Intrusion Detection Using Variable-Length Audit Trail Patterns

Andreas Wespi, Marc Dacier, and Hervé Debar

IBM Research, Zurich Research Laboratory,
Säumerstrasse 4, CH-8803 Rüschlikon, Switzerland
{anw,dac,deb}@zurich.ibm.com

Abstract. Audit trail patterns generated on behalf of a Unix process can be used to model the process behavior. Most of the approaches proposed so far use a table of fixed-length patterns to represent the process model. However, variable-length patterns seem to be more naturally suited to model the process behavior, but they are also more difficult to construct. In this paper, we present a novel technique to build a table of variable-length patterns. This technique is based on Teiresias, an algorithm initially developed for discovering rigid patterns in unaligned biological sequences. We evaluate the quality of our technique in a testbed environment, and compare it with the intrusion-detection system proposed by Forrest *et al.* [8], which is based on fixed-length patterns. The results achieved with our novel method are significantly better than those obtained with the original method based on fixed-length patterns.

Keywords: Intrusion detection, Teiresias, pattern discovery, pattern matching, variable-length patterns, C2 audit trail, functionality verification tests.

1 Introduction

In [9], Forrest *et al.* introduced a new approach to the problem of protecting computer systems. The problem is viewed as an instance of the more general problem of distinguishing *self* (i.e. normal process execution) from *other* (i.e. anomalous process execution). Based on the way natural immune systems distinguish *self* from *other*, Forrest *et al.* have developed a change-detection method that can be applied to virus detection [9] and intrusion detection [8]. The method models the way an application or service running on a machine normally behaves by registering characteristic subsequences, i.e. patterns, of system calls invoked. An intrusion is assumed to pursue abnormal paths in the executable code, and is detected when new sequences are observed that cannot be matched with registered patterns (see also [6, 7]).

Forrest *et al.* use fixed-length patterns to represent the process model. However, a main limitation of this approach is that there is no rationale for selecting the optimal pattern length. As shown in [10], the pattern length has an influence on the detection capabilities of the intrusion-detection system. Therefore, in [2]

H. Debar, L. Mé, and F. Wu (Eds.): RAID 2000, LNCS 1907, pp. 110–129, 2000.

the concept of using variable-length patterns to model the process behavior was introduced. However, preliminary results obtained with variable-length patterns revealed no clear advantage of that method. In this paper, we present a novel method to generate variable-length patterns. We can show that the results obtained with variable-length patterns clearly outperform those achieved with the original method, which is based on fixed-length patterns.

The structure of the paper is as follows. Section 2 describes the basic principles of detecting suspicious process behavior by analyzing the sequences of system calls a process can generate. Readers familiar with the previous work on this topic [2, 3, 8, 10, 11, 13, 14, 15] can skip this section and go directly to Section 3 where our novel intrusion-detection method, which uses variable-length patterns, is presented. Section 4 compares our novel method with the one proposed by Forrest *et al.* [8, 10] based on experiments performed in a testbed [5] environment. Section 5 concludes the paper by summarizing the results obtained and offering ideas for future work. In the Appendix, formal descriptions of the variable-length pattern-extraction and the variable-length pattern-matching algorithm are given.

2 Background

We describe the basic principles of intrusion-detection systems that use characteristic subsequences of system call traces to model the process behavior and to detect intrusions by looking for deviations from the process model. First, we show the generic architecture of such intrusion-detection systems. Then we describe in more detail the intrusion-detection system proposed by Forrest *et al.* [8, 10], which will be used as the reference system to evaluate the quality of our novel approach.

2.1 Architecture

The intrusion-detection system proposed by Forrest *et al.* [8, 10] is a behavior-based [4] intrusion-detection system. In a training phase, normal process behavior is defined. During real-time operation, it is decided whether the observed process behavior corresponds to the learned normal behavior, or whether significant deviations are observed, which may be an indication of an intrusion.

There are different interpretations of what the expression *"normal" behavior* means. In [10] the authors differentiate between *synthetic* normal and *real* normal behavior. Synthetic normal behavior is created by exercising a program in an isolated environment in as many modes as possible and recording its behavior. Real normal behavior is observed by tracing the behavior of a program in a live user environment. For a discussion of the advantages and disadvantages of each approach see [2].

Forrest *et al.* [8, 10] are mainly interested in real normal behavior because this allows them to detect abnormal but legitimate behavior, i.e., behavior that is valid according to the process specification but has not been seen during the

training phase. In our work, we concentrate on synthetic normal behavior and furthermore try to learn the normal process behavior exhaustively. We achieve this by using functionality verification test suites (FVT) that systematically exercise all valid process invocations. Our objective is to detect attacks against the process itself, i.e. attacks that succeed in exercising process execution paths that were hitherto unknown and do not correspond to the process specification. However, it is important to note that the intrusion-detection technique itself, specifically whether to use fixed- or variable-length patterns, does not depend on the method used to learn normal behavior.

The architecture of our intrusion-detection system is depicted in Figure 1. The system comprises two main parts: an off-line part, which corresponds to the training system, and an on-line part, which corresponds to the detection system. The main components of each part are described in the next two subsections.

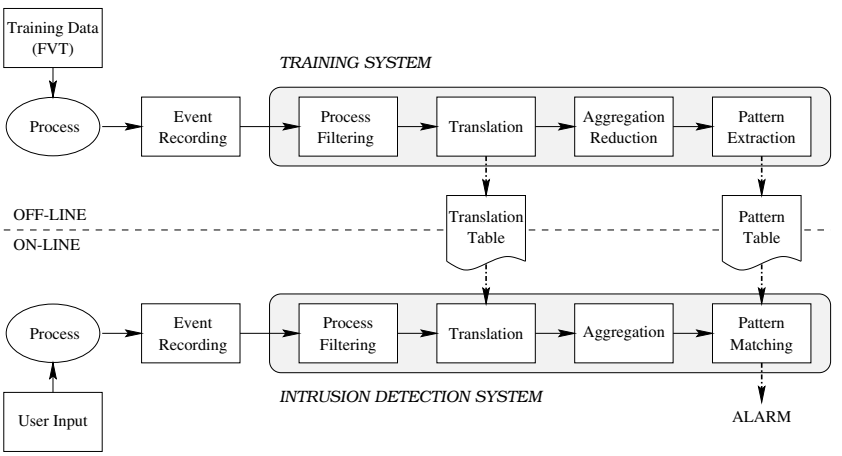

Fig. 1. Intrusion-detection system

Training System The behavior of the process under study is traced by recording either the system calls or the audit events generated on behalf of the process. In [8], system calls are used. Although on most operating systems not every system call is represented as an audit event, it has been shown in [2] that audit events are a viable alternative offering the same detection capabilities. For our work we use audit events because, as our experiments have shown, collecting audit events is a less intrusive technique than recording system calls.

The audit events generated on behalf of different process executions are sent to a *filtering module*. Its task is to sort the events by process id while keeping the chronological event order. The events are given as tuples comprising the process and event name. For easier processing, the *translation module* translates the events into an internal format. We use characters to represent this internal

format. The translation rules are generated on the fly and stored in a *translation table*. Figure 2 shows the translation steps from the stream of audit events to the sequences of characters.

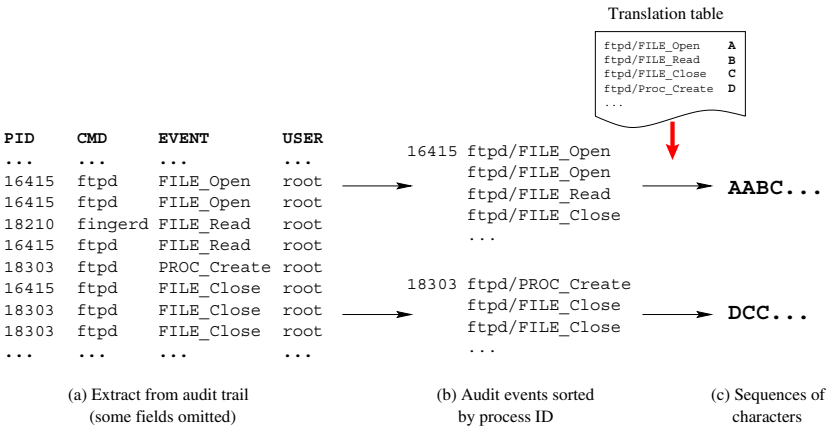

Fig. 2. Translation of audit events to characters

The translated sequences are forwarded to the *aggregation and reduction module*. The purpose of this module is twofold:

- It aggregates consecutive occurrences of the same character, i.e., of the same event.
- It removes duplicate sequences.

The following example shows the usefulness of the aggregation of consecutive identical characters. When a new instance of the *ftpd* process is invoked, it inherits several file handles from *inetd*, its parent process. As one of its first tasks, the ftp daemon closes the file handles, resulting in consecutive `FILE_close` events. The number of inherited file handles may vary because the inetd process is not always in the same state. Closing all unneeded file handles will therefore result in a varying number of `FILE_close` events. As a consequence, the resulting sequence of system calls is dependent on the environment in which the process runs. Because we would like to have a process description that is independent of the environment, we aggregate consecutive occurrences of the same character.

There are different ways to do the aggregation. We follow the approach proposed in [2] and aggregate identical consecutive characters in a single character, i.e., $A = A+$ in regular expression formalism. We make no claim of equivalence between the simplified event sequence and the original one. The aggregation is an experimental choice and would be removed if any negative impact on the detection capabilities of the intrusion-detection system were observed.

During the training phase, duplicate event sequences may occur. Because they contain no new patterns, duplicate event sequences do not have to be considered and are hence removed.

After all process executions have taken place, the preprocessed sequences are forwarded to the *pattern-extraction module* where the *pattern table* is generated.

Detection System The structure of the detection system is similar to that of the training system. In the detection system, events generated on behalf of the process under study are collected and processed in real time. The *filtering module* is identical to that of the training system. The *translation module* is slightly different from its counterpart in that audit events are translated based on the entries already contained in the *translation table*. Events without a corresponding entry in the translation table constitute quite an unusual event because they have not been seen in the training phase of the process. They are translated into a dummy character, and one may consider issuing an alarm whenever such a character is observed. In the current implementation of our intrusion-detection system, they are treated the same way as unmatched characters.

The reduction component of the *reduction and aggregation module* is no longer needed. Pattern matching is done in real time, and initiated as soon as possible for each sequence. This means we do not wait until the complete sequence has been received before the pattern matching is started, and therefore the reduction of entire sequences is not applicable.

The task of the *pattern-matching module* is to match the arriving event sequences with the entries in the *pattern table*. Based on how well the pattern matching can be done, it is decided whether anomalous behavior is observed and thus an *alarm* has to be raised.

2.2 A Review of Forrest *et al.*'s Approach

This description of Forrest *et al.*'s work is based on the original paper [8] as well as a more recent publication [10] in which some modifications of the original concepts are described. We show the techniques applied for pattern extraction and pattern matching as well as the metrics used to differentiate between normal and abnormal behavior.

Pattern Extraction The algorithm to build the table of fixed-length patterns is very simple. From the sequences sent to the pattern-extraction module, all unique subsequences, i.e. patterns, of a given length k are extracted. This is achieved by sliding a window of length k across all input sequences and recording the encountered subsequences. Duplicates are not considered.

The construction of the pattern table is best illustrated with an example. For $k = 3$ and the sample training sequence `ABCCABC`, we obtain the following pattern table:

$$\{\ \texttt{ABC, BCC, CCA, CAB}\ \}.$$

Note that the pattern `ABC` shows up only once in the pattern table although it is encountered at two window positions, namely the first and the last position.

Pattern Matching The pattern-matching technique is similar to the pattern-generation technique. We move a window of length k across the sequence that is recorded during real operation. Each window position is checked for a *match*, i.e., whether there is a pattern that matches the subsequence in the window. If no matching pattern exists, we speak of a *mismatch*.

Given the pattern table of the previous example and the sample sequence `ABCCACC`, we observe three matches, namely {`ABC, BCC, CCA`}, and two mismatches, namely {`CAC, ACC`}.

Metric Note that the measure for raising an alarm must not depend on the sequence length. Arriving events have to be processed in real time, and we do not want to wait until all events of a process have arrived before we check them for possible signs of intrusions. This would be problematic, for example, in cases of continuously running processes. In [10], three measures are given to differentiate between normal and abnormal behavior. However, only the measure we are going to describe in this section is independent of the sequence length.

Let a and b be two sequences of length k. The expression a_i designates the character at position i. The difference $d(a, b)$ between a and b is defined as

$$d(a,b) = \sum_{i=1}^{k} f_i(a,b) \qquad \text{where } f_i(a,b) = \begin{cases} 0 \text{ if } a_i = b_i \\ 1 \text{ otherwise.} \end{cases}$$

During pattern-matching, we determine for each subsequence u of the translated event sequence the *minimum* distance $d_{\min}(u)$ between u and the entries in the pattern table:

$$d_{\min}(u) = \min \{d(u,p) \ \ \forall \text{ patterns } p\} .$$

To detect an attack, at least one of the subsequences generated by the attack must be classified as anomalous. In terms of the above measure, there is at least one subsequence u for which

$$d_{\min}(u) > 0.$$

It is assumed that the higher the $d_{\min}$ value, the more likely it is that the subsequence was actually generated by an intrusion. In practice, the maximum $d_{\min}$ value observed is used as the measure for an intrusion because it represents the strongest anomalous signal. The signal of anomaly, S_A, is defined as

$$S_A = \max \{d_{\min}(u) \ \ \forall \text{ subsequences } u\} .$$

In the ideal case, an S_A value that is greater than 0 can be considered a sign of an intrusion. However, as experimental results show, a complete match cannot always be achieved [10]. Therefore, a threshold is defined such that only sequences whose S_A value is above this threshold are considered suspicious.

3 Variable-Length Patterns

Before building a table of fixed-length patterns, one has to decide which pattern length to use. However, selecting the most appropriate pattern length is not straightforward:

- Long patterns are expected to be more process-specific than short patterns. The longer a pattern, the lower the probability that a pattern would match part of an event sequence generated on behalf of an attack.
- It is desirable to have a small pattern table because it reduces the amount of computation needed for the detection process. As experimental results show, increasing the pattern length to a certain length also increases the size of the corresponding pattern table [2].

Using variable-length patterns enables us to cope with these two apparently contradictory constraints. To describe the normal behavior of a process, variable-length patterns appear to be more naturally suitable than fixed-length patterns. A careful look at the sequences of events that can be generated by a process shows that there are many cases in which very long subsequences are repeated frequently. For example, more than 50% of the process images we have obtained for the *ftpd* process start with the same string. After aggregation, this string contains 40 audit events and should be incorporated as a whole in the pattern table. However, approaches based on fixed-length patterns use much shorter pattern lengths and would therefore not detect such a long pattern.

Variable-length patterns are also motivated by the fact that, for example, the ftp daemon answers user commands, and that each such command can probably be represented by its own sequence of audit events.

Variable-length patterns are not as easy to generate as fixed-length patterns. A technique based on building and pruning suffix trees [2] showed that variable-length patterns are an interesting alternative to fixed-length patterns, but it also showed some limitations of the chosen pattern-generation technique.

3.1 Pattern Extraction

We present a novel method to generate the table of variable-length patterns. This method comprises two steps. In the first step, all maximal variable-length patterns contained in the set of training sequences are determined. Because the patterns can share common subsequences, not all patterns may be needed to cover, i.e. fully match, the training sequences. Therefore, in the second step, a reduction algorithm is applied to prune entries in the pattern table. The goal is to obtain the minimum pattern set that still covers all training sequences.

Generating the Pattern Set The input to the pattern-extraction module (see Fig. 1) are sequences of audit events that have been preprocessed as described in Section 2.1. We define a variable-length pattern as a subsequence that has a

minimum length of two and occurs at least twice, be it in the same or in different sequences. Furthermore, we consider only maximal variable-length patterns. A pattern p is maximal if there is no other pattern q that contains the pattern p as a subsequence and has the same number of occurrences as pattern p. For example, if there are two patterns `DEA` and `EA`, pattern `EA` is considered maximal only if it occurs more often than pattern `DEA`.

There are several algorithms to determine variable-length patterns [1]. We use the Teiresias algorithm [12], an algorithm developed initially to discover rigid patterns in unaligned biological sequences. Teiresias has many interesting properties. It is well suited to our problem for the following two main reasons:

- It finds the maximal variable-length patterns by avoiding the generation of non-maximal intermediate patterns during the pattern-extraction process [12].
- Its performance scales quasilinearly with the size of the output [12].

It follows that Teiresias very efficiently finds all the maximal variable-length patterns in the set of training sequences.

Reducing the Pattern Set We want the pattern set to be as process-specific as possible. This means that the pattern set should contain all the patterns needed to cover the training sequences but not more. The set of maximal variable-length patterns usually contains overlapping patterns, i.e. patterns that have common subsequences. Let us have a look at the following sample set of training sequences:

{ `ABCDEAFDE, BCFDEABCD, BCEADEFDE` }.

Extracting the maximal variable-length patterns results in the following pattern table:

{ `ABCD, DEA, FDE, BC, DE, EA` }.

The question arises whether all patterns are needed to cover the training sequences. Let us decompose the training sequences such that the resulting subsequences correspond to entries in the pattern table. A possible decomposition of the training sequences is listed below. We use the symbol "–" to mark the decomposition points:

{ `ABCD-EA-FDE, BC-FDE-ABCD, BC-EA-DE-FDE` }.

As we can see, of the six patterns in the pattern table only five are needed in the above decomposition. The pattern `DEA` is not used. We conclude that the pattern set determined by Teiresias can be reduced.

There are various ways to construct the reduced pattern set. The rationales for the approach described in the remainder of this section are based on the observation that there are patterns that have a clear semantical representation.

A pattern may, for example, represent a subroutine that is invoked several times or the statements that are executed in a loop. Such patterns can be regarded as building blocks out of which the event sequences of any possible process instantiation can be composed.

In our experiments, we observed that many training sequences have the same beginning and end, i.e., the same initialization and termination routine is executed for different process instantiations. As a first step, we can add the corresponding pattern to the reduced pattern set. Subsequences that match this pattern are removed from the training sequences, and the reduction process continues with the pruned training sequences. This procedure is reiterated until no training sequences are left, i.e., until all training sequences can be covered with the patterns added to the reduced pattern set.

There is a single requirement that must be fulfilled by the reduced pattern set:

- The training sequences must be covered by the patterns in the reduced pattern set.

In addition, as explained at the beginning of Section 3, the following properties are desirable:

- The reduced pattern set should contain long patterns.
- The number of patterns in the reduced pattern set should be small.

The two inputs for the reduction algorithm are the pattern table as produced by the Teiresias algorithm and the set of training sequences. The algorithm itself comprises four steps, which are executed repeatedly until all training sequences have been processed. We outline here only the basic steps of the algorithm. A detailed description can be found in Appendix A.2.

Step 1

The function $\mathrm{bCover}(p, s)$ returns the number of characters covered at the beginning and at the end of a sequence s by a pattern p. $\mathrm{bCover}(p, s)$ considers the fact that a pattern may match several times at the beginning or end of a sequence, e.g.

$$\mathrm{bCover}(\texttt{AB}, \texttt{ABCDEABAB}) = 6.$$

If S designates a set of sequences, $\mathrm{bCover}(p, S)$ is the sum of all events matched at the beginning and end of all sequences by the pattern p. We call the returned value the *boundary coverage*.

For each entry in the pattern table, we calculate its boundary coverage of the set of training sequences. The pattern with the highest boundary coverage is added to the reduced pattern set. This pattern is used further in Steps 2 and 3.

Step 2

All the subsequences at the beginning and end of the training sequences that are matched by the pattern determined in Step 1 are removed. For example, if

the pattern AB is selected in Step 1, the sequence ABABCDAB will be transformed as follows:

$$\texttt{ABABCDAB} \rightarrow \texttt{CD}.$$

Furthermore, we have to avoid training sequences being reduced to sequences that are shorter than the minimal pattern length. By definition, there would be no pattern to match such a short sequence. For example, if the minimal pattern length is two and ABC is an entry in the pattern table, the following transformation of the sequence ABCD is invalid:

$$\texttt{ABCD} \rightarrow \texttt{D}.$$

because the remaining sequence D is shorter than the minimum pattern length.

Step 3

After removing the matching subsequences at the boundary of the training sequences, we now also remove matching subsequences p that are not adjacent to the boundary. We call this process nonboundary matching. Removing such subsequences results in splitting the original sequence into two new sequences. As in the case of boundary matching, it has to be ensured that the length of the resulting sequences is equal to or greater than the minimum pattern length. If a sequence has several subsequences that can be matched, the longest subsequence is removed first. Nonboundary matching may again be applied to the resulting sequences. For example, given the pattern AB and a maximal pattern length of two, the following transformation can be applied to the sequence CDABABEFABGH:

$$\texttt{CDABABEFABGH} \rightarrow \{\ \texttt{CD, EFABGH}\ \} \rightarrow \{\ \texttt{CD, EF, GH}\ \}.$$

Step 4

No further transformation can be applied to sequences whose length is less than two times the minimal pattern length. Any further transformation would result in a new sequence that is less than the minimal pattern length, which contradicts our requirements. As a result, any sequence that cannot be further reduced will be added to the reduced pattern set. However, they are first moved to the pattern table and treated the same way as the patterns determined by the Teiresias algorithm. Note that, as a consequence, the reduced pattern set may contain entries that were not determined by Teiresias.

If after execution of Step 4 no sequences remain in the training set, the reduction algorithm terminates, otherwise execution continues at Step 1.

Figure 3 illustrates how the algorithm works for a sample training set of three sequences. In this example, five steps are needed to derive the reduced pattern table. For each step, we show the state of the training sequences, the entries in the pattern table, their corresponding bCover values, and the state of the reduced pattern table.

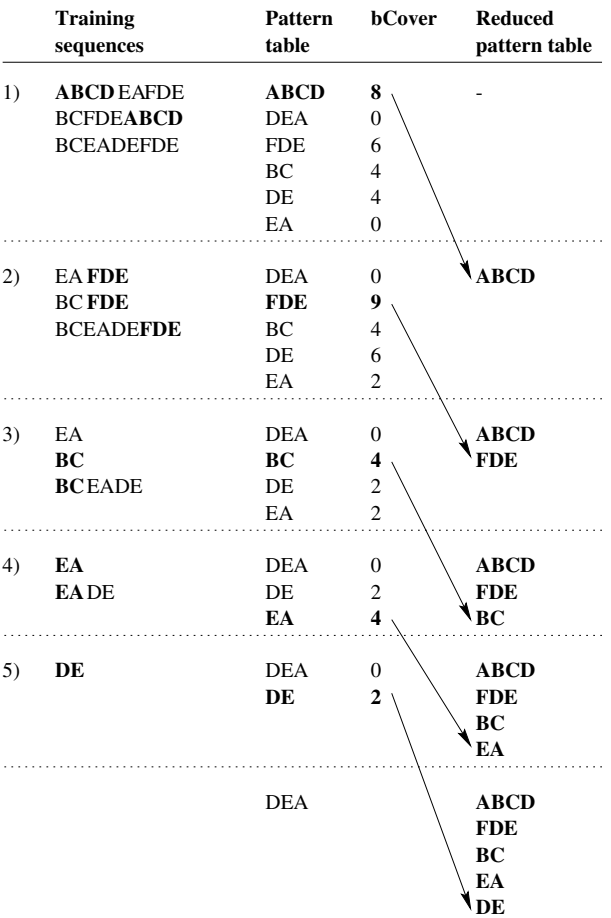

Fig. 3. Reduction algorithm

To show the importance of the reduction algorithm, let us take a look at the following numbers. The training sequences of the experiment that we are going to describe in Section 4 contained a total of 167,187 patterns. Of this total, 554 patterns are maximal. These are the patterns that we generate using the Teiresias algorithm. It becomes obvious that generating the maximal patterns directly as Teiresias does offers a significant advantage over other approaches that also generate the intermediate patterns. Of the 554 maximal variable-length patterns, a pattern set of only 71 patterns can be constructed that covers all the training sequences. This shows the usefulness of reducing the pattern sets generated by Teiresias. A pattern-matching process that has to consider only 71 patterns will run faster than one that has to consider 554 entries. The statistics for the variable-length patterns are summarized in Table 1.

Table 1. Example of table sizes of variable-length patterns

Patterns	167,187
Maximal patterns	554
Covering patterns	71

3.2 Pattern Matching

As stated in the previous section, variable-length patterns can be seen as building blocks out of which any valid event sequence can be constructed. This idea is also reflected in the juxtaposed pattern-matching technique we apply for variable-length patterns.

The sequence to be matched is processed starting from the beginning of the sequence to its end. One out of three conditions holds at a given point of the pattern-matching process.

1. Exactly one pattern matches at a given position. The corresponding events are marked as matched and the pattern matching continues right after the last event marked.
2. Several patterns match at a given position. To decide which of the matching patterns to select, a look-ahead algorithm determines for a predefined value of n whether a sequence of up to n patterns can be found that matches the continuation of the sequence. The pattern whose continuation results in the longest match is selected, the corresponding events are marked as matched, and the pattern matching continues right after the last event marked.
3. No matching pattern can be found. The event at the current position of the pattern-matching process is marked as unmatched and skipped. The pattern matching continues right after the skipped event.

A detailed description of the pattern-matching algorithm can be found in Appendix A.3.

3.3 Metric

For each sequence, the pattern-matching algorithm returns the g groups of consecutive uncovered events and the length l_i, $i = 1 \ldots g$, of each of these groups. It is assumed that the greater the length l_i, the more likely it is that an intrusion is observed. Based on the length of the longest group of uncovered events, T, it has to be decided whether an attack is observed. T is defined as follows:

$$T = \max(l_i), i = 1 \ldots g.$$

4 Results

We have set up a test environment [5] to evaluate the quality of Forrest *et al.*'s intrusion-detection method and our novel method, which is based on variable-length patterns. We report the results obtained for the *ftpd* process. We focus

on this process because it is widely used and is known to contain many vulnerabilities (either due to software flaws or configuration errors). Furthermore, it provides a host of possibilities for user interaction and is therefore a challenging process from an intrusion-detection point of view. We have also successfully applied our intrusion-detection approach to other network services, e.g. finger and sendmail. For space reasons, only the results obtained for the ftp service are presented in this paper.

To train the system, we use the functionality verification test suites (FVT) running under AIX [3]. The test suite allows us to automatically exercise all ftp subcommands and thus to learn the complete process behavior.

4.1 Problem Size

The FVT for the ftp process consists of 487 individual tests. Because many of these tests do not differ in the subcommands invoked but only in the arguments used, they result in identical sequences of audit events. When running the ftp test suite, 68 unique sequences (after aggregation and reduction) were recorded comprising a total of 23,302 audit events. Table 2 summarizes these numbers.

Table 2. Problem size of the ftp experiment

Tests	487
Training sequences	68
Events	23,302

For the comparison of the fixed- and variable-length approaches, we use two tables of fixed-length patterns and one of variable-length patterns. The pattern sizes of the fixed-length pattern tables are six and ten, respectively. Six was selected because it is stated in [10] that the pattern size makes only little difference for the normalized signal of anomaly, i.e. S_A/k, once we have a length of at least six, and ten because this is the pattern size used in the experiments reported in [10]. It is worth noting that, coincidentally, the mean pattern length of the variable-length pattern table is ten. Table 3 lists the size of the respective pattern tables. We see that the size of the variable-length pattern table is much smaller than that of the fixed-length pattern tables.

4.2 Normal User Sessions

In our testbed [5], we simulated series of user sessions. The simulation resulted in 65 unique sequences comprising a total of 26,025 audit events. We used these sequences to evaluate the quality of the fixed-length and variable-length pattern-matching techniques in combination with the respective pattern tables. As we have used the FVT to learn the complete process behavior and the user sessions contain no attacks, the intrusion-detection system should not generate an alarm.

Table 3. Table sizes of fixed-length patterns

Pattern type	(Mean) pattern size	Table size
Fixed-length	6	396
Fixed-length	10	702
Variable-length	10	71

Table 4 shows the results obtained. The first column lists the number of unique sequences recorded. The second column specifies a value n that is used as a comparison value for columns three to five. Columns three and four give a measure of how well the normal user sessions could be matched with the entries of the respective fixed-length pattern tables, and column five does the same for the variable-length pattern table.

To understand the content of columns three to five, we have to recall the meaning of the two metrics S_A and T. The metric S_A is the signal of anomaly defined in Section 2.2 and is used to differentiate between normal and abnormal behavior when fixed-length patterns are used. The values of S_A lie between 0 and k, where k is the pattern size. The higher the value of S_A, the more likely it is that an intrusion is observed. The metric T is the number of consecutive uncovered characters defined in Section 3.3 and is the metric used in our intrusion-detection system that is based on variable-length patterns.

The entries in columns three to four list the number of sequences for which S_A is equal to the comparison value n of the same row. For example, the row with $n = 4$ indicates that for a window size of six (ten), we have observed a maximum of four uncovered characters in all subsequences of six (ten) characters. This has been seen in five (eight) out of 65 sequences.

The last column lists the results obtained for the variable-length approach. Here, the row with $n = 4$ indicates that there are two sequences out of 65 where the maximum number of consecutive uncovered characters is 4.

Table 4. Experimental results for normal user sessions

Number of sequences	n	Fixed $k = 6$	Fixed $k = 10$	Variable
		$S_A = n$	$S_A = n$	$T = n$
65	0	11	11	47
	1	19	0	14
	2	19	25	1
	3	11	15	1
	4	5	8	2
	5	0	4	0
	6	0	2	0
	> 6	–	0	0

In the ideal case, we would like to see $S_A = 0$ and $T = 0$, i.e. full coverage of all test sequences. However, Table 4 shows that with the two fixed-length approaches only 17% of the sequences, i.e. 11 out of 65, can be fully matched. With the variable-length pattern approach, 72% of the sequences, i.e. 47 out of 65, can be covered. We see that variable-length patterns result in much a better coverage of the test sequences than fixed-length patterns.

Table 4 also allows us to set thresholds to differentiate between normal and abnormal behavior. Any value of S_A or T that is above the threshold would be considered a sign of an intrusion. If we do not want to issue false alarms, the threshold for S_A has to be set to four (six) in the case of fixed-length patterns. In the case of variable-length patterns, the threshold of T has to be set to four.

4.3 Attacks

We have implemented seven attacks against the ftp service. Some of the attacks exploit server misconfigurations, others take advantage of vulnerabilities in older versions of the ftp daemon.

The *put forward* attack consists of putting a *.forward* file in the home directory of the ftp user, and then sending a mail to the ftp user. This vulnerability results from a misconfiguration of the ftp service because this directory should obviously not be world writable.

The *site exec* suite of attacks exploits a vulnerability that was enabled by wrongly setting the `_PATH_EXECPATH` variable when compiling the ftpd program. Precompiled binaries containing this vulnerability were shipped with an older release of the Linux Slackware distribution. Two different attack scripts were executed, *ftpbug* and *copy*. To make the two scripts difficult to detect, they were given the names *ftpd* and *ls* (hence the code names of the attacks).

The *tar exec* type of attacks use the option of the GNU *tar* program to specify a compression program in combination with the tar command. The attack becomes possible because some older versions of the ftp daemon do not release their root privileges quickly enough before forking other processes. To exploit this vulnerability, we let the tar program invoke renamed copies of the *ftpbug* and *copy* program as compression programs.

A detailed description of the attacks can be found in [5]. The results we obtained are shown in Table 5.

We see that all three approaches can be used to detect the attacks. For the two fixed-length pattern tables, S_A is equal to the maximum value of six (ten) for all attacks. All the values obtained are above the threshold we have defined for S_A. For the variable-length method, the values for T vary between 13 and 36. These values are significantly higher than the threshold we have set for T, namely four.

4.4 Discussion

The quality of an intrusion-detection method is given by its capability to differentiate between normal and abnormal behavior. For the intrusion-detection

Table 5. Experimental results for attacks

Attack description	Fixed $k = 6$	Fixed $k = 10$	Variable
	S_A	S_A	T
put forward	6	10	36
site copy	6	10	18
site exec copy ftpd	6	10	16
site exec copy ls	6	10	18
site exec ftpbug ftpd	6	10	16
tar exec ftpbug ftpd	6	10	13
tar exec ftpbug ls	6	10	14

methods we investigate, the differentiator is the threshold that has to be set for the measures S_A and T. In the case of fixed-length patterns, the threshold for S_A, i.e. four or six, is relatively high compared to the pattern length of six or ten, and implies an increased risk to miss an attack. In the case of variable-length patterns, we observe quite a difference between the threshold for T, i.e. four, and the minimum value of T obtained for the attack sequences, namely 13. Therefore, the risk of issuing a false alarm is quite low if variable-length patterns are used.

We conclude that intrusion-detection methods based on variable-length patterns can be more reliably used to differentiate between normal and abnormal behavior. This is mainly because variable-length patterns better match normal user sessions.

5 Conclusions

We have presented a host-based intrusion-detection system that can model the normal process behavior based on the audit sequences created on behalf of the process. The process model is a pattern table whose entries are subsequences of the audit event sequences determined during a training phase.

Because the fixed-length pattern approach has certain limitations, including the inability to represent long, meaningful substrings, it appears to be more natural to use variable-length patterns to build the process model. We have developed a novel technique to generate tables of variable-length patterns automatically. To construct the patterns, the Teiresias algorithm, a method initially developed for discovering rigid patterns in unaligned biological sequences, is used in combination with a pattern-reduction algorithm.

We have shown that the variable-length pattern model has several advantages over the fixed-length model. Fewer patterns are needed to describe the normal process behavior, and the quality of the results achieved is significantly better than that of the results obtained with fixed-length patterns. Our results also show that behavior-based intrusion-detection systems can be built that do not suffer

from one of the main problems observed in behavior-based intrusion detection, namely generating (too) many false alarms.

Future work will concentrate on validating our approach for other network services and on investigating techniques that would result in 100% coverage of normal user sessions. Furthermore, as our technique to build variable-length pattern tables has some similarities with techniques used for data compression, we plan to investigate the potential of this technology for intrusion-detection purposes.

A Algorithms

The pattern-reduction and pattern-matching algorithms have been briefly described in Sections 3.1 and 3.2, respectively. Here, we describe them in more detail.

A.1 Terminology and Notation

Consider a finite set of characters $\Sigma = c_1, c_2, \ldots, c_n$. The set Σ is called alphabet. To denote a string of $n, n > 0$, consecutive identical characters $c \in \Sigma$, we write c^n. c^+ denotes a string of identical consecutive characters of arbitrary length l, $l > 0$.
The length of a string s is written as $|s|$. We write $c \in s$ if the character c is contained in the string s.

Given is a set of strings $S = \{s_1, s_2, ..., s_n\}$ over the alphabet Σ. A substring p that

- occurs at least twice in the set of strings S, and
- has a length $|p|$ of two or more characters

is called a pattern.

p^n denotes the pattern p repeated $n, n > 0$ times. p^+ denotes the pattern p repeated $l, l > 0$ times.

A pattern p is maximal if there is no pattern q for which holds that

- p is a substring of q with $|p| < |q|$, and
- the number of occurrences of the pattern q in S is equal to or larger than the number of occurrences of the pattern p in S.

A character $c \in s$ is said to be covered by the pattern p if $c \in p$ and p is a substring of s.

A string s is said to be covered by a set of patterns P if for each character c, $c \in s$, there is a pattern $p, p \in P$, such that c is covered by p.

A set of strings S is said to be covered by a set of patterns P if each string $s, s \in S$, is covered by P. Additionally, P is said to cover S.

Given are a pattern p and a string s. Let us decompose the string s as follows:

$$s = p^l s' p^r \quad l, r >= 0, \quad |s'| >= 0$$

It is assumed that the decomposition is maximal, i.e., there is no l' and r' for which holds $l' + r' > l + r$.

The expression $(l+r) \cdot |p|$, i.e. the sum $l+r$ times the pattern length $|p|$, is called boundary coverage of pattern p and string s. It is written as bCover(p, s).

The boundary coverage of a pattern p and a string set $S = s_1, s_2, \ldots, s_n$, written as bCover$(p, S)$, is defined as

$$\text{bCover}(p, S) = \sum_{i=1}^{n} \text{bCover}(p, s_i).$$

A.2 Pattern Reduction

Out of the set of patterns P consisting of all the maximal patterns found for the string set S, a subset of patterns $R, R \subset P$, is selected that covers S. μ denotes the minimal pattern length that was used to generate the set of maximal variable-length patterns. The reduced pattern set R is constructed as follows:

1. If $P = \emptyset$, then add all $s \in S$ to the reduced pattern set R and exit.
2. For each $p \in P$ calculate bCover(p, S).
3. Select a pattern $r \in P$ for which bCover(r, S) is maximal, i.e., there is no other pattern $p \in P$ for which holds:
   - bCover$(p, S) >$ bCover(r, S), or
   - bCover$(p, S) =$ bCover$(r, S) \wedge |p| > |r|$.
4. Add r to the reduced pattern set R and remove it from P.
5. Remove all matching substrings adjacent to the beginning or end of a string, i.e., remove strings of the form $s = r^+$, and replace strings of the form $s = r^+ s'$, $|s'| > \mu$, or $s = s'' r^+$, $|s''| > \mu$, with s' or s'', respectively.
6. Remove the matching substrings that are not adjacent to the beginning or end of a string, i.e., as long as there is an $s \in S, s = s' r s'', |s'| \geq \mu, |s''| \geq \mu$, replace s with the two strings s' and s''.
7. If there is an $s \in S$ with length $|s| < 2 \cdot \mu$, remove s from the set of strings S and add it to the pattern set P.
8. If $S \neq \emptyset$, go to Step 1, otherwise exit.

A.3 Pattern Matching

At certain points of the pattern-matching process, there may be several patterns that match the input stream. To decide which pattern to select, the algorithm uses a look-ahead approach. A pattern is selected if $\delta, \delta > 0$, patterns can be found that match the continuation of the string. We designate δ as *look-ahead parameter*. An alarm is raised if the number of consecutive uncovered characters exceeds a threshold τ.

The pattern matching is done as follows:

1. Set the look-ahead parameter to a value $\delta > 0$, and set the threshold for the number of consecutive uncovered characters to a value $\tau > 0$.
2. Set the counter of consecutive uncovered characters, κ, to 0.
3. When there is a sufficient number of characters in the input stream I, find a pattern $p \in P$ that covers the beginning of the input stream I. If no pattern can be found, go to Step 6.
4. Find $\delta > 0$ patterns $q_1, q_2, \ldots, q_\delta$, such that the string $t = pq_1q_2...q_d$ covers the beginning of the stream. If there are ϵ patterns $q_1, q_2, ..., q_\epsilon$, $0 < \epsilon < \delta$, that cover the entire input sequence, set $t = pq_1q_2...q_\epsilon$.
   (a) If t matches the entire input sequence, remove it and go to Step 2.
   (b) If δ patterns can be found that cover the beginning of the input stream, remove pattern p from the input stream, and go to Step 2.
5. Determine all pattern combinations that match the beginning of the input stream. If there is a match, select the pattern combination that covers the longest input sequence, remove it from the input stream, and go to Step 2.
6. Skip one character, and increase κ by 1.
7. If $\kappa = \tau + 1$, raise an alarm.
8. Go to Step 2.

References

[1] A. Brazma, I. Jonassen, I. Eidhammer, and D. Gilbert. Approaches to the automatic discovery of patterns in biosequences. Technical report, Department of Informatics, University of Bergen, 1995. 117

[2] Hervé Debar, Marc Dacier, Medhi Nassehi, and Andreas Wespi. Fixed vs. variable-length patterns for detecting suspicious process behavior. In Jean-Jacques Quisquater, Yves Deswarte, Catherine Meadows, and Dieter Gollmann, editors, *Computer Security - ESORICS 98, 5th European Symposium on Research in Computer Security*, LNCS, pages 1–15, Louvain-la-Neuve, Belgium, September 1998. Springer. 110, 111, 112, 113, 116

[3] Hervé Debar, Marc Dacier, and Andreas Wespi. Reference Audit Information Generation for Intrusion Detection Systems. In Reinhard Posch and György Papp, editors, *Information Systems Security, Proceedings of the 14th International Information Security Conference IFIP SEC'98*, pages 405–417, Vienna, Austria and Budapest, Hungaria, August 31–September 4 1998. 111, 122

[4] Hervé Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion detection systems. *Computer Networks*, 31(8):805–822, April 1999. Special issue on Computer Network Security. 111

[5] Hervé Debar, Marc Dacier, Andreas Wespi, and Stefan Lampart. A workbench for intrusion detection systems. Technical Report RZ 6519, IBM Zurich Research Laboratory, Säumerstrasse 4, CH-8803 Rüschlikon, Switzerland, March 1998. 111, 121, 122, 124

[6] Patrick D'haeseleer, Stephanie Forrest, and Paul Helman. An immunological approach to change detection: algorithms, analysis, and implications. In *Proceedings of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 110–119. IEEE Computer Society, IEEE Computer Society Press, May 1996. 110

[7] Stephanie Forrest, Steven A. Hofmeyr, and Anil Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, October 1997. 110

[8] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society, IEEE Computer Society Press, May 1996. 110, 111, 112, 114

[9] Stephanie Forrest, Alan S. Perelson, Lawrence Allen, and Rajesh Cherukuri. Self-nonself discrimination. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, pages 202–212. IEEE Computer Society, IEEE Computer Society Press, May 1994. 110

[10] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998. 110, 111, 114, 115, 122

[11] Andrew P. Kosoresow and Steven A. Hofmeyr. Intrusion detection via system call traces. *IEEE Software*, pages 35–42, September/October 1997. 111

[12] Isidore Rigoutsos and Aris Floratos. Combinatorial pattern discovery in biological sequences. *Bioinformatics*, 14(1):55–67, 1998. 117

[13] Andreas Wespi, Marc Dacier, and Hervé Debar. An intrusion-detection system based on the Teiresias pattern-discovery algorithm. In Urs E. Gattiker, Pia Pedersen, and Karsten Petersen, editors, *Proceedings of EICAR '99*, Aalborg, Denmark, February 1999. European Institute for Computer Anti-Virus Research. ISBN 87-987271-0-9. 111

[14] Andreas Wespi, Marc Dacier, Hervé Debar, and Mehdi M. Nassehi. Audit trail pattern analysis for detecting suspicious process behavior. In *Proceedings of RAID 98, Workshop on Recent Advances in Intrusion Detection*, Louvain-la-Neuve, Belgium, September 1998. 111

[15] Andreas Wespi and Hervé Debar. Building an intrusion-detection system to detect suspicious process behavior. In *Proceedings of RAID 99, Workshop on Recent Advances in Intrusion Detection*, West Lafayette, Indiana, USA, September 1999. 111