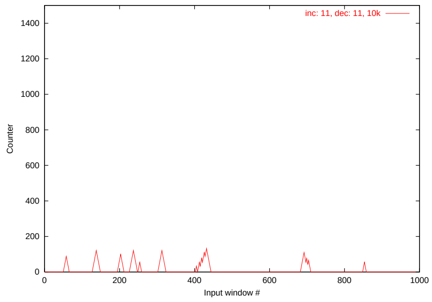


(7) $\hat{S_A}$: ls Sample

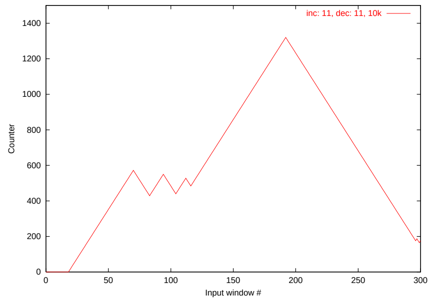


(8) $\hat{S_A}$: ls Exploit

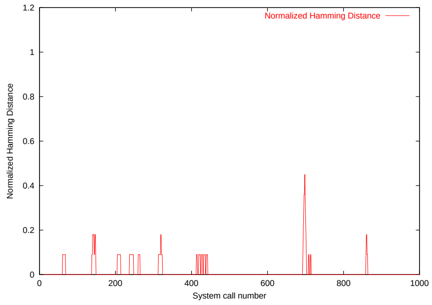
Figure 5.10. /bin/ls using window size 6. Graphs (1)–(4) were compiled using a training size of 10,000 sequences. Graphs (5)–(8) were compiled using a training size of 1000 sequences.



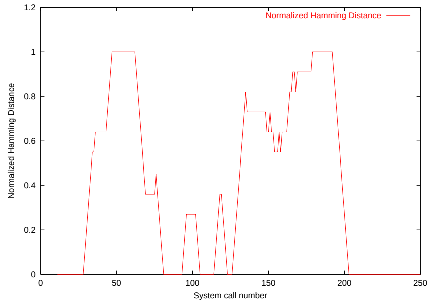
(1) Event Counter: ls Sample



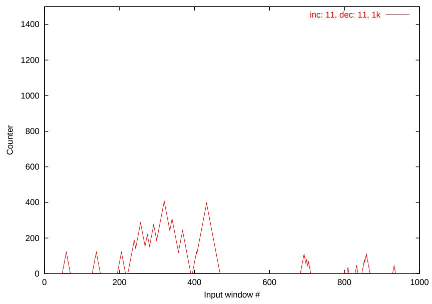
(2) Event Counter: ls Exploit



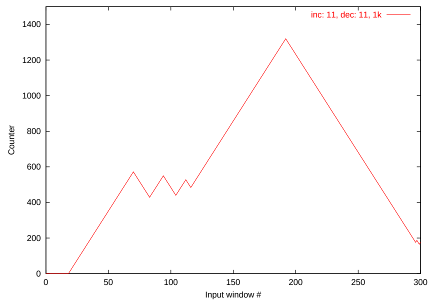
(3) $\hat{S_A}$: ls Sample



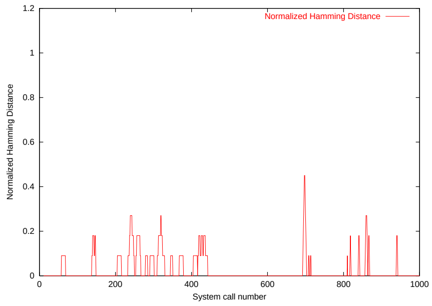
(4) $\hat{S_A}$: ls Exploit



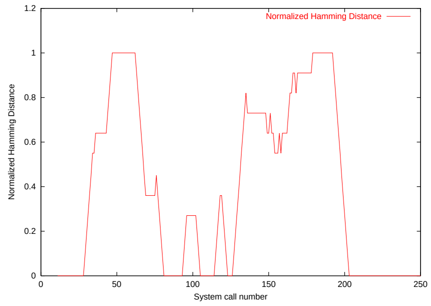
(5) Event Counter: ls Sample



(6) Event Counter: ls Exploit

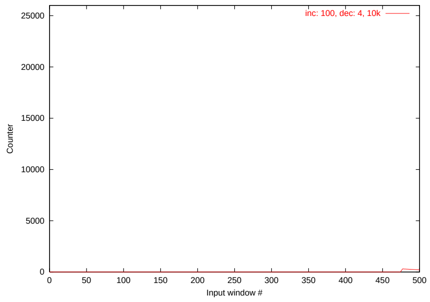


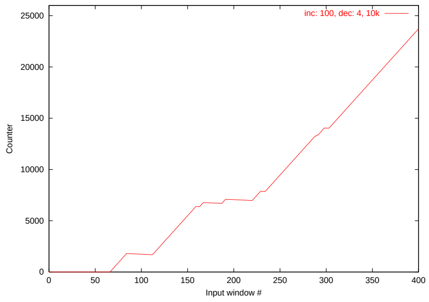
(7) $\hat{S_A}$: ls Sample



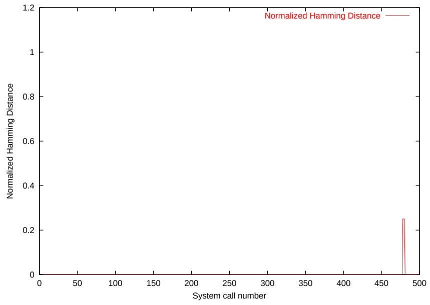
(8) $\hat{S_A}$: ls Exploit

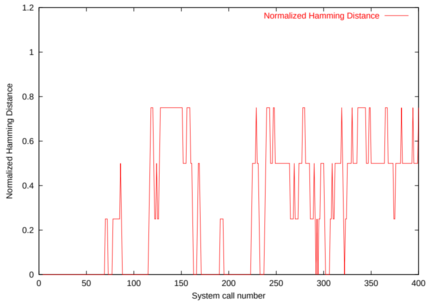
Figure 5.11. /bin/ls using window size 11. Graphs (1)–(4) were compiled using a training size of 10,000 sequences. Graphs (5)–(8) were compiled using a training size of 1000 sequences.



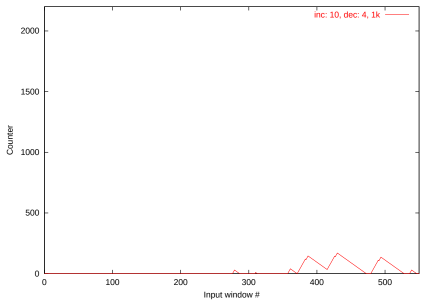


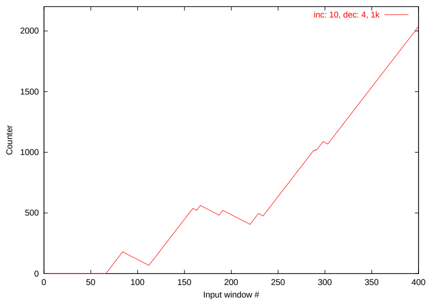
(1) Event Counter: su Sample (2) Event Counter: su Exploit



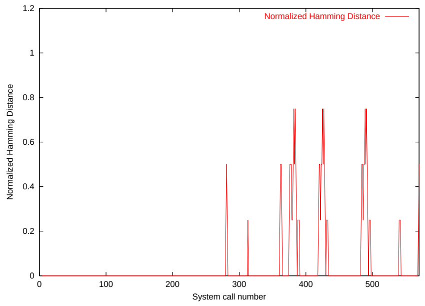


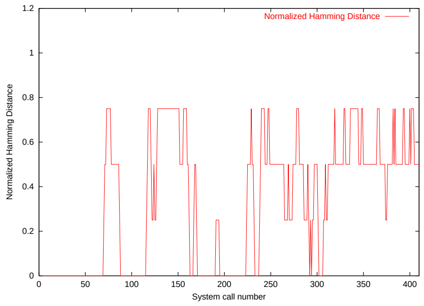
(3) $\hat{S_A}$: su Sample (4) $\hat{S_A}$: su Exploit





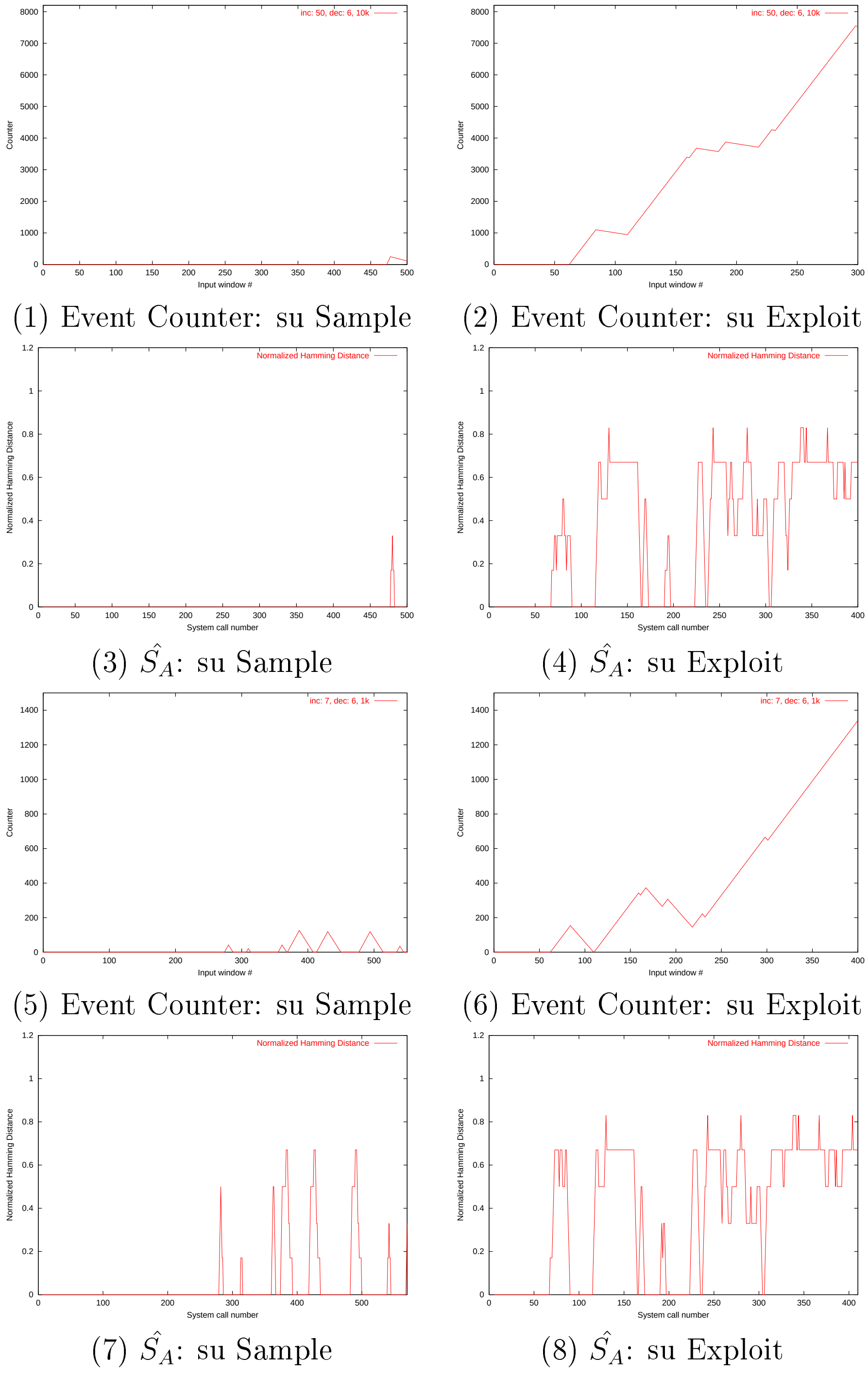
(5) Event Counter: su Sample (6) Event Counter: su Exploit





(7) $\hat{S_A}$: su Sample (8) $\hat{S_A}$: su Exploit

Figure 5.12. /bin/su using window size 4. Graphs (1)–(4) were compiled using a training size of 10,000 sequences. Graphs (5)–(8) were compiled using a training size of 1000 sequences.



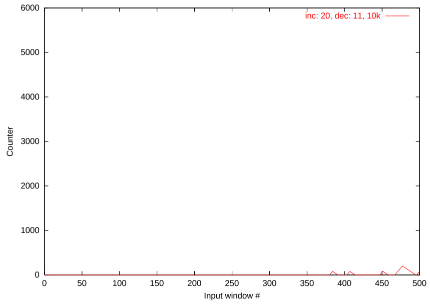
(1) Event Counter: su Sample (2) Event Counter: su Exploit

(3) $\hat{S_A}$: su Sample (4) $\hat{S_A}$: su Exploit

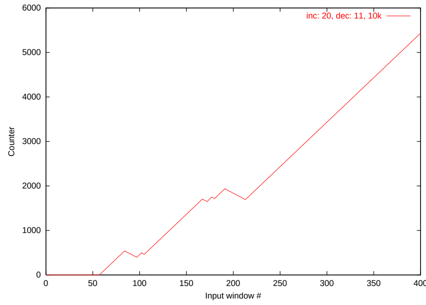
(5) Event Counter: su Sample (6) Event Counter: su Exploit

(7) $\hat{S_A}$: su Sample (8) $\hat{S_A}$: su Exploit

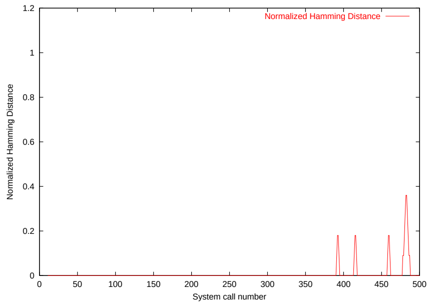
Figure 5.13. /bin/su using window wize 6. Graphs (1)–(4) were compiled using a training size of 10,000 sequences. Graphs (5)–(8) were compiled using a training size of 1000 sequences.



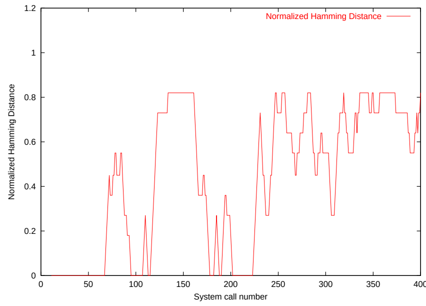
(1) Event Counter: su Sample



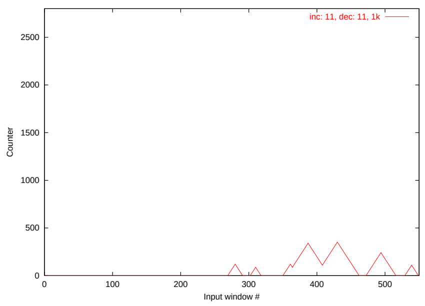
(2) Event Counter: su Exploit



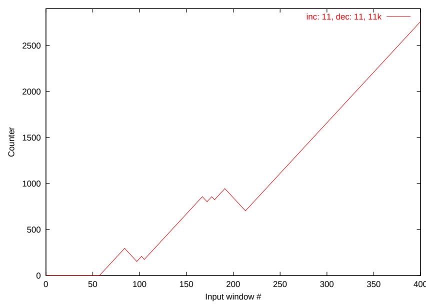
(3) $\hat{S_A}$: su Sample



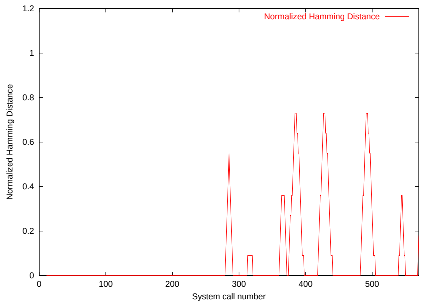
(4) $\hat{S_A}$: su Exploit



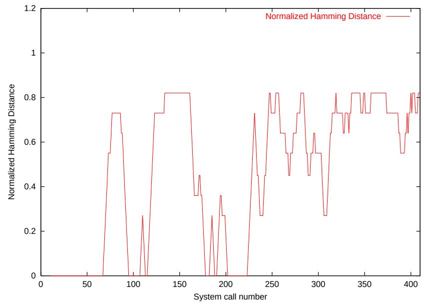
(5) Event Counter: su Sample



(6) Event Counter: su Exploit

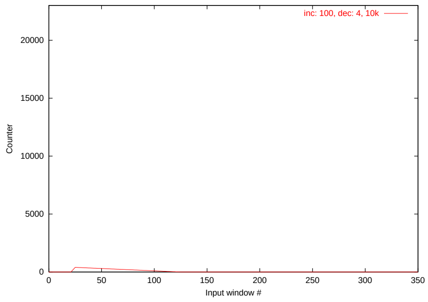


(7) $\hat{S_A}$: su Sample

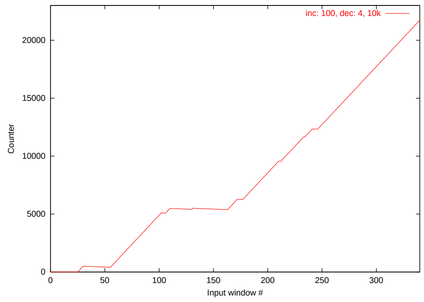


(8) $\hat{S_A}$: su Exploit

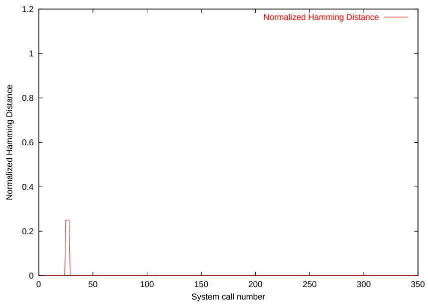
Figure 5.14. /bin/su using window size 11. Graphs (1)–(4) were compiled using a training size of 10,000 sequences. Graphs (5)–(8) were compiled using a training size of 1000 sequences.



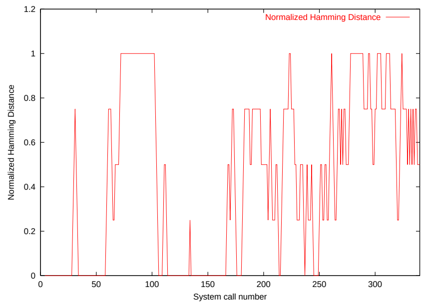
(1) Event Counter: traceroute Sample



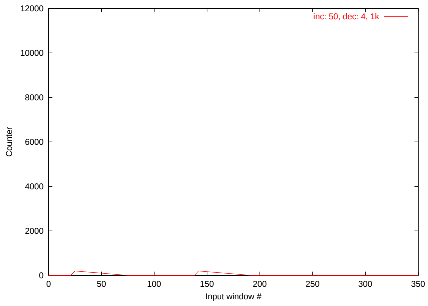
(2) Event Counter: traceroute Exploit



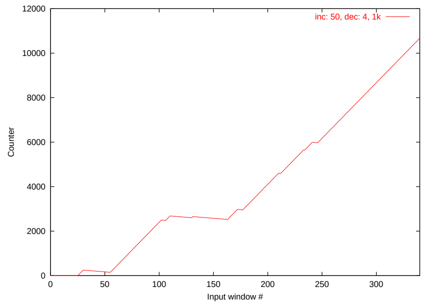
(3) $\hat{S_A}$: traceroute Sample



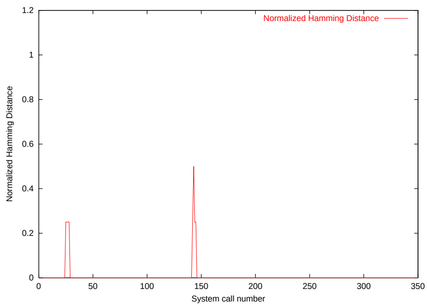
(4) $\hat{S_A}$: traceroute Exploit



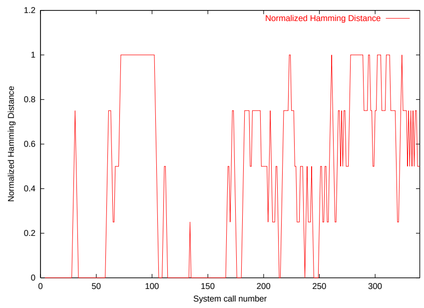
(5) Event Counter: traceroute Sample



(6) Event Counter: traceroute Exploit

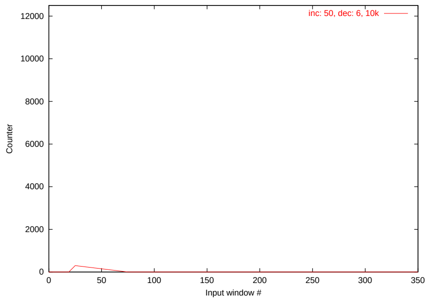


(7) $\hat{S_A}$: traceroute Sample

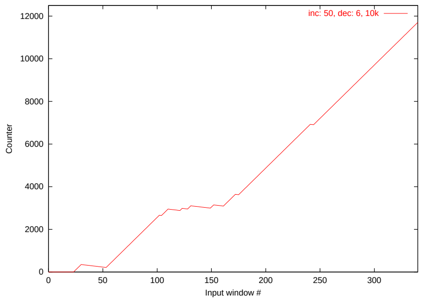


(8) $\hat{S_A}$: traceroute Exploit

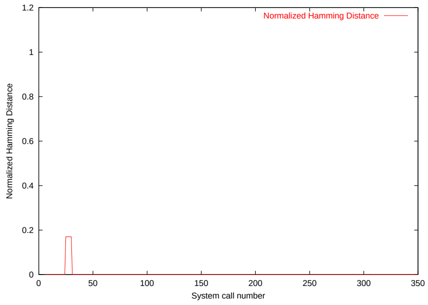
Figure 5.15. /usr/sbin/traceroute using window size 4. Graphs (1)–(4) were compiled using a training size of 10,000 sequences. Graphs (5)–(8) were compiled using a training size of 1000 sequences.



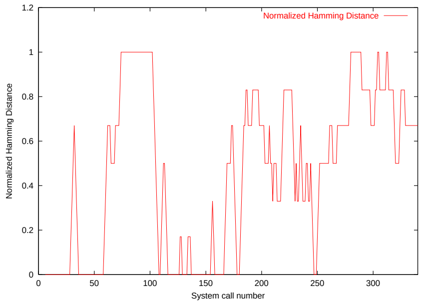
(1) Event Counter: traceroute Sample



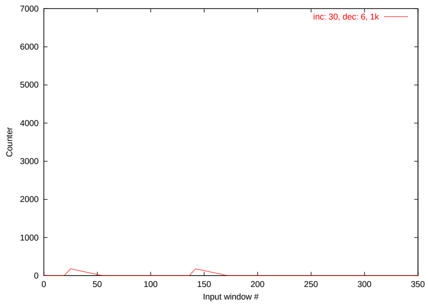
(2) Event Counter: traceroute Exploit



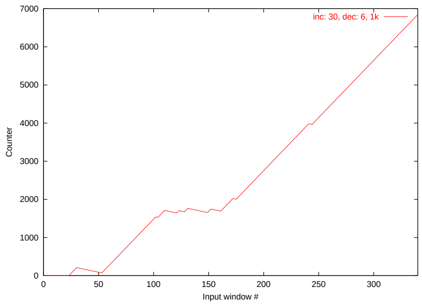
(3) $\hat{S_A}$: traceroute Sample



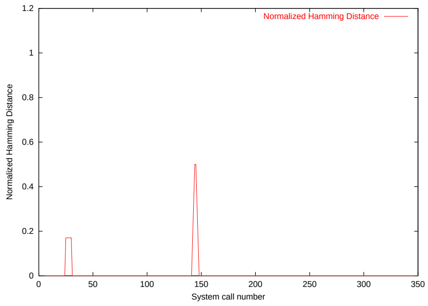
(4) $\hat{S_A}$: traceroute Exploit



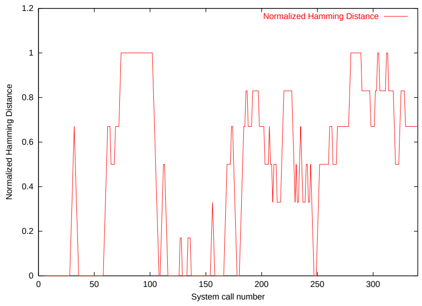
(5) Event Counter: traceroute Sample



(6) Event Counter: traceroute Exploit

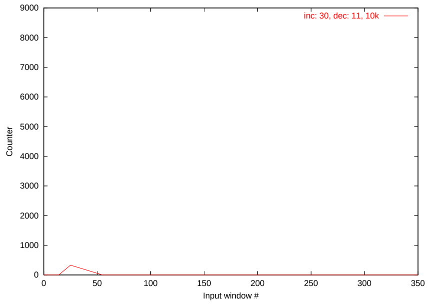


(7) $\hat{S_A}$: traceroute Sample

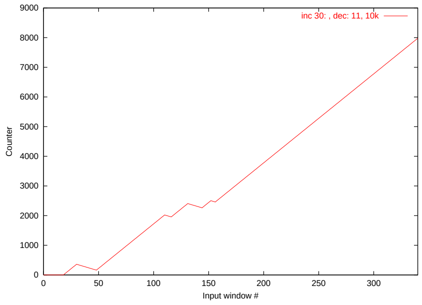


(8) $\hat{S_A}$: traceroute Exploit

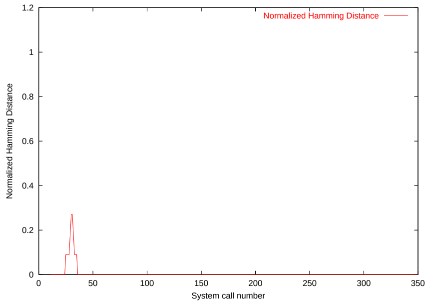
Figure 5.16. /usr/sbin/traceroute using window size 6. Graphs (1)–(4) were compiled using a training size of 10,000 sequences. Graphs (5)–(8) were compiled using a training size of 1000 sequences.



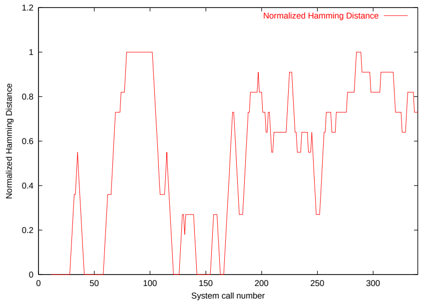
(1) Event Counter: traceroute Sample



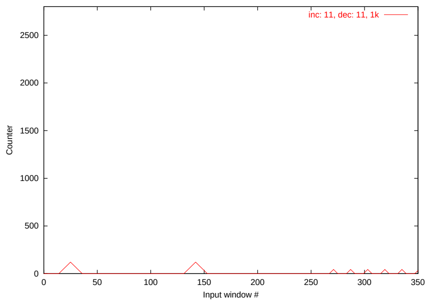
(2) Event Counter: traceroute Exploit



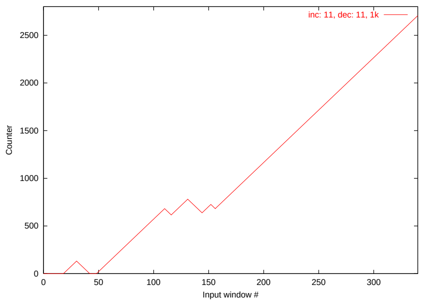
(3) $\hat{S_A}$: traceroute Sample



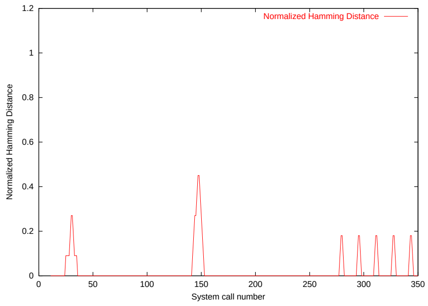
(4) $\hat{S_A}$: traceroute Exploit



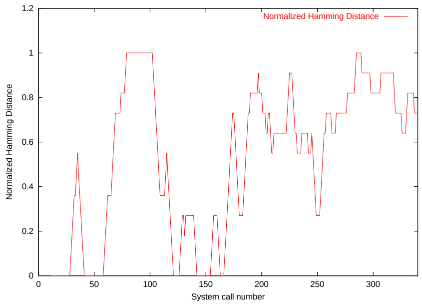
(5) Event Counter: traceroute Sample



(6) Event Counter: traceroute Exploit



(7) $\hat{S_A}$: traceroute Sample



(8) $\hat{S_A}$: traceroute Exploit

Figure 5.17. /usr/sbin/traceroute using window size 11. Graphs (1)–(4) were compiled using a training size of 10,000 sequences. Graphs (5)–(8) were compiled using a training size of 1000 sequences.

an IDS in kernel-space are different than those placed on a similar IDS in user-space, one might also evaluate the overhead of using a given metric in kernel-space.

Finally, an effective intrusion detection system should have a large signal to noise ratio. This is based on the assumption [3] that some measure of noise is unavoidable. Furthermore, the larger the signal to noise ratio, the more flexibility the security administrator has in setting the tolerance for false positives.

It is by these three criteria that the event counter metric and $\hat{S_A}$ will be compared. These metrics will be evaluated using Figures 5.6–5.17 along with the analysis from Chapter 4.

5.3.3.2 Comparing Intrusion Detection Approaches

This section will compare the event counter approach and the Hamming distance approach based on their ability to detect intrusions. The analysis presented will attempt to show that the event counter approach is at least as effective in detecting intrusion as the hamming distance approach and more effective when small windows are used with less training.

From the results presented in Figures 5.6–5.17 it is clear from graphs representing a training size of 10,000 that both the event counter approach and the Hamming distance approach are capable of detecting intrusions. The exploit graphs on the right are easily distinguishable from the sample graphs on the left in terms of the height of the anomalous signal. It is not so clear when the training size is reduced to 1,000.

As previously stated, the threshold value for the $\hat{S_A}$ metric is a function of the signal observed from an intrusion. Furthermore, in order to eliminate noise entirely while still catching the intrusion, a threshold T must be set one unit of measurement

[3] This assumption is not unreasonable. During the training of `traceroute`, sliding a window of size 11 across 10,000 system calls produced a database with 439 unique sequences, composed of only 29 different system calls. Assuming that the sampling of `traceroute` included all of the system calls used, the database of normal sequences contains 4^{-14}% (439 / 29^{11}) of the total possible size of the database.

below the highest $\hat{S_A}$ value observed during an intrusion. Using Figure 5.8, graphs (7) and (8), as an example, the highest $\hat{S_A}$ value observed in graph (8) is 7/11. According to [11], T must be set at 6/11 to insure detection of this exploit. However, in graph (7), the peak $\hat{S_A}$ value observed is also 6/11. This implies that a Hamming distance of one determines normal behavior from an intrusion.

A problem is illustrated by `su` with a training size of 1,000. Note the Hamming distance graphs in Figure 5.12, graphs (7) and (8). In this case, the noise observed during the sample trace is indistinguishable from the signal observed during the exploit trace. Recall from Section 5.3.1, that the threshold T for the Hamming distance approach is based on the signal observed during an intrusion. Based on this, the signal observed in Figure 5.12 graph (7) (the sample trace) would be reported as an intrusion (a false positive).

A small window size is regarded as beneficial to on-line intrusion detection. Figure 5.5 illustrates this fact. Furthermore, it is also assumed that the training of the database representing normal behavior is not likely to be complete. The queuing theory model presented in Section 4.5 is tested under these circumstances. Figure 5.18 is an experiment to validate the threshold model from queueing theory. This experiment demonstrates that the simple queuing model is valid under these circumstances.

Figure 5.18 depicts the sample traces side by side with the exploit traces using the event counter metric for window size 4, and a training size of 1,000 sequences. The upper and lower bound to the threshold have been superimposed for the purpose of this discussion (see Section 4.5 for a detail discussion of the upper and lower bounds).

The height of the event counter curve in Figure 5.18 graph (1) remains within an acceptable tolerance between the lower bound and the upper bound of the threshold. The upper bound of the threshold is never exceeded during the sample trace. On the other hand, the exploit trace in Figure 5.18 graph (2) clearly exceeds the upper bound threshold by system call 80. The signal produced by the sample traces in graphs (3) and (5) also produce the same behavior. Likewise, the signal observed during the intrusion in graphs (4) and (6) also clearly exceed the upper bound threshold.

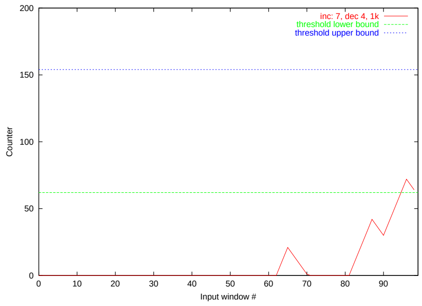
The noise produced by the sample trace in Figure 5.18 graph (7) is the only exception. In this graph, the signal observed during normal behavior never exceeds the lower bound threshold. This is the ideal situation. This give the user the greatest flexibility in choosing a threshold that accurately reflects the local tolerances for false positives and false negatives.

Although three of these examples exceed the lower bound of the threshold, this does not present a problem. The range between the lower bound and the upper bound is intended to supply the user with a suggested threshold depending on the environment and the tolerance for false positives and false negatives. Choosing a threshold near the lower bound increases the sensitivity and the likely hood of false positives. Choosing a higher threshold reduces the sensitivity and the possibility for false positives.

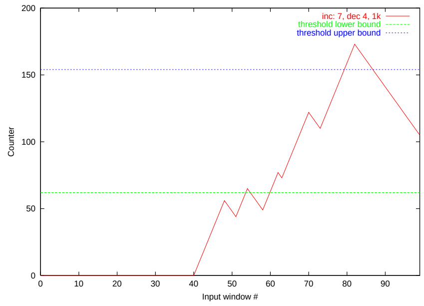
The observations presented in this section give enough leading evidence to suggest that event counter metric is at least as effective in detecting intrusions as the Hamming distance approach. These results also suggest that the determination of the threshold based on the queuing theory model in combination with the event counter metric can be used effectively to detect these intrusions. Furthermore, these observations also suggest that the event counter metric is more effective in detecting intrusions given a reduced training set and a smaller window size. Finally, the event counter approach provides more flexibility in setting the threshold which increases the users flexibility in determining the tolerance for false positives and false negatives.

5.3.3.3 Signal to Noise Ratio

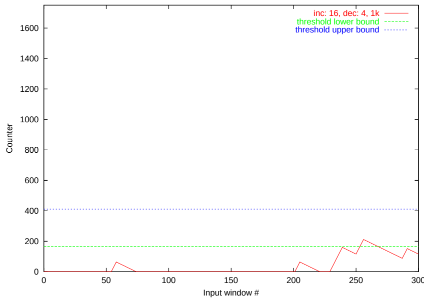
By eyeballing the graphs in Figures 5.6–5.17 it is clear that the signal to noise ratio is larger using the event counter metric. Table 5.3 summarizes the average signal to noise ratio over all four applications for window sizes 4, 6, and 11.



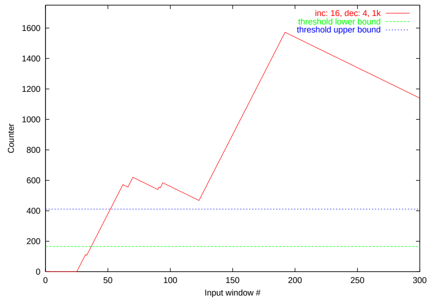
(1) Event Counter: dump Sample



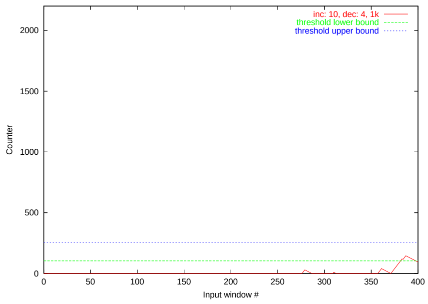
(2) Event Counter: dump Exploit



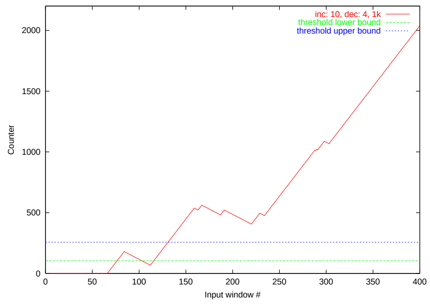
(3) Event Counter: ls Sample



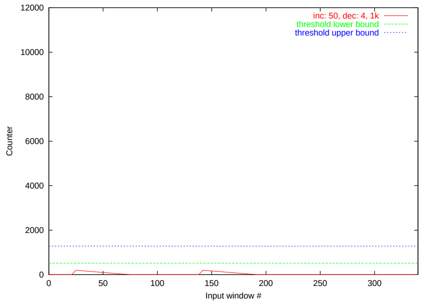
(4) Event Counter: traceroute Exploit



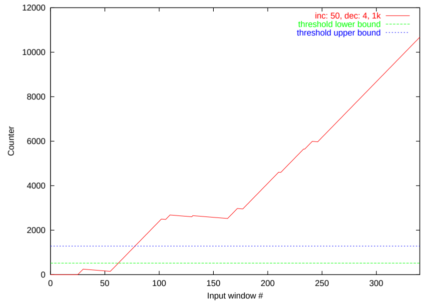
(5) Event Counter: su Sample



(6) Event Counter: su Exploit



(7) Event Counter: traceroute Sample



(8) Event Counter: traceroute Exploit

Figure 5.18. dump, ls. su, traceroute with theoretical upper and lower bounds. Training size 1,000.

Table 5.3. Signal to noise ratio for each application.

Window Size	$\hat{S_A}$	EC	Training Size
4	2.62	39.59	10,000
4	1.46	20.07	1000
6	3.49	31.05	10,000
6	1.60	15.46	1000
11	2.63	17.04	10,000
11	1.47	9.29	1000

This table suggests that on average, the event counter metric yields nearly a ten fold increase in the signal to noise ratio over $\hat{S_A}$. The result is a better separation between the security policy (tolerance for false positive/negative) and the mechanism by which it is implemented.

5.3.3.4 Applying these Metrics in Kernel-space

This criterion is not as simple as the previous. In Section 4.2, the overhead required to determine the Hamming distance between an input sequence and the database was discussed. To summarize, in order to determine if a sequence is accepted by the database requires $k-1$ comparisons (using the tree representation). However, when a sequence is not in the database, determining the Hamming distance requires $O(Nk)$, where N is the size of the database and k is the window size. This is a high price to pay.

Similarly, since the event counter metric is implemented using a DFA (the tree representation and a DFA are analogous), determining if a sequence is accepted or rejected requires k comparisons. The increment and the decrement of the counter are insignificant.

In terms of the storage requirements, the two implementations are also the same. Their storage is bound by $O(|\Sigma|^k)$, where Σ is the alphabet of system calls with size $|\Sigma|$. However, the use of $\hat{S_A}$ is bound to the tree implementation because accurately determining the Hamming distance is only possible when the definition of normal is stored as explicit sequences. Therefore, if space is more important than accuracy, the event counter metric provides more flexibility with its ability to accommodate either storage implementation (See Section 5.2.3.1).

Based on these observations, in the event of anomalies, $\hat{S_A}$ becomes much more expensive than the event counter metric. Therefore, since the storage requirements are the same, the evaluation time is the same $O(k)$ and $\hat{S_A}$ is more costly overall with respect to time, the event counter metric is more suitable for implementation in kernel-space.

CHAPTER 6

ON-LINE INTRUSION DETECTION

This Chapter will present an overview of an on-line intrusion detection system using sequences of system calls. The methods used to characterize normal behavior will be presented and the details regarding the on-line intrusion detection system will be introduced.

6.1 Characterization of Normal Using Lex

The implementation presented by this thesis uses Lex to characterize normal behavior using sequences of system calls. The mechanics behind this usage of Lex are introduced in Section 3.5. This Section will address some of the tradeoffs for using Lex in this capacity.

Lex has been used extensively by the compiler community to specify lexical analyzers for a variety of languages. The result is a lexical analyzer generator that is easy to use and simplifies the characterization of normal behavior using sequences of system calls. The resulting lexical analyzer, by default, is automatically optimized to reduce storage and maximize speed in recognition.

One drawback to using this approach is the rapid growth of the transition table used to represent the DFA when explicit sequences of system calls are used to represent normal behavior. Figure 5.5 illustrates this growth.

Another potential drawback to using this approach in a on-line IDS is the default use of dynamic memory allocation in the skeleton lexical analyzer used by Lex to create a lexical analyzer. Although this property is acceptable when employed in user-space, it can have negative consequences in kernel-space. The implementors of `flex` (the version of Lex popular to Linux machines) dynamically allocate memory with the expectation that it will be freed upon program termination. This is acceptable in user-space. However, in the context of a loadable kernel module (see Section 6.3) this memory will not be returned to the memory management subsystem unless the module is unloaded or the storage is explicitly freed. A few simple modifications to the skeleton lexical analyzer resolve this problem.

6.2 Kernel-based Implementation

The kernel-based implementation was developed using Linux kernel 2.4.0 on the Intel i386 architecture. In order to understand how this kernel based IDS is implemented, it helps to understand how the system call interface works on the this architecture. A description of how the transition is made from user-space to kernel-space will be presented in this section.

For brevity, the term i386 to refer the class of Intel CPU's including the Celeron, the Pentium II, and the Pentium III. All were used in the development of this implementation.

The following example illustrates how the string "Hello, World!" could be written to `stdout` using the `write` system call.

```
write(1, "Hello, world!", 14);
```

The following assembly statements illustrate how this system call is referenced using an interrupt [1]:

```
msg     db      "Hello, world!",0xa
...
```

[1] This file was written for use with nasm. See Appendix A for the complete listing of this program and more details about nasm.

```
mov     edx,14  ;third argument: message length
mov     ecx,msg ;second argument: pointer to message to write
mov     ebx,1   ;first argument: file handle (stdout)
mov     eax,4   ;system call number (sys_write)
int     0x80    ;call kernel
....
```

The system call is referenced by first loading the arguments to `write` into the registers used for function call arguments. First the length, then the string, and finally the file descriptor to write to. Next, the system call number is loaded into register `eax`. The different system calls and their corresponding numbers are mapped in the file include/unistd.h under the Linux source tree. In this case, `sys_write` is system call 4.

When the processor receives an interrupt (int 0x80), it halts the execution of the current process and switches execution to an interrupt handler designed specifically to handle the interrupt condition. In this case, a system call. The interrupt handler for system calls is the `system_call` function written in assembly in the Linux kernel. This function can be found in `arch/i386/kernel/entry.S`.

Next, what happens after the transition to kernel-space will be discussed. To explore this transition, the `system_call` function defined in `arch/i386/kernel/entry.S` will be described in detail. The important part of this function, relative to this discussion, is the beginning[2]:

[2] If this looks a little strange, it is because the the kernel uses AT&T assembler syntax. For more information, see the GNU assembler documentation.

```
ENTRY(system_call)
      pushl %eax                      # save orig_eax
      SAVE_ALL
      GET_CURRENT(%ebx)
      cmpl $(NR_syscalls),%eax
      jae badsys
      testb $0x02,tsk_ptrace(%ebx)    # PT_TRACESYS
      jne tracesys
      call *SYMBOL_NAME(sys_call_table)(,%eax,4)
      movl %eax,EAX(%esp)             # save the return value
      ...
```

The first argument to `system_call` is 4, the number of `sys_write`, which is stored in register `eax`. The `eax` register is then pushed on the stack (taking the position of `orig_eax` in the stack, see the top of entry.S for the stack layout). The next operation, `SAVE_ALL` copies all of the registers onto the stack so that `system_call` can use these registers. The `GET_CURRENT` macro then retrieves a pointer to the current processes' `task_struct` for use in the `write` system call. The `system_call` function then checks to see if the first argument, the number of the system call, is a legal value. If not, it branches to handle this condition. Otherwise, it then proceeds to check if this process is being traced. If so, it has a special routine that is invoked before the execution of the system call and after the system call returns (this is how programs like `strace` report information about the system calls used by a process). The next line is the most important. Every system call in the Linux kernel is stored as a table of function pointers. The system call `sys_write`, is at offset 4 in this table (see the bottom of `arch/i386/kernel/entry.S` for the table of system calls). `sys_write` is then referenced by looking up its offset in the table. The return value of call is then placed in `eax` (on the stack). This is how the calling process gets the return value.

When considering how to implement a prototype IDS into the Linux kernel, it is important to focus on finding an implementation that is fast, architecturally independent, and easy to implement. It is also desirable to avoid making any drastic changes to the Linux kernel. Two potential options were reviewed; re-writing the `system_call` function or wrapping the system calls individually.

It was decided that wrapping the system calls individually was more suitable. This can be done with minimal modification to the core Linux kernel. Furthermore, this approach can be implemented using a loadable kernel module, which gives the user the highest degree of flexibility, is easy to implement, and is (mostly) architecture independent.

6.3 Implementation: System Call Wrappers

Wrapping the system calls individually allows for the flexibility to treat each individual system call differently. This flexibility will be elaborated upon in Section 7.1. It also allows the integration our IDS into the Linux kernel without having to make extensive modifications.

Continuing with the example from Section 6.2, the following describes how the `write` system call would be wrapped. Recall that system calls are stored as a table of function pointers. Using a loadable module, an individual system call used by the kernel can be replaced by reassigning the corresponding function pointer in the table. The following code illustrates this:

```
asmlinkage ssize_t (*orig_sys_write)(unsigned int fd,
      const char * buf, size_t count);

asmlinkage ssize_t new_sys_write(unsigned int fd, const char *
      buf,size_t count) {
      int ret;
      ret = compare_sequence_to_database(current->window,
                                          __NR_write);
      /* make decision about ret here */

      return orig_sys_write(fd, buf, count);
}


int init_module(void) {
      orig_sys_write = sys_call_table[__NR_write];
      sys_call_table[__NR_write] = new_sys_write;

      return 0;
}
```

The `orig_sys_write` pointer is used to hold the function pointer to the original `sys_write` system call. The `new_sys_write` function is the wrapper. After the module is loaded (`init_module` is called), every call to the `write` system call will be invoking our wrapper. Inside this wrapper, the number of the system call, in this case 4 (`__NR_write` is set to 4 in `include/asm/unistd.h`), can be inserted into the sliding window for this process, the new window can then be compared to the definition of normal. The return value from the lexical analyzer will be placed in the variable `ret` (normal or anomalous). At this point, before `orig_sys_write` is called, modifications can be made to the metric quantifying normal behavior depending on the result. Furthermore, if this "slide" of the window causes a threshold T to be exceeded based on the metric being used, the process can be terminated before the system call is carried out.

This approach is different from the implementations presented in [3] and [12] because it wraps the system calls individually, and not the *interface*. Wrapping the interface requires assembly modifications to the system call dispatcher `system_call`. Making modifications at this level requires that close attention be paid to branches– an ill placed branch in the dispatcher could introduce delays in the instruction pipeline. Furthermore, modifying the dispatcher ties the implementation to the i386 architecture.

6.4 Performance Impact

This section explores the performance impact of the system call wrappers introduced in previous Section 6.3. To determine the impact, the `find` command was run on the `/usr` file system on the RedHat 6.1 machine mentioned in Chapter 5. This test was done ten times with no monitoring done, and ten times each with window sizes 4, 6, and 11. The results are recorded in Table 6.1. The user column indicates how much time was spent in user-space. The system column indicates how much time was spent in kernel-space, and the elapsed time indicates the total elapsed time for the find command. All of the numeric values represent the average over the ten runs. The top row was run using an unmodified 2.4.0 kernel. Row 2 was run using a 2.4.0 kernel with the modifications presented in Section 6.3. Row 3 was run using an unmodified 2.2.16 kernel. Rows 4 and 5 were run using kernel 2.2.16, with pH installed and turned off and on respectively.

The results presented in Table 6.1 suggest that modifying the system call dispatcher, as in the pH implementation, incurs a slightly higher performance penalty

Table 6.1. Averaged user, system, and elapsed times for `find`.

run	kernel version	user	system	elapsed
No IDS	2.4.0	0.28	1.51	1.79
Window Size 11	2.4.0	0.28	1.52	1.80
no IDS	2.2.16	0.31	0.73	1.04
pH off	2.2.16	0.31	0.74	1.04
pH on	2.2.16	0.28	0.96	1.23

than wrapping system call individually. However, caution must be taken when drawing conclusions from these results. Since there are two different kernels being used, and their corresponding times are not the same without an IDS present, it is difficult to determine how much of an impact the difference between the kernels has on the these results.

CHAPTER 7

CONCLUSIONS

The analysis of the different sliding window implementations was intended to fill the gaps in the current research regarding the practical implications of using a forward or backward tabular representation vs. the tree representation. In summary, the observations presented indicate that the forward implementation takes slightly longer to recognize an anomalous system call due to the increasing variability in the last column of the compressed tabular representation. Furthermore, it appears that the backward tabular implementation detects anomalous system calls just as early as the tree implementation. However, at the expense of increased storage, the tree implementation is slightly more effective at detecting anomalies over the tabular implementations. When storage is not an issue, as in off-line analysis, it is clear that the tree implementation is the preferred choice. However, if storage is a concern, as in a kernel-based implementation, the backward tabular implementation is preferred due to its compressed representation of normal behavior.

The observations presented in this thesis give enough leading evidence to indicate that the event counter metric is at least as effective at detecting intrusions as the Hamming distance metric, and less sensitive to training and window size. It is more conducive to implementation within the kernel because it is faster and does not require the additional storage needed to calculate the Hamming distance. It is more more flexible with respect to the threshold, T, because the signal to noise ratio is much greater than the $\hat{S_A}$ metric. Additionally, the event counter metric also performs well with small window sizes. Therefore, memory requirements can be reduced. The event counter allows for dynamic fine tuning of the increment to reflect changes in the environment without requiring retraining of the IDS. Finally, this thesis provides enough leading evidence to indicate that the event counter approach, in combination with the sliding window characterization of normal behavior, provides a means for more rigorously generating an intrusion detection system based on noise analysis.

7.1 Future Work

The event counter metric and the kernel-based IDS framework presented in this thesis provide an infrastructure for enhancing this field of intrusion detection. The following sections present ideas that can be used for future investigation.

7.1.1 Reducing the Alphabet

Including the entire system call alphabet of an application may be excessive. It may be possible to maintain the same level of detection while reducing the size of the alphabet. For example, if an application's trace output during legitimate behavior was composed of the `write` system call evenly distributed half of the time, then ignoring the `write` system call would effectively double the window size and half the overhead incurred by the DFA in kernel-space.

Of course, it is unlikely that any system call would be predictably distributed all of the time. It is also unclear which system calls would be acceptable to remove from the alphabet without effecting the detection capabilities. Therefore, a significant amount of empirical analysis may be required to determine if the same level of detection would be possible.

7.1.2 Filtering Based on Alphabet

Most of the buffer overflow exploits for UNIX applications today involve the replacement of the instruction pointer (on the stack) with a pointer to a specially crafted sequence of assembly instructions, called shell code, that exec's a common UNIX shell while the application still has root privileges. Once the shell code begins execution an abrupt change takes place in the system calls being used. This abrupt change corresponds to a sharp increase in anomalies. This abrupt change is normally initiated by a conspicuous `execve` system call. The `traceroute` exploit presented in this thesis is a example of this behavior. In the case of `traceroute`, the `execve` system call is not in the alphabet of normal behavior (except for the first system call of the trace). By determining the alphabet empirically, and rejecting system calls not in the alphabet, it may be possible to prevent these types of intrusions before they begin.

Furthermore, static analysis of the application could be used to determine the alphabet of system calls. This can give a higher degree of confidence in rejecting system calls to prevent intrusions.

7.1.2.1 Specialized Selection of Increment

It may be possible to determine the system calls that frequently precede successful intrusions and assign different increments to these system calls in order to increase the signal of an intrusion. Clearly, many of the buffer overflows begin intrusions with the `execve` system call. It is conceivable that other system calls mark the beginning of an intrusion as well.

7.1.3 Adding an Exploit DFA to Increase Accuracy

Although misuse detection is more complicated and requires a constant update of intrusion signatures, it is conceivable that the addition of an DFA containing sequences of system calls found in known exploits could compliment the detection capabilities of an anomaly based intrusion detection system [5]. By combining such a DFA with the event counter metric, it may be possible to exaggerate the increment in order to amplify the anomaly signal.

7.1.4 Hybrid Anomaly Detection DFA and Specification Language

In Section 2.1.3, the use of a specification language to track anomalies in system call traces was described. This technique defines an acceptable domain of operation for an application and any activity outside this domain is considered anomalous. The behavior-based system call techniques (sliding window implementations) could be used to help ensure normal behavior is maintained within the domain. Conversely, the specification language approach could be used to augment a sliding window implementation. If a sequence is rejected by the sliding window implementation and also rejected by the specification language, the IDS may be able to make a better informed decision about the anomaly or modify the increment to reflect this information.

REFERENCES

[1] Acharya, A., and Raje, M. Mapbox: Using parameterized behavior classes to confine applications. Tech. Rep. TRCS99-15, University of California Santa Barbara, May 1999. http://www.cs.ucsb.edu/TRs/Docs/TRCS99-15.ps.

[2] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[3] Bowen, T., Segal, M., and Sekar, R. On preventing intrusions by process behavior monitoring. In *Proceedings of the Workshop on Intrusion Detection and Network Monitoring* (April 1999).

[4] Bugtraq vulnerability database statistics, March 2001. http://www.securityfocus.com/frames/?content=/vdb/stats.html.

[5] Denning, D. E. An intrusion detection model. *IEEE Transactions on Software Engineering SE-13*, 2 (February 1987), 222–232.

[6] fish stiqz fish@analog.org. Redhat linux restore insecure environment variables vulnerability, November 2000. BUGTRAQ id 1914: http://www.securityfocus.com/bid/1914.

[7] Forrest, S., Hofmeyr, S., Somayaji, A., and Longstaff, T. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy* (1996), IEEE Computer Society Press, pp. 120–128.

[8] Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. A secure environment for untrusted helper applications — confining the wily hacker. In *In Proceedings of the 1996 USENIX Security Symposium* (1996).

[9] Grimmett, G. R., and Stirzaker, D. R. *Probability and Random Processes.* Oxford University Press, 1989.

[10] guidob@mainnet.nl, G. B. Multiple vendor locale subsystem format string vulnerability, September 2000. BUGTRAQ id 1634: http://www.securityfocus.com/bid/1634.

[11] Hofmeyr, S., Forrest, S., and Somayaji, A. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6 (1998), 151–180.

[12] Hofmeyr, S., Forrest, S., and Somayaji, A. Automated response using system-call delays. In *Usenix Security Syposium 2000, submitted.* (2000).

[13] Ko, C., Fink, G., and Levitt, K. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *In Proceedings of the 10th Annual Computer Security Applications Conference* (December 1994), pp. 134–144.

[14] Loscocco, P., Smalley, S., P. Muckelbauer, R. T., and S. Turner, J. F. The inevitability of failure: The flawed assumption of security in modern computing environments. Tech. rep., National Security Agency, 1998.

[15] NSA. Security-enhanced linux. http://www.nsa.gov/selinux/. The National Security Agency.

[16] Ousterhout, J. K., Levy, J. Y., and Welch, B. B. The safe-tcl security model., 1997. Sun Microsystems Laboratories http://www.scriptics.com/people/john.ousterhout/safeTcl.ps.

[17] Pekka Savola, p. Bnl traceroute heap corruption vulnerability, September 2000. BUGTRAQ id 1739: http://www.securityfocus.com/bid/1739.

[18] Proceedings of the DARPA/NSA Workshop on Operating System Security. *DARPA/NSA/DISA Joint Technology Office: Research Challenges in Operating System Security* (May 1996).

[19] Rehmeyer, J. User documentation for the stide software package. http://www.cs.unm.edu/ immsec/software/, April 1998.

[20] Saltzer, J. Protection and the control of information sharing in multics. *Communications of the ACM 17*, 7 (July 1974), 388–402.

[21] Warrender, C., Forrest, S., and Pearlmutter, B. A. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy* (1999), pp. 133–145.

[22] Xia, H., and Biondi, P. Linux intrusion detection system (lids), 2000. http://www.lids.org/.

BIOGRAPHICAL SKETCH

Damon Snyder

Damon Snyder graduated from Florida State University in 1999 with a B.S degree in Computer Science and immediately began work on his M.S. degree in Computer Science. During this time, he also worked as a system administrator for the Computer Science Department. Since 2000 he has been a research assistant investigating computer security and a part time system administrator.