Application of Loop Reduction to Learning Program Behaviors for Anomaly Detection*

Jidong Long, Daniel G. Schwartz, Sara Stoecklin
Department of Computer Science
Florida State University
Tallahassee, FL 32306, U.S.A.
{jidolong, schwartz, stoeckli}@cs.fsu.edu

Mahesh K. Patel
Rivermine Software,
3975 Fair Ridge Drive, 210 South
Fairfax, VA 22033, U.S.A.
mahesh.patel@rivermine.com

Abstract: Evidence of some attacks can be manifested by abnormal sequences of system calls of programs. Most approaches that have been developed so far mainly concentrate on some program-specific behaviors and ignore some plain behaviors of programs. According to the concept of locality of reference, programs tend to spend most of their time on a few lines of code rather than other parts of the program. We use this finding to propose a method of loop reduction. A loop reduction algorithm, when applied to a series of system calls, eliminates redundant data. We did experiments for the comparison before and after loop reduction with the same detection approach. The preliminary results show that loop reduction improves the quality of training data by removing redundancy.

Keywords: Intrusion Detection, Anomaly Detection, System Call Sequences, Loop Reduction

1 Introduction

Intrusion detection systems (IDSs) in the field of computer security are systems that identify suspicious activity when an intrusion is taking place or after it has taken place. IDSs can be classified according to the resource it is watching over [1]. A broad classification distinguishes between network-based and host-based IDSs. Network-based systems look at the network traffic and host-based systems watch the activity on the host. Another method of classification is on the basis of mode of detection: misuse-based versus anomaly-based. Misuse detection systems rely on the known knowledge or signatures of attacks to ascertain intrusion. Anomaly-based systems look for deviations from normal activity. There are both advantages and disadvantages of these detection methods. Anomaly-based IDSs can detect unknown attacks, while misuse-based IDSs can detect known attacks; but the former tend to produce more false alerts, while the latter cannot detect new kinds of attacks.

The novel idea of examining system call traces was introduced by Forrest et. al. [2] in a host-based anomaly detection system. Sequences of system calls were collected for normal runs of the sendmail program. These were used as training data. A record of all distinct subsequences of some fixed length (eg. 6) was stored in a database (implemented as a simple table). The database in turn then became the basis for detecting abnormal runs. If some new run of sendmail had subsequences that did not match any of those in the database, these were flagged as being abnormal, and the measure of mismatch rate is used to indicate the maliciousness of the execution trace.

In an early version of this approach [2] the patterns in the database were pairs of system calls with a look-ahead value. For example, (open, read, 1) says that a call to open may be followed by a call to read in position 1, i.e., the next immediately following position. Unfortunately, this was found to produce many false positives. A later approach [3, 4, 10] divided the input into a set of fixed-length sequences and then represented those collections of sequences having common initial segments as trees (cf. [3] for details). This led to the now well-known system STIDE (Sequence Time-Delay Embedding), which was found to give much better discrimination and to do so with a smaller database.

A further development of this idea was proposed in [11, 12]. Here, instead of using fixed-length patterns, the tables accommodate patterns with variable-length. This technique, in combination with a pattern-reduction algorithm, is based on Teiresias [7], an algorithm initially developed for the discovery of rigid patterns in unaligned bi-
ological sequence. It outperforms the fixed-length pattern approach in its ability to represent long, meaningful sub-strings. As a consequence, fewer patterns are needed to cover the training sequence; they are more process-specific, and the pattern-matching can be implemented more efficiently.

In these sequence-based approaches, the percentage of mismatched sequences is used to distinguish normal from abnormal behaviors according to the following formula:

\[
\text{mismatch}\% = \frac{\text{mismatch}\#}{\text{mismatch}\# + \text{match}\#}.
\]

It is determined that an attack has occurred whenever this mismatch rate (%) exceeds some threshold.

If the number of matched sequences is large, the mismatch rate can fall below the threshold, even though an attack has occurred. This can happen because there exists a long normal loop in the sequence of system calls. For example, suppose there are two sequences of system calls generating the following sequence of patterns, where \(N\) stands for normal and \(A\) stands for abnormal:

\[
N_1 N_2 N_3 N_4 N_5
\]

\[
N_1 N_2 N_3 N_4 N_5 \ldots N_4 N_5
\]

By applying the formula above, the first sequence generates a mismatch rate of 20%, which likely would exceed any reasonable threshold. The second sequence contains the same normal and abnormal sequences, but the mismatch rate is lower because \(N_4\) repeats itself. The mismatch rate can always reach a value lower than any given threshold as long as \(N_4\) repeats sufficiently many times. Thus execution loops make the abnormal behavior hard to detect among patterns that are outside of these loops. Accordingly, IDSs that rely on sequence-based approaches will fail to detect attacks that occur in such regions of the execution trace.

This paper addresses this problem by preprocessing system event sequences so as to remove sub-sequence repetitions that result from loops. In effect, this only removes redundant information that contributes nothing to the intrusion detection process. An algorithm has been designed and implemented in Perl that scans an input sequence of system calls and identifies all loops, working from the innermost levels of nesting to the outermost, and for each such loop removes all except its first occurrence. Several tests were performed where a “loop reduced” sequence was input to the abovementioned system STIDE, and the results of running STIDE on this input were compared with the results obtained by inputting the same sequences without loop reduction. The results were that loop reduction greatly enhanced the ability of the system to accurately detect attacks. In addition, because the reduced sequences are considerably shorter than the originals, using reduced normal sequences as training data for building the system’s database is much faster.

This paper is organized as follows. First, a loop reduction algorithm is presented in Section 2. Then, in Section 3, we present experimental results showing the advantages of applying this algorithm to the event streams before they are input to STIDE. Section 4 discusses some weakness of the loop reduction approach and suggests some ways to improve it. Section 5 surveys other IDS models for which these same loop reduction methods should be applicable. Section 6 summarizes with some concluding remarks.

### 2 A Loop Reduction Algorithm

Normally, a program has loops; otherwise, it ends very quickly. A program may have only one simple loop, but more often it contains more complex control structures, involving concatenated loops and nested loops. Figure 1 illustrates these three common control structures and the typical sequences of system calls they may generate.

The loop reduction algorithm is inspired by the observation that in a program with loops there will be many identical consecutive subsequences of system calls in the execution traces. We make the assumption that these identical consecutive subsequences are the reflection of loops in the executed code. Loops in real programs may be more complex than those in Figure 1, because a loop may have nested loops and concatenated loops within it, and those loops may further have inner loops, and so on. Thus some complex loops are difficult to find by just looking for identical consecutive subsequences; different inner loops can generate different system call sequences. We expect the loop reduction algorithm to be able to find as many loops in the system call sequence as possible.

The loop reduction process is illustrated in Figure 2. Here A, B, C, etc. are individual system calls. The top line represents a sequence of system calls, as might be recorded in some system audit log. The first step is to find all repeating sequences of just one system call. These are underlined in the second line of Figure 2. These are each assumed
to each represent some innermost loop in the associated program code. For each of these, all but one system call is removed. This gives the sequence shown in line three. For this sequence we find all repeating sequences of two system calls. These are underlined in line four. Then in each of these, all but one occurrence of the given pair are removed. This gives the sequence shown in line five. Next, we find all repeating subsequences of length 3. There are three of these identified in line six. All but one of these is removed, giving line seven.

In general, the process just described repeats for subsequences of increasing length until no more repeating subsequences can be found. A typical example of repeating sequence of just one system call would be a sequence of repetitions of the call close(2). A typical pair of calls that might repeat are mmap(2), open(2).

This algorithm cannot always find all loops in an arbitrary sequence of system calls, since some loops have conditional branches and thus give rise to different system calls in different iterations. In such cases it is difficult to track loops by only looking at sequences of system calls. Thus, in the present algorithm, such loops will not be reduced. Moreover, this algorithm may treat some repeating sequences of system calls as being the result of loops when they actually are not. Such instances we regard as occasional aberrations, however, and as not significantly affecting the overall results.

It should also be noted that this technique is best applied only to situations where the arguments of the system calls do not play a role in the underlying intrusion detection model. This is because, when reduction occurs, some of this information is removed. Fortunately, nearly all current detection models based on analyzing system call sequences don’t use the arguments information. One would not, however, use this technique in conjunction with the model described in [9].

3 Experiments with STIDE on the sendmail System Call Data

For our experiments we used the same sendmail data as employed by Forrest, et al. [2] at the University of New Mexico for developing and testing STIDE. There are two sets of sendmail system call data: UNM synthetic sendmail data, and CERT synthetic sendmail data. Each set contains data generated by normal runs of sendmail, used for training, and some data containing attacks, used for testing. This same data has been used for similar purposes by Lee, et al. [5]. The files of normal data are listed in the first column of Table 1. The data from both UNM and CERT each contain a trace of the sendmail daemon and one or more invocations of sendmail. In addition, the UNM data contains a trace of the process used to produce a sendmail log. A rationale for the choice of names, “bounce”, “bounce-1”, etc. for the invocations of sendmail was not provided.

Each trace file is a table with two columns of integers. The first is a process id, and the second is the system call number. The latter is an index into a list of system call names (open, read, mmap, etc.). These appear in the chronological order in which the system calls were executed. These files were first preprocessed, gathering together all the
system calls associated with each process. Thus each resulting file consisted of a list of all the processes invoked during that run of sendmail, where each process is given as a list of all its system calls.

Then these separate processes were subjected to loop reduction. After doing this, it was found that, often, several different processes reduce to the exact same process. This indicates that the different processes were actually executions of the same loops, but which ran through these loops for different numbers of iterations. Thus, from the standpoint of describing normal program behavior, all these event sequences can be considered as equivalent, and so only one of these need to be retained.

The final results of this reduction process for these various files are shown in Table 1. The first column indicates different traces. The second column indicates the number of system calls before (without) loop reduction. The third column indicates the number of system calls after (with) loop reduction. The last column indicates the percentage of system calls that have been removed. As can be seen, the amount of reduction in overall file size is significant. Moreover, the larger the files, the more they can be reduced. The first three traces, having less than 1000 system calls, were reduced by less than 50%, while the larger traces files were reduced by over 95%, and up to 99.83%. Intuitively, large trace files are more likely to benefit from loop reduction, since more loops can be found.

We similarly applied loop reduction to both the UNM and CERT testing data. We then ran several applications of STIDE. First, in the manner described in Section 1, we trained STIDE with the unreduced training data, and then tested it with the unreduced testing data. Then we trained and tested with the loop-reduced training and testing data. The overall approach is depicted in Figure 3.

Moreover, these experiments were done first with training and testing based on system event patterns of length 6, and then again with patterns of length 10. These were used various reasons. First, Forrest used look-ahead pairs with a look-ahead value of 6 in the early work [2], and then used exact sequences of length 10 in later work [3]. Second, Lee [6] indicated that a length of 6 was optimal, using information theory. Third, Tan [8] has also asserted that 6 is optimal. The results of our experiment are shown in Table 2.

We find that loop reduction leads to a stronger attack signal in general, with the exception that the attack signals for the two syslog-remote traces using sequences of length 10 are weaker. The latter may happen when loop reduction reduces the abnormal behavior in certain loops. This phenomenon is discussed in the section below. The results also show that patterns of length 10 always lead to more mismatches than patterns of length 6. This is consistent with Forrest’s results in [2]. The overall results are therefore that, after loop reduction, the attack information has been enlarged while the behaviors of programs, to some degree, are preserved. In other words, for the data employed, loop reduction gave a better indication of an attack when an attack actually occurred.

4 Discussion

Although the results are generally positive regarding the loop reduction algorithm, there are potential drawbacks stemming from the resulting information loss due to the way we represent loops. First, it loses information regarding the number of iterations of the loops. Second, it loses information regarding loop structures in that that two different loops might be considered as two different iterations of the same loop and thus be reduced in error. For example, if the concatenated loop and the nested loop in Figure 1 are contiguous, they will be treated as two iterations of the same loop and thereby reduced.

Another critical issue is whether the abnormal behavior occurs inside or outside of the loops. If the attack is occurring outside of loops, then loop reduction will tend to increase the attack signal. We conjecture that this is what happened for most of the results reported above. If the attack occurs inside a loop, however, and this attack is repeated for each iteration of the loop, then loop reduction will tend to weaken the attack signal, given that there is significant normal processing elsewhere in
<table>
<thead>
<tr>
<th>Traces</th>
<th>Without loop reduction</th>
<th>With loop reduction</th>
<th>Percentage reduced</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bounce</td>
<td>818</td>
<td>460</td>
<td>43.77%</td>
</tr>
<tr>
<td>bounce-1</td>
<td>291</td>
<td>163</td>
<td>43.99%</td>
</tr>
<tr>
<td>bounce-2</td>
<td>739</td>
<td>389</td>
<td>47.36%</td>
</tr>
<tr>
<td>plus</td>
<td>98180</td>
<td>1490</td>
<td>98.48%</td>
</tr>
<tr>
<td>queue</td>
<td>96330</td>
<td>724</td>
<td>99.25%</td>
</tr>
<tr>
<td>sendmail.daemon</td>
<td>1571783</td>
<td>4480</td>
<td>99.71%</td>
</tr>
<tr>
<td>sendmail.log</td>
<td>31821</td>
<td>1462</td>
<td>95.41%</td>
</tr>
<tr>
<td>CERT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sendmail.daemon</td>
<td>1556560</td>
<td>2689</td>
<td>99.83%</td>
</tr>
<tr>
<td>sendmail</td>
<td>19526</td>
<td>863</td>
<td>95.58%</td>
</tr>
<tr>
<td>TOTAL</td>
<td>3376048</td>
<td>12720</td>
<td>99.62%</td>
</tr>
</tbody>
</table>

Table 1: The Comparison of trace files in the number of system calls before and after loop reduction. The last column indicates the percentage of system calls reduced.

It is possible to encapsulate these notions in a few simple formulas. Let be the number of abnormal patterns, and be the number of normal patterns. The original mismatch rate, i.e., before loop reduction, is given as:

\[ A\% = \frac{A}{A + N}. \]

Ideally, we would like that loop reduction reduces the normal subsequences in loops while keeping the abnormal information unchanged. The mismatch rate after loop reduction under these assumptions is

\[ A_{\text{new}}\% = \frac{A}{A + N - N_r}, \]

where \( N_r \) is the number of normal sequence removed by loop reduction. This mismatch rate certainly provides a stronger abnormal signal than \( A\% \). However, if some abnormal sequences occur inside of loops, then some of these are also removed by the loop reduction process. Thus the general case can be given as

\[ A'_{\text{new}}\% = \frac{A - A_r}{A - A_r + N - N_r}, \]

where \( A_r \) is the number of abnormal patterns removed by loop reduction. In order to have \( A'_{\text{new}}\% > A\% \), the condition

\[ \frac{A_r}{A} < \frac{N_r}{N} \]

must be satisfied. In other words, if this condition doesn’t hold, the original STIDE gives a better result than STIDE applying loop reduction. This probably explains why those two syslog-remote traces under the sequence of 10 yield weaker results after loop reduction. This also shows that, roughly speaking, loop reduction is more useful for detecting attacks that occur outside of execution loops since in that case there is no abnormal sequence than can be reduced outside loops.

The present loop reduction algorithm can easily be modified, however, to resolve this shortcoming. For each repeating sequence, representing iterations of a loop, keep a count of the number of repetitions. Then, if some number of pattern mismatches occurs in each such sequence, simply multiply this number by the number of repetitions. This guarantees that the actual mismatch rate is preserved.

## 5 Concluding Remarks

A loop reduction approach was introduced in this paper in an effort to mitigate the burden of managing large volumes of training data in intrusion detection systems that are based on the learning of program behaviors. Our approach is based on the assumption that identical consecutive subsequences of system calls are generated by execution loops. We applied the approach of loop reduction to the pre-processing of the sequences of system calls of processes. This approach, due to its ability to capture loop structures in system call sequences, not only reduces the size of training data and thus the training time considerably, but also keeps the new training data informative for intrusion detection. Our experiments on system call sequences of normal and abnormal executions of the Unix sendmail process demonstrates that, using the same method introduced by [2], loop reduction results in a much better performance in anomaly detection than the original data, even though the reduced data sets are much smaller than the originals.
Anomaly Traces

<table>
<thead>
<tr>
<th>Anomaly Traces</th>
<th>Mismatch rate of sequences of fixed length 6</th>
<th>Mismatch rate of sequences of fixed length 10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Without loop reduction</td>
<td>With loop reduction</td>
</tr>
<tr>
<td>UMN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>decode-1</td>
<td>0.07</td>
<td>0.42</td>
</tr>
<tr>
<td>decode-2</td>
<td>0.80</td>
<td>1.81</td>
</tr>
<tr>
<td>fwd-loops-1</td>
<td>10.58</td>
<td>15.53</td>
</tr>
<tr>
<td>fwd-loops-2</td>
<td>4.75</td>
<td>5.58</td>
</tr>
<tr>
<td>fwd-loops-3</td>
<td>10.24</td>
<td>15.57</td>
</tr>
<tr>
<td>fwd-loops-4</td>
<td>10.56</td>
<td>15.49</td>
</tr>
<tr>
<td>fwd-loops-5</td>
<td>5.86</td>
<td>8.13</td>
</tr>
<tr>
<td>sscp-1</td>
<td>18.48</td>
<td>21.43</td>
</tr>
<tr>
<td>sscp-2</td>
<td>18.48</td>
<td>21.43</td>
</tr>
<tr>
<td>sscp-3</td>
<td>18.48</td>
<td>21.43</td>
</tr>
<tr>
<td>CERT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cert-sm555a</td>
<td>8.08</td>
<td>15.04</td>
</tr>
<tr>
<td>cert-sm5x</td>
<td>14.43</td>
<td>17.83</td>
</tr>
<tr>
<td>syslog-remote-1</td>
<td>14.18</td>
<td>21.30</td>
</tr>
<tr>
<td>syslog-remote-2</td>
<td>7.76</td>
<td>8.38</td>
</tr>
<tr>
<td>syslog-local-1</td>
<td>9.56</td>
<td>16.15</td>
</tr>
<tr>
<td>syslog-local-2</td>
<td>12.24</td>
<td>20.42</td>
</tr>
</tbody>
</table>

Table 2: Comparing Detection of Anomalies.

References


