Fixed vs. Variable-Length Patterns for Detecting Suspicious Process Behavior

Hervé Debar, Marc Dacier, Mehdi Nassehi, and Andreas Wespi

IBM Research Division, Zurich Research Laboratory, CH-8803 Rüschlikon, Switzerland
{deb,dac,mmn,anw}@zurich.ibm.com

Abstract. This paper addresses the problem of creating patterns that can be used to model the normal behavior of a given process. These models can be used for intrusion detection purposes. In a previous work, we presented a novel method to generate input data sets that enable us to observe the normal behavior of a process in a secure environment. Using this method, we propose various techniques to generate either fixed-length or variable-length patterns. We show the advantages and drawbacks of each technique, based on the results of the experiments we have run on our testbed.

1 Introduction

In [1] Forrest *et al.* introduced a change-detection algorithm for detecting computer viruses that is based on the way that natural immune systems distinguish "self" from "nonself". In [2] they reported preliminary results extending this approach in the intrusion-detection area by establishing such a definition of self for Unix processes. This technique models the way an application or service running on a machine normally behaves by registering the sequences of system calls invoked. An intrusion is assumed to exercise abnormal paths in the executable code, and is detected when new sequences are observed (see also [3], [4], [5]).

Like every behavior-based technique, an intrusion-detection system needs to be trained to learn what the "normal" behavior of a process is. This is usually done by recording the activity of the process running in a real environment during a given period. This procedure has a number of drawbacks:

- It leaves open the possibility of obtaining false negatives [6] if not all possible behaviors have been exercised during the recording period, because new users, new applications or configuration changes introduce new usage patterns.
- It leaves open the possibility of obtaining false positives [6] if the application has been hacked during the recording period.
- The observed behavior is a function of the environment in which it is running and thus prevents the distribution of an "initialized" intrusion-detection system that could immediately be plugged in and activated.

Another approach consists of artificially creating input data sets that exercise all normal modes of operation of the process. For example, Forrest *et al.* [2] use a set of 112 messages to study the behavior of the *sendmail* daemon. This method eliminates the risk of obtaining a false positive but not that of obtaining a false negative. Furthermore, it is an extremely complex and time-consuming task to come up with a good input data set.

In [7] we have shown how to use the functionality verification test (FVT) suites provided by application developers to observe all the *specified* behaviors of an application. Not only are we able to eliminate the risk of obtaining a false positive with this approach, but we also dramatically reduce the risk of obtaining a false negative because, by design, our input data set exercises all the normal (in the sense of having been specified by the designer of the application) modes of operation of the application under study.

In this paper, we present the results of experiments we have been running with this technique. Besides the use of the FVT suites, our work differs from that of Forrest *et al.* in three main ways:

- The algorithm used to create fixed-length patterns is slightly different and results in shorter tables of patterns.
- We consider in addition the opportunity of using variable-length patterns [8] and various ways of doing this.
- The decision to raise an alarm is based on a new paradigm that is simpler than the Hamming distance described in [2], thus allowing possible real-time countermeasure.

The structure of the paper is as follows. Section 2 presents the principles of the intrusion-detection techniques we are working with. It explains how patterns are generated and used to cover sequences. Section 3 presents the experimental results obtained when applying a fixed-length pattern-generation algorithm. Section 4 does the same for a variable-length pattern-generation algorithm. These two sections also highlight advantages and disadvantages of each technique. Section 5 concludes the paper by offering a comparison of the results as well as ideas for future work.

2 Principles of the Approach

In our approach, UNIX processes are described by the sequence of audit events[1] that they generate, from start (*fork*) to finish (*exit*). Their normal behavior is modeled by a table of patterns which are subsequences extracted from these sequences. The detection process relies on the assumption that when an attack exploits vulnerabilities in the code, new (i.e., not included in the model) subsequences of audit events will appear. Fig. 1 describes the complete chain used to configure the detecting tool.

[1] Note that Forrest *et al.* [1], [2] use the sequences of system calls instead of audit events. Obtaining system calls constitutes a more intrusive technique than the one we present here.

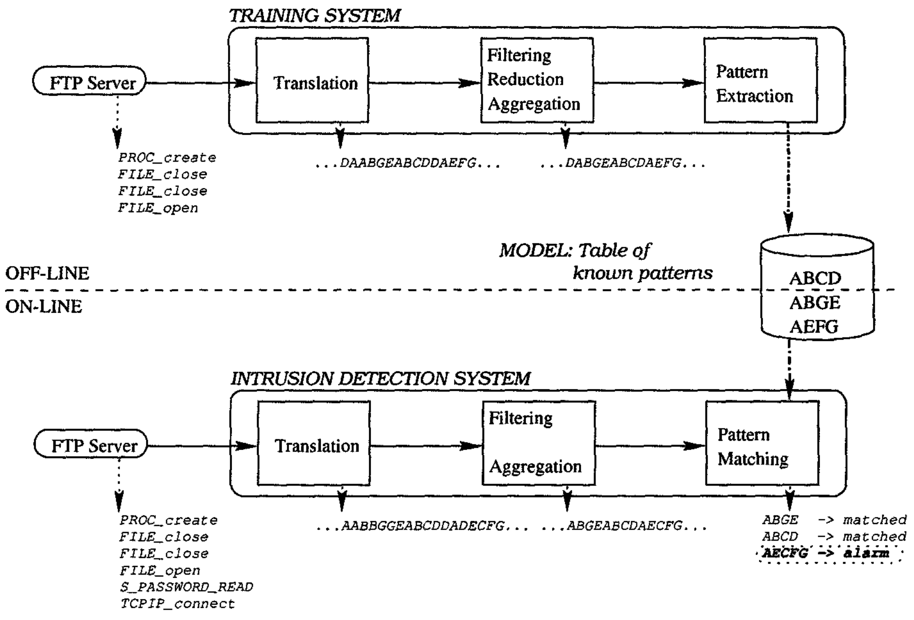

Fig. 1. Intrusion-detection system

2.1 Off-Line Treatment

The upper part of Fig. 1 represents how the model of normal behavior is created off-line. Audit events are recorded from the ftp daemon, which has been triggered by an experiment process, and translated as letters for easier handling. Such a letter in our system actually represents the combination of the audit event and the name of the process generating it, both pieces of information being provided by the Unix C2 audit trail. This recording runs through a filtering and reduction process whose purpose is explained later. When the experiment has been completed, the entire audit information is used to generate the table of patterns that constitute our model of the normal behavior of the system.

The purpose of the filtering/reduction/aggregation box is threefold (explained in more detail in the remainder of the section):

filtering to eliminate irrelevant events (processes that are not related to the services we monitor) and to sort the remaining ones by process number;

reduction to remove duplicate sequences due to processes generating exactly the same stream of audit events from start (*fork*) to finish (*exit*),[2] and

aggregation to remove consecutive occurrences of the same system calls.

[2] This is not used in the on-line version, because of the memory cost and algorithmic complexity of keeping all known process images.

Sorting prevents the introduction of arbitrary context switches into the audit stream by the operating system. Processes are sorted so that the intrusion-detection process is applied to each process individually. The events inside the process remain in the order in which they have been written to the audit trail.

The reduction keeps only unique process images for model extraction. We use test suites to record the normal behavior of the ftp daemon. These test suites carry many repetitive actions and therefore result in many instances of the same process image (see [7] for a complete explanation of this issue). We remove these duplicates as they are not used by the current algorithm extracting the patterns from the reference audit data.

The aggregation comes from the observation that strings of N consecutive audit events ($N > 20$) are quite frequent for certain events, with N exhibiting small variations. A good example is the "ftp login" session, where the *ftp daemon* closes several file handles inherited from *inetd*, the number of these closes changing without obvious cause. Therefore, we decided to simplify the audit trail in order to obtain a model containing fewer and shorter patterns. We did not observe any difference in either the false positive or false negative rates between those experiments in which the aggregation was used and those in which it was not. There is no claim of equivalence between the reduced (simplified) audit trail and the original one; the new one possibly has less semantic content. This is an experimental choice, and we would remove the aggregation part if the true/false alarm performance of the intrusion-detection system were not satisfactory. This aggregation phase would also be skipped if the intrusion-detection technique requires redundancy, e.g. neural networks.

The most obvious aggregation consists of replacing identical consecutive audit events with an extra "virtual" audit event. However, doing so enriches the vocabulary (the number of registered audit events) and possibly the number of patterns, which we want to keep small. Our solution simply aggregates these identical consecutive audit events. Therefore, any audit event represents "one or more" such events (i.e., $A = A+$ in regular expression formalism) at the output of the aggregation box.

2.2 Real-Time Detection

The lower part of Fig. 1 shows the real-time intrusion-detection process. Audit events are again generated by the ftp daemon, and go through the same sorting and reduction mechanism in real time. Then, we apply a pattern-matching algorithm (Section 2.3) to cover the sequences on the fly. When no known pattern appropriately matches the current stream of audit events, a decision has to be taken whether to raise an alarm (see Section 2.4).

2.3 Pattern-Matching Algorithm

The pattern-matching algorithm is quite critical for the performance of the intrusion-detection system. We wish to maximize both speed and detection capabilities. Therefore, we require that patterns match exactly, i.e., they cannot be matched as regular expressions using wildcards. We use a two-step algorithm: the first step looks for an exact match, and the second step looks for the best partial match possible. These two steps are illustrated with the examples presented in Figs. 2 and 3.

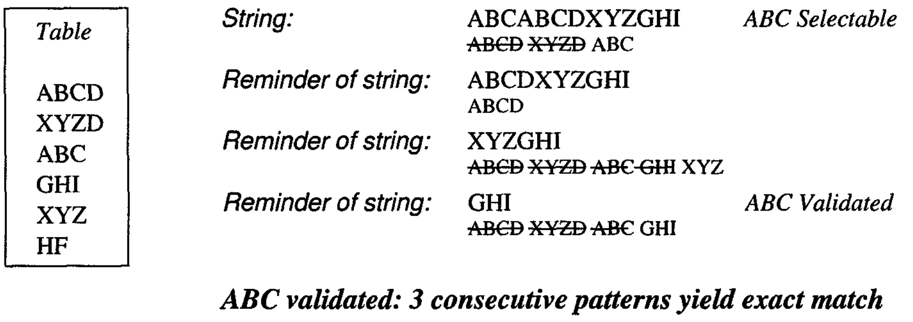
Table
ABCD
XYZD
ABC
GHI
XYZ
HF
String: ABCABCDXYZGHI
~~ABCD~~ ~~XYZD~~ ABC
ABC Selectable
Reminder of string: ABCDXYZGHI
ABCD
Reminder of string: XYZGHI
~~ABCD~~ ~~XYZD~~ ~~ABC~~ ~~GHI~~ XYZ
Reminder of string: GHI
~~ABCD~~ ~~XYZD~~ ~~ABC~~ GHI
ABC Validated
ABC validated: 3 consecutive patterns yield exact match

Fig. 2. Exact pattern-matching sequence

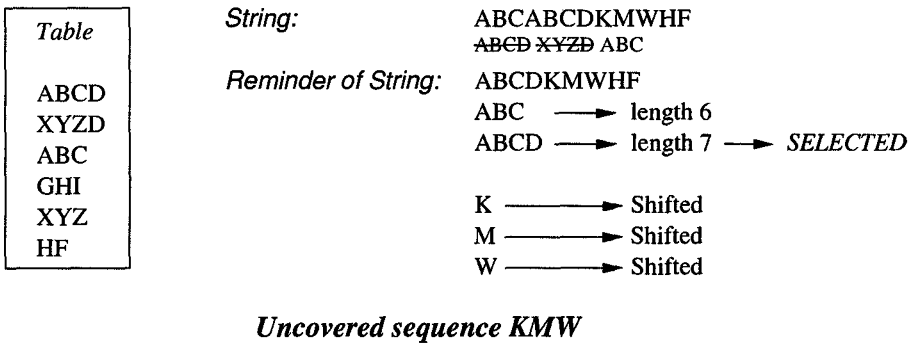

Fig. 3. Approximate pattern-matching sequence

The first step of the algorithm is illustrated in Fig. 2. A pattern that matches the beginning of the string is selected from the pattern table. If no pattern matches the beginning of the string, then the first event is counted as uncovered and removed. The algorithm is subsequently applied to the remainder of the string.

Once a selectable pattern has been found, the algorithm will recursively look for other patterns that exactly cover the continuation of the string up to a given depth D or to the end of the string, whichever comes first ($D = 3$ in Fig. 2). This means that, for a selectable pattern to be chosen, we must find D other patterns that also completely match the events following that pattern in the sequence. If such a sequence of D patterns does not exist, step 2 of the algorithm is used. The rationale behind this is that we want to know whether the selected pattern has a positive influence on the coverage of the audit events that are in its vicinity. When such a D-pattern sequence has been found, the algorithm pops the head pattern out of the sequence and goes on with the remainder of the string.

If we cannot find three other patterns that match the sequence after the selected pattern, we deselect it and try the next selectable pattern in the table. The fact that there were selectable patterns, but none that fulfill the depth requirement, triggers the second phase of the algorithm.

The second step of the algorithm deals with failure cases and is illustrated in Fig. 3. The algorithm looks for the sequence of N patterns that covers the largest number of audit events. This sequence is removed from the string and the algorithm goes back to its first part, looking again for a selectable pattern, and shortening the string as long as one is not found.

2.4 Intrusion Detection

For each string, the algorithm extracts both the number of groups of audit events that were not covered and the length of each of these groups. The decision whether to raise an alarm is based on the length of the uncovered sequences. It is worth noting that this is different from the measures used for the same purpose by Forrest *et al.*, namely the number and the percentage of uncovered events in a string [2].

There are three reasons for this.

- Processes can generate a large number of events. An attack can be hidden in the midst of a set of normal actions. Using the percentage of uncovered characters would fail to detect the attack in this case.
- Small processes could be falsely flagged as anomalous because of a few uncovered events if we use the percentage of uncovered characters.
- Long processes could be falsely flagged as anomalous because of many isolated uncovered events if we use the amount of uncovered characters.

During our experiments we observed that the trace of the attacks was represented by a number of consecutive uncovered characters. This is consistent with the findings of Kosoresow and Hofmeyr [5], who note that mismatches due to intrusions occur in fairly distinct bursts in the sequences of system calls they monitored. This means that an attack cannot be carried out in fewer than T consecutive characters. Our experiments led us to choose the value 7 for T. In other words, each group of more than six consecutive uncovered events is considered anomalous and flagged as an attack.

This amounts to defining an intrusion as any event that generates a sequence of at least T consecutive events not covered by the algorithm defined above.

To validate the concepts presented here, we have developed an intrusion-detection testbed [9]. We have used it to compare various strategies to build the table of patterns used in the detector. The results of these experiments are reported in the next two sections. They focus on the *ftp* service, which is widely used, is known to contain many vulnerabilities, and provides rich possibilities for user interaction.

3 Generating Fixed-Length Patterns

3.1 Pattern Generation

Fixed-length patterns are the most immediate approach to generate the table of patterns. We have evolved four ways of obtaining these patterns. For each of them, we use a parameter L, the length of the patterns.

Technique 1 is an exhaustive generation of all possible patterns by sliding a window of size L across the sequence and recording each unique L-event sequence. This technique is easy to implement, but has obvious drawbacks. First, a large number of patterns in the reference table will not be used by the matching algorithm on the reference data set. Moreover, the covering algorithm generates boundary mismatches when the length of the process image is not a multiple of L, because it uses a juxtaposing window instead of a sliding one (which would not make sense in this context). It is worth noting that this is the technique used to run the experiments described in [2], [4], and [5].

To solve the first problem, technique 2 is introduced, which shifts the string by L events, thus creating juxtaposed patterns instead of overlapping ones. In other words, the events in positions 1 to L constitute the first pattern, events in the positions $L + 1$ to $2L$ constitute the second one, and so on ... The table we obtain is thus much shorter than the table generated by technique 1, which makes the pattern-matching algorithm faster. Technique 2 drops the remainder of the sequence of events when its length is not a multiple of L, therefore the second problem mentioned remains.

To solve it, we have introduced techniques 3 and 4. They are similar to technique 2 but keep the events remaining at the end of the string as a stand-alone pattern or concatenated to the last generated pattern, respectively. Techniques 3 and 4 both result in 100% coverage of the test sets. As they are the ones that were most promising during the experiments, we focus on them in the remainder of this paper. Note that some of the patterns are smaller or larger than the specified size, and these can only match at the end of the process execution, as they necessarily contain the end-of-process event.

3.2 Experimental Results

We have used three test sets in our experiments. The first test is solely based on the FVT. The second test set includes additional information on tools that are used by ftp users, but are not in the testing of the ftp daemon *per se*. The third data set is based on the recording of several hundred real user sessions. The experimental results are presented for the second test set. The reference table generated from the first set does not contain patterns for commands like *tar*, which are used by normal users. Using the third set breaks the rule fixed in the introduction of the paper (no user recording for training), thus this set has only been used to evaluate the false negative rates obtained with our approach.

The output of the intrusion-detection system is shown in Fig. 4. Each line reports the analysis for one process related to the ftp service.

```
|Number of |Number of |Percentage |Number of |Average |Number of |Length of
|audit     |uncovered |of covered |patterns  |size of |groups    |each group
|events    |events    |events     |needed    |pattern |          |
|          |          |           |          |        |          |
| 569      | 4        | 99.30     | 188      | 3.01   | 4        | 1 1 1 1
| 97       | 0        | 100.00    | 32       | 3.03   | 0        |
| 928      | 3        | 99.68     | 308      | 3.00   | 3        | 1 1 1
| 735      | 3        | 99.59     | 244      | 3.00   | 3        | 1 1 1
| 6        | 0        | 100.00    | 2        | 3.00   | 0        |
| 878      | 8        | 99.09     | 290      | 3.00   | 8        | 1 1 1 1 1 1 1 1
| 839      | 4        | 99.52     | 278      | 3.00   | 4        | 1 1 1 1
| 8        | 6        | 25.00     | 1        | 2.00   | 1        | 6
| 963      | 2        | 99.79     | 320      | 3.00   | 1        | 2
| 168      | 2        | 98.81     | 55       | 3.02   | 2        | 1 1
| 476      | 1        | 99.79     | 158      | 3.01   | 1        | 1
| 366      | 3        | 99.18     | 121      | 3.00   | 3        | 1 1 1
```

Fig. 4. Output of the intrusion-detection system

Table 1 shows the results of the experiments to monitor our ftp server under various conditions, using the intrusion-detection system loaded with several reference tables. The first column identifies the experiment. The second column gives the number of audit events in the experiment. The third column indicates the two parameters corresponding to the reference table used in the intrusion-detection system, t denotes the generation technique (3 or 4) and s the size of the patterns (2, 3 and 4). The fourth column gives the overall number of uncovered characters. The fifth column gives the number of patterns used for the coverage. The sixth column gives the average size of patterns,[3] and the seventh column indicates the number of false alarms triggered by the intrusion-detection system.

Table 1 shows that our intrusion-detection system has the potential to perform well without generating false alarms. Indeed, for patterns of length 2, independent of the technique used (3 or 4), the intrusion-detection system does not generate false alarms. For patterns of length 4, independent of the technique used, the intrusion-detection system generates false alarms. For patterns of length 3, the generation of false alarms is dependent on the technique used for the generation. With technique 3, which introduces more patterns but also more freedom (some patterns can be smaller than the required length of 3), the intrusion-detection system does not generate false alarms, whereas it will generate false alarms if technique 4 is used.

It is also worth mentioning that if we use the third test set, which is closely based on user-behavior observations, instead of the second one, we observe no false alarms for any of the combinations (pattern size, technique). However, using user-generated data to train the intrusion-detection system breaches the requirements expressed in the introduction.

[3] This value is truncated after the second decimal.

Table 1. Experimental results for normal user sessions

Simulation description	Number of audit events	Technique and size	Number of uncovered events	Number of patterns required	Average size of patterns	Number of false alarms
Anonymous user sessions	6067	$t = 3, s = 2$	2	3028	2.00	0
		$t = 3, s = 3$	79	1994	3.00	0
		$t = 3, s = 4$	180	1477	3.99	3
		$t = 4, s = 2$	2	3013	2.01	0
		$t = 4, s = 3$	82	1983	3.02	0
		$t = 4, s = 4$	186	1455	4.04	3
Normal user sessions not using "site exec"	19439	$t = 3, s = 2$	9	9705	2.00	0
		$t = 3, s = 3$	108	6443	3.00	0
		$t = 3, s = 4$	257	4802	3.99	1
		$t = 4, s = 2$	10	9685	2.01	0
		$t = 4, s = 3$	116	6416	3.01	0
		$t = 4, s = 4$	264	4769	4.02	1
Normal user sessions	20135	$t = 3, s = 2$	9	10052	2.00	0
		$t = 3, s = 3$	108	6675	3.00	0
		$t = 3, s = 4$	257	4978	3.99	1
		$t = 4, s = 2$	10	10025	2.01	0
		$t = 4, s = 3$	116	6643	3.01	0
		$t = 4, s = 4$	264	4936	4.03	1
Sequence of all available user simulations	26094	$t = 3, s = 2$	11	13029	2.00	0
		$t = 3, s = 3$	199	8621	3.00	0
		$t = 3, s = 4$	444	6431	3.99	4
		$t = 4, s = 2$	12	12985	2.01	0
		$t = 4, s = 3$	213	8586	3.01	0
		$t = 4, s = 4$	511	6362	4.02	4
Simulation of user activity on loaded ftp server	26148	$t = 3, s = 2$	11	13053	2.00	0
		$t = 3, s = 3$	185	8652	3.00	0
		$t = 3, s = 4$	437	6440	3.99	4
		$t = 4, s = 2$	12	13015	2.01	0
		$t = 4, s = 3$	196	8611	3.01	0
		$t = 4, s = 4$	450	6380	4.03	4

To evaluate the risk of not detecting an attack,[4] Table 2 shows the number of valid sequences of six audit events that can be created out of the table (six is there because our $T = 7$). This is the most obvious measure, but it does not account for the full risk, as an attack could be the combination of legitimate sequences with short illegal ones. The table clearly shows that a small increase in the length of the pattern dramatically reduces the degrees of freedom in the table, and we consider this a very important result. Also, the technique used to

[4] By this, we denote the possibility of running an attack that would create a new sequence of audit events without being detected. This is possible if the new sequence consists of subsequences that are valid, i.e. that are present in the reference table.

Table 2. Statistics on fixed-length tables

Technique and size	Size of table	Size of patterns	Number of legitimate combinations of length $< T(6)$
$t = 3, s = 2$	87	{1, 2}	1142100
$t = 3, s = 3$	114	{1, 2, 3}	11559
$t = 3, s = 4$	152	{1, 2, 3, 4}	590
$t = 4, s = 2$	86	{2, 3}	456456
$t = 4, s = 3$	113	{3, 4, 5}	9905
$t = 4, s = 4$	147	{4, 5, 6, 7}	143

generate the patterns has a dramatic impact on the number of combinations, as small patterns (typically of length 1, potentially generated by the use of technique 3) significantly increase the probability of covering attacks, i.e., of not detecting them.

We have also carried out a number of attacks against our ftp server to validate our approach. Table 3 shows our experimentation results for nine attacks. An X in the column indicates that the attack has been detected.

Table 3 reveals that three attacks pose a problem to the intrusion-detection system, but for very different reasons.

The "put rhosts" attack consists of putting a *.rhosts* file in the home directory of the ftp user. This vulnerability results from a misconfiguration of the ftp service because this directory should obviously not be world writable. Actually, the "put forward" and the "put rhosts" attacks use the same vulnerability, except that one of them puts a *.forward* file, and the other one a *.rhosts*. The first one is detected because our attack script activates the attack by sending a mail to the ftp user, thus triggering the execution of the *.forward* file by the sendmail daemon. The second attack is not triggered, because our attack script does not log on to the attacked machine. Any login attempt would immediately

Table 3. Experimental results for attack detection

Attack description	t=3, s=2	t=3, s=3	t=3, s=4	t=4, s=2	t=4, s=3	t=4, s=4
put forward	X	X	X	X	X	X
put rhost						
site copy	X	X	X	X	X	X
site exec copy ftpd	X	X	X	X	X	X
site exec copy ls	X	X	X	X	X	X
site exec ftpbug ftpd	X	X	X	X	X	X
tar exec copy ftpd						X
tar exec copy ls		X	X		X	X
tar exec ftpbug ftpd	X	X	X	X	X	X
tar exec ftpbug ls	X	X	X	X	X	X

be detected, as a shell would be spawned for the ftp user, and this behavior does not belong to the reference data set.

The two "tar exec copy" attacks use the option of *GNU tar* to specify a compression program that will be used by *tar* to uncompress the data before extracting it. In our case, we have developed a Trojan horse that does no decompression, but copies */bin/sh* to */tmp/.sh* as a setuid executable. This attack exploits the fact that some ftp daemons do not release their root privileges quickly enough before forking other processes. Therefore the *tar* and the Trojan horse both run as root and the */tmp/.sh* is created suid root. Our Trojan horse is compiled and saved under the name of either *ls* or *ftpd* (hence the code name of the attacks). The attacks are only detected by the more restrictive table of patterns of length 4. We observe that the patterns modeling the normal behavior of an *ls* process in our reference table are more restrictive than those modeling *ftpd*, as the masquerading of our attack program under the process name "ls" is detected by $(t = 4, s = 3)$ but not the masquerading under the process name "ftpd" with the same pair $(t = 4, s = 3)$.

This second case also highlights one of the difficulties of this intrusion-detection technique: the reference data set must be truly restrictive and exhaustive, so that the patterns can faithfully represent the information. In our case, *tar* is not a command that is tested by the test suites we use to generate the reference data set, as it is not considered to be strictly related to ftpd. We have added some examples of usage of the *tar* command to our reference data set, but forking commands is a legitimate behavior of the *tar* command, and both the ftpd and ls patterns cover the execution of our Trojan horse. Lowering the threshold L would improve detection using the smaller reference tables, but it would also increase the false alarm rate.

4 Generating Variable-Length Patterns

4.1 Rationale for Variable-Length Patterns

The fixed-length patterns approach yields interesting results, but a careful look at the sequence of audit events shows that there are very long subsequences which repeat themselves frequently. For example, more than 50% of the process images we have obtained start with the same string. This string contains 40 audit events. Therefore, having these very long sequences would better represent our data. This is intuitively correlated by the fact that the ftp daemon answers to commands given by the user, and that each command can probably be represented by a (set of) long sequence(s) of audit events. Therefore, having the capability to extract variable-length patterns from our data makes sense [8].

However, we also are looking for an automated approach. Manual pattern extraction might be feasible for small-scale experiments, but is very time consuming. Therefore, we have looked for a method that will allow us to generate variable-length patterns.

4.2 Description of the Pattern-Generation Technique

A block diagram of the variable-length pattern generator is shown in Fig. 5. It consists of a suffix-tree constructor, a pattern pruner, and a pattern selector. The suffix-tree constructor processes the sample of user behavior as a sequence of symbols, i.e., a string to generate its suffix tree. The well-known suffix-tree structure is a dictionary of all substrings in the string [10]. An example of a string and its corresponding suffix tree are given in Fig. 6.

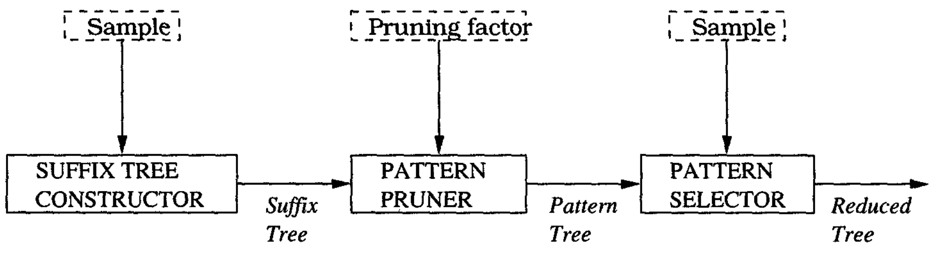

Fig. 5. Variable-length pattern generation

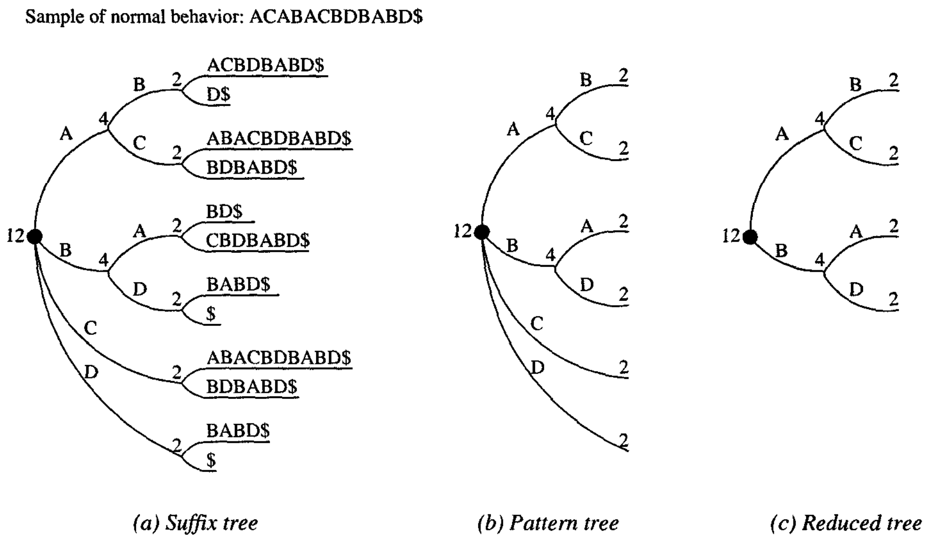

Fig. 6. Pattern extraction

The suffix tree initially consists of only a root. Then all suffixes of the sample string are added one by one until the complete suffix tree is obtained. Note that a "$", representing an end delimiter, is added to the end of the string to ensure that the suffix tree contains one leaf corresponding to each suffix. Every node corresponds to the substring represented by the symbols on the path from the root to that node. The number next to the node represents the number of occurrences of the substring in the sample string.

The pattern pruner prunes the *suffix tree* in order to generate a tree, referred to as a *pattern tree.* The pruning is done based on the pruning factor, which is a real number between 0 and 1. This factor is multiplied by the length of the sample of user behavior to obtain a lower limit on the occurrences of patterns. Every pattern with a number of occurrences less than this minimum is clipped, i.e., all its outgoing branches are removed. Fig. 6b shows the pattern tree when the suffix tree is pruned with a pruning factor of 3/12. The number of occurrences of each pattern in the sample string is indicated next to the node corresponding to that pattern. As the branching factor is increased, the suffix tree is pruned more and, hence, the pattern tree contains fewer and shorter patterns. We can use this flexibility to try different sets of patterns and choose the one that results in the most satisfactory detector performance.

From the set of pruned patterns the selector removes those patterns that are not required for covering the sample of user behavior. In our example, patterns AB, AC, BA, BD in the pattern tree in Fig. 6c are sufficient to cover the sample of user behavior; patterns C, D are not needed. The selector identifies the needed patterns by processing the sample with a pattern matcher identical to that of the detector. First, it loads the matcher with the set of the pruned-pattern tree and sets all the nodes to not marked. It then performs matching on the sample. For every matched-pattern event generated by the matcher, it marks the corresponding pattern in the pruned-pattern tree. After the entire sample has been processed, the marked patterns are selected as the set of patterns to be used by the detector. Note that currently the matcher used here is different from and much simpler than the matcher of the intrusion-detection system.

4.3 Experimental Results

Using the second test set, we have built three reference tables of patterns. The first reference table contains all the possible sequences extracted from the pattern tree,[5] denoted *all patterns* in Table 4. The second reference table contains only the sequences going from the root to the leaf in the pattern tree,[6] denoted *leaf patterns* in Table 4. The last one contains only the sequences going from the root to the leaf in the *reduced tree*[7] (see Fig. 6c). The statistics for the reference tables are given in Table 4, according to both the kind of reference table and the pruning factor used in the tree generation. The information still has to be compared with the complete number of combinations that we can derive from all the registered pairs (application, audit event) in our intrusion-detection system, which gives $56^2 * 55^4$ combinations[8].

[5] $A, AB, AC, B, BA, BD, C, D$ in the example in Fig. 6.

[6] AB, AC, BA, BD, C, D in the example in Fig. 6.

[7] AB, AC, BA, BD in the example in Fig. 6

[8] In our experiments, we have observed 55 different pairs (audit event, process name); two fictitious events have been added to represent the beginning and end of a process.

Table 4. Statistics on variable-length tables

Reference table	Size of table	Pruning factor	Number of legitimate combinations of length $< T(6)$
All patterns	87	0.1	$> 24 * 22^5$
Leaf patterns	82	0.1	$> 21 * 19^5$
Selected patterns	51	0.1	$> 13 * 12^5$
Leaf patterns	146	0.05	$> 17 * 16^5$
Selected patterns	74	0.05	$> 12 * 11^5$
Leaf patterns	215	0.02	$> 16 * 14^5$
Selected patterns	80	0.02	$> 12 * 11^5$

These reference tables actually contain patterns of length 1 to 50. This approach identifies the long sequences we found by browsing the data sets. However, they are actually less restrictive that the ones obtained with fixed-length patterns, owing to the presence of very small patterns. This unexpected result is a consequence of the presence of numerous patterns of length one, along with much longer patterns, in the table. This is the reason we are now working to improve the variable-length pattern tables by introducing a lower limit for the size of the patterns. Lowering the pruning factor results in longer patterns, but the very small ones are still required to obtain a good match, thus the number of legitimate combinations is still very large.

We have experimented with these three tables on the normal user behavior, and found that they also do not generate false alarms. Concerning the attacks, we obtain the same results as in the previous section, i.e., the ".rhosts" and "tar exec copy" attacks are not detected.

5 Conclusion and Future Work

In this paper, we have proposed two different families of methods, fixed- vs. variable-length pattern generation, to generate tables of patterns that model the normal behavior of a process. We have also proposed a new paradigm to detect intrusions based on the length of uncovered sequences and relying on a specific pattern-matching algorithm.

Based on the results of the experiments run on our testbed, we have shown that the appeal of the new paradigm for both families of patterns (fixed length and variable length) is that it minimizes the false negative alarm rate. We have explained the limitation of the fixed-length pattern approach, namely its inability to represent long meaningful substrings. We have defined a new family of methods based on a suffix-tree representation to obtain variable-length patterns. Preliminary results exhibit their ability to detect real intrusions, but a detailed analysis indicates that they are also more prone to issue false alarms.

Further work will concentrate on enhancing the algorithm for variable-length generation of patterns in order to reduce the risk of obtaining false positive alarms while keeping the good results regarding the true positive rate. Another proposed enhancement is to associate frequencies with the patterns, to customize the intrusion-detection patterns locally by flagging the infrequent use of registered patterns.

References

1. Forrest, S., Perelson, A.S., Allen, L., Cherukuri, R.: Self-Nonself Discrimination. In: Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy. IEEE Computer Society Press, Los Alamitos, CA (1994) 202–212.
2. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A Sense of Self for Unix Processes. In: Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy. IEEE Computer Society Press, Los Alamitos, CA (1996) 120–128.
3. D'haeseleer, P., Forrest, S., Helman, P.: An Immunological Approach to Change Detection: Algorithms, Analysis, and Implications. In: Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy. IEEE Computer Society Press, Los Alamitos, CA (1996) 110–119.
4. Forrest, S., Hofmeyr, S.A., Somayaji. A.: Computer Immunology. Commun. ACM **40** (1997) 88–96.
5. Kosoresow, A.P., Hofmeyr. S.A.: Intrusion Detection via System Call Traces. IEEE Software **14**(5) (1997) 35–42.
6. Esmaili, M., Safavi-Naini, R., Pieprzyk, J.: Computer Intrusion Detection: A Comparative Survey. Technical Report 95-07, Center for Computer Security Research, University of Wollongong, Wollongong, NSW 2522, Australia (May 1995).
7. Debar, H., Dacier, M., Wespi, A.: Reference Audit Information Generation for Intrusion Detection Systems. In: Posch, R., Papp, G. (eds).: Information Systems Security, Proceedings of the 14th International Information Security Conference (IFIP SEC'98), Vienna, Austria, and Budapest, Hungary, Aug. 31–Sept. 4, 1998 (in press).
8. Teng, H.S., Chen, K., Lu, S. C-Y: Adaptive Real-Time Anomaly Detection Using Inductively Generated Sequential Patterns. In: Proceedings of the IEEE Symposium on Research in Security and Privacy. IEEE Computer Society Press, Los Alamitos, CA (1990) 278–284.
9. Debar, H., Dacier, M., Wespi, A., Lampart. S.: A Workbench for Intrusion Detection Systems. IBM Research Report RZ 2998, IBM Research Division, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland (1998).
10. Stephen. G.A.: String Searching Algorithms. World Scientific, Singapore (1994).