Available at
www.ElsevierMathematics.com
POWERED BY SCIENCE DIRECT®
Computational Statistics & Data Analysis 45 (2004) 51–67
COMPUTATIONAL STATISTICS & DATA ANALYSIS
www.elsevier.com/locate/csda

Statistical anomaly detection via httpd data analysis

Daniel Q. Naiman[1]

Department of Mathematical Sciences, Johns Hopkins University, Baltimore, MD 21218, USA

Received 2 January 2002; received in revised form 15 July 2002

Abstract

The use of *statistical anomaly detection* for the purpose of computer intrusion detection is introduced, and a data analytic approach to the problem is illustrated. A preliminary study of the http daemon is used to demonstrate some of the more elementary yet informative investigations that can be carried out. We describe a sequence of findings of regularities in processes that the daemon creates, and in the system calls that comprise these processes that seem potentially useful in constructing an intrusion detection system based on traces of the daemon. General tools for studying the process traces are described.

Keywords: Anomaly detection; System calls; Process trace

1. Introduction

From a statistician's point of view, the problem of computer intrusion detection is quite attractive for a variety of reasons. To be told of the existence of a problem of as great practical importance as protecting the information infrastructure (which, incidentally, happens to include our own data) from compromise, and that the solution to the problem requires data analytic skill, it is difficult for a data analyst not to feel compelled to get involved. There are other quite practical aspects of the problem that make it appealing. Data for investigation are quite easy to come by, and in fact, anyone administering a computer that provides a service over a network to several users is likely to be able to collect vast quantities of data in just a matter of days. Also, the field is very much in its infancy, and just about any plausible proposal for

[1] This research was supported in part by a grant from the National Science Foundation, Grant No. DMI-0087032, and a grant from the Naval Surface Warfare Center, Grant No. N00178-99-1-9004.
E-mail address: daniel.naiman@jhu.edu (D.Q. Naiman).

0167-9473/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/S0167-9473(03)00115-4

analyzing data merits attention as it might have the potential for producing interesting new observations and to lead to a better understanding of the intrusion detection issue. A case will be made here that some of the most basic information to be gleaned from these data is waiting to be discovered using simple statistical ideas, as opposed to complex data mining and pattern recognition techniques. In addition, many of the approaches taken to analyze data appearing in the literature emphasize *machine learning* techniques, which leaves some people with traditional data analytic views wondering whether there remains a strong possibility that we might benefit a great deal from *human learning.* Statisticians, with their emphasis on data analysis and modeling, have an opportunity to contribute in a meaningful way to the development of successful approaches to deal with this problem.

There are two related but distinct approaches to computer intrusion detection referred to in the current literature: *misuse detection* and *anomaly detection*. In misuse detection, also referred to as *signature-based detection* known methods for hacking into computers are studied and categorized, and characteristic patterns of these methods are determined and stored. A detector is devised that monitors computer activity and attempts to recognize characteristics of intrusion immersed in data representing normal activity. On the other hand, anomaly detection involves the collection of data representing normal activity (without misuse) and the development of statistical models describing typical patterns of variation in this activity. A detector then is devised based on monitoring the system for general classes of deviations from normal behavior.

Thus, a critical difference between the two approaches is in the degree to which known misuse information is relied upon. One would expect that implementation of a misuse detector would lead to better defense against known types of attacks, while anomaly detectors would be more effective against newly devised attacks. The former approach would typically require a better understanding of actual attacks, while the latter approach is more statistical in nature. It is likely that the most successful intrusion detection systems will involve the conjunction of both methods.

While the focus here is on the anomaly detection of web-based intrusions, there are many other potential attacks on a computer host in which anomaly detection ought to prove useful. For example, anomaly detection could be potentially used in order to identify a intruder masquerading as a legitimate user, exploitation of vulnerabilities in a mail server, and corruption of program code. Web-based intrusion is of particular interest because, for so many institutions, allowing some level of web service to clients from the outside is necessary, and opening up this service leads to a concomitant desire to reduce the risk of attack.

2. Tracing process system calls

Statistical analysis of system trace data has been used in many approaches to computer anomaly detection. It is expected that an intruder will exploit vulnerabilities in a particular program in order to gain access to functionality that is not intended for users by the programmer. The exploited program will execute system calls that are not commonly seen when the program is operating normally and, consequently, it is

```
28323: execve("/usr/mts/sbin/httpd", 0xEFFFF8AC, 0xEFFFF8BC)  argc = 3
28323: stat("/usr/mts/sbin/httpd", 0xEFFFF608) = 0
28323: open("/dev/zero", O_RDONLY) = 3
28323: mmap(0x00000000, 4096, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0xEF7B0000
28323: open("/opt/gnu/lib/libm.so.1", O_RDONLY) Err#2 ENOENT
28323: open("/usr/local/lib/libm.so.1", O_RDONLY) Err#2 ENOENT
28323: open("/usr/lib/libm.so.1", O_RDONLY) = 4
28323: fstat(4, 0xEFFFF3D4) = 0
28323: mmap(0x00000000, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE, 4, 0) = 0xEF780000
28323: mmap(0x00000000, 155648, PROT_READ|PROT_EXEC, MAP_PRIVATE, 4, 0) = 0xEF740000
28323: munmap(0xEF755000, 61440) = 0
28323: mmap(0xEF764000, 6054, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE|MAP_FIXED, 4, 81920)
28323: close(4) = 0
28323: open("/opt/gnu/lib/libdl.so.1", O_RDONLY) Err#2 ENOENT
28323: open("/usr/local/lib/libdl.so.1", O_RDONLY) Err#2 ENOENT
28323: open("/usr/lib/libdl.so.1", O_RDONLY) = 4
28323: fstat(4, 0xEFFFF3D4) = 0
28323: mmap(0xEF780000, 4096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED, 4, 0) = 0xEF780000
28323: close(4) = 0
28323: open("/opt/gnu/lib/libsocket.so.1", O_RDONLY) Err#2 ENOENT
28323: open("/usr/local/lib/libsocket.so.1", O_RDONLY) Err#2 ENOENT
```

Fig. 1. Sample of the output when a process is traced.

hoped that a trace of the program will reveal to the computer operator that an attack is underway.

Particularly, interesting approaches to intrusion detection are described in Forrest et al. (1996), Hofmeyr et al. (1998), and Warrander et al. (1999) (see also the references cited therein). The idea is to record the sequence of system kernel calls associated with a given process (a capability available in most flavors of UNIX), statistically model the normal behavior of the process, and detect anomalies based on sufficient evidence that new data could not plausibly have been produced under such a model. Among the ideas considered in these papers is that of modeling the occurrences of new contiguous sequences of n system calls (*n-grams*), the motivation being that occurrences of sufficiently many new n-grams in some localized time frame constitutes evidence of innovative behavior. In this context, we are motivated to ask whether any plausible statistical model could be devised that appropriately describes the data being observed, since fitting such models allows one to set critical thresholds for intrusion detection systems having predictable error rates. In fact, an early goal of this investigation was to determine if the importance sampling methodology described in Naiman and Priebe (2001) could be applied successfully to the intrusion detection problem.

Traces of process system calls produce output with varying amounts of information. A portion of output using `ptrace`, the Solaris version of the trace command, is shown in Fig. 1.

Each line of trace output starts with numerical identifier for the process that is responsible for the system call. This identifier is followed by a description of the system call actually made together with its arguments in parentheses, and the last bit of information is a system return value. The analyses of system calls that have been suggested in the literature as well as our analyses make use only of the names of the system calls and their process identifiers.

```
3132: fork() = 3136
3136: fork() (returning as child ...) = 3132
3136: getpid() = 3136 [3132]
3136: brk(0x00039870) = 0
3136: brk(0x00039C70) = 0
.
.
.
3136: unlink("/tmp/countfile-3130") = 0
3136: fchdir(3) = 0
3136: lseek(0, 0, SEEK_CUR) Err#29 ESPIPE
3136: _exit(0)
```

Fig. 2. Trace of process forking a new process.

To complicate matters, the initial process being traced can spawn new processes. These *subprocesses* are always created using the `fork()` system call, and end with an `_exit()`. For example, the trace output shown in Fig. 2 shows a process being forked. The output indicates that process number 3132 spawns process 3136 which performs a sequence of operations then dies as indicated by the execution of an `_exit()`.

At any given time a parent process might have several live descendants, and some of the apparent variability in structure of traced system calls can be resolved by accounting for this phenomenon.

3. A study of httpd system calls

Given the ubiquity of computer networks, and the degree of reliance upon them, the problem of protecting computer and network infrastructures from unwanted use has become recognized as a critical one.

An http daemon is a program that allows a web site to operate as one would hope. It is meant to run without further interaction by the host system and provide web service, processing valid requests by visitors (clients) to the site, while refusing invalid ones. A request can take several forms: downloading of a file, e.g. a web page, an image file, a document, etc. In more sophisticated applications, a client might cause the server to initiate a process that results in some action taking place on the server. For example, data provided by the client in some form might be added to a database. A simple and perhaps familiar example occurs when a server records the number of visits to a site and then displays that information on the web page. Thus, a process may be initiated that modifies data in the server's file system. This in turn might lead to a complex chain of responses across the network. It is instructive to try to imagine what the consequences are of pressing the button that causes a credit card transaction, or the transfer of money from a financial institution to billing agency, to commence.

In an effort to develop an anomaly detection system based on httpd system calls, we traced the httpd daemon running on the web server at the Johns Hopkins University's Department of Mathematical Sciences over a period of about 3 months from May through August of 2001. This web server uses the http software version 1.3.2 from

Table 1
Number of processes and system calls obtained in each run (trace) of the http daemon

Data set no.	No. of processes	No. of system calls
1	2096	2,519,011
2	698	2,211,242
3	690	1,995,029
4	799	1,513,370
5	646	1,848,217
6	539	1,835,507

```
128.220.30.179  [01/May/2001:12:36:13 -0400] "GET /mts/image...
141.157.91.101  [01/May/2001:12:36:16 -0400] "GET /~userdir/311...
141.157.91.101  [01/May/2001:12:36:24 -0400] "GET /~userdir/311...
128.220.30.179  [01/May/2001:12:36:25 -0400] "GET /mts/under...
```

Fig. 3. Sample entries from the http daemon log files.

the Apache Software Foundation, and runs it on a Sun Sparcstation. The Apache Software Foundation claims that their software is the most popular server software on the internet, representing about 60% of web servers participating in a recent survey. Even if a considerably smaller percentage of servers used this program, its ubiquity makes their software an ideal choice for the first study of this kind. Further analysis should focus on more recent versions of the software and other web servers that perform more services than are provided by our department's server.

The data were collected in six batches. Each time a large enough batch is collected, the server is shut down, the trace is stored. The server trace begins upon startup and this initial activity is recorded. The six data sets consist of between 1.5 and 2.6 million system calls each and there are 63 system calls that are found to be in use by the program. Table 1 gives the number of processes and system calls obtained in each trace.

Additional information with requests to the httpd daemon is recorded when the server is active. Each time a user downloads a page from the web site, a line is appended to the daemon's log file consisting of the user's internet host identifier, the date and time at which the request was made, and the specific file requested (see Fig. 3). Initially, when this project was undertaken, it was our intention to attempt to match up information in these responses to requests with the processes appearing in the program trace. Unfortunately, despite the fact that the tracing program as well as the access logs provide time stamps, the matching of specific system calls to requests does not appear to be a simple matter.

This raises an important question that helps to shed some light on the philosophy of our approach. Why not examine the program's source code to answer this question? Certainly, if we have access to it then we would have a great deal more information about when the access log is written, and which functions are carried out in the meantime. In fact, without altering the functionality of the program in any appreciable way one could create alternative methods for tracing the behavior of the program that record

activity on a level higher than at the system call level. However, while it is the case that the httpd source code is currently available, we would like to avoid examining it, because we are interested in determining the extent to which intrusion detection can be achieved under general conditions, and generally program source code is not readily available.

While on the subject of source code, to the uninitiated it might be tempting to dismiss the data analytic approach to the intrusion detection problem and advocate devoting all of ones resources to designing secure programs and operating systems. The data analytic approach certainly has its drawbacks, but there are good reasons to take the view that secure programming is unlikely to provide a complete solution to the problem in the foreseeable future. For one thing, it is unlikely that programmers will conform to a formal set of rules deemed to lead to secure programs, and even if they can be convinced to do so, the resulting problem of determining if the system arrived at is *secure* is likely to be undecidable (see Harrison et al., 1976; Landwehr, 1981).

4. Process pedigrees

Since there is a system call pedigree, a natural first step in obtaining a *big picture* view of the httpd daemon's behavior is to examine the family tree for a process. If we presume that an intruder will exploit weaknesses in a program to spawn a new shell, from which to engage in further exploits, then the new process would appear as a subprocess of the program being run, and hence would appear in the process tree. Detection might be enabled by finding obvious anomalies in the process tree, provided we attain some idea of the usual tree structure.

A simple tool for studying process pedigrees has been developed by the author and is based on the following idea. Create a directory in the filesystem that is associated with the original process, proceed recursively to create a create a subdirectory of the directory corresponding to the parent process each time a new child process is spawned. Once the directories and files are written, the process directory can be examined using the UNIX `tree` command (in conjection with `more`) which lists the contents of a directory in a tree-like format. Fig. 4 gives a sample of what the output looks like.

In addition to having a directory associated with a subprocess, additional identifying information about a process can be incorporated into the name of a file placed in the directory corresponding to the process. For example, we would like to get some idea of the number of system calls there are for the various processes, and how these numbers vary depending on how deep the processes are in the process tree. For each child process, an empty file is created whose name is of the form `CALLS_n`, where n is the number of system calls made by that process. The first httpd process for which data have been collected consists of 2087 processes, so it is not possible to show the complete output here, but a partial listing provides one with a feel for the idea. Fig. 5 shows what the output looks like as a result of adding this additional bit of information.

Already some regular patterns begin to emerge from this analysis. The httpd daemon creates a root process with only 431 associated system calls (not visible in Fig. 5) and

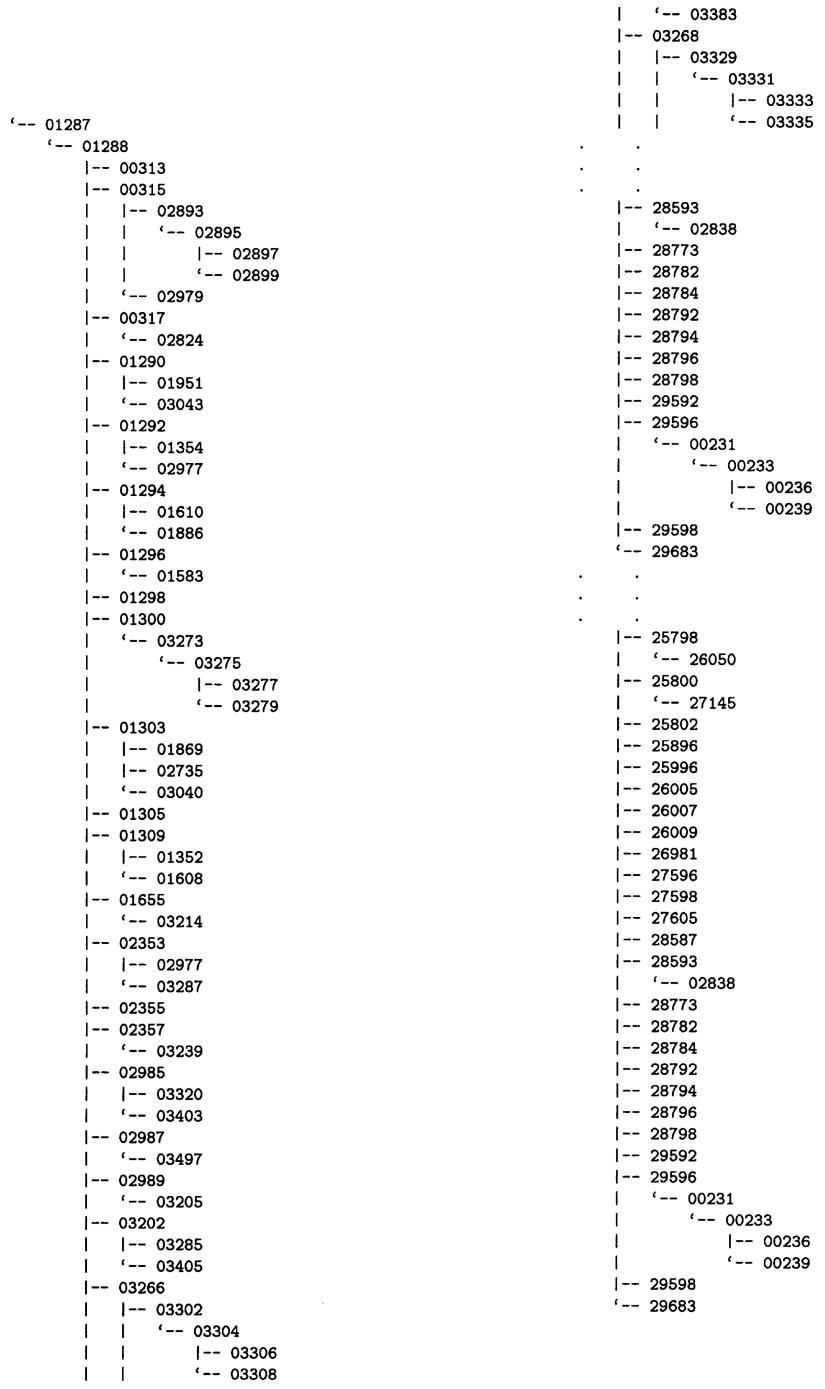

Fig. 4. Process pedigree for httpd trace (portion of second data set).

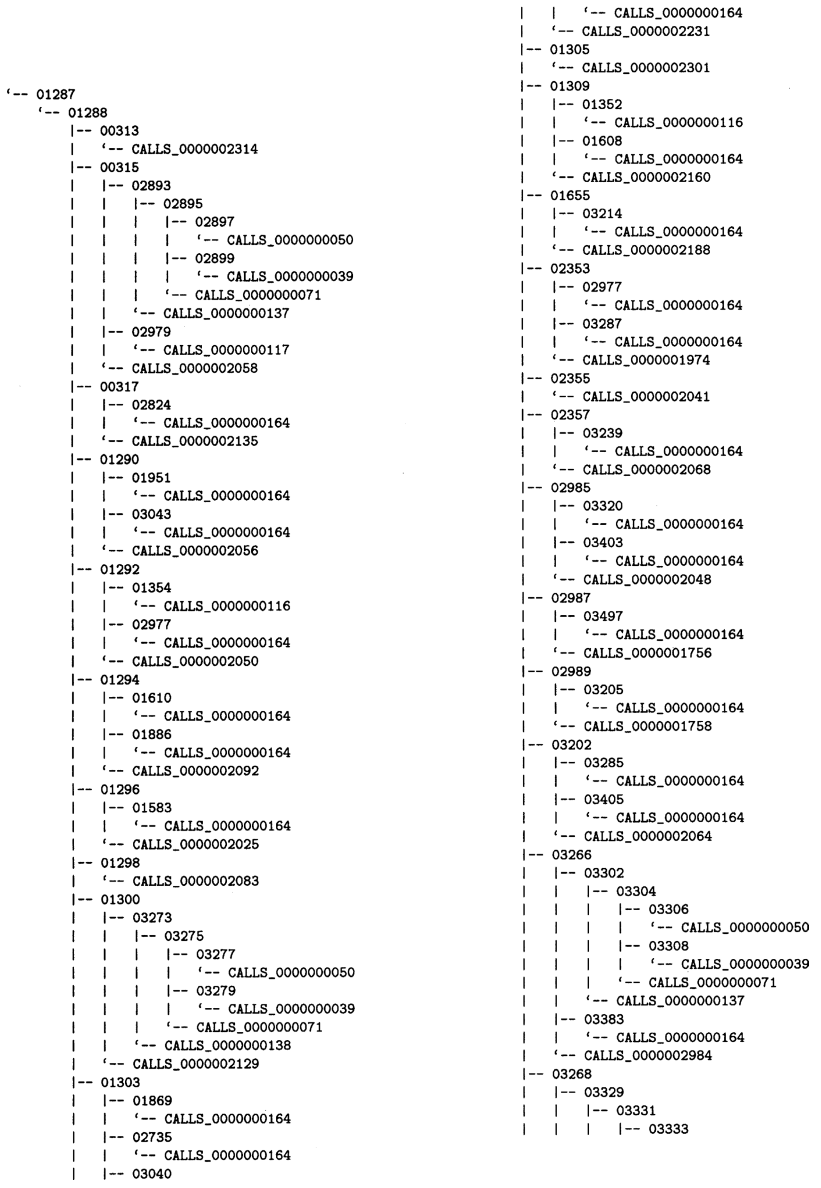

Fig. 5. Process pedigree for httpd trace including process lengths (in system calls).

it spawns a unique subprocess which is then an ancestor to every subsequent process and has roughly a half-million system calls associated with it. The separation between descendants and the root process is at most five generations.

The term *depth* will be used to refer to the number of generations that separate a process from the root process. Examination of the full process tree reveals immediately

Table 2
Frequencies of system calls at depth 3 of various lengths

Length	116	117	118	137	138	139	140	148	164	166	167	178	205
Frequency	5	37	40	22	174	281	5	2	100	342	2	1	2

a strong degree of determinacy in the structure of the tree at higher depths. The processes at depth 5 always come in pairs with one of the pair consisting of 50 system calls and the other consisting of 39 system calls. Of the 484 times these process pairs appear, 423 times their common parent always consists of 72 system calls, and 61 times their parent consists of 71 system calls, and this accounts for all of the processes at depths 4 and 5. The calls at depth 3 (great grandchildren) come in a relatively small number of sizes. Table 2 gives the frequency of lengths (in system calls) of all 1013 processes at depth 3.

Clearly, the number of calls for a process at higher depths are quite constrained. These regularities uncovered lead immediately to some rather natural detection rules. If a process

- appears at depth 4 or 5 having a number of system calls that has not been seen before,
- appears at depth 5 with no sibling,
- appears at depth greater than 5,

then it ought to be viewed with suspicion.

On the other hand, there is quite a bit of variation in the lengths of the 2991 depth 2 processes (grandchildren); the length distribution for these processes is quite smooth in appearance and bimodal. This distribution is shown in Fig. 6.

5. Sequence regularity

Having uncovered evidence of common process *sizes*, in terms of system calls, it is natural to investigate the degree of variation in the sequences. For this purpose, we store each distinct process, which is now viewed as a word (typically huge) in a 63 system call alphabet, and the number of times it appears in the datasets we explore. Our analysis treats identical system calls in succession as a single one. The decision to do this is somewhat arbitrary and an analysis of the repeated calls would probably lead to similar results. Interestingly, we find that only 1877 of the 5456 processes (at depth 2 and above) obtained appear only once in the exact same form, while the other 3579 of them appear multiple times, leaving us with a collection of only 1959 distinct processes.

Nearly all of the processes appearing exactly once are grandchildren of the root process. In fact, at depth 3 there are only four processes that are unique and at depth 4 there is one. As mentioned above, depth 5 has only two processes. So we see that the vast majority of processes for which there is any variation appear at depth 2. Again, lack of variation can be used to suggest strategies for eliminating processes to study

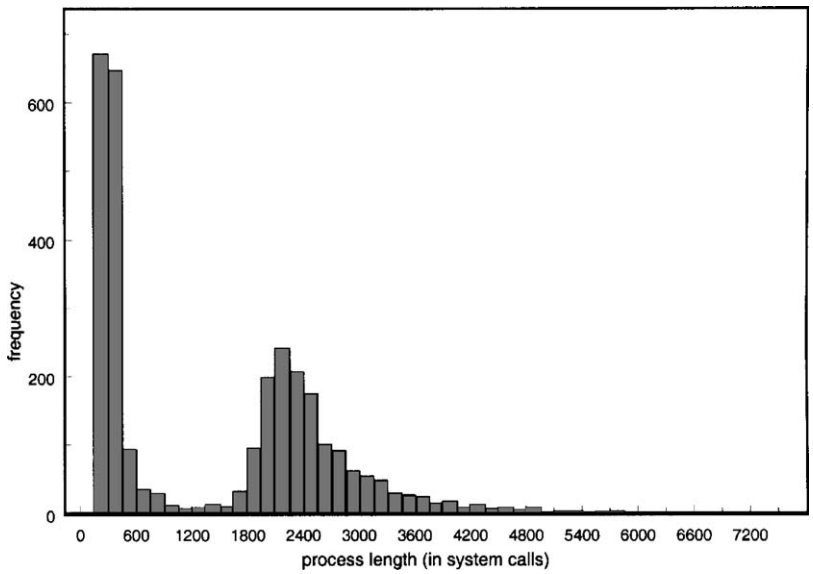

Fig. 6. Distribution of lengths (in system calls) for processes at depth 2.

for attack. Processes that are found in a table listing repeaters can be deemed *safe* and it is probably best to focus attention on processes that deviate from these. That is, because we anticipate being able to discard as non-malicious, any process that appears in exactly the same form multiple times, we narrow our focus further to the processes at this depth that appear in exactly the same form only once.

The distribution of lengths of the 1872 uniquely appearing depth 2 processes is shown in Fig. 7. The distribution is smooth and bimodal, and while the appearance of the distribution is similar to the process length distribution, there is a tendency for the smaller-length processes to repeat, as shown by the fact that the height of the peak in Fig. 6 has diminished considerably.

A natural next step in the analysis is to investigate variation in the uniquely occurring processes. We begin by asking where the variation in the processes resides. To answer this, consider the number of distinct system calls that are found in each possible position in the process. Fig. 8 shows how this number varies with the system call position. Here system calls found in succession that are of the same type are interpreted as a single call. Not surprisingly, the first system call in every process is exactly the same. What is somewhat surprising is that it is not until the 19th system call in a process that the sequence of calls starts to vary. For every process, the first 18 system calls take the form `fork, siggaction, getpid, getuid, sysconfig, open, fstat, mmap, munmap, mmap, close, munmap, waitid, open, fstat, ioctl, read, and lseek.`

Even with the variation in the possible system calls introduced at position 19, it remains the case that for each of the first 146 positions, there are at most two possible system calls to be found, and even then the system call in 40 of the positions from 33

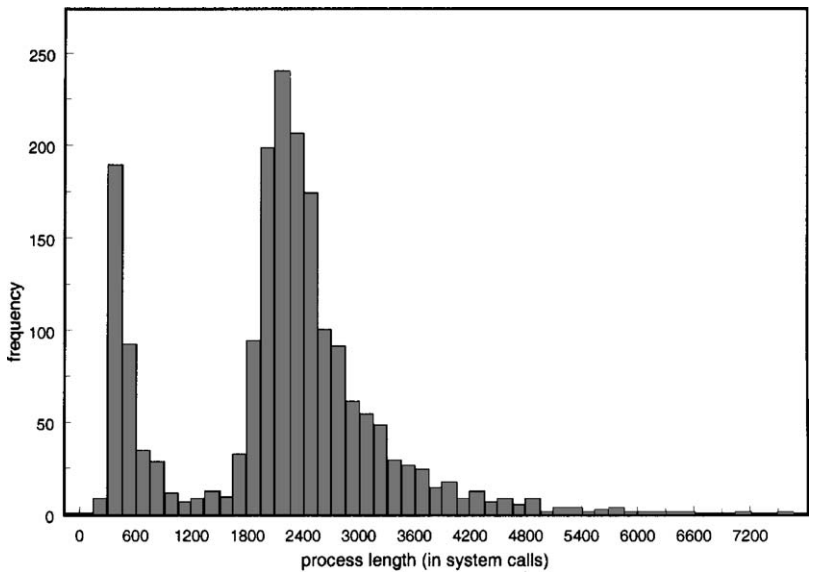

Fig. 7. Lengths of processes (in system calls) at depth 2 that appear exactly once.

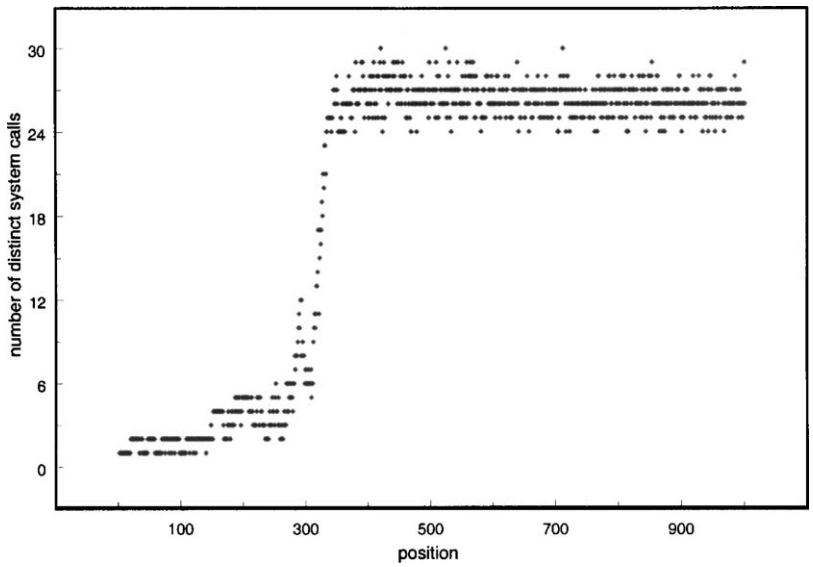

Fig. 8. Number of possible system calls in each position for processes at depth 2 appearing only once.

through 139 are unique. In fact, the system calls in positions 38 through 45, 58 through 68, and 100 through 107 are the same for every process. A dramatic climb in system call variability takes place around 300 calls into the sequences and the variability remains roughly the same from then on. Apparently, it would be rather difficult to hide

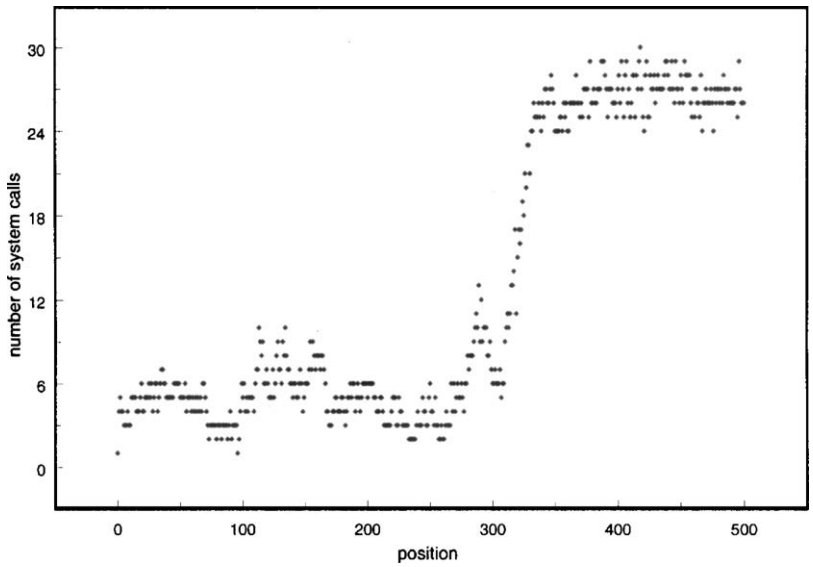

Fig. 9. Number of possible system calls in each position for all processes.

a process from detection if traces of deviation from normalcy appear in the early stages of the processes lifetime. This observation suggests that a study of intrusions ought to be carried out to determine whether the process in which they are embedded can be constructed in such a way that the intrusion appears later.

The degree to which this set of processes is restricted leads one to wonder if this property is shared by *all* of the httpd processes. Fig. 9 shows number of system calls in each position for all such processes. It remains the case that a relatively small number of possible system calls (mean of 5, standard deviation of 2) can be found in any of the first 300 positions. The 97th system call is always `ioctl`. Also, and the same dramatic growth in the number of possible system calls occurs a bit after the 300th position. Still, there is considerably greater variation than is found in the restricted set of processes described above. Again it appears that by focusing on processes that we have not been able to eliminate by other means, we have uncovered new sources of determinacy, and additional criteria for detection could be introduced based on this discovery.

6. Modeling what is left

Having eliminated a large portion of the data for investigation, we are left with the problem of making sense of the variability that exists in processes at depth 2 that appear without identical copies, and including only the system calls appearing beyond the 330th position. From this point on we investigate this chunk of data, which will be referred to as the *remaining* data.

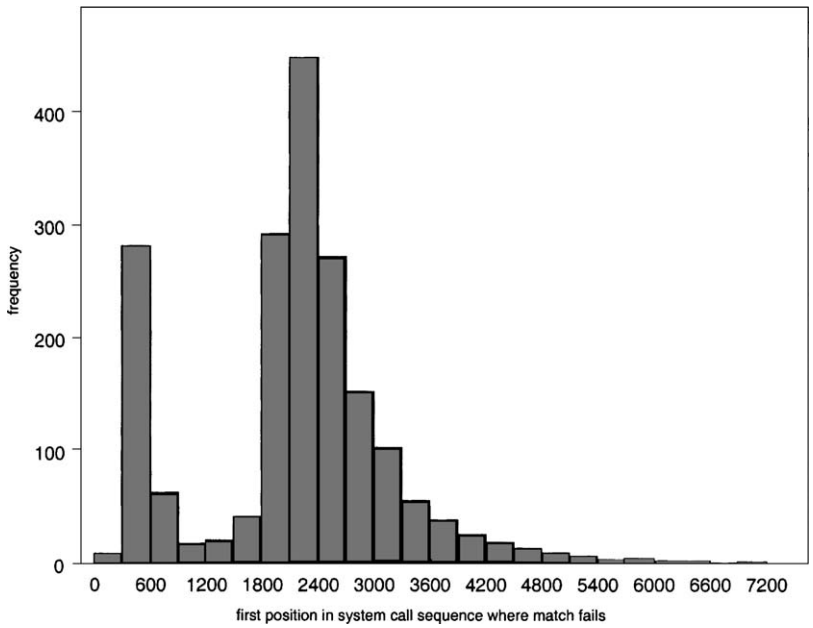

Fig. 10. Distribution of position in the unique process where the sequence of system calls fails to match with that of some process appearing multiple times.

Some preliminary observations about the remaining data can be made. Since the processes have been selected on the basis of their uniqueness, it is natural to wonder at what point in their lifetime they begin to deviate from processes that appear at multiple times in the database. For each of the remaining processes, the position where its system call sequence first fails to match that of some process appearing multiple times in the data set is calculated. The distribution of these positions is given in Fig. 10. For 80% of the remaining processes, there is some process that appears multiple times for which the call sequence matches in the first 1000 positions. The number of possible places to hide anomalous system calls in the early stages of a process appears to be relatively small.

The remaining processes are comprised of roughly 3.5 million system calls but only 34 of the 63 possible system calls actually appear. Table 3 gives a listing of these system calls and their frequencies. We see that a small number of system calls appear rather rarely and most system calls are quite common.

If one were to attempt to base an intrusion detection on the occurrence of rare system calls, one ought to attempt to determine the degree to which rare calls are clustered in normal data. In an attempt to answer this question, we compute for each process how many times each of the rarest 7 system calls `fork`, `_exit`, `brk`, `getpid`, `kill`, `alarm`, and `sigsuspend` appears in a sequence. The distribution of this number of times is given in Table 4.

There are some processes with a relatively large number of rare system calls, so we investigate these for evidence of clustering. The process having 17 rare calls contains

Table 3
Frequencies of system calls for the remaining data

System call	Frequency	Rel. freq.
stat	146,573	0.0428
open	300,347	0.0877
mmap	38,499	0.0112
fstat	189,277	0.0553
munmap	38,615	0.0113
close	293,143	0.0856
brk	42	0.0000
ioctl	414,447	0.1211
read	318,689	0.0931
lseek	238,185	0.0696
getpid	501	0.0001
sigprocmask	300,064	0.0876
poll	87,905	0.0257
getmsg	83,876	0.0245
fcntl	94,736	0.0277
time	177,901	0.0520
door_info	66,088	0.0193
door_call	66,100	0.0193
chdir	91,912	0.0268
fork	1010	0.0003
_exit	1910	0.0006
write	159,483	0.0466
sigaction	231,874	0.0677
waitid	1018	0.0003
getcontext	11,696	0.0034
kill	16	0.0000
setcontext	11,780	0.0034
writev	29,396	0.0086
getdents	14,160	0.0041
pipe	1010	0.0003
alarm	24	0.0000
sigsuspend	8	0.0000
door	10,992	0.0032
context	2288	0.0007

Table 4
Number of processes with a given number of rare system calls

No. of rare calls	0	1	2	3	4	5	6	7	8	9	10	11	14	15	17
No. of processes	12	988	503	200	92	36	19	5	7	4	1	2	1	1	1

14 occurrences of fork, two occurrences of brk and one exit. A plot of the positions of these calls appears in Fig. 11 and it is clear from this that the rare calls are close to uniformly distributed in the sequence of system calls for the process. Identical observations can be made about the processes with 10, 14 and 15 rare system calls. If

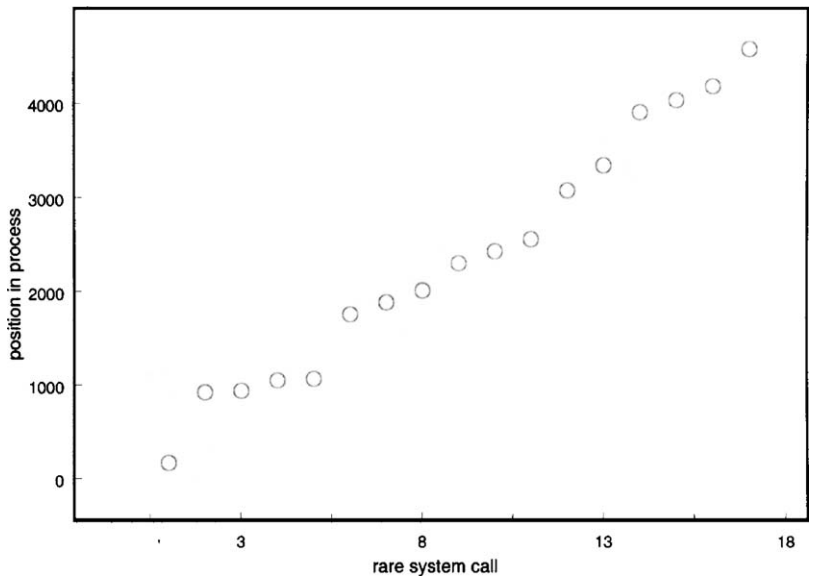

Fig. 11. Positions where rare system calls appear for the process with 17 rare system calls.

Table 5
Number of system calls with each possible number of successors

Number of successors	1	2	3	4	5	6	7	8	9	10	12	14	15	21
Number of system calls	4	5	5	3	3	3	2	1	1	1	2	1	1	1

an intrusion detection system is based on a large number of new n-grams localized in the process sequence, then these anomalies are not likely to trigger false alarms.

It is natural to wonder whether a Markov chain with state space consisting of the system calls, would appropriately describe the remaining processes. If such a model is valid, the transition matrix is relatively sparse. Among the 33 system calls that have successors (`_exit` has none), on average 5.7 possible successors are observed. Four of the system calls (`door_info`, `fork`, `get_context`, and `pipe`) have unique successors, so that the state space can be reduced in size to 28. One of the system calls `close` has 21 possible successors. Table 5 gives the distribution of the number of successors for the 33 system calls. We see that many system calls have relatively few successors and at the same time, many have a relatively large number of successors.

To determine how the system call frequencies depend on their position in the overall process sequences, the data were broken up into strips consisting of 500 system calls. The relative frequency of each system call is computed in each range of positions: positions 330–829, 830–1329, and so on, and these relative frequencies are plotted using a different color for each of the 34 possible system calls in Fig. 12. From the figure, it seems clear that the system call frequency remains constant in each range.

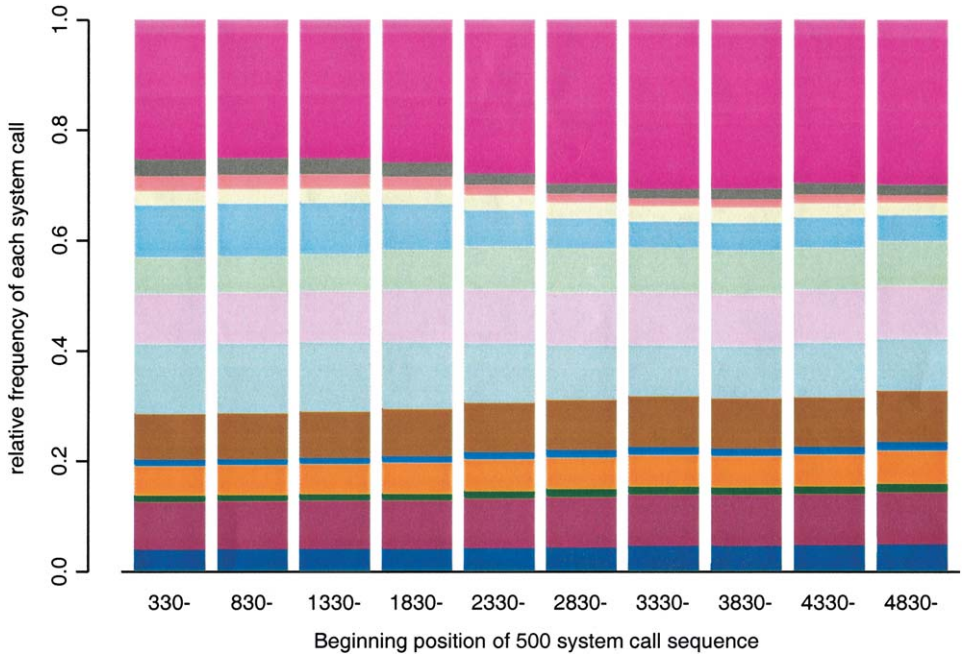

Fig. 12. Evidence of stationarity: frequencies of system calls in each strip consisting of 500 calls for the remaining data.

If a Markov chain model were to fit the data, the assumption of stationarity appears to be a reasonable one.

Unfortunately, attempts to fit a Markov chain model of some order to the data have not been successful, and there is strong evidence that a different sort of stochastic model involving brief bursts of randomness and long stretches of determinism seems more appropriate. The author hopes to investigate the fitting of *hidden Markov models* (see Rabiner and Juang, 1986; Charniak, 1993), or their more exotic relatives, such as *stochastic context-free grammars* (Sakakibara et al., 1994; Jelinek et al., 1992), or in devising new alternative models. Discussion of these further modeling efforts will be the subject of a future paper.

7. Conclusions and future directions

Data analysis has been applied to rather large data sets produced by tracing the httpd daemon, and many observations have been made as a result that seem pertinent to the design of an intrusion detection system, and to the probabilistic analysis of an intrusion detection system. In terms of design, regularities have been discovered that create a context for describing what would naturally constitute what is meant by irregularity. In terms of probabilistic analysis of intrusion detection system, particularly

in setting of error rates, such analysis ought to rely upon evidence that regularities of a statistical nature exist for the process under investigation. Thus far, while some statistical regularities seem present in the processes' system calls we are far from having constructed what appears to be a valid statistical model. One clear conclusion that can be drawn from the analyses presented is that it seems difficult to hide anomalous processes if they alter the system call sequence early in a normally occurring process. Tracing some known attacks should lead to insights into the extent to which hiding anomalous processes is possible.

Acknowledgements

The author wishes to acknowledge David Marchette at NWSC Dahlgren for introducing him to the problem and many discussions, Jonathan Shapiro from Johns Hopkins Computer Science Department for his occasional insights, and Jeff Solka at NWSC Dahlgren for his encouragement.

References

Charniak, E., 1993. Statistical Language Learning. MIT Press, Cambridge, MA.

Forrest, S., Hofmeyr, S.A., Somayaji, A., 1996. A sense of self for unix processes. Proceedings of the 1996 IEEE Symposium on Security and Privacy. IEEE Computer Society Press, Los Alamitos, CA.

Harrison, M.A., Ruzzo, W.L., Ullman, J.D., 1976. Protection in operating systems. Commun. ACM 19, 461–470.

Hofmeyr, S., Forrest, S., Somayaji, A., 1998. Intrusion detection using sequences of system calls. J. Comput. Security 6, 151–180.

Jelinek, F., Lafferty, J.D., Mercer, R.L., 1992. Basic methods of probabilistic context free grammars. In: P. Laface, R. De Mori (Ed.), Speech Recognition and Understanding. Recent Advances, Trends and Applications. NATO ASI, Springer, Berlin, pp. 345–360.

Landwehr, C.E., 1981. Formal models for computer security. ACM Comput. Surveys 13, 247–278.

Naiman, D.Q., Priebe, C.E., 2001. Computing scan statistic p-values using importance sampling with applications to genetics and medical image analysis. J. Comput. Graph. Statist. 10, 296–328.

Rabiner, L.R., Juang, B.H., 1986. An introduction to hidden Markov models. IEEE ASSP Magazine 3, 4–16.

Sakakibara, Y., Brown, M., Hughey, R., Saira Mian, I., Sjölander, K., Underwood, R.C., Haussler, D., 1994. Recent methods for RNA modeling using stochastic context-free grammars. In: M. Crochemore, D. Gusfield (Ed.), Proceedings of the 5th Annual Asilomar Symposium on Combinatorial Pattern Matching, Springer, Berlin, pp. 289–306.

Warrander, C., Forrest, S., Pearlmutter, B., 1999. Detecting intrusions using system calls: alternative data models. 1999 IEEE Symposium on Security and Privacy. IEEE Computer Soc. Press, Silver Spring, MD, pp. 133–145.