# Temporal Signatures for Intrusion Detection

Anita Jones
jones@virginia.edu
*University of Virginia*

Song Li
mail2ls@yahoo.com
*University of Massachusetts*

## Abstract

*We introduce a new method for detecting intrusions based on the temporal behavior of applications. It builds on an existing method of application intrusion detection developed at the University of New Mexico that uses a system call sequence as a signature. Intrusions are detected by comparing the signature of the intrusion and that of the normal application. But when the system call sequences generated by the intrusion and the normal application are sufficiently similar, this method cannot work. By extending system call signature to incorporate temporal information related to the application, we form a richer signature. Analysis shows that the temporal behavior for many applications is relatively stable. We exclude high variance data when creating a normal database to characterize an application with a temporal signature. It can then be the basis for future comparisons in an intrusion detection system. This paper discusses experiments that test the effectiveness of the temporal signature on different applications, alternative intrusions, and in various environments. The results show that by choosing appropriate analysis methods and experimentally adjusting the parameters, intrusions are readily detected. Finally, we give some comparisons between the temporal signature method and the system call method.*

Keywords: security, application intrusion detection, temporal signature

## 1. Introduction

Modern computer systems are vulnerable to numerous intrusions. Both the long standing UNIX buffer overflow flaw and the recently successful denial of services attacks illustrate that applications and operating systems harbor many security flaws. The use of Intrusion Detection Systems (IDS) to assure security is based on the assumption that a system will not be secure, but that violations of security policy (intrusions) can be detected by analyzing system behavior [1]. Anomaly detection techniques assume that all intrusive activities are necessarily anomalous. This means that given a "normal activity profile" for a system, intrusion attempts produce behavior that varies from the established profile by statistically significant amounts [2, 3].

An anomaly detection method created at the University of New Mexico defines sequences of system calls created by a particular application to be its signature [1, 4]. Anomalous behavior of that application, hopefully, produces system call sequences that are different from those of the normally executing application. Of course, if all system call sequences produced by an intrusion are a subset of those of the application executing normally, then the intrusion cannot be detected.

The University of New Mexico method depends on the creation of a database for each application that consists of call sequences that reflect normal activity of that application. To detect intrusions, sequences in this *normal database* are compared with the system call sequences generated by monitored runs of the application.

Hamming distance, which is a measure of the distance or difference between two system call sequences, is used as the criteria for determining abnormal from normal signatures.

## 2. Temporal signature introduced

Our work builds on the University of New Mexico method; we use sequences of system calls, but also incorporate timing properties to create signatures that more richly and uniquely related to the application. We have experimented with four different time measure definitions. In this paper, we discuss the measure that we consider the most straightforward and most effective. That measure is the time duration between sequence calls. Time measurements are on a per process basis and are made so that only the time duration (between calls) that the process is actually executing is measured. Context swap and process sleep time are elided. For convenience, we will denote this time measurement as *r*.

The elapsed time between two system calls can vary because the path through the application code can vary. So, we take a number of measurements for each unique sequence. We compute time distributions for each time interval between two calls in a unique sequence. Then we build the normal database containing signatures that are – in effect – summaries of multiple measurements.

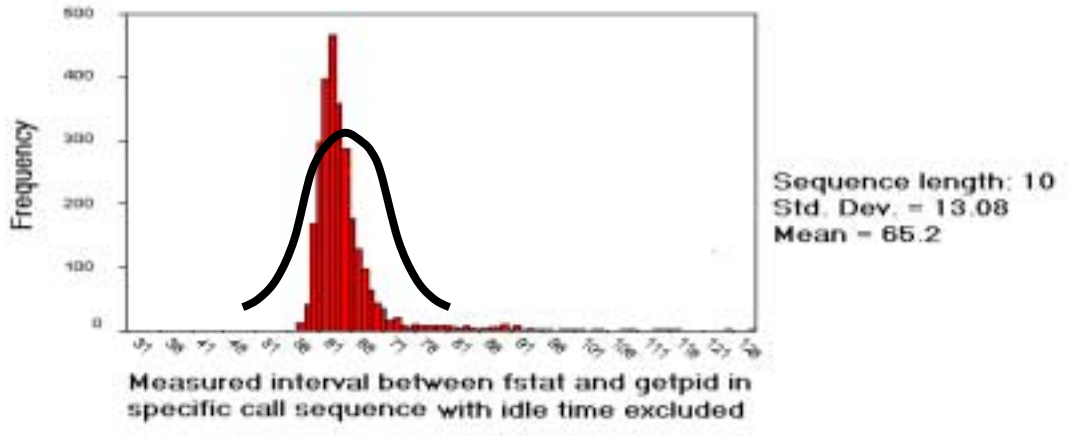


Figure 1. The frequency distribution of the time intervals associated with a particular pair of adjacent system calls.

## 3. Building the raw database

In this section we describe how a raw database is constructed, and in the next section we describe how to refine it to be a normal database that can be considered a signature representative of the application. Our discussion will focus on the construction of a single element in the database.

We monitor application execution multiple times either in a production environment, or when synthetically stimulating the application, with a variety of inputs. That execution generates a sequence of system calls. Given a sequence length *k*, we can pass a window across the recorded sequence and detect all unique sub-sequences of length *k*. We will develop a database element for each such *k*-length sequence.

In the following explanation, we use an illustrative system call sequence of length 6:

open, fstat, seteuid, socket, setsockopt, bind.

During the time that the database measurements are taken, there are typically many cases in which this unique sequence of calls is detected. We collect the multiple *cases* of time measurements related to the unique call sequence into groups called *sequence clusters*. Each case is a sequence of time interval measurements.

Table 1 illustrates one sequence cluster. The first row representing the system call sequence is called the *title* of the cluster. All other rows represent one time interval sequence or case. Table 1 has a total of twelve cases, i.e. the sequence in the title was measured 12 times.

The raw *temporal signature* database is defined as a collection of sequence clusters, or clusters. There is a different database of temporal signatures for each application.

Table 1. Illustration of one sequence cluster composed of one title and 12 cases.

| System interval sequence | calls | open | fstat | seteuid | socket | setsockopt | bind |
|---|---|---|---|---|---|---|---|
| Case 1 | 108 | 24 | 22 | 43 | 21 | 27 | |
| Case 2 | 84 | 23 | 23 | 37 | 21 | 27 | |
| Case 3 | 84 | 23 | 22 | 37 | 23 | 27 | |
| Case 4 | 84 | 24 | 22 | 38 | 21 | 27 | |
| Case 5 | 116 | 25 | 23 | 41 | 22 | 28 | |
| Case 6 | 85 | 23 | 23 | 40 | 23 | 28 | |
| Case 7 | 87 | 25 | 23 | 41 | 23 | 29 | |
| Case 8 | 87 | 24 | 24 | 42 | 23 | 29 | |
| Case 9 | 114 | 24 | 23 | 43 | 22 | 28 | |
| Case 10 | 83 | 23 | 23 | 41 | 24 | 27 | |
| Case 11 | 120 | 24 | 50 | 40 | 22 | 27 | |
| Case 12 | 98 | 23 | 23 | 41 | 22 | 27 | |

The database of an application might be large or relatively small. Usually, the signature of an application is generated by one or more executions of the application. Sometimes, several sequence clusters for an application are created at different times. Two clusters for the same application can be merged to create a new cluster. The new cluster is the union of all the clusters that share a common title. Two clusters can be merged only if their sequence lengths are identical.

Next we consider the time elapsed between two system adjacent calls in a cluster. Figure 1 illustrates that the

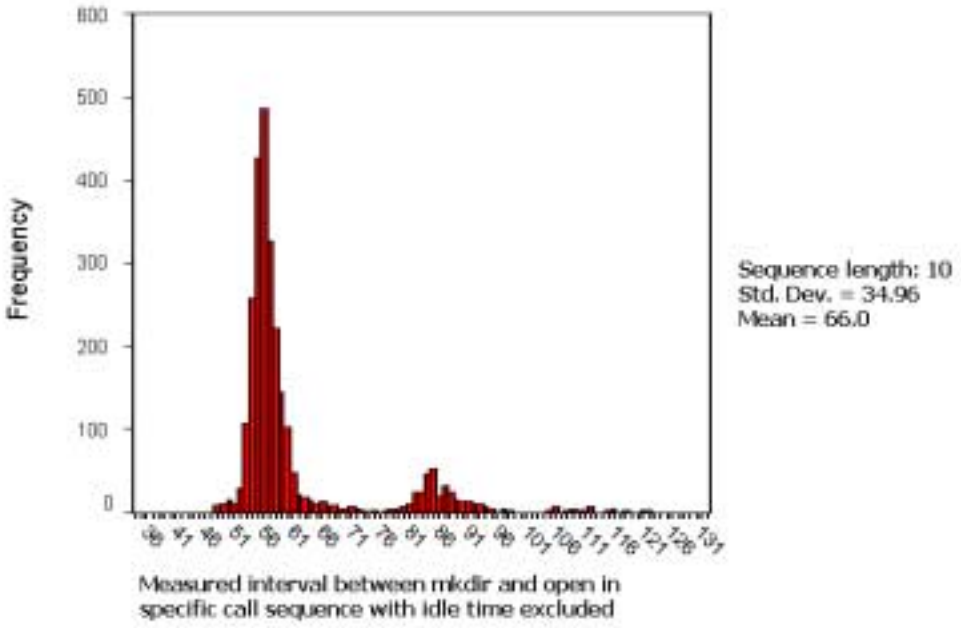


Figure 2. The frequency distribution of the time intervals with two peaks.

time intervals corresponding to the interval between *fstat* and *getpid* in a particular cluster, i.e. one column of a sequence cluster. In Figure 1, the x axis indicates the time interval duration, measured in μs, between the two successive system calls. The y axis is the number of the occurrences of each value of a particular time interval value. So, for about 400 times, the interval measured was 61(μs). Figure 1 depicts 2832 time interval measures. Note that the time intervals between the pair of calls deviate from one another very little. We found this to be typical.

However, in some situations we observed distributions with multiple peaks. Figure 2 shows two clear peaks, and both of them are similar to normal distributions. We observed multiple peaks (usually two) in only 3-10% of the cases.

The reason for such two-peak distribution of time intervals is likely because two different execution paths through the application code that exhibits the same system call sequence.

Because some time intervals exhibit large deviations and are not well distributed, the raw cluster summaries should not be used directly. High variance data should be removed. The next step is to reduce the raw sequence cluster data illustrated by Table 1 to a summary form We refer to the process as qualifying the database.

## 3. Qualification of cluster summaries and derivation of the normal database

First, we calculate an initial summary of the time interval sequences. We define a *cluster summary* to be the combination of the title of the sequence cluster, the *mean vector* of all the cases, and the *standard deviation vector* of all the cases.

In Table 2, the initial cluster summary appears at the bottom of the table. The row labeled *m* is the vector of mean values. The row labeled *s* is the vector of standard deviations.

In Table 2, most standard deviations are small because most case values are quite close to their associated mean values. However, note that four time intervals diverge greatly from their corresponding mean. Three are associated with system call *open*: 116, 114, 120 (cases 5, 9, 11, respectively), and also the value 50 for case 11 and system call *seteuid*. Analysis shows that most such high variance data are due to I/O related system calls, as is the case here with *open*. Basically this occurs for I/O devices whose actions rely on (slow) mechanical components. However, not all I/O related system calls would result in high variance time intervals.

Table 2. Time intervals in a cluster, *m* is the mean of time intervals in each column, *s* is the standard deviation of these time intervals

| System interval calls sequence | open | fstat | seteuid | socket | setsockopt | bind |
|---|---|---|---|---|---|---|
| Case 1 | 108 | 24 | 22 | 43 | 21 | 27 |
| Case 2 | 84 | 23 | 23 | 37 | 21 | 27 |
| Case 3 | 84 | 23 | 23 | 37 | 21 | 27 |
| Case 4 | 84 | 24 | 22 | 38 | 21 | 27 |
| Case 5 | 116 | 25 | 23 | 41 | 22 | 28 |
| Case 6 | 85 | 23 | 23 | 40 | 23 | 28 |
| Case 7 | 87 | 25 | 23 | 41 | 23 | 29 |
| Case 8 | 87 | 24 | 24 | 42 | 23 | 29 |
| Case 9 | 114 | 24 | 23 | 43 | 22 | 28 |
| Case 10 | 83 | 23 | 23 | 41 | 24 | 27 |
| Case 11 | 120 | 24 | 50 | 40 | 22 | 27 |
| Case 12 | 98 | 23 | 23 | 41 | 22 | 27 |
| *m* (mean) | 95.8 | 23.8 | 25.2 | 40.3 | 22.1 | 27.6 |
| *s* | 14.6 | 0.75 | 7.84 | 2.06 | 0.996 | 0.793 |

## 3.1 Removing high variance (unusable) data

We want to remove high variance. Based on the normal distribution assumption of the time intervals, we develop a quantitative method for the exclusion of high variance data. We use three steps: remove high variance cases, exclude data for high variance system calls, and mark unusable clusters.

**3.1.1 Removing high variance cases.** First, we remove high variance cases. Consider case 11 that has the single divergent time measure associated with *seteuid* that varies greatly from the mean. Based on the normal distribution assumption of time intervals, the probability of high variance time intervals is low, therefore to remove the high variance value, we exclude the entire case 11 so that it does not contribute to the cluster summary.

To determine criteria for excluding such data, we use a common notation for the variance of each interval, called the *z-score* [13]. It is the number of standard deviations that an interval is from its corresponding mean. Let $t_i$, $m_i$, and $s_i$ represent the time interval, mean, and standard deviation of the $i^{th}$ system call. Then the *z-score* can be expressed as:

$$z_i = (t_i - m_i)/s_i$$

The *z-score* is a balanced and normalized time interval. It will satisfy the standard normal distribution, which is the normal distribution with a mean of zero and standard deviation of one. The *z-scores* can thus be compared to each other. Furthermore, given a certain probability criteria of being normal or abnormal, there is a corresponding value, which can be compared with the *z-scores*. For example, if we regard all data that will appear with probability less than 5% as abnormal, then we can compare the *z-score* with 2.33 which is the maximum permitted z value with 5% abnormal cases. If the *z-score* of a time interval is larger than 2.33, it will be regarded as abnormal. Because the time intervals within one case are related to each other, whenever one time interval is excluded, the whole case will be excluded.

Table 3 shows the mean vector, the standard deviation vector and the max *z-score* for each case. The gray cell (for *seteuid* case 11) is unusable data (because the *z*-score z is large) and case 11 will be removed in the next iteration.

Table 3. Given mean *m* and standard deviation *s*, each interval in a column has a corresponding *z-score*, which is supposed to satisfy the standard normal distribution.

| System interval calls sequence | open | fstat | seteuid | socket | setsockop | bind | z |
|---|---|---|---|---|---|---|---|
| Case 1 | 108 | 24 | 22 | 43 | 21 | 27 | 1.29 |
| Case 2 | 84 | 23 | 23 | 37 | 21 | 27 | 1.62 |
| Case 3 | 84 | 23 | 23 | 37 | 21 | 27 | 1.62 |
| Case 4 | 84 | 24 | 22 | 38 | 21 | 27 | 1.13 |
| Case 5 | 116 | 25 | 23 | 41 | 22 | 28 | 1.66 |
| Case 6 | 85 | 23 | 23 | 40 | 23 | 28 | 0.99 |
| Case 7 | 87 | 25 | 23 | 41 | 23 | 29 | 1.79 |
| Case 8 | 87 | 24 | 24 | 42 | 23 | 29 | 1.79 |
| Case 9 | 114 | 24 | 23 | 43 | 22 | 28 | 1.29 |
| Case 10 | 83 | 23 | 23 | 41 | 24 | 27 | 1.92 |
| Case 11 | 120 | 24 | 50 | 40 | 22 | 27 | 3.17 |
| Case 12 | 98 | 23 | 23 | 41 | 22 | 27 | 0.99 |
| *m* | 95.8 | 23.8 | 25.2 | 40.3 | 22.1 | 27.6 | 0 |
| *s* | 14.6 | 0.75 | 7.84 | 2.06 | 0.996 | 0.793 | N/a |

The mean and standard deviation for each column will change after exclusions are made. So, *m s* and *z-score* are recalculated. It is possible that this recalculation will need to be performed several times because additional intervals might satisfy the criteria of high variance intervals. Such exclusions should be performed iteratively, until no case is to be excluded. Typically, two or three iterations will be sufficient to exclude all the high variance cases. In our example, one iteration is sufficient.

**3.1.2 Excluding data of high variance system calls.** After excluding high variance cases the possibility of high

variance systems call data should be considered. The standard deviation of the *open* column in Table 3 remains high. Again, I/O (disk access) is typically the source of high variance (columns). Network operations and, in rare cases, code algorithms may exhibit high variance. (Timing of such system calls should not be used as the basis of the comparison because their high variance.) So we remove the timing values of such system calls entirely from the cluster summary.

Although we do not want the high variance timing data to contribute to the cluster summary, we do not remove the system call, *open*, from the title in order to maintain a single sequence length for all clusters.

Because experimental data shows that the standard deviation is usually higher when the mean is higher, we compute a *normalized standard deviation*. We define $C_s$ to be a threshold that *s/m* cannot exceed. $C_s$ is called the *criteria of high variance system calls*. Any system call with normalized standard deviation larger than $C_s$ will be regarded as a high variance system call. This model is experimentally based and $C_s$ is an adjustable parameter. Typically, $C_s$ takes a value in the range of .1 to .3 based on experimental observation.

Table 4 shows the final values for *m'*, *s'*, *s'/m'*, and *z-score* after all the high variance cases have been removed. The column for *open* in Table 4 has been shaded gray to show a high variance system call values that will be elided.

Table 4. High variance case 11 has been removed. *m, s, s/m* are recomputed.

| System interval calls sequence | open | fstat | seteuid | socket | setsockopt | bind | z´ |
|---|---|---|---|---|---|---|---|
| Case1 | 108 | 24 | 22 | 43 | 21 | 27 | 1.69 |
| Case 2 | 84 | 23 | 23 | 37 | 21 | 27 | 1.56 |
| Case 3 | 84 | 23 | 23 | 37 | 21 | 27 | 1.69 |
| Case 4 | 84 | 24 | 22 | 38 | 21 | 27 | 1.69 |
| Case 5 | 116 | 25 | 23 | 41 | 22 | 28 | 1.62 |
| Case 6 | 85 | 23 | 23 | 40 | 23 | 28 | 0.92 |
| Case 7 | 87 | 25 | 23 | 41 | 23 | 29 | 1.69 |
| Case 8 | 87 | 24 | 24 | 42 | 23 | 29 | 2.02 |
| Case 9 | 114 | 24 | 23 | 43 | 22 | 28 | 1.22 |
| Case 10 | 83 | 23 | 23 | 41 | 24 | 27 | 1.83 |
| Case12 | 98 | 23 | 23 | 41 | 22 | 27 | 0.92 |
| *m´* | 93.6 | 23.7 | 22.9 | 40.4 | 22.1 | 27.6 | 0 |
| *s´* | 13.02 | 0.786 | 0.539 | 2.16 | 1.044 | 0.809 | n/a |
| *s´/m´* | 0.139 | 0.033 | 0.024 | 0.053 | 0.047 | 0.029 | n/a |

Using $C_s$ as a criterion is a simple method for determining what system call time measures to exclude. There are more formal and strict statistical methods to check whether the distribution of a sample is a normal distribution or not, for example, using the equation $W=\Sigma(t_j-m)^2/n$ which observes the chi-square distribution. We did not use this approach, however, because our model does not satisfy all the assumptions required to apply such methods. Instead of the strict methods, we used the relatively simple model and empirical criteria.

The exclusion of the timing information of high variance system calls will not affect the calculation of mean and deviation for other system calls, so they do not need to be recalculated. Note that the high variance cases must be excluded before the exclusion of high variance system calls. If we exclude the high variance system calls first, those columns with just one or several abnormal time intervals in the columns will be excluded as high variance system calls, while typically such columns are still usable.

**3.1.3 Mark unusable clusters.** Our experiments have shown that typically, only a small percentage of the total cases and system calls are excluded. However, it is necessary to check whether exclusions have made the whole cluster unusable because of insufficient valid data.

We define the percentage of valid cases for a cluster to be *cluster* $P_v$, i.e. the percentage of cases remaining after high variance cases are excluded. We can also compute a similar $P_v$ value for an entire database or for a comparison between clusters and a normal database. The *database* $P_v$ is the percentage of valid cases in a normal database compared to the total number of cases in the raw database. The *comparison* $P_v$ is the percentage of valid cases in a monitored behavior of an application compared to the number in the normal database of the application. Whenever the cluster $P_v$ is lower than a threshold due to the above exclusions, we mark the cluster as unusable. We call this threshold as $T_v$. It is empirically defined.

## 3.2 Put it all together

After all high variance exclusions are made, the cluster is typically found to be usable. It is referred to as being a *qualified cluster summary*. The normal database consists of all qualified cluster summaries for the application. Recall that each cluster summary has a distinct title. Each database is then characterized by the following parameters:

- set of unique titles
- sequence length characterizing the database
- database $P_v$
- $C_s$, criteria for excluding invalid cases

Table 5 shows three different example cluster summaries for *wu.ftpd*. The annotation, "*", on *open* indicates that

the system call *open* is a high variance system call. During intrusion detection no comparisons will be made to the mean and standard deviation values for *open* in the summary; they are non-numeric. The third summary shows the cluster for a cluster that has been marked unusable. Note that the title of the unusable cluster is maintained for comparison purposes.

Table 5. Three cluster summaries from the normal database for application *wu.ftpd*.

| system calls | open* | fstat | seteuid | socket | setsockopt | bind |
|---|---|---|---|---|---|---|
| m | x | 23.7 | 23.7 | 40.4 | 22.1 | 27.6 |
| s | x | 0.786 | 0.539 | 2.16 | 1.044 | 0.809 |

| system calls | seteuid | read | read | fstat | write | fcntl |
|---|---|---|---|---|---|---|
| *m* | 32.3 | 23.7 | 22.5 | 30.4 | 52.1 | 22.8 |
| s | 1.02 | 1.786 | 1.539 | 0.16 | 2.02 | 0.34 |

| system calls* | msgget | read | sbrk | write | open | dup |
|---|---|---|---|---|---|---|
| m | x | x | x | x | x | x |
| s | x | x | x | x | x | x |

Table 6 shows a summary of the properties that characterize four different normal databases for the application *wu.ftpd*. We used different *z-scores* and sequence lengths. The database $P_v$ values indicate that fewer high variances are encountered when the *z-score* is larger and when sequence lengths are shorter.

Table 6. Summary of a normal database for application *wu.ftpd*.

| Sequence length | $z_a$ | Time interval class | Database $P_v$ |
|---|---|---|---|
| 6 | 2.33 | *r* | 93.4% |
| 10 | 2.33 | *r* | 78.8% |
| 6 | 2.58 | *r* | 95.1% |
| 10 | 2.58 | *r* | 84.2% |

## 4. Intrusion detection systems

At this point we have defined the temporal signature and described in detail how the normal database might be constructed to be a robust, usable signature for the application. We next experimented with database construction and then with building an experimental intrusion detection system (IDS) in order to test its efficacy in detecting intrusions using the temporal signature.

The IDS consists of two major parts: the normal database builder and the run-time monitor. The normal database builder creates the normal database for future comparison as described earlier. The run-time monitor observes the execution of the application much in the same way that the application was monitored in order to build the normal database. When it finds a sequence of length *k* with the associated inter-call timing measurements, the monitor has (effectively) a case. It compares the case to the database to see if there exists a sequence summary whose title is the same as the *k*-length sequence. For the full or partial title match, the monitor computes the difference between the monitored case and the most relevant cluster summary. Based on that comparison, it makes a decision whether or not anomalistic behavior is occurring. We will discuss the details of this comparison in the context of experimentation.

## 5. Experiments and results

The major objective of our experiments is to validate the hypotheses stated in previous sections: we can build a normal temporal signature database; and we can detect intrusions. We also discuss the effects of the environment on our experiments, the effectiveness of our system, and the selection of parameter settings.

The first step is to actually construct the normal database. There are two construction methods: "synthetic" normal and "real" [12]. A "synthetic" normal database is generated by exercising the application in as many normal modes as possible while tracing its behavior. A "real" normal database is generated by tracing the normal behavior of the application in a live user environment [12].

A synthetic normal database is quite useful when there is a need to replicate results and to compare performance using different parameter settings. A real normal database is more problematic to collect and evaluate because of the difficulties of avoiding abnormal sequences (intrusions during construction) and determining if the traced behavior is sufficiently comprehensive. However, we need a real normal database to determine how our system is likely to perform in realistic environments.

Our experiments were performed using FreeBSD. We built both synthetic and real normal databases. We experimented with the following applications: *wu-ftpd*, a server implementing FTP, the File Transfer Protocol; *popper*, a mail server implementing POP3, the Post Office Protocol; *delegate*, a proxy server; and *htdig*, a html search engine.

In our experiments, we must select settings for the following adjustable parameters:

- $T_v$, threshold of the percentage of valid cases in a cluster.

- $C_s$, criteria of high variance system call.
- *z-score*, criteria for the high variance cases.
- *sequence length*, the length of the sequence cut from the stream of system calls.

Database $P_v$ serves as an indicator whether our database is acceptable for IDS use. Because no other criterion is available at this time, we judge the goodness of a normal database based on empirical analysis. Besides these parameters, we also measured the database $P_v$ when we build the normal database and then use it as the criterion for intrusion detection

All these parameters were defined and discussed in detail in previous sections. Typical settings of the parameters are:

- $T_v$,: *85%*;
- $C_s$: 0.1~0.3;
- *z-score*: 2.33 or 2.58
- *sequence length*: 6~10;

The first parameter value was determined based on the experiments we performed. The *z-score* was determined based on our assumption that no more than 5%-10% high variance data was to be allowed. That is, we assumed that less than 5% (for *z-score* 2.33) or 10% (for *z-score* 2.58) high variance data deviate from the mean by this value (after database normalization).

The selection of sequence length is also an important issue. Some early works showed that sequence lengths of 6 to 10 are effective choices [2, 17]. We adopted these suggestions and develop our own preference in following sections.

## 5.1 Results of building normal databases

We built both the "synthetic" normal database and the "real" normal database for the applications, *wu-ftpd* and *popper*. We generated the synthetic normal databases by building a script that invoked every command that the application defines. Each command is invoked multiple times with parameters designed to exercise the full functionality of the application. *wu-ftpd* has 23 commands and *popper* has 10 commands. The summaries for the several generated normal databases for *wu-ftpd* and *popper* are discussed in detail in the next section.

**5.1.1 Building normal databases in a synthetic environment.** Table 7 summarizes the qualities of four slightly different normal databases for *wu-ftpd*. As mentioned earlier we timed different kinds of time intervals – e.g. between system calls and within system calls. All data in this paper are for the measurements of *r*, between system calls. All were generated using the same (synthetic) script. Each line in the table represents a database using different parameter settings. Database $P_v$ values are higher for the shorter sequence length, and when the *z-score* is higher and admits more case variance. The database $P_v$'s are the percentage of retained cases after the qualification of cluster summaries. High $P_v$ values indicate the quality of the databases. Database $P_v$'s using time interval class *r* were higher than when measuring time elapsed during system calls.

Table 7. Synthetic normal database of *wu-ftpd* using time interval classes *r*, $C_s$ = 2.0, $T_v$ = 85%.

| Sequence length | $z_a$ | Time interval class | Database $P_v$ |
|---|---|---|---|
| 6 | 2.33 | *r* | 93.4% |
| 10 | 2.33 | *r* | 78.8% |
| 6 | 2.58 | *r* | 95.1% |
| 10 | 2.58 | *r* | 84.2% |

Our second set of database construction experiments involves the application *popper*. It runs as a daemon. *popper* (based on POP3, the Post Office Protocol) performs many fewer commands than those of the *wu-ftpd*. However, each command has a variety of parameters. To explore all the commands with different parameters, we use two kinds of clients, MS Outlook and Netscape Messenger, to logon, quit, receive, retrieve attachment, list, and delete mail on the server. The following list suggests the ranges of parameter settings:

- mail size: less than 10bytes – 1Mbytes;
- with/without attachment;
- with/without subject;
- amount of mail on the server: 0-1000;
- delete/keep the mail when retrieving;
- correct/incorrect logon on the server.

Table 8 describes the qualities of 4 different synthetic databases for *popper*. Each line in the table represents a database. Again we explored all time interval classes although they are not shown here.

Table 8. Synthetic normal database for *popper* using time interval class *r*, $C_s$ = 2.0, and $T_v$ = 85%

| Sequence length | $z_a$ | Time interval class | Database $P_v$ |
|---|---|---|---|
| 6 | 2.33 | *r* | 90.1% |
| 10 | 2.33 | *r* | 80.8% |
| 6 | 2.58 | *r* | 96.4% |
| 10 | 2.58 | *r* | 88.2% |

The results in Table 8 are similar to those for *wu-ftpd*; all the $P_v$'s in Table 8 show a good quality normal database.

**5.1.2 Building normal databases in a real environment.** We generated the "real" normal databases for the

application *wu-ftpd* by executing the application in an actual, open environment (the computing environment in the Computer Science Department, University of Virginia), while monitoring the environment carefully to ensure that no intrusions occurred during our data collection.

Table 9. Statistics for four real normal database of *wu-ftpd* using time interval class *r*, $C_s$ = 2.0, and $T_v$ = 85%

| Sequence length | $z_a$ | Time interval class | Database $P_v$ |
|---|---|---|---|
| 6 | 2.33 | *r* | 95.4% |
| 10 | 2.33 | *r* | 82.8% |
| 6 | 2.58 | *r* | 93.1% |
| 10 | 2.58 | *r* | 85.2% |

Table 10. Statistics for four real normal database of *popper* using time interval class *r*, $C_s$ = 2.0, and $T_v$ = 85%

| Sequence length | $z_a$ | Time interval class | Database $P_v$ |
|---|---|---|---|
| 6 | 2.33 | *r* | 93.9% |
| 10 | 2.33 | *r* | 85.1% |
| 6 | 2.58 | *r* | 92.5% |
| 10 | 2.58 | *r* | 84.6% |

Results in Table 9 and Table 10 illustrate results that are similar to those for the synthetic normal databases. All the $P_v$'s in Table 9 and Table 10 show a good quality database. We can also see that in both normal databases, using sequence length 6 is better than sequence length 10.

There is an argument in favor of using longer sequence lengths. Assume that we measure *r*, the duration of application code execution between system calls. Consider two application code sequences, A and B, that invoke the same sequence of three system calls, C1, C2, and C3 followed in one case by C4 and in the other by C5. Assume that in Figure 3 the length of the horizontal line between calls indicates the measured execution time interval between calls.

A: C1---------C2----------C3------C4
B: C1---------C2----------------------------------C3----------C5

Figure 3.

If the sequence length is 4, then the titles for A and B are different and they are in different clusters. However, if the sequence length is 3, then A and B have the same title: C1-C2-C3. The time interval between C2 and C3 is differs greatly for A and B. Therefore, the shorter sequence length results in combining cases that may lead to higher standard deviation. And that may cause cases to be excluded because of high variance. That in turn lowers $P_v$. Hence, this argues in favor of longer sequence lengths.

In contrast there is an argument in favor of using shorter sequence lengths with fewer system calls. There is less probability that one interval measure will result in a case being removed for high variance. This argues that shorter sequence lengths should yield higher $P_v$ values.

Actual experimental evidence leads us to conclude that short sequences are better. So we used only a sequence length of 6 for experiment in the intrusion detection experiments that are described next.

## 5.2 Intrusion detection

To determine how effective our method is in actually detecting intrusion, we build a prototype IDS of the form described in section 4. The monitor generated cases for the application and compared each case to relevant cluster summaries. We used the same criteria for high variance cases to decide whether a case matched or mismatched the cluster summary in the normal database. We define $P_m$ to be the percentage of matched cases. We define anomaly to be detected when $P_m << P_v$.

We detected intrusions for the applications *wu-ftpd*, *popper*, and *delegate* and compared the results of the first two applications with the University of New Mexico's system call results. We also performed an intrusion detection experiment to show that our method is *not* applicable to some applications.

**5.2.1 Detecting intrusion into *wu-ftpd*.** Because of improper bounds checking, it is possible for an intruder to overwrite static memory in certain configurations of the *wu-ftpd* daemon. The overflow occurs in the MAPPING_CHDIR portion of the source code and is caused by creating directories with carefully chosen names [14]. We exploited the flaw in order to time intruding codes which will try to decrypt the password in the /etc/shadow file.

We performed the comparison between the signatures of the intrusion behavior and those in the real normal database of *wu-ftpd*. That real normal database was generated using the following parameters: sequence length: 6; time interval class: *r*; $C_s$ = 2.0; and $T_v$ = 85%.

Table 11. Intrusion detection using two real normal databases for *wu-ftpd*

| Sequence length | $z_a$ | Time interval class | Database $P_v$ | $P_m$ |
|---|---|---|---|---|
| 6 | 2.33 | *r* | 93.4% | 82.7% |
| 6 | 2.58 | *r* | 95.1% | 79.2% |

Table 11 shows the $P_m$ value for the intrusion behavior compared to $P_v$ value determined when the database was constructed. The $P_m$ values in Table 11 are much lower than the corresponding database $P_v$'s. This indicates that intrusion is detected

**5.2.2 Detecting intrusion into *popper*.** There is a buffer overflow flaw in the *popper* [16]. We performed a intrusion similar to the above and constructed two real normal databases for *popper* using the same parameters as for *wu-ftpd* database construction.

Table 12. Intrusion detection using two real normal databases for *popper*

| Sequence length | $z_a$ | Time interval class | Database $P_v$ | $P_m$ |
|---|---|---|---|---|
| 6 | 2.33 | *r* | 93.9% | 84.7% |
| 6 | 2.58 | *r* | 92.5% | 81.2% |

Table 12 shows two $P_m$ values are significantly lower than corresponding $P_v$ values. This indicates intrusion. We also performed the same intrusion and used the system call method [12].

Table 13. Intrusion detection fails for the system call method

| Application | *popper* | *wu-ftpd* |
|---|---|---|
| Normal | 0.8% | 1.3% |
| Buffer overflow | 1.1% | 1.0% |

The sequence length used in the experiment is 6. The percentages are the ratio of the number of mismatched sequences to the total sequences in the normal database of system call sequences. In Table 13, the percentages for normal behavior of the applications and the behavior under buffer overflow attack exhibit little difference. We can see that in this case the system call method cannot detect the intrusion effectively. This is because that the intrusion did not launch any abnormal system calls; it generated mostly cases that matched the cluster summaries in the normal databases.

**5.2.3 Detecting intrusion into *delegate.*** *delegate* is a versatile application-level proxy, including the http proxy. For our *delegate* experiment, we built the normal database by monitoring two days of regular use of the proxy server, including the *http* and *ftp*, in the Computer Science department at the University of Virginia. We employed a buffer overflow attack to successfully intrude into *delegate*. We caused an input buffer of *delegate*'s *strcpy* to overflow. Again, we inserted code that performed simple intensive calculation.

Table 14. Intrusion detection using two normal databases for *delegate*

| Sequence length | $z_a$ | Time interval class | Database $P_v$ | $P_m$ |
|---|---|---|---|---|
| 6 | 2.33 | *r* | 91.5% | 71.7% |
| 6 | 2.58 | *r* | 95.4% | 73.3% |

Table 14 shows the $P_m$ value for the intrusion behavior compared to $P_v$ value determined when the database was constructed. The $P_m$ values in Table 14 are much lower than the corresponding database $P_v$'s. Therefore, intrusion is detected. This demonstrates an example intrusion in which the system calls alone are an insufficient signature to detect the intrusion, but the temporal signature can be used to detect intrusion.

**5.2.4 Detecting intrusion into *htdig.*** There are also some intrusions our method cannot detect. When an intrusion does not incur any time intensive computation, the temporal signature when the intrusion occurs will not be different – from the point of view of timing – from the normal database. An intrusion on *htdig*, a small html search engine, provides such an example.

The program *htdig* was set up to allow for file inclusion from configuration files. Any string surrounded by the opening single quote character ( ` ) is interpreted as a path to a file for inclusion. *Htdig* will also allow included files to be specified via the form input method in the http protocol. Therefore, any file can be specified for inclusion into a variable by any web user and that file will thus be shown on the web page. As a result, the client can read any file on the server, which can be read by the invoker of Apache [15].

Table 15. Comparison without intrusion.

| Sequence length | $z_a$ | Time interval class | Database $P_v$ | $P_m$ |
|---|---|---|---|---|
| 6 | 2.33 | *r* | 95.0% | 93.5% |
| 6 | 2.58 | *r* | 90.1% | 91.3% |

Table 16. Comparison with intrusion into the arbitrary file inclusion vulnerability.

| Sequence length | $z_a$ | Time interval class | Database $P_v$ | $P_m$ |
|---|---|---|---|---|
| 6 | 2.33 | *r* | 94.8% | 92.7% |
| 6 | 2.58 | *r* | 91.3% | 94.3% |

The intrusion does not issue any abnormal system calls, so the intrusion cannot be detected. Therefore, $P_m$ does not substantially differ from $P_{v.}$

## 6. Summary and conclusion

We presented a method for anomaly intrusion detection using the timing information of the monitored application, a temporal signature. The idea comes from the observation that after removing various effects of the environment, most of the time intervals between or within system calls are uniformly distributed. We can reasonably assume the normal distribution of the time intervals, remove high variance data based on this assumption, and build a qualified normal database. Then, we can compare the timing behavior of an application with that database.

Our method is empirical, using adjustable parameters. There are many factors affecting the timing behavior of an application. The distribution of the time intervals is not strictly a normal distribution. In practice, adjusting the parameters individually for different applications will be more effective than using fixed parameters.

Our method builds on the system call method of Forrest [12]. Moreover, our method expanded the system call method so that it will work well when the observed application has the same system sequences as those in the normal database, as shown experimentally. We believe that our temporal signature method provides an effective approach to the detecting anomalistic behaviors.

## 8. References


[1] S.A. Hofmeyr, S. Forrest, and A.Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6, *pp 151-180*, 1998.

[2] A. Somayaji, S. Hofmeyr, and S. Forrest. Principles of a computer immune system. *1997 New Security Paradigms Workshop, pp 75-82*, ACM, 1998.

[3] A. Sundaram. An introduction to intrusion Detection. *ACM Crossroads Student Magazine*, *http://www.acm.org/crossroads/xrds2-4/intrus.html*.

[4] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for UNIX processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy, pp 120-128*, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[5] P. Helman and J. Bhangoo. A statistically based system for prioritizing information exploration under uncertainty. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans, 27(4), pp 449-466*, July 1997.

[6] H. S. Javitz and A. Valdes. The NIDES statistical component: description and justification. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 1993.

[7] G. G. Helmer, J. S. K. Wong, V. Honavar, and L. Miller. Intelligent agents for intrusion detection. In *Proceedings, IEEE Information Technology Conference, pp 121-134*, Syracuse, NY, September 1998.

[8] W. Lee, S. J. Stolfo, and P. K. Chan. Learning patterns from UNIX process execution traces for intrusion detection. In *AAAI Workshop on AI Approaches to Fraud Detection and Risk Management, pp 50-56*. AAAI Press, July 1997.

[9] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting Intrusions Using System Calls: Alternative Data Models. In *The 1999 IEEE Symposium on Security and Privacy*, May 1999.

[10] D. Dasgupta and S. Forrest. Novelty detection in time series data using ideas from immunology. In *The fifth International Conference on Intelligent Systems*, Reno, Nevada, June 1996.

[11] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri. Self-nonself discrimination in a computer. In *Proceedings of IEEE Symposium on Research in Security and Privacy, pp 202-212*, Oakland, CA, 16-18 May 1994.

[12] W. Lee and S. J. Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[13] John A. Rice *Mathematical statistics and data analysis*, Wadsworth & Brooks, 1988.

[14] http://www.cert.org/advisories/CA-1999-13.html

[15] http://www.securityfocus.com/vdb/bottom.html?vid=1026

[16] http://sabre.unix-security.net/pub/exploits/SendMail/qpush.c%7B-bufroverflwscript

[17] Y. Lin, A. K. Jones. *Application Intrusion Detection using Language Library Calls.* Annual Computer Security Applications Conference, December 2001.