Intrusion Detection using Sequences of System Calls

Steven A. Hofmeyr
Stephanie Forrest
Anil Somayaji

Dept. of Computer Science
University of New Mexico
Albuquerque, NM 87131-1386
{steveah,forrest,soma}@cs.unm.edu

Revision: 12/17/97

A method is introduced for detecting intrusions at the level of privileged processes. Evidence is given that short sequences of system calls executed by running programs are a good discriminator between normal and abnormal operating characteristics of several common UNIX programs. Normal behavior is collected in two ways: Synthetically, by exercising as many normal modes of usage of a program as possible, and in a live user environment by tracing the actual execution of the program. In the former case several types of intrusive behavior were studied; in the latter case, we analyze results were analyzed for false positives.

1. Introduction

Modern computer systems are plagued by security vulnerabilities. Whether it is the latest UNIX buffer overflow or bug in Microsoft Internet Explorer, our applications and operating systems are full of security flaws on many levels. From the viewpoint of the traditional security paradigm, it should be possible to eliminate such problems through more extensive use of formal methods and better software engineering. This view rests on several assumptions: That security policy can be explicitly and correctly specified, that programs can be correctly implemented, and that systems can be correctly configured. Although these assumptions may be theoretically reasonable, in practice none of them holds. Computers systems are not static: They are continually changed by vendors, system administrators, and users. Programs are added and removed, and configurations are changed. Formal verification of a statically defined system is time-consuming and hard to do correctly; formal verification of a dynamic system is impractical. Without formal verifications, tools such as encryption, access controls, firewalls, and audit trails all become fallible, making perfect implementation of a security policy impossible, even if a correct policy could be devised in the first place. If we accept that our security policies, our implementations, and our configurations are flawed in practice, then we must also accept that we will have imperfect security. We can incrementally improve security through the use of tools such as Intrusion Detection Systems (IDS). The IDS approach to security is based on the assumption that a system will not be secure, but that violations of security policy (intrusions) can be detected by monitoring and analyzing system behavior.

There are many different levels on which an IDS can monitor system behavior. It is critical to profile normal behavior at a level that is both robust to variations in normal and perturbed by intrusions. In the work reported here, we chose to monitor behavior at the level of *privileged processes*. Privileged processes are running programs that perform services (such as sending or receiving mail), which require access to system resources that are inaccessible to the ordinary user. To enable these processes to perform their jobs, they are given privileges over and above those of an ordinary user (even though they can be invoked by ordinary users). In UNIX, processes usually run with the privileges of the user that invoked them. However, privileged processes can change their privileges to that of the superuser by means of the setuid mechanism. One of the security problems with privileged processes in UNIX is that the granularity of permissions is too coarse: Privileged processes need superuser status to access system resources, but granting them such status gives them more permission than necessary to perform their specific tasks [30]. Consequently, they have permission to access *all* system resources, not just those that are relevant to their operation. Privileged processes are trusted to access only relevant system resources, but in cases where there is some programming error in the code that the privileged process is running, or if the privileged process is incorrectly configured, an ordinary user may be able to gain superuser privileges by exploiting the problem in the program. For the sake of brevity, we usually refer to privileged processes (or programs) simply as "processes" (or "programs"), and use the qualifier only to resolve ambiguities.

It is clear that privileged processes are a good level to focus on because exploitation of vulnerabilities in privileged processes can give an intruder super-user status. Furthermore, privileged processes constitute a natural boundary for a computer, especially processes that listen to a particular port. In UNIX, privileged programs, such as `telnetd` and `logind`, function as servers that control access into the system. Corruption of these servers can allow an intruder to access the system remotely. Monitoring privileged processes also offers some advantages over monitoring user behavior, which has been the most common method to date (for example, see [5],[12],[26],[35],[39]). The range of behaviors of privileged processes is limited compared to the range of behaviors of users; privileged processes usually perform a specific, limited function, whereas users can carry out a wide variety of actions. Finally, the behavior of privileged processes is relatively stable over time, especially compared to user behavior. Not only do users perform a wider variety of actions, but the actions performed may change considerably over time, whereas the actions (or at least the functions) of privileged processes usually do not vary much with time.

Our approach to detecting irregularities in the behavior of privileged programs is to regard the program as a black box, which, when run, emits some *observable*. We believe that this observable should be a dynamic characteristic of that program; although code stored on disk may have the potential to do harm, it has to be actually running to realize that potential. If we regard the program as a black-box, we do not need specialized knowledge of the internal functioning or the intended role of the program; we can infer these indirectly by observing its normal behavior[1]. A natural observable for processes in UNIX would be based on *system calls*, because UNIX processes accesses system resources through the use of system calls. We have chosen *short sequences* of system calls as our observable.

In an earlier study we reported preliminary evidence that short sequences of system calls are a good simple discriminator for several types of intrusions [18]. The results reported here extend the earlier study, with several important differences. First, we have slightly changed how we record sequences of system calls: Previously, we used look-ahead pairs, with a look-ahead value of 6; here we use exact sequences of length 10. Consequently, the database sizes reported here are smaller than in the earlier study. Next, we have used a measure of anomalous behavior that is independent of trace length (based on Hamming matches between sequences). Finally, we have collected normal behavior in a real, live[2] environment, and analyzed it for false positives.

We want an IDS that is stable and lightweight (efficient), all of which depends on the discriminator (observable) that we use to distinguish between acceptable and unacceptable behavior. By stable we mean that the discriminator reliably distinguishes between acceptable and unacceptable behavior. Our approach is experimental because we believe that current theories do not adequately describe how implemented systems really

---

[1] There are other approaches that require knowledge of the internals and intended role of a program, most notably the program specification method [30], which attempts to constrain the process in such a way that it can perform only those operations that the program is designed to do, and no more, i.e. the method refines the permissions structure to accommodate specific privileged programs. The differences between our method and this are discussed more fully in section 6.

[2] We use the words "real" and "live" to refer to a production environment, i.e. an environment which is currently in normal, everyday use. We contrast this to our "synthetic" environment, which is an isolated test environment.

run. In this paper we are primarily concerned with determining empirically if the discriminator is stable. Efficiency is a secondary consideration, and is addressed in this paper to the extent that we analyze the complexity of our algorithm; however, we do not report actual running times for the method on a production system.

Our work is inspired by the defenses of natural immune systems. There are compelling similarities between the problems faced by immune systems and by computer security [17]. Both systems must protect a highly complex system from penetration by inimical agents; to do this, they must be able to discriminate between broad ranges of normal and abnormal behavior. In the immune system, this discrimination task is known as the problem of distinguishing "self" (the harmless molecules normally within the body) from "nonself" (dangerous pathogens and other foreign materials). Discrimination in the immune system is based on a characteristic structure called a peptide (a short protein fragment) that is both compact and universal in the body. This limits the effectiveness of the immune system; for example, the immune system cannot protect the body against radiation. However, proteins are a component of all living matter, and generally differ between self and nonself, so they provide a good distinguishing characteristic. We view our chosen discriminator (short sequences of system calls) as analogous to a peptide.

The structure of this paper is as follows. In section 2 we review related work in intrusion detection. Section 3 describes our method of anomaly intrusion detection: First we describe how to build up profiles of normal program behavior, and then we define three ways of detecting anomalies. We then use the method to build a synthetic normal profile in section 4, demonstrating its effectiveness at detecting intrusions and other anomalies. In section 5 we consider the consequences of collecting our normal data in online, functioning environments, discuss false positives, and present experimental results on false positive rates. The limitations and implications of our approach are discussed in section 6. A brief appendix is included which details the various intrusions that we used in our experiments, the methods we used to generate synthetic normal, and a brief overview of UNIX.

2. Related Work

An Intrusion Detection System (IDS) continuously monitors some dynamic behavioral characteristics of a computer system to determine if an intrusion has occurred. This definition excludes many useful computer security methods. Security analysis tools, such as SATAN [16] and COPS [15] are used to scan a system for weaknesses and possible security holes. They are not IDS because they do not monitor some dynamic characteristic of the system for intrusions or evidence of intrusions, rather they scan the system for weaknesses such as configuration errors or poor password choices that could *lead* to intrusions. Other important non-IDS solutions to computer security problems are provided by cryptography [13], which is especially useful for authentication and secure communications [36]. Virus protection schemes such as that described in [28] are also not IDS under our definition, because they scan static code, not dynamic behavioral characteristics. Some approaches are not easily classified, for example, integrity checking systems such as TRIPWIRE [29] monitor important files for changes that could indicate intrusions. Although such files are static code, they become a dynamic characteristic indicative of intrusions when modified by intrusive activities, and so TRIPWIRE could be classified as an IDS.

There are many different architectures for IDS. IDS can be centralized (i.e. processing is performed on a single machine) or distributed across many machines. Almost all IDS are centralized; the autonomous agents approach [10] is one of the few proposed IDS that is truly distributed. Furthermore, an IDS can be host-based or network-based; the former type monitors activity on a single computer, whereas the latter type monitors activity over a network. Network-based IDS can monitor information collated from audit trails from many different hosts (multi-host monitoring) or they can monitor network traffic. NADIR [26] and DIDs [25] are examples of IDS that do both multi-host and network traffic monitoring; NSM [24] is an IDS that monitors only network traffic. Regardless of other architectural considerations, any IDS must have three components: Data collection (and reduction), data classification and data reporting. Data reporting is usually very simple, with system administrators being informed of anomalous or intrusive behavior; few IDS take it upon themselves to act rapidly to deal with irregularities. Various methods for data collection and classification are discussed below.

An IDS that monitors for intrusive behavior needs to collect data on the dynamic state of the system. Selecting a set of dynamic behavioral characteristics to monitor is a key design decision for an IDS, one which will influence the types of analyses that can be performed and the amount of data that will be collected. Most systems (for example, IDES/NIDES [34], [35], [5], Wisdom&Sense [33] and TIM [39]) collect profiles of user behavior, generated by audit logs. Other systems look at network traffic, for example, NSM and the system presented in [23]. Other approaches attempt to characterize the behavior of privileged processes, as in the program specification method [30]. Different behavioral characteristics will generate different amounts of data; as an extreme example, systems monitoring user profiles process large volumes of raw data (an average user will generate from 3 to 35MB of audit data per day [22]). In the latter case the data may need to be reduced to a manageable size.

Once a behavioral characteristic is selected, it is used to classify data. In the simplest case, this is a binary decision problem: The data is classified as either normal (acceptable) or anomalous (and possibly intrusive). Data classification can be more complex, for instance, trying to identify the particular type of intrusion associated with anomalous behavior. A plethora of methods have been used for data classification, the majority of them using artificial intelligence techniques (see [22] for a detailed overview). Classification techniques can be divided into two categories, depending on whether they look for known intrusion signatures (*misuse intrusion detection*), or for anomalous behavior (*anomaly intrusion detection*). Misuse-IDS encode intrusion signatures or scenarios and scan for occurrences of these, which requires prior knowledge of the nature of the intrusion. By contrast, in anomaly-IDS, it is assumed that the nature of the intrusion is unknown, but that the intrusion will result in behavior different from that normally seen in the system. Anomaly IDS use models of normal or expected behavior to monitor systems; deviations from the normal model indicate possible intrusions. Some systems incorporate both categories, a good example being NIDES, or Denning's generic model of an IDS [12].

Relatively few IDS deal with misuse intrusion detection. One type of implementation uses an expert system to fit data to known intrusion signatures, for example, in IDES/NIDES, or Stalker [37], knowledge of past intrusions is encoded by human experts in expert system rules. Other approaches attempt to generate intrusion signatures automatically, for example, one approach uses a pattern matching model based on colored Petri nets [31] and [32], while USTAT [27] represents potential intrusions as sequences of system states in the form of state transition diagrams.

Because of the difficulty of encoding known intrusions, and the continual occurrence of new intrusions, many systems focus on anomaly intrusion detection. A wide variety of methods have been used. TRIPWIRE monitors the state of special files (such as the `/etc./hosts.equiv` file on a UNIX system, or UNIX daemon binaries) for change; normal is simply the static MD5 checksum of a file. A program specification language is used in [30] to define normal for privileged programs in terms of the allowed operations for that program. Rule-based induction systems such as TIM have been used to generate temporal models of normal user behavior. Wisdom&Sense incorporates an unsupervised tree-learning algorithm to build models of patterns in user transactions. Other systems, such as NIDES, have employed statistical methods to generate models of normal user behavior in terms of frequency distributions. NSM uses a hierarchical model in combination with a statistical approach to determine network traffic usage profiles. On the biologically inspired side, connectionist or neural nets have been used to classify data [21], and genetic programming has been proposed as a means of developing classifications [10].

3. Anomaly Intrusion Detection

The method we present here performs anomaly intrusion detection (although it could also be used for misuse detection---see section 6). We build up a profile of normal behavior for a program of interest, treating deviations from this profile as anomalies. There are two stages to the anomaly detection: In the first stage we build up profiles or *databases* of normal behavior (this is analogous to the training phase for a learning system); in the second stage we use these databases to monitor process behavior for significant deviations from normal (analogous to the test phase).

Recall that we have chosen to define normal in terms of short sequences of system calls. In the interests of simplicity, we ignore the parameters passed to the system calls, and look only at their temporal orderings. This definition of normal behavior ignores many other important aspects of process behavior, such as timing information, instruction sequences between system calls, and interactions with other processes. Certain intrusions may only be detectable by examining these other aspects of process behavior, and so we may need to consider them later. Our philosophy is to see how far we can go with the simplest possible assumption.

3.1 Profiling Normal Behavior

The algorithm used to build the normal databases is extremely simple. We scan traces of system calls generated by a particular program, and build up a database of all unique sequences of a given length, k, that occurred during the trace. Each program of interest has a different database, which is specific to a particular architecture, software version and configuration, local administrative policies, and usage patterns. Once a stable database is constructed for a given program, the database can be used to monitor the ongoing behavior of the processes invoked by that program.

This method is complicated by the fact that in UNIX a program can invoke more than one process. Processes are created via the `fork` system call or its virtual variant `vfork`. The essential difference between the two is that a `fork` creates a new process which is an instance of the same program (i.e. a copy), whereas a `vfork` replaces the existing process with a new one, without changing the process ID. We trace forks individually and include their traces as part of normal, but we do not yet trace virtual forks because a virtual fork executes a new program. In the future, we will switch databases dynamically to follow the virtual fork.

Given the large variability in how individual systems are currently configured, patched, and used, we conjecture that individual databases will provide a unique definition of normal for most systems. We believe that such uniqueness, and the resulting diversity of systems, is an important feature of the immune system, increasing the robustness of populations to infectious diseases [20]. The immune system of each individual is vulnerable to different pathogens, greatly limiting the spread of a disease across a population. Traditionally, computer systems have been biased towards increased uniformity because of the advantages offered, such as portability and maintainability. However, all the advantages of uniformity become potential weaknesses when errors can be exploited by an attacker. Once a method is discovered for penetrating the security of one computer, all computers with the same configuration become similarly vulnerable.

The construction of the normal database is best illustrated with an example. Suppose we observe the following trace of system calls (excluding parameters):

open, read, mmap, mmap, open, read, mmap

We slide a window of size k across the trace, recording each unique sequence of length k that is encountered. For example, if $k = 3$, then we get the unique sequences:

open, read, mmap
read, mmap, mmap
mmap, mmap, open
mmap, open, read

For efficiency, these sequences are currently stored as trees, with each tree rooted at a particular system call. The set of trees corresponding to our example is given in Figure 1.

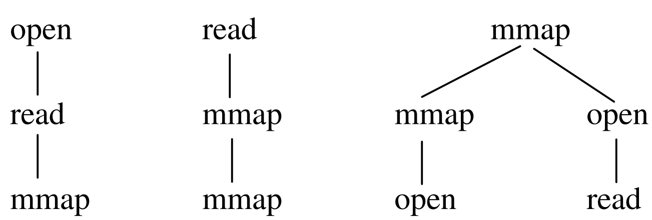

Figure 1. An example of a forest of system call sequence trees.

We record the size of the database in terms of the number of unique sequences N, (in the example just given, $N = 4$) so an upper bound on the storage requirements for the normal database is $O(Nk)$. In practice, the storage requirements are much lower because the sequences are stored as trees. For example, the sendmail database, which contains 1318 unique sequences of length 10, has 7578 nodes in the forest, where each node corresponds to a system call. If we had a node for every single system call in all 1318 sequences, we would have 13180 nodes.

3.2 Measuring Anomalous Behavior

Once we have a database of normal behavior, we use the same method that we used to generate the database to check new traces of behavior. We look at all overlapping sequences of length k in the new trace and determine if they are represented in the normal database. Sequences that do not occur in the normal database are considered to be *mismatches*. By recording the number of mismatches, we can determine the strength of an anomalous signal. Thus the number of mismatches occurring in a new trace is the simplest determinant of anomalous behavior. We report these counts both as a raw number and as a percentage of the total number of matches performed in the trace, which reflects the length of the trace. Ideally, we would like these numbers to be zero for new examples of normal behavior, and for them to jump significantly when abnormalities occur.

We make a clear distinction here between *normal* and *legal* behavior. In the ideal case we want the normal database to contain all variations in normal behavior, but we do not want it to contain every single possible path of legal behavior, because our approach is based upon the assumption that normal behavior forms only a subset of the possible legal execution paths through a program, and unusual behavior that deviates from those normal paths signifies an intrusion or some other undesirable condition. We want to be able to detect not only intrusions, but also unusual conditions that are indicative of system problems. For example, when a process runs out of disk space, it may execute some error code that results in an unusual execution sequence (path through the program). Clearly such a path is *legal*, but certainly it should not be regarded as *normal*.

If the normal database does contain all variations in normal behavior, then when we encounter a sequence that is not present in the normal database, we can regard it as anomalous, i.e. we can consider a single mismatch to be significant. In reality, it is likely to be impossible to collect all normal variations in behavior (these issues are discussed more fully in sections 4 and 5), so we must face the possibility that a normal database provides incomplete coverage of normal behavior. One solution is to count the number of mismatches occurring in a trace, and only regard as anomalous those traces that produce more than a certain number of mismatches. This is problematic however, because the count is dependent on trace length, which might be indefinite for continuously running processes.

An alternative is to constrain the measure locally. The anomalies we have studied are temporally clumped: Anomalous sequences due to intrusions seem to occur in local bursts. However, defining a local measure is difficult because we have an unordered state space, i.e. we have no true notion of locality---how "close" one system call is to another, or how "close" one system call sequence is to another. We have chosen "Hamming distance"[3] between sequences as the measure. Although this choice is somewhat arbitrary, it is related to how closely anomalies are clumped. We cannot theoretically justify this measure, so we determine its worth empirically.

We use the "Hamming distance" between two sequences to compute how much a new sequence actually differs from existing normal sequences. The similarity between two sequences can be computed using a matching rule that determines how the two sequences are compared. The matching rule used here is based on Hamming distance, i.e. the difference between two sequences *i* and *j* is indicated by the Hamming distance $d(i, j)$ between them. For each new sequence *i*, we determine the *minimal* Hamming distance $d_{\min}(i)$ between it and the set of normal sequences:

$$d_{\min}(i) = \min\{d(i, j) \text{ for all normal sequences } j\}.$$

The $d_{\min}$ value represents the strength of the anomalous signal, i.e. how much it deviates from a known pattern. Note that this measure is not dependent on trace length and is still amenable to the use of thresholds for binary decision making.

The various measures can be illustrated with a small example. Again, consider the trace shown in the previous example:

open, read, mmap, mmap, open, read, mmap

that generated the normal database consisting of:

open, read, mmap
read, mmap, mmap
mmap, mmap, open
mmap, open, read

Now, if we have a trace in which one call (the sixth in the trace) is changed from read to mmap:

open, read, mmap, mmap, open, *mmap*, mmap

[3] Although we are not using a binary alphabet, the measure we use is analogous to a binary Hamming distance, i.e. it is the number of positions in which the two sequences differ.

then we will have the following new sequences:

> mmap, open, *mmap*
> open, *mmap*, mmap

This corresponds to 2 mismatches, which is 40% of the trace, and two $d_{\min}$ values of 1.

These three different measures have different time-complexities. To determine that a new sequence is a mismatch requires at most $k-1$ comparisons, because the normal sequences are stored in a forest of trees, where the root of each tree corresponds to a different system call. Similarly, it will take $k-1$ comparisons to confirm that a sequence is actually in the normal database. If the sequence is not in the normal database, then computing $d_{\min}$ for that sequence is much more expensive. Because $d_{\min}(i)$ is the smallest Hamming distance between *i* and all normal sequences, we have to check every single sequence in normal before we can determine $d_{\min}(i)$, which will require a total of $N(k-1)$ comparisons (recall that N is the number of sequences in the database). However, we expect anomalies to be rare, so most of the time, the algorithm will be confirming normal sequences, which is much cheaper to do. If our rate of anomalous to normal sequences is R_A, then the average complexity of computing $d_{\min}(i)$ per sequence is $N(k-1)R_A + (k-1)(1-R_A)$, which is $O(k(R_A N + 1))$.

3.3 Classification Errors

An IDS using these measures will be making decisions based on the observed values of the measures. In the simplest case, these are binary decisions: Either a sequence is anomalous, or it is normal. With binary decision making, there are two types of classification errors: False positives and false negatives. We define these errors asymmetrically: A *false positive* occurs when a single sequence generated by legitimate behavior is classified as anomalous; and 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. In statistical decision theory, false negatives and false positives are called type I and type II errors, respectively.

To detect an intrusion, *at least one* of the sequences generated by the intrusion must be classified as anomalous. In terms of our measures, what we require is that at least one of the sequences generated by the intrusion has $d_{\min} > 0$. We measure the strength of the anomaly by $d_{\min}$, and because we want intrusions to generate strong anomalies, we assume that the higher the $d_{\min}$ the more likely it is that the sequence was actually generated by an intrusion. In practice, we report the maximum $d_{\min}$ value that was encountered during a trace, because that represents the strongest anomalous signal found in the trace, i.e. we compute the signal of the anomaly, S_A, as:

$$S_A = \max\left\{d_{\min}(i) \text{ for all new sequences } i\right\}.$$

In our example above, $S_A = 1$. Generally, we do not report the actual S_A value, but rather the S_A value normalized over the sequence length *k*, to enable us to compare S_A values for different values of *k*, i.e.:

$$\hat{S}_A = S_A / k \,.$$

Although we would like to minimize both kinds of errors, we are more willing to tolerate false negatives than false positives. False negatives can be reduced by adding layers of defense, whereas layering will not reduce overall false positive rates. A simple example illustrates this. Consider a system with *L* layers of defense that an intruder must penetrate, where at each layer there is a probability p_n that the intruder will escape detection (i.e. p_n is the false negative rate). If the probability of detection is independent for each layer, then the probability that the intruder will penetrate all layers undetected is ${p_n}^L$. So, in this example, the overall false negative rate is exponentially reduced by adding layers of protection (provided we have independence). By contrast, if we assume that at each layer we have an (independent) probability p_f of generating a false positive, then the expected number of false positives is $p_f L$. In this case layering compounds false positives.

False positives can be measured when we collect normal behavior in live environments (see section 5). If we are collecting normal empirically, the occurrence of rare but acceptable events could result in an incomplete normal database. If the normal were incomplete, false positives could be the result, as we encounter acceptable sequences that are not yet included in our normal database. To limit false positives, we set thresholds on the $d_{\min}(i)$ values, i.e. we regard as anomalous any sequence *i* for which

$$d_{\min}(i) \geq C \,.$$

where $1 \leq C \leq k$ is the threshold value. To summarize, if a sequence *i* of length *k* is sufficiently different from all normal sequences it is flagged as anomalous. The validity of the assumption that intrusive behavior is characterized by increased Hamming distance from normal sequences is tested empirically in the sections that follow.

4. Behavior in a Synthetic Environment

There are two methods for choosing the normal behavior that is used to define the normal database: (1) We can generate a "synthetic" normal by exercising a program in as many normal modes as possible and tracing its behavior; (2) we can generate a "real" normal by tracing the normal behavior of a program in a live user environment. A synthetic normal is useful for replicating results, comparing performance in different settings, and other kinds of controlled experiments. Real normal is more problematic to collect and evaluate (these issues are discussed in section 5); however, we need real normal to determine how our system is likely to perform in realistic settings. For example, if we generate normal synthetically we have no idea what false positive rates we will get in realistic settings because our synthetic, by definition, includes all variations on normal behavior (although not all variations on legal behavior). We could exclude some synthetically generated traces from normal and see what false positives resulted, but it is not clear which traces to exclude---the choice is arbitrary and the resulting false positives would be equally arbitrary. In this section we present results using a synthetic normal; in section 5 we present results using a real normal.

4.1 Building a Synthetic Normal Database

We studied normal behavior for three different programs in UNIX: `sendmail`, `lpr` and `wu.ftpd` (the first two were running under SunOS 4.1.x, and the last one was running under Linux). `Sendmail` is a program that sends and receives mail, `lpr` is a program that enables users to print documents on a printer, and `ftpd` is a program for the transfer of files between local and remote hosts. Because `sendmail` is the most complex of these programs, we will briefly describe how we exercised `sendmail` to produce a profile of normal behavior (the methods for constructing synthetic normal for the other two programs are described in Appendix 2). We considered variations in message length, number of messages, message content (text, binary, encoded, encrypted), message subject line, who sent the mail, who received the mail, and mailers. In addition, we looked at the effects of forwarding, bounced mail and queuing. Lastly, we considered the effects of the origin of all these variations in the cases of remote and local delivery.

A suite of 112 artificially constructed messages was used to exercise `sendmail` (version 5), producing a trace of a combined length of over 1.5 million system calls. Table 1 shows how many messages of each type were used to generate the normal databases. We began with message length, testing 12 different message lengths, ranging from 1 line to 300,000 bytes. From this, we selected the shortest length that produced the most varied pattern of system calls (50,000 bytes), and then used that as the standard message length for the remaining test messages. Similarly, with the number of messages in a `sendmail` run, we first sent 1 message and traced `sendmail`, then we sent 5 messages, tracing `sendmail`, and so forth, up to 20 messages. This was intended to test the response of `sendmail` to bursts of messages. We tested message content by sending messages containing ASCII text, uuencoded data, gzipped data, and a pgp encrypted file. In each case, a number of variations was tested and the one that generated the most variations in system call patterns was selected as a single default before moving on to the next stage.

These messages constituted our corpus of normal behavior. We reran this set of standard messages on each different operating system and `sendmail.cf` (the `sendmail` configuration file) variant that we tried, thus generating a normal database that was tailored to the exact operating conditions under which `sendmail` was running.

Of the features considered, the following seemed to have little or no effect: Number of messages, message content, subject line, who sent the mail, who received the mail, mail programs and queuing. Message length has a considerably different effect on the sequence of system calls, depending on the message origin: Remote mail produces traces of system calls that are proportional to the length of the message, with little sequence variation in these traces; local mail produces traces that are roughly the same length, regardless of the size of message, but the sequence of system calls used changes considerably as the message size increases. In both cases, once a large enough message size (50K) is used to generate normal, message size makes no difference. The effect of forwarding mail on remote traces is negligible, whereas it has a small but noticeable affect on local traces. Bounced mail had more of an effect remotely, but the effects are still evident in the local case.

Type of Behavior	# of Mail Messages
message length	12
number of messages	70
message content	6
subject	2
sender/receiver	4
different mailers	4
forwarding	4
bounced mail	4
queuing	4
vacation	2
Total	112

Table 1. Number of messages of each type used to generate synthetic `sendmail` normal. Each number in the table indicates the number of variants used, for example, we used 12 different message lengths.

For each test, we generated databases for different values of k for each of the three programs tested, i.e. `sendmail`, `lpr` and `ftpd`. The results for $k = 10$ are shown in Table 2. Our choice of sequence length was determined by two conflicting criteria. On the one hand we want a sequence length as short as possible to minimize the size of the database and the computation involved in detection (recall that the time complexity of detection is proportional to k). On the other hand, if the sequence length is too small we will not be able to discriminate between normal and anomalous behavior. Our choice of 10 is based on empirical observations (see section 4.2.2).

Program	Database Size *N*
sendmail	1318
lpr	198
ftpd	1017

Table 2. Synthetic normal database size *N* for sequence length of 10, for `sendmail`, `lpr` and `ftpd`.

These databases are remarkably compact, for example, the `sendmail` database contains only 1318 unique sequences of length 10, which requires 9085 bytes to store in our current implementation. `sendmail` is one of the most complex of the privileged programs currently used in UNIX systems, and if its behavior can be described so compactly, then we can expect that other privileged programs will have normals at least as compact. The data are encouraging because they indicate that the range of normal behavior of these programs is limited. Too much variability in normal would preclude detecting anomalies; in the worst case, if all possible sequences of length *k* show up as legal normal behavior, then no anomalies could ever be detected.

How many possible sequences of length *k* are there? If we have an alphabet Σ of system calls, with size $|\Sigma|$, then there are $|\Sigma|^k$ possible sequences of length *k*. Choosing the alphabet size can be problematic without knowing exactly which system calls are used by `sendmail`, considering that there are a total of 182 system calls in the SunOS 4.1.x operating system. As a conservative estimate, we assume that `sendmail` uses no more than 53 calls (the number in the synthetic normal database), so, for $k = 10$ there are 53^{10}, or approximately 10^{17} possible sequences. Thus our `sendmail` normal database only contains about 10^{-13} percent of the total possible number of patterns. Of course, this is not completely accurate, because the number of possible sequences that `sendmail` can actually use is limited by the structure of the code. To determine this would require a detailed analysis of the source code, which is precisely what we wish to avoid because one of the strengths of our approach is that it does not require specialized knowledge of any particular program.

4.2 Detecting Anomalous Behavior

Is intrusive behavior anomalous under this definition of normal? Ideally, we want most, if not all, intrusive behavior to be anomalous. To test this, we compared the normal databases against a range of known abnormal behavior.

In these experiments, we report the number of mismatches, the percentage of mismatches, and the normalized anomaly signal $\hat{S}_A$. Because $\hat{S}_A$ is not dependent on the length of the trace, it is our preferred measure. However, $\hat{S}_A$ values are meaningful only in the context of detection thresholds, and thresholds are dependent on the acceptable level of false positives. Because of the way we constructed normal, we have zero false positives for synthetic data; thus, in principle, any $\hat{S}_A > 0$ indicates an anomaly (although our goal is clear separation between the anomaly and normal, i.e. we want the $\hat{S}_A$ values to be large). The issue of false positives in a real environment is explored in section 5.

4.2.1 Distinguishing Between Programs

The first experiments we performed compared `sendmail` with other UNIX programs. If we could not distinguish between `sendmail` and other programs, then we would be unlikely to detect small deviations in the behavior of a single program. We have done this comparison for varying sequence lengths. When the sequence length is very low, ($k = 1$), there are very few mismatches, in the range of 0 to 7%. When the sequence length reaches $k = 30$ there are 100% mismatches against all programs. Results of comparisons for $k = 10$ are presented in Table 3.

Each program showed a significant number of anomalous sequences (at least 57%), and at least one anomalous sequence is quite different from the normal `sendmail` sequences, as evinced by $\hat{S}_A$, which is at least 0.6, indicating that the most anomalous sequence differs from the normal sequences in over half of its positions. The programs shown are distinct from `sendmail` because the actions they perform are considerably different from those of `sendmail`. We also tested the normal database for `lpr` and achieved similar results (data not shown). `lpr` exhibits even more separation than that shown in Table 3, presumably because it is a smaller program with more limited behavior. These results suggest that the behavior of different programs is easily distinguishable using sequence information alone.

Program	Number Mismatches	% Mismatches	$\hat{S}_A$
ls	42	75	0.6
ls -l	134	91	1.0
ls -a	44	76	0.6
ps	539	97	0.6
ps -ux	1123	99	0.6
finger	67	83	0.6
ping	41	57	0.6
ftp	271	90	0.7
pine	430	77	1.0

Table 3. Distinguishing `sendmail` from other programs. Each column reports results for a single anomalous measures: Mismatches (column 2), percentage of mismatches over a trace (column 3), and $\hat{S}_A$ (column 4). The results shown are for a sequence length of $k = 10$. There are no mismatches against `sendmail` itself because the database includes all variations.

4.2.2 Detecting Intrusions

The second set of experiments was to detect intrusions that exploit flaws in three programs: `sendmail`, `lpr` and `wu.ftpd`. Some of the intrusions were successful, and others unsuccessful because of updates and patches in software. We report results for both. We would like to be able to detect most (if not all) of these attempted intrusions, even if they fail. Detection of failed intrusions would be a useful warning sign that an attacker is attempting to break into a system. A third behavioral category that we would like to be able to detect is the occurrence of error states, such as `sendmail` forwarding loops. Although these error states are technically legal behavior, they are properly regarded as abnormal because they indicate the existence of problems.

We compared system call traces for each of the three categories (successful exploits, unsuccessful exploits and error conditions) with the normal database for the relevant program and recorded the number of mismatches, percentage of mismatches over the trace, and $\hat{S}_A$ values. Table 4 shows results for successful intrusions. Each row in the table reports data for one typical trace. In most cases, we have conducted multiple runs of the intrusion with identical or nearly identical results; where runs differed significantly, we report a range of values. To date, we have collected data on five successful intrusions, three of them for `sendmail`, one for `lpr` [4] and one for `ftpd` [8] **CERT. swinstall vulnerability. Ftp://info.cert.org/pub/ cert_advisories/ CA-96:27, December 1996.**
[9]. The three `sendmail` intrusions were: `sunsendmailcp` [2], `syslogd` [3], [7], and a decode alias intrusion. These intrusions are described in the appendix. Most of the

successful intrusions are clearly detected, with $\hat{S}_A$ values of 0.5 to 0.7. The exception to this is the decode intrusion, which, on the low end of the range, generates only 7 mismatches and a $\hat{S}_A$ value of 0.2. These results suggest approximate detection thresholds that we would need in an online system to detect intrusions.

Anomaly	Number Mismatches	% Mismatches	$\hat{S}_A$
syslogd	248 - 529	17 - 30	0.7
sunsendmailcp	92	25	0.6
decode	7 - 22	1 - 2	0.2 - 0.5
lprcp	242	9	0.5
ftpd	496	38	0.7

Table 4. Detection of successful intrusions for `sendmail`, `lpr` and `ftpd`. The data for the `syslogd` intrusion show the results of tracing `sendmail` (rather than tracing `syslogd` itself). The three columns list the results for various anomalous measures, from mismatches, percentage of mismatches over a trace, to $\hat{S}_A$. In some cases the columns list a range of values, from minimum to maximum. The results are for $k = 10$.

The results for unsuccessful intrusions and error conditions are shown in Table 5. The unsuccessful intrusions are based on attack scripts called sm565a and sm5x. SunOS 4.1.4 has patches that prevent these particular intrusions. Overall, these unsuccessful intrusions are as clearly detectable as the successful intrusions. Error conditions are also detectable within a similar range of $\hat{S}_A$ values. As a clear case of undesirable errors, we have studied local forwarding loops in `sendmail` (see appendix for a description).

Anomaly	Number Mismatches	% Mismatches	$\hat{S}_A$
sm565a	54	22	0.6
sm5x	472	33	0.6
forward loop	21 - 108	10 - 18	0.4 - 0.6

Table 5. Detection of unsuccessful intrusions and error conditions for `sendmail`. The three columns list the results for various anomalous measures, from mismatches, percentage of mismatches over a trace, to $\hat{S}_A$. The results are for $k = 10$.

In summary, we are able to detect *all* the abnormal behaviors we tested against, including successful intrusions, failed intrusion attempts, and unusual error conditions.

We have only reported results for $k = 10$ because experiments show that varying sequence length has little effect on detection, in terms of the $\hat{S}_A$ measure. We analyzed sequences of length $k = 2$ to $k = 30$. The minimum sequence length used was 2, because

$k = 1$ will just give $\hat{S}_A = 0$ or $\hat{S}_A = 1$, which is not sufficiently informative. The maximum sequence length used was 30 because the cost of computation scales with sequence length. The results are reported in Figure 2. The decode intrusion is not detectable for $k < 6$, but beyond this value of *k*, sequence length seems to make little difference for $\hat{S}_A$. Sometimes an increased sequence length results in a decreased anomaly signal. This could happen if the anomalies consisted of short clumps of system calls separated by large gaps: As sequence length increases, longer sequences would be more similar to normal sequences. For example, say we had a normal sequence *open, read, mmap, mmap, open, read*, which an intrusion disrupted in the first three positions to give ***close, close, close***, *mmap, open, read*. Then $k = 3$ would give $\hat{S}_A = 3/3 = 1.0$ (from the first three system calls), and $k = 6$ would give $\hat{S}_A = 3/6 = 0.5$. Figure 2 implies that the best sequence length to use would be 6 or slightly larger than 6, because that will allow detection of anomalies while minimizing computation, which is directly proportional to *k*. We have chosen $k = 10$ because that gives a margin for error.

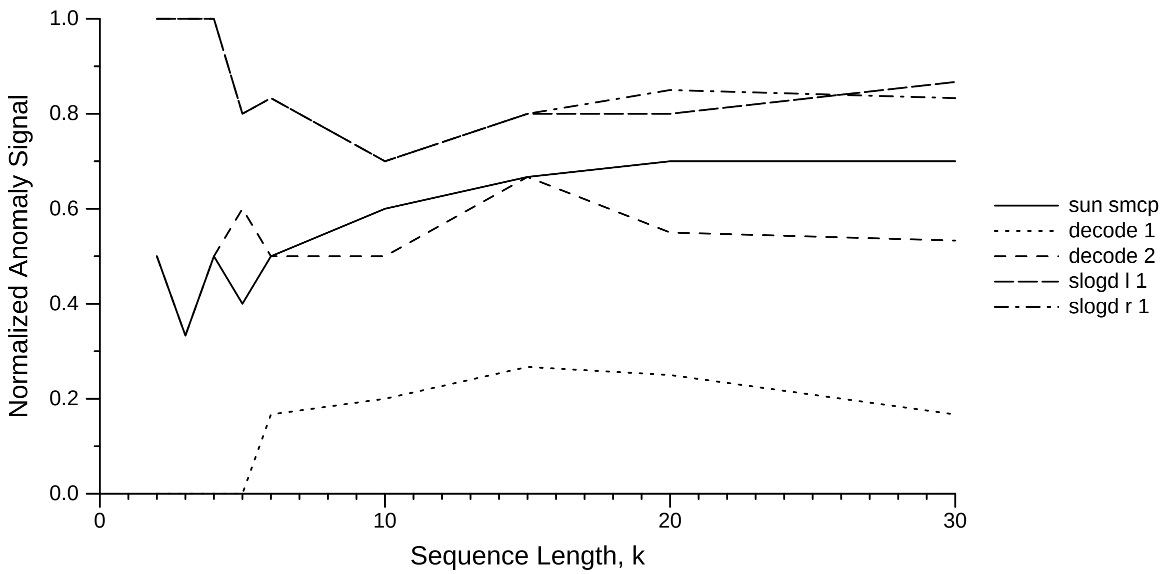

Figure 2. $\hat{S}_A$ **plotted against sequence length *k*. From this plot we infer that sequence length makes little difference once we have a length of at least 6.**

Considering only the three anomaly measures gives a limited picture of the sorts of perturbations caused by intrusions and other unacceptable behaviors. For example, the $\hat{S}_A$ values indicate only the most anomalous sequence without giving any clear idea of how anomalous sequences are temporally distributed. The anomaly profile in Figure 3 shows the temporal distribution of anomalous sequences for a successful `sendmail` intrusion, one of the `syslogd` intrusion runs. From this figure we can see how noticeable intrusions are, and how anomalies are clumped. It also indicates that if we were doing real-time monitoring, we might be able to detect some intrusions *before* an intruder gains access, right at the start of the intrusive behavior.

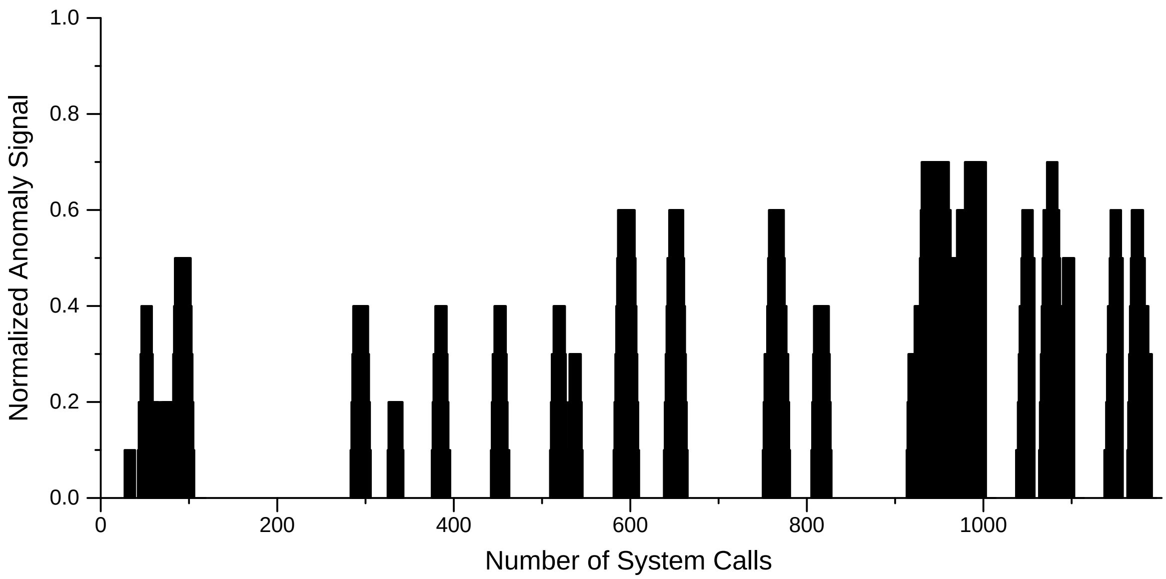

Figure 3. Anomaly profile for a run of the `syslogd` intrusion. The data represents a trace of system calls that is a concatenation of 5 forked `sendmail` processes. The $\hat{S}_A$ value for this intrusion is 0.7, i.e. the highest point reached on the y axis.

5. Behavior in a Real Environment

The results reported in section 4 were based on normal databases generated synthetically, i.e. we attempted to exercise all normal modes of behavior of a given program and used the resulting traces to build our normal databases. For an IDS that is deployed to protect a functioning system, this may not be the best way to generate normal. The real normal behavior of a given program on a particular machine could be quite different from the synthetic normal. Some synthetic normal behaviors may be absent in an actual system; on the other hand, the real normal might include behavior that we had not thought of, or were unable to incorporate into the synthetic. In this section we attempt to build normal in a real environment.

Several questions arise when we consider collecting real normal on a running system:

1. How do we ensure that we have not included abnormal sequences? That is, how do we ensure that the system is not being exploited as we collect the normal traces? Including abnormal sequences could result in false negatives.
2. How do we ensure that our normal is sufficiently comprehensive? How long do we collect normal for? How much normal is enough? An incomplete normal could result in false positives.
3. Are intrusions still detectable as we increase the size of the normal? As the size of normal increases, we include rare normal sequences that could overlap more with

abnormal sequences, thus reducing detection rates, i.e. increasing false negatives.

5.1 Collecting Real Normal

We have collected normal for `lpr` in two different real environments, at the Massachusetts Institute of Technology's Artificial Intelligence laboratory (MIT), and at the University of New Mexico's Computer Science Department (UNM). In both cases, we used a very simple solution to question 1 posed above: How do we ensure that intrusive behavior is not included in these normals? For the `lpr` we have studied, we are aware of only one intrusion (reported in section 4.2.2 above) which requires that `lpr` generate a 1000 print jobs in close succession, which is something we as observers could easily detect on a system that never generates more than 200 jobs in a day. This does not guarantee that our normal is free of intrusion traces, but at least we have excluded the intrusion against which we perform our analysis. In general, however, the problem will not be so trivial, particularly if we do not know the nature of the intrusion beforehand, i.e. if we are concerned with true anomaly detection. Possible ways of excluding intrusive behavior from the normal trace include:

- Collect normal in the real, open environment, whilst monitoring the environment very carefully to ensure that no intrusions have happened during our collection of normal. This is what we did for `lpr`.
- Collect normal in an isolated environment where we are sure no intrusions can happen. The disadvantage of this solution is that the normal will possibly be incomplete, because the environment is of necessity limited, particularly in the case of programs, such as `sendmail`, that communicate with the outside world.

In the MIT environment, we traced `lpr` running on 77 different hosts, each running SunOS, for two weeks, to obtain traces of a total of 2766 print jobs. The growth of the size *N* of the normal database is shown in Figure 4. As more print jobs are traced and the traces added into normal, so the number of unique sequences *N* in the normal database grows. Initially, the growth is very rapid, but then tapers off, in particular, for $k = 6$ and $k = 10$, there is minimal database growth past 1000 print jobs. This reinforces the idea of choosing as short a sequence length as possible, because we can accumulate the full range of normal sequences much more rapidly for short sequences. We regard Figure 4 as promising, because it indicates that normal behavior is limited and can be collected in a short period of time (depending on how much the system is used).

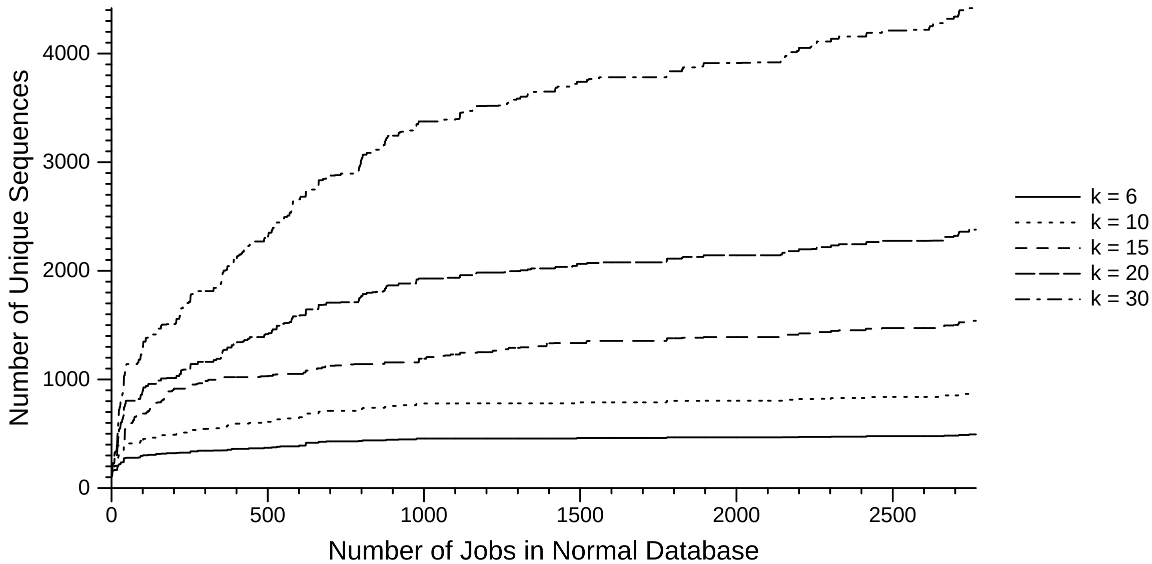

Figure 4. Growth of database size for `lpr` real normal collected at MIT. The x axis indicates the number of print jobs traced, and the y axis indicates the number of unique sequences *N* in the normal database.

How much does normal vary between different environments? We have some answers in the case of `lpr` because we have two normals collected independently at MIT and UNM, for the identical program and operating system. These represent considerably different environments, as can be seen from the differences listed in Table 6, for example, we traced `lpr` on only one host at UNM, whereas we traced it on 77 hosts at MIT. Despite the differences in environment, the patterns of database growth in the UNM environment are similar to those at MIT (data not shown), although the resulting database sizes are different: 569 unique sequences for UNM and 876 for MIT. These databases not only differ in size, but also in content: For example, a comparison of the unique sequences in both databases for $k = 6$ indicates that only 141 of the sequences are the same between the databases, which represents 40% of the UNM database and 29% of the MIT database.

	UNM	MIT
Number of hosts	1	77
Number of print jobs	1234	2766
Time period (weeks)	13	2
DB Size N	569	876
Detection of lprcp:		
# mismatches	11009	11006
% mismatches	7	7
$\hat{S}_A$	0.4	0.4

Table 6. Comparison of `lpr` normals collected at MIT and at UNM. These results are for $k = 10$.

Although these databases are very different, they both detect the `lprcp` intrusion almost identically. When we analyze the anomalous sequences generated by the intrusion, we find that there are 16 *unique* anomalous sequences detected by the UNM database, which are identical to 16 of the 17 unique anomalous sequences detected by the MIT database, i.e. the anomaly is almost identical for both databases. This suggests that intrusion signatures could be encoded in sequences of system calls, i.e. the system call signature could be the basis of a misuse-IDS, or an IDS that does both anomaly and misuse detection (for a further exploration of these ideas see [38]).

5.2 How much Normal is enough?

This section addresses questions 2 and 3 posed above: How much normal is enough? And, are intrusions still detectable as the size of normal increases? In our experiments we used the `lpr` data we collected in the real environments at MIT and UNM. In both cases, we divided the set of data into two, the first set is used as the training set, and the second set as the test set. The training data are used to build up a normal database, and the test data are scanned using this normal database (we explain below how we choose the test and training sets). A false positive is then any sequence i in the test set for which

$$d_{\min}(i) \geq C.$$

We determine the lowest false positive rate $\boldsymbol{\varepsilon}_{fp}$ by setting the threshold C to be the maximum value needed for the normal database to detect the `lprcp` intrusion. Because we only have one intrusion to test against, and we set the threshold so that we always detect it, we have zero false negatives. The false positive rate is simply the number of false positives per print job.

The *expected* false positive rate was calculated using the bootstrap technique, which is a procedure for estimating (approximating) the distribution of a statistic from a random sample [14]. We divided the jobs into test and training sets as follows: up to 700 jobs were chosen randomly with replacement for the training set, and the remaining jobs were used for the test set (thus we had a test set of 2066 jobs for MIT and one of 534 jobs for UNM). This process was repeated 100 times to get the bootstrap estimate. The bootstrap is applicable here because the data appear to be stationary. We checked for stationarity by sampling the jobs both randomly, and in small chronologically consecutive groups, and comparing the means produced by the two sampling methods. A two-tailed, two sample t-test between these two samples gives a P-value of 0.19. Thus the probability that these means are different is insignificant.

The expected false positive rates and standard deviations are shown in Figure 5 for varying sizes of the normal database. The data shown are for the MIT `lpr`, with $k = 10$, and $C = 4$. Similar results were obtained with the `lpr` data collected at UNM (data not shown).

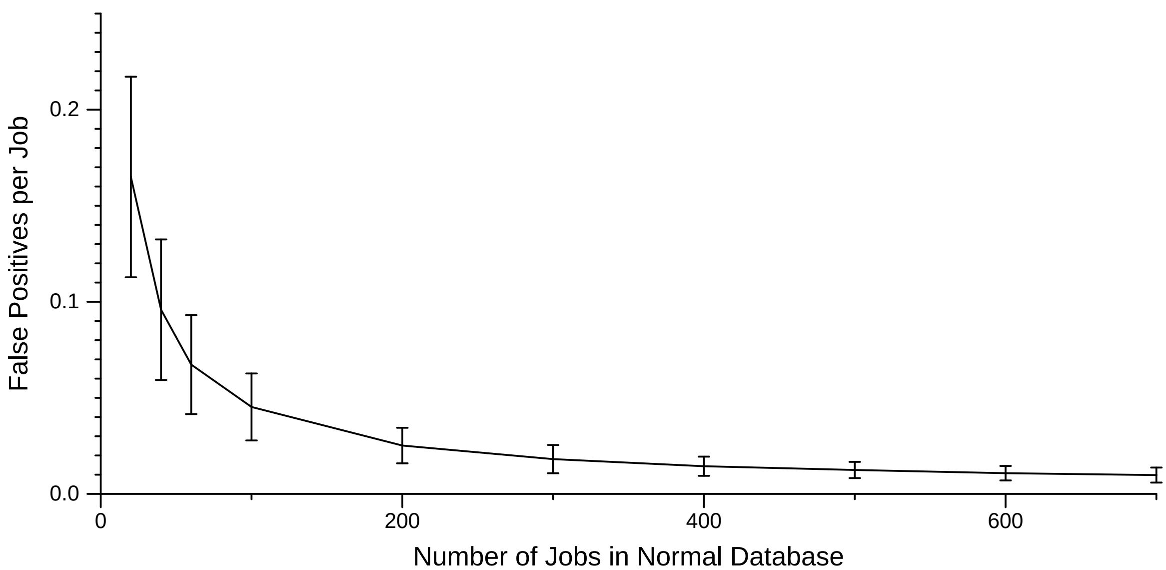

Figure 5. Bootstrap estimate of change in expected false positive rate as normal database size increases.

To summarize, the lowest expected false positive rate in Figure 5 is 0.01 ± 0.004. This is about 1 false positive in every 100 jobs, or, on the MIT system, an average of 2 false positives per day. This rate was computed for a normal database of 700 jobs, with 2066 jobs in the test set. From Figure 5 the false positive rate appears to be leveling off. However, when we increase the size of the normal database to 1400 jobs (not shown in the figure), with a test set of 1366 jobs, the rate drops to 0.005 ± 0.002, which is one false positive per day. We are hesitant to draw too many conclusions from these data because they are derived from a single program for which we have only one true positive (an intrusion), and so we cannot get an accurate measure of false negatives, or the false positive rate we could expect if we had to detect several different intrusions. Furthermore, although we have done tests to check for stationarity, we cannot be absolutely sure that there are no time-dependent effects in the data.

If we build the normal database chronologically from the first 700 jobs and compare that to the remaining 2066 jobs, we get a false positive rate of 0.004 for a sequence length of 10. Although this is within the bootstrap distribution, there is a probability of only 0.05 of getting a false positive rate that low when the jobs are randomly selected. So it may be that there are temporal dependencies not detected by our tests for stationarity. In an on-line system, normal would be constructed from the first jobs encountered, and so in this case we could expect lower false positive rates.

It is worth noting that these false positive rates are computed for a system in which we have only spent 3 or 4 days collecting normal behavior. Provided the size of the normal database does not grow indefinitely, we could expect our false positive rates to reduce as we spend more days on normal collection. This is illustrated by the fact that when we increase the size of the normal database to include 1400 jobs (7 days), our false positive rate halves. Furthermore, even if we use all of the normal behavior traced over two weeks to build the normal database, the threshold for detection of the `lprcp` intrusion does not drop (see Table 6).

5.3 Analysis of False Positives

We looked at the sequences which were responsible for the false positives to get an idea of what could be causing rare but acceptable behavior. We investigated several false positives and found unusual circumstances behind all of them, including:

1. Trying to print on a machine where the file `/dev/printer` did not exist. This file is a named local socket that connects to `lpd` running on the machine. Apparently `lpr` would place a job in the queue, but could not communicate with `lpd`. It is unclear whether `lpd` indicated an error. It is likely that the job did not print.
2. Printing from symbolic links. `lpr` was told to print a file in the current directory using the -s flag. It seems that the file to be printed was actually a symbolic link to another file, so `lpr` followed the symbolic link to the original file, and then placed a symbolic link to the real file in the spool directory.
3. Printing from a separately administered machine with a very different configuration.
4. Trying to print a job so large that `lpr` ran out of disk space for the log file.

When the normal database is built chronologically, there are only 6 false positives, 3 of which are caused by the first case (1) above, and 3 of which are caused by the second case (2).

Are these really false positives? A false positive is some sort of acceptable behavior that is classified as anomalous. If the behavior is unacceptable, even if it is not caused by an intrusion, we would want to know about it, because it indicates that the system is not functioning properly or efficiently. Points 1 and 4 above are both instances of irregular behavior symptomatic of a problem with the system; both indicate conditions that need to be rectified. In this sense, neither 1 nor 4 are false positives. This kind of analysis indicates that our actual false positive rate is lower than the reported values, for example, in the case of a chronological normal, the number of false positives would be reduced from 6 to 3.

6. Discussion

The previous two sections have presented evidence that short sequences of system calls are good discriminators between normal and abnormal operating characteristics of several common UNIX programs. In essence, we have found a regularity in executing programs that is highly likely to be perturbed by intrusive activities. These results are interesting for several reasons: They suggest a possible implementation path for a lightweight intrusion-detection system; the techniques might be applicable to security problems in other computational settings; they illustrate the value of studying the empirical behavior of actual systems; and they suggest a strategy for approaching other on-line problems in computing that are not well solved by conventional methods.

Although the results presented in Sections 4 and 5 are suggestive, much more testing needs to be completed to validate the approach. In particular, extensive testing on a wider variety of UNIX programs being subjected to large numbers of different kinds of intrusions is desirable. For each of these programs, we would ideally like to have results both in controlled environments (in which we could run large numbers of intrusions) and in live user environments. Overall, we expect that discrimination will be more difficult in highly stressed environments (high user loads, overloaded networks, etc.) in which many exceptional conditions are raised. Furthermore, we would like to test these ideas in different operating systems, such as Windows NT. Recently, we have successfully detected intrusions in two other programs: a buffer overflow in the `xlock` program running in Linux, and a symbolic link vulnerability in the `swinstall` program running under HP-UX [8][4].

However, there are some logistical problems associated with collecting data in live user environments. Most operating systems are not shipped with robust tracing facilities, and

[4] This data was collected by Mark Crosbie, at Hewlett Packard.

as much as possible, we would like to collect data in standardized environments. It is difficult to justify installing code with known vulnerabilities (needed to run large numbers of different intrusions) in a production environment, thus putting the user community at risk of real intrusions. Finally, there are no obvious stopping criteria. Every system is slightly different---when can we say that we have collected enough data on enough different programs in enough different environments?

Assuming that more detailed experiments confirm our results, there are a host of systems-engineering questions that need to be addressed before an IDS based on these principles could be implemented and deployed. First, what combination of synthetic and actual behavior should be collected to define a normal database? In many user environments, certain (legitimate) features of programs might be seldom used, and so a database generated from live user traces might generate false positives, whereas constructing a synthetic database appropriately could prevent these false positives. It would also be much easier to distribute an IDS that did not require a lot of customization at the time it is installed---an IDS should make systems administration easier not harder. Thus, the collection of real usage data at install-time would have to be highly automated. A related complication is how to guarantee that no intrusions take place during the collection of normal behavior. Second, which UNIX programs should be monitored, and how (and when) should databases be switched when different processes are started? We could use a completely different database for each program---earlier we emphasized that normal behavior for different programs is significantly different (ranging from 40% to 80%). However, these percentages also imply that there is much behavior in common between different programs, and so in a running implementation we might be able to reduce resource requirements by exploiting this commonality. Finally, we envision our IDS as a real-time, on-line system that could potentially discover and interrupt some intrusions before they were successful. The feasibility of this is highly dependent on efficient design and implementation of both the tracing facility and the algorithms that detect mismatches.

Our emphasis has been on determining if our approach can be successful at all. We were not too concerned with efficiency issues in this paper. However, for the system to be able to detect intrusions in real-time---as they are happening---will require careful attention to efficiency issues. As a first step towards this we have analyzed the complexity of our algorithm, although we have not been able to measure its efficiency in a production environment. Should the implementation prove too inefficient, there are numerous simplifications we could experiment with, such as looking only at specific kinds of calls, or only at every tenth call, etc.

An important question in the context of an IDS is what response is most appropriate once a possible intrusion has been detected. This is a deep topic and largely beyond the scope of our paper. Most IDS respond by sending an alarm to a human operator. In the long run, however, we believe that the response side will have to be largely automated if IDS technology is going to be widely deployed. We have some evidence that intrusions generate highly regular signatures, so it might be possible to store these signatures for known intrusions and respond more aggressively when those signatures are detected. Then for new anomalies more cautious actions could be taken. One advantage of monitoring at the process level is that a wide range of responses is possible, ranging from shutting down the computer completely (most radical) to simply running the process at lower priority.

The method we propose is not a panacea---it will certainly miss some forms of intrusions. One example is race condition attacks, which typically involve stealing a resource (such as a file) created by a process running as root, before the process has had a chance to restrict access to the resource. If the root process does not detect an unusual error state, a normal set of system calls will be made, defeating our method. Other examples of intrusions that would be missed are password hijacking (when one user masquerades as another), and cases in which a user violates policy without using privileged processes.

The idea of looking at short sequences of behavior is quite general and might be applicable to several other security problems. For example, people have suggested applying the technique to several other computational systems, including: The Java virtual machine, the CORBA distributed object system, security for ATM switches, and network security. For each of these potential applications, it would be necessary first to determine empirically whether simple definitions (analogous to sequences of system calls) give a clear and compact signature of normal behavior, and then to determine if the signature is perturbed by intrusive behavior.

Our approach is similar to several other approaches, although the differences are critical. Ko et al [30] have also chosen the level of privileged processes, but they characterize the behavior of a privileged process by a program specification or policy, which is a description of what the program should be able to do. This policy is derived from the program code and so requires specialized knowledge of program function. Writing a policy can be prone to the same sorts of errors as writing the program, i.e. it is hard to guarantee correctness. Most importantly, from our perspective, such a policy could easily include behavior that is legal but not normal because it is hard to determine beforehand what behavior should be normal. We avoid these issues by treating the program as a black box, and relying purely on empirical observation to ascertain program behavior. Another key difference is that we rely exclusively on sequencing information, unlike the specification approach, which monitors individual operations. However, there are other approaches, such as TIM [39], that consider sequencing information. These differ from our approach in that they look at the domain of user behavior, and use a probabilitistic approach for detecting anomalies. Because our results are sufficiently promising the added complexity of using probabilities seems unnecessary. It is possible that our simple deterministic approach is successful because our data is well-structured. If this is the case, it may well be that probabilities are necessary in less structured domains, such as user behavior.

In earlier papers, we have advocated a comprehensive approach to computer security based on a collection of organizing principles derived from our study of the immune system [38]. The immune-system perspective has certainly influenced many of our design decisions, but in this paper we are emphasizing concrete computational mechanisms and largely ignoring the immune system connection. Details of how our approach to IDS fits into the overall immune-system vision are given in [17]. Extensions are suggested by analogy.

An important bias underlying our approach is that modern computers are "complex systems" in the sense that they are comprised of a large number of components, many of which interact nonlinearly. These components are continually evolving, as well as the environments in which they are embedded, their users, and the programmers who implement them. This complexity threatens to overwhelm design strategies based on functional decomposition. Furthermore, it implies that although we design and build computers, we do not necessarily understand how they behave. An example of this is the fact that the normal behavior of a highly complex program such as `sendmail` can be captured by such a small number of system call sequences---it would have been hard to predict this. Rather than making assumptions about how we believe that programs or users will behave, or trying to prespecify their behavior (and being surprised), this paper asks the question: What behavior do we observe? That is, we take existing artifacts and study their behavior rigorously. Although such an approach might be dismissed as "merely empirical" rather than theoretical, our point is that we need to spend more time asking to what extent our existing theories describe our existing artifacts.

7. Conclusions

We presented a method for anomaly intrusion detection at the process level. Normal was defined in terms of short sequences of system calls executed by running privileged processes. Our profiles of normal behavior, which consisted of unique sequences of length 10, were remarkably compact, for example, the `sendmail` database contained only 1318 such sequences. Three measures were used to detect abnormal behavior as deviations from profiles of normal. These measures allowed us to successfully detect several classes of abnormal behavior, including: Intrusions in the UNIX programs `sendmail`, `lpr` and `ftpd`; failed intrusion attempts against `sendmail`; and error conditions in `sendmail`. We studied two different methods of accumulating normal profiles: Generating normal synthetically by attempting to exercise the program in as many modes of normal operation as possible, and tracing a process in a live user environment. In the latter case we have analyzed the data for false positives. Our false positive rates for `lpr` were about 1 in every 100 print jobs (and explainable in terms of system problems), but these results are tentative because we did not have sufficient data for a comprehensive analysis. In future we intend to expand our base of intrusions and gather more data for more programs running in real environments, so we can get more realistic estimates of false positive and false negative rates.

References

[1] 8LGM Advisories. http://www.8lgm.org/advisories.html.

[2] 8LGM.[8lgm]-advisory-16.unix.sendmail-6-dec-1994. Http://www.8lgm.org/ advisories.html, December 94.

[3] 8LGM.[8lgm]-advisory-22.unix.syslog.2-aug-95. Http://www.8lgm.org/ advisories.html, August 95.

[4] 8LGM.[8lgm]-advisory-3.unix.lpr.19-aug-1991. Http://www.8lgm.org/ advisories.html, August 91.

[5] Anderson D, Frivold T, Valdes A. Next Generation Intrusion Detection Expert Systems (NIDES): A Summary. Technical Report SRI-CSL-95-07, Computer Science Laboratory, SRI International, May 1995.

[6] CERT Advisories. ftp://info.cert.org/pub/cert-advisories/

[7] CERT. Syslog vulnerability - a workaround for sendmail. Ftp://info.cert.org/pub/ cert_advisories/ CA-95:13.syslog.vul, October 95.

[8] CERT. swinstall vulnerability. Ftp://info.cert.org/pub/ cert_advisories/ CA-96:27, December 1996.

[9] ftp://ftp.cert.org/pub/cert_advisories/CA-93%3A06.wuarchive.ftpd.vulnerability

[10] Crosbie M, Spafford G. Defending a Computer System using Autonomous Agents. Technical Report No. 95-022, Dept. of Comp. Sciences, Purdue University, March 1996.

[11] D'haeseleer P, Forrest S, Helman P. An Immunological Approach to Change Detection: Algorithms, Analysis and Implications. In *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*, IEEE Press, 1996.

[12] Denning D E. An Intrusion Detection Model. In *IEEE Transactions on Software Engineering*, Los Alamos, CA, IEEE Press, 1987.

[13] Denning D E. Cryptography and Data Security. Addison-Wesley Inc, 1992.

[14] Efron B, Tibshirani R J. An introduction to the bootstrap. New York, Chapman & Hall, 1993.

[15] Farmer D, Spafford E H. The COPS Security Checker System. Technical Report

CSD-TR-993, Purdue University, Comp. Science, 1996.

[16] Farmer D, Venema W. Improving the Security of your Site by Breaking into It. ftp://ftp.win.tue.nl/pub/security/admin-guide-to-cracking.101.Z. 1995.

[17] Forrest S, Hofmeyr S A, Somajayi A. Computer Immunology. *Communications of the ACM*, 40(10):88-96, 1997.

[18] Forrest S, Hofmeyr S A, Somajayi A. A Sense of Self for UNIX Processes. In *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*, IEEE Press, 1996.

[19] Forrest S, Perelson A S, Allen L, Cherukuri R. Self-Nonself discrimination in a Computer. In *Proceedings of 1994 IEEE Symposium on Computer Security and Privacy*, IEEE Press, 1994.

[20] Forrest S, Somayaji A, Ackley D. Building diverse computer systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, Computer Society Press, Los Alamitos, CA, pp. 67-72, 1997.

[21] Fox K L, Henning R R, Reed J H, Simonian R. A Neural Network Approach Towards Intrusion Detection. In *Proceedings of the 13th National Computer Security Conference,* Washington DC, Oct 1990.

[22] Frank J. Artificial Intelligence and Intrusion Detection: Current and Future Directions. In *Proceedings of the 17th National Computer Security Conference*, 1994.

[23] Heady R, Luger G, Maccabe A, Servilla M. The Architecture of a Network Level Intrusion Detection System. Technical Report, Dept. Comp. Science, University of New Mexico, August 1990.

[24] Heberlein L, Dias G, Levitt K, Mukherjee B, Wood J, Wolber D. A Network Security Monitor. In *Proceedings of the IEEE Symposium on Research in Computer Security and Privacy*, 1990.

[25] Heberlein L, Mukherjee B, Levitt K. Internet Security Monitor: An Intrusion Detection System for Large Scale Networks. *Proceedings of 15th National Computer Security Conference*, 1992.

[26] Hochberg J, Jackson K, Stallings C, McClary J F, DuBois D, Ford J. NADIR: An Automated System for Detecting Network Intrusion and Misuse. *Computers and Security*, 12(3):235-248, 1993.

[27] Illgun K. Kemmerer R A, Porras Ph A. State Transition Analysis: A Rule-Based Intrusion Detection Approach. In *IEEE Transactions on Software Engineering* 3, 1995.

[28] Kephart J O. A Biologically Inspired Immune System for Computers. *Artificial Life IV*, eds Brooks R, Maes P, MIT Press, 1994.

[29] Kim G H, Spafford E H. The Design and Implementation of Tripwire: A File System Integrity Checker. In *Proceedings of the 2cnd ACM Conference on Computer and Communications Security*, 1994.

[30] Ko C, Fink G, Levitt K. Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring. In *Proceedings of the 1994 Computer Security*

Applications Conference, 1994.
[31] Kumar S, Spafford E H. An Application of Pattern Matching in Intrusion Detection. Technical Report CSD-TR-94-013, Purdue University, Dept. Comp. Science, June 1994.
[32] Kumar S. Classification and Detection of Computer Intrusions. Phd Thesis, Purdue University, August 95.
[33] Liepens G, Vaccaro H. Intrusion Detection: Its Role and Validation. *Computers and Security*, 11: 347 - 355, 1992.
[34] Lunt T F. Detecting Intruders in Computer Systems. In *Conference on Auditing and Computer Technology*, 1993.
[35] Lunt T, Tamura A, Gilham F, Jagannathan R, Neumann P, Javitz H, Valdes A, Garvey T. A Real-Time Intrusion Detection Expert System (IDES). Final Technical Report, Computer Science Laboratory, SRI International, Menlo Park, Calif., Feb. 1992.
[36] Schuba C L, Spafford E H. Countering Abuse of Name-Based Authentication. *22cnd Annual Telecommunications Policy Research Conference*, 1996.
[37] Smaha S E, Winslow J. Misuse Detection Tools. *Computer Security Journal* 10 (1):39 - 49, Spring 1994.
[38] Somayaji A, Hofmeyr S A, Forrest S. Principles of a computer immune system In *New Security Paradigms Workshop 97*, in press.
[39] Teng H, Chen K, Lu S. Adaptive Real-Time Anomaly Detection Using Inductively Generated Sequential Patterns. In *Proceedings of IEEE Symposium on Research in Computer Security and Privacy*, 1990.

Appendix 1: Description of Intrusions

This appendix gives more detailed descriptions of intrusions and error conditions that we tested against.

- **sunsendmailcp:** The sunsendmailcp script uses a special command line option to cause `sendmail` to append an email message to a file. By using this script on a file such as `/.rhosts`, a local user may obtain root access.
- **syslogd:** The `syslogd` attack uses the syslog interface to overflow a buffer in `sendmail`. A message is sent to the `sendmail` on the victim machine, causing it to log a very long, specially created error message. The log entry overflows a buffer in `sendmail`, replacing part of the `sendmail`'s running image with the attacker's machine code. The new code is then executed, causing the standard I/O of a root-owned shell to be attached to a port. The attacker may then attach to this port at his or her leisure. This attack can be run either locally or remotely; we have tested both modes. We also varied the number of commands issued as root after a successful attack.
- **decode:** In older `sendmail` installations, the alias database contains an entry called

"decode," which resolves to `uudecode`, a UNIX program that converts a binary file encoded in plain text into its original form and name. `uudecode` respects absolute filenames, so if a file "bar.uu" says that the original file is "/home/foo/.rhosts" then when `uudecode` is given "bar.uu", it will attempt to create foo's `.rhosts` file. `sendmail` will generally run `uudecode` as the semi-privileged user daemon, so email sent to decode cannot overwrite any file on the system; however, if the target file happens to be world-writable, the decode alias entry allows these files to be modified by a remote user.

- **lprcp:** The lprcp attack script uses `lpr` to replace the contents of an arbitrary file with those of another. This attack exploits the fact that older versions of `lpr` use only 1000 different names for printer queue files, and they do not remove the old queue files before reusing them. The attack consists of getting `lpr` to place a symbolic link to the victim file in the queue, incrementing `lpr`'s counter 1000 times, and then printing the new file, overwriting the victim file's contents.
- **`ftpd`**: This is a configuration problem. `Wu.ftpd` is misconfigured at compile time, allowing users SITE EXEC access to /bin. Users can then run executables such as bash with root privilege.
- **unsuccessful intrusions:** sm5x, sm565a.
- **forwarding loops:** A local forwarding loops occurs in `sendmail` when a set of `$HOME/.forward` files form a logical circle. We considered the simplest case, with the following setup:

Email address	.forward file
foo@host1	bar@host2
bar@host2	foo@host1

Appendix 2: Synthetic Normal Generation

This appendix describes briefly how synthetic normals were generated for `ftpd` and `lpr`.

The synthetic `ftpd` was generated by tracing the execution of ftpd, using every option on the `ftpd` man page at least once.

The synthetic `lpr` was generated by tracing the following types of `lpr` jobs: Printing a text file, printing a postscript file, attempting to print a nonexistent file, printing to several different printers, printing with and without burst pages, printing with symbolic links (the -s option).

Acknowledgements

The authors gratefully acknowledge support from the Defense Advanced Research Projects Agency (grant N00014-96-1-0680), the Office of Naval Research (grant N00014-95-1-0364), and the National Science Foundation (grant IR-9157644). Some of this work was performed whilst the authors were at the MIT Artificial Intelligence Laboratory, where a very helpful support staff provided a good environment for experimentation. Many people have contributed important ideas and suggestions for this paper, including D. Ackley, A. Koseresow, B. Sanchez, B. LeBaron, P. D'haeseleer, A. B. Maccabe, K. McCurly, N. Minar, G. Hunsicker and M. Crosbie. Some of the data presented in Table 4 and in Table 5 were generated with the help of L. Rogers and T. Longstaff of the Computer Emergency Response Team (CERT).

8. Revisions made to the Paper

We list all of the major changes here, indexed by section and, where appropriate, the number of the paragraph within the *section*. We also indicate the page numbers of the changes.

Title.
Changed the title on page 1 to:
"Intrusion Detection using Sequences of System Calls"

Section 1.
Modified paragraph 2 on page 2:
"There are many different levels on which an IDS can monitor system behavior. It is critical to profile normal behavior at a level that is both robust to variations in normal and perturbed by intrusions. In the work reported here, we chose to monitor behavior at the level of *privileged processes*. Privileged processes are running programs that perform services (such as sending or receiving mail), which require access to system resources that are inaccessible to the ordinary user. To enable these processes to perform their jobs, they are given privileges over and above those of an ordinary user (even though they can be invoked by ordinary users). In UNIX, processes usually run with the privileges of the user that invoked them. However, privileged processes can change their privileges to that of the superuser by means of the setuid mechanism. One of the security problems with privileged processes in UNIX is that the granularity of permissions is too coarse: Privileged processes need superuser status to access system resources, but granting them such status gives them more permission than necessary to perform their specific tasks [30]. Consequently, they have permission to access *all* system resources, not just those that are relevant to their operation. Privileged processes are trusted to access only relevant system resources, but in cases where there is some programming error in the code that the privileged process is running, or if the privileged process is incorrectly configured, an ordinary user may be able to gain superuser privileges by exploiting the problem in the program. For the sake of brevity, we usually refer to privileged processes (or programs) simply as "processes" (or "programs"), and use the qualifier only to resolve ambiguities."

Modified paragraph 3 on pages 2-3:
"It is clear that privileged processes are a good level to focus on because exploitation of vulnerabilities in privileged processes can give an intruder super-user status. Furthermore, privileged processes constitute a natural boundary for a computer, especially processes that listen to a particular port. In UNIX, privileged programs, such as `telnetd` and `logind`, function as servers that control access into the system. Corruption of these

servers can allow an intruder to access the system remotely. Monitoring privileged processes also offers some advantages over monitoring user behavior, which has been the most common method to date (for example, see [5],[12],[26],[35],[39]). The range of behaviors of privileged processes is limited compared to the range of behaviors of users; privileged processes usually perform a specific, limited function, whereas users can carry out a wide variety of actions. Finally, the behavior of privileged processes is relatively stable over time, especially compared to user behavior. Not only do users perform a wider variety of actions, but the actions performed may change considerably over time, whereas the actions (or at least the functions) of privileged processes usually do not vary much with time."

Modified paragraph 4 on page 3:

"Our approach to detecting irregularities in the behavior of privileged programs is to regard the program as a black box, which, when run, emits some *observable*. We believe that this observable should be a dynamic characteristic of that program; although code stored on disk may have the potential to do harm, it has to be actually running to realize that potential. If we regard the program as a black-box, we do not need specialized knowledge of the internal functioning or the intended role of the program; we can infer these indirectly by observing its normal behavior. A natural observable for processes in UNIX would be based on *system calls*, because UNIX processes accesses system resources through the use of system calls. We have chosen *short sequences* of system calls as our observable."

Added paragraph 6 on pages 3-4:

"We want an IDS that is stable and lightweight (efficient), all of which depends on the discriminator (observable) that we use to distinguish between acceptable and unacceptable behavior. By stable we mean that the discriminator reliably distinguishes between acceptable and unacceptable behavior. Our approach is experimental because we believe that current theories do not adequately describe how implemented systems really run. In this paper we are primarily concerned with determining empirically if the discriminator is stable. Efficiency is a secondary consideration, and is addressed in this paper to the extent that we analyze the complexity of our algorithm; however, we do not report actual running times for the method on a production system. "

Removed the paragraph:

"One of our goals is lightweight intrusion detection. By lightweight we mean methods that could be implemented in an IDS to run on-line, and that would use minimal computing resources. Because an IDS is a preventative measure that does not enhance system performance (quite the reverse), it is hard to justify the use of a computationally expensive IDS. In many cases, if an IDS is not sufficiently lightweight, it simply will not be used. Another advantage of having an IDS be sufficiently lightweight to run on-line is the potential to detect intrusions in progress, perhaps to prevent the intrusion attempt from succeeding."

Section 3.1.

Added paragraph 2 on page 7:

"This method is complicated by the fact that in UNIX a program can invoke more than one process. Processes are created via the `fork` system call or its virtual variant `vfork`. The essential difference between the two is that a `fork` creates a new process which is an instance of the same program (i.e. a copy), whereas a `vfork` replaces the existing process with a new one, without changing the process ID. We trace forks individually and include their traces as part of normal, but we do not yet trace virtual forks because a virtual fork executes a new program. In the future, we will switch databases dynamically to follow the virtual fork."

Section 3.2.

Added paragraph 2 on page 9:

"We make a clear distinction here between *normal* and *legal* behavior. In the ideal case we want the normal database to contain all variations in normal behavior, but we do not want it to contain every single possible path of legal behavior, because our approach is based upon the assumption that normal behavior forms only a subset of the possible legal execution paths through a program, and unusual behavior that deviates from those normal paths signifies an intrusion or some other undesirable condition. We want to be able to detect not only intrusions, but also unusual conditions that are indicative of system problems. For example, when a process runs out of disk space, it may execute some error code that results in an unusual execution sequence (path through the program). Clearly such a path is *legal*, but certainly it should not be regarded as *normal*."

Added paragraph 4 on pages 9-10:

"An alternative is to constrain the measure locally. The anomalies we have studied are temporally clumped: Anomalous sequences due to intrusions seem to occur in local bursts. However, defining a local measure is difficult because we have an unordered state space, i.e. we have no true notion of locality---how "close" one system call is to another, or how "close" one system call sequence is to another. We have chosen "Hamming distance" between sequences as the measure. Although this choice is somewhat arbitrary, it is related to how closely anomalies are clumped. We cannot theoretically justify this measure, so we determine its worth empirically."

Section 3.3.

Modifed paragraph 2 on page 11:

"To detect an intrusion, *at least one* of the sequences generated by the intrusion must be classified as anomalous. In terms of our measures, what we require is that at least one of the sequences generated by the intrusion has $d_{\min} > 0$. We measure the strength of the anomaly by $d_{\min}$, and because we want intrusions to generate strong anomalies, we assume that the higher the $d_{\min}$ the more likely it is that the sequence was actually generated by an intrusion. In practice, we report the maximum $d_{\min}$ value that was encountered during a trace, because that represents the strongest anomalous signal found in the trace, i.e. we compute the signal of the anomaly, S_A, as:"

Added the last sentence to paragraph 5 on page 12:

"The validity of the assumption that intrusive behavior is characterized by increased Hamming distance from normal sequences is tested empirically in the sections that follow."

Section 4.

Modified the fourth sentence of paragraph 1 on page 12-13:

"For example, if we generate normal synthetically we have no idea what false positive rates we will get in realistic settings because our synthetic, by definition, includes all variations on normal behavior (although not all variations on legal behavior)."

Section 4.1.

Modified paragraph 1 on page 13:

"We studied normal behavior for three different programs in UNIX: `sendmail`, `lpr` and `wu.ftpd` (the first two were running under SunOS 4.1.x, and the last one was running under Linux). `Sendmail` is a program that sends and receives mail, `lpr` is a program that enables users to print documents on a printer, and `ftpd` is a program for the transfer of files between local and remote hosts. Because `sendmail` is the most complex of these programs, we will briefly describe how we exercised `sendmail` to produce a profile of normal behavior (the methods for constructing synthetic normal for the other two programs are described in Appendix 2). We considered variations in message length, number of messages, message content (text, binary, encoded, encrypted), message subject line, who sent the mail, who received the mail, and mailers. In addition, we looked at the effects of forwarding, bounced mail and queuing. Lastly, we considered the effects of the origin of all these variations in the cases of remote and local delivery."

Section 4.2.2.

Modified paragraph 3 on pages 17-18:

We compared system call traces for each of the three categories (successful exploits, unsuccessful exploits and error conditions) with the normal database for the relevant program and recorded the number of mismatches, percentage of mismatches over the trace, and $\hat{S}_A$ values. Table 4 shows results for successful intrusions. Each row in the table reports data for one typical trace. In most cases, we have conducted multiple runs of the intrusion with identical or nearly identical results; where runs differed significantly, we report a range of values. To date, we have collected data on five successful intrusions, three of them for `sendmail`, one for `lpr` [4] and one for `ftpd` [8] **CERT. swinstall vulnerability. Ftp://info.cert.org/pub/ cert_advisories/ CA-96:27, December 1996.** **[9**]. The three `sendmail` intrusions were: `sunsendmailcp` [2], `syslogd` [3], [7], and a decode alias intrusion. These intrusions are described in the appendix. Most of the successful intrusions are clearly detected, with $\hat{S}_A$ values of 0.5 to 0.7. The exception to this is the decode intrusion, which, on the low end of the range, generates only 7 mismatches and a $\hat{S}_A$ value of 0.2. These results suggest approximate detection thresholds that we would need in an online system to detect intrusions.

Modified the first and third rows of table 4 on page 18 :

Anomaly	Number Mismatches	% Mismatches	$\hat{S}_A$
syslogd	248 - 529	17 - 30	0.7
sunsendmailcp	92	25	0.6
decode	7 - 22	1 - 2	0.2 - 0.5
lprcp	242	9	0.5
ftpd	496	38	0.7

Added the fourth sentence to the caption of table 4 on page 18:

"In some cases the columns list a range of values, from minimum to maximum."

Modified the third row of table 5 on page 18:

Anomaly	Number Mismatches	% Mismatches	$\hat{S}_A$
sm565a	54	22	0.6
sm5x	472	33	0.6
forward loop	21 - 108	10 - 18	0.4 - 0.6

Section 5.1.

Added paragraph 3 on page 22:

"How much does normal vary between different environments? We have some answers in the case of `lpr` because we have two normals collected independently at MIT and UNM, for the identical program and operating system. These represent considerably different environments, as can be seen from the differences listed in Table 6, for example, we traced `lpr` on only one host at UNM, whereas we traced it on 77 hosts at MIT. Despite the differences in environment, the patterns of database growth in the UNM environment are similar to those at MIT (data not shown), although the resulting database sizes are different: 569 unique sequences for UNM and 876 for MIT. These databases not only differ in size, but also in content: For example, a comparison of the unique sequences in both databases for $k = 6$ indicates that only 141 of the sequences are the same between the databases, which represents 40% of the UNM database and 29% of the MIT database."

Section 5.3.

Added paragraph 3 on page 26:

"Are these really false positives? A false positive is some sort of acceptable behavior that is classified as anomalous. If the behavior is unacceptable, even if it is not caused by an intrusion, we would want to know about it, because it indicates that the system is not functioning properly or efficiently. Points 1 and 4 above are both instances of irregular behavior symptomatic of a problem with the system; both indicate conditions that need to be rectified. In this sense, neither 1 nor 4 are false positives. This kind of analysis indicates that our actual false positive rate is lower than the reported values, for example, in the case of a chronological normal, the number of false positives would be reduced from 6 to 3."

Section 6.

Added the last sentence to paragraph 2 on page 26:

"Recently, we have successfully detected intrusions in two other programs: a buffer overflow in the `xlock` program running in Linux, and a symbolic link vulnerability in the `swinstall` program running under HP-UX."

Added paragraph 5 on page 27:

"Our emphasis has been on determining if our approach can be successful at all. We were not too concerned with efficiency issues in this paper. However, for the system to be able to detect intrusions in real-time---as they are happening---will require careful attention to efficiency issues. As a first step towards this we have analyzed the complexity of our algorithm, although we have not been able to measure its efficiency in a production environment. Should the implementation prove too inefficient, there are numerous simplifications we could experiment with, such as looking only at specific kinds of calls, or only at every tenth call, etc."

Added paragraph 9 on pages 28-29:

"Our approach is similar to several other approaches, although the differences are critical. Ko et al [30] have also chosen the level of privileged processes, but they characterize the behavior of a privileged process by a program specification or policy, which is a description of what the program should be able to do. This policy is derived from the program code and so requires specialized knowledge of program function. Writing a policy can be prone to the same sorts of errors as writing the program, i.e. it is hard to guarantee correctness. Most importantly, from our perspective, such a policy could easily include behavior that is legal but not normal because it is hard to determine beforehand what behavior should be normal. We avoid these issues by treating the program as a black box, and relying purely on empirical observation to ascertain program behavior. Another key difference is that we rely exclusively on sequencing information, unlike the specification approach, which monitors individual operations. However, there are other approaches, such as TIM [39], that consider sequencing information. These differ from our approach in that they look at the domain of user behavior, and use a probabilitistic approach for detecting anomalies. Because our results are sufficiently promising the added complexity of using probabilities seems unnecessary. It is possible that our simple deterministic approach is successful because our data is well-structured. If this is the case, it may well be that probabilities are necessary in less structured domains, such as user behavior."