An Intrusion-Detection System Based on the Teiresias Pattern-Discovery Algorithm

Andreas Wespi, Marc Dacier and Hervé Debar
IBM Research Division, Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland

About the Authors

Andreas Wespi is a research scientist at the IBM Zurich Research Laboratory in the Information Technology Solutions department. He holds an M.Sc. in Computer Science from the University of Berne, Switzerland. His research interests include intrusion detection, network security in general, and distributed and parallel computing.

Marc Dacier holds an M.Sc. in Computer Sciences from the University of Louvain, Belgium, where he worked from 1989 to 1991 as a research assistant. In 1994, he obtained a Ph.D. in computer security from the INPT in Toulouse, France. In 1995, he worked as a security consultant at Firstel, Paris (France). He joined the IBM Zurich Research Laboratory in 1996, where he is the manager of the Global Security Analysis Laboratory (GSAL).

Hervé Debar is a research scientist in the Global Security Analysis Laboratory at the IBM Zurich Research Laboratory, where he works on system and network security (in particular intrusion detection) as well as system management. His interests include secure systems and artificial intelligence. Dr. Debar holds a Ph.D. from the University of Paris, France, and a Telecommunications Engineering degree from the Institut National des Télécommunications in Evry (France).

Mailing Address: Andreas Wespi, Marc Dacier and Hervé Debar, IBM Research Division, Zurich Research Laboratory, CH-8803 Rüschlikon, Switzerland. Telephone: +41-1-724-8264; Fax: +41-1-724-8953; E-mail: {anw,dac,deb}@zurich.ibm.com

Descriptors

intrusion detection, Teiresias, pattern discovery, pattern matching, variable-length patterns, UNIX processes, C2 audit logs, functional verification tests (FVT)

EICAR Prodeedings 1999 Edited by Urs E. Gattiker, Pia Pedersen and Karsten Petersen

An Intrusion-Detection System Based on the Teiresias Pattern-Discovery Algorithm

Abstract

This paper addresses the problem of creating a pattern table that can be used to model the normal behavior of a given process. The model can be used for intrusion-detection purposes. So far, most of the approaches proposed have been based on fixed-length patterns, although variable-length patterns seem to be more naturally suited to model the normal process behavior. We have developed a novel technique to build tables of variable-length patterns. This technique is based on Teiresias, an algorithm initially developed for the discovery of rigid patterns in unaligned biological sequences. We evaluate the quality of our technique in a testbed environment and compare it with techniques based on fixed-length patterns.

1. Introduction

In this paper, we present a new technique to detect attacks against UNIX processes by observing behavior deviations in those processes. Initial results were presented in (Debar, Dacier, Nassehi & Wespi, 1998a). In the following, we present the latest results obtained by applying a new algorithm, namely Teiresias, which was initially developed for the discovery of rigid patterns in unaligned biological sequences. Readers aware of our previous work may want to skip the first sections and directly go to Section 3.

Forrest, Perelson, Allen & Cherukuri (1994) introduced a change-detection algorithm that is based on the way natural immune systems distinguish “self” from “nonself” for detecting computer viruses. Forrest, Hofmeyr, Somayaji & Longstaff (1996) reported preliminary results of extending this approach to 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 pursue abnormal paths in the executable code, and is detected when new sequences are observed [see also D'haeseleer, Forrest & Helman (1996); Forrest, Hofmeyr & Somayaji (1997); Kosoresow & Hofmeyr (1997)].

Like every behavior-based technique, an intrusion-detection system must 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 several drawbacks:

- It has the potential to produce false negatives (Esmaili, Safavi-Naini & Pieprzyk, 1995) if not all possible behaviors have been exercised during the recording period, because new users, new applications, or configuration changes may introduce new usage patterns.
- It has the potential to produce false positives (Esmaili *et al*., 1995) 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 be immediately 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.* (1996) 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 (Debar, Dacier & Wespi, 1998b) 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. With this approach, not only are we able to eliminate the risk of obtaining a false positive, 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 a new algorithm, Teiresias, to create what we call “variable-length” sequences to represent the normal behavior of a process. Besides the use of the FVT suites, our work differs from that of Forrest *et al.* in two ways:

- We use variable-length patterns instead of fixed-length patterns.
- The decision to raise an alarm is based on a new paradigm that is simpler than the Hamming distance described in (Forrest *et al*., 1996), thus allowing possible real-time countermeasures to be taken.

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 goes into the details of applying the Teiresias algorithm to intrusion detection. Section 4 presents the experimental results and compares our approach with fixed-length based approaches. Section 5 concludes the paper by offering ideas for future work.

2. Principle 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*). The normal behavior of such processes is modeled by a table of patterns, which are sub-sequences 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) sub-sequences of audit events will appear. Fig. 1 describes the complete chain used to configure the detecting tool.

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 into letters for easier handling. Such a letter in our system 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 (and will be

[1] Note that Forrest *et al*. (1994, 1996) use sequences of system calls instead of audit events. Obtaining system calls constitutes a more intrusive technique than the one we present here.

explained in more detail in the remainder of the section):

filtering to eliminate irrelevant events (processes that are not related to the services that 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]
aggregation to remove consecutive occurrences of the same system calls.

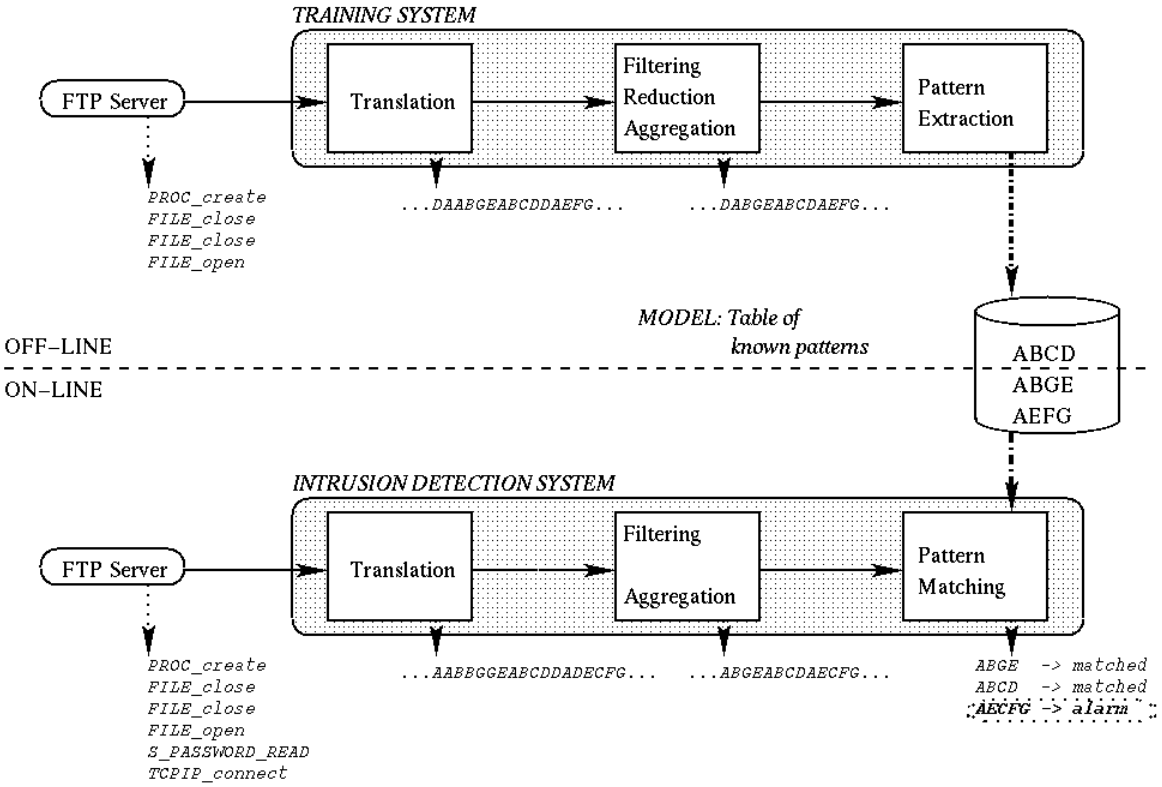

Figure 1: Intrusion-detection system.

Sorting prevents the introduction in the audit stream of arbitrary context switches by the operating system. Processes are sorted so that the intrusion-detection process can be 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 (Debar *et al*., 1998b) for a complete explanation of this issue]. We remove these duplicates as they are not used by the current algorithm to extract the patterns from the reference audit data.

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

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 changes for no apparent reason. Therefore, we have simplified the audit trail to obtain a model that contains fewer and shorter patterns, and have observed no differences in experiments that do or do not use the aggregation. There is no claim that the reduced (simplified) audit trail and the original one are equivalent; 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 omitted if the intrusion-detection technique requires redundancy, e.g. neural networks.

The most obvious aggregation consists of replacing identical consecutive audit events with an additional “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 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 its speed and detection capabilities. Therefore, we require that patterns match exactly, i.e., they cannot be matched like ordinary 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.

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 ejects the head pattern out of the sequence and continues with the remainder of the string.

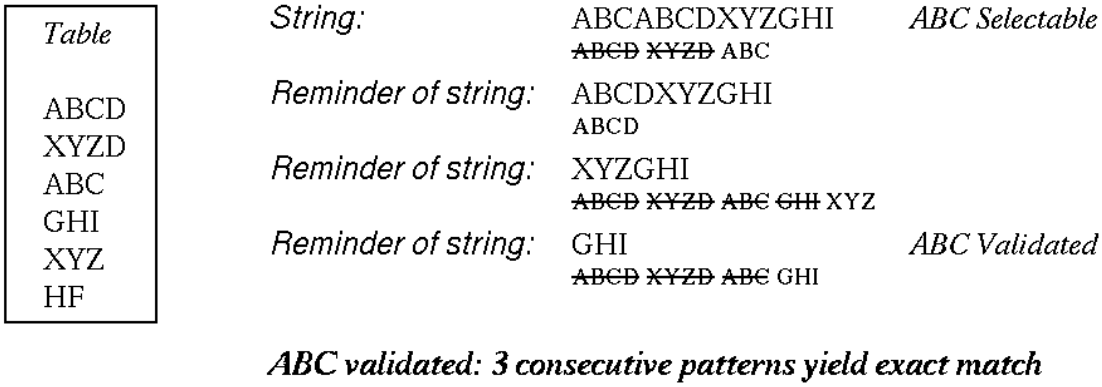

Figure 2: Exact pattern-matching sequence.

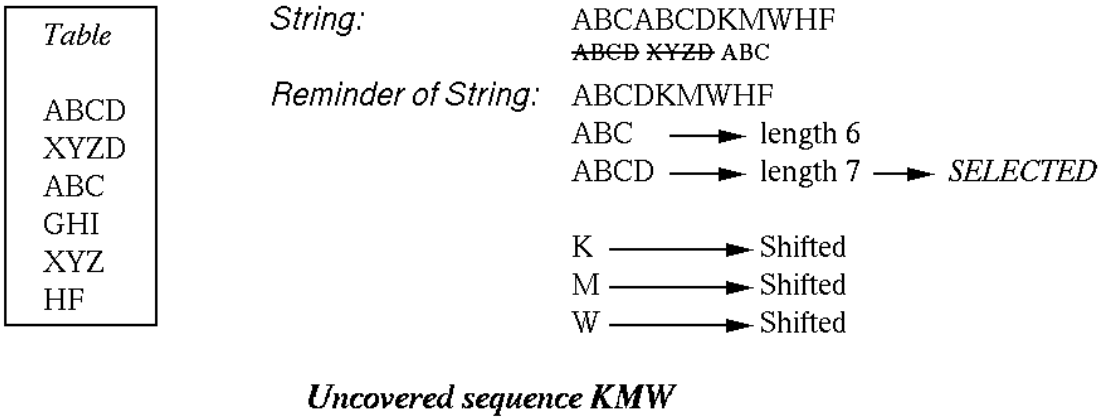

Figure 3: Approximate pattern-matching sequence.

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 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.* (1996), namely the amount and the percentage of uncovered events in a string.

The reasons for not choosing the same measures are threefold:

- 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.

We observed during our experiments that the trace of the attacks was represented by a number of consecutive noncovered characters. This is consistent with the findings of Kosoresow and Hofmeyr (1997), who note that mismatches due to intrusions occur in fairly distinct bursts in the sequences of system calls that 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 noncovered 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 (Debar, Dacier, Wespi & Lampart, 1998c) and have used it to compare various strategies to build the table of patterns used in the detector. These strategies focus on the *ftp* service, which is widely used, is known to contain many vulnerabilities, and provides rich possibilities for user interaction.

3. Generating variable-length patterns with Teiresias

Because they are easy to generate, fixed-length patterns are the most immediate approach to building the pattern table. Techniques to generate tables of fixed-length patterns are described by Forrest *et al.* (1996) and Debar *et al.* (1998a). However, variable-length patterns appear to be more naturally suitable to describe the normal behavior of a process. As a careful look at the sequences of audit events that can be

generated on behalf of a process shows, there are many cases where very long sub-sequences are repeated frequently. For example, more than 50% of the process images we have obtained for the ftp daemon start with the same string. This string contains 40 audit events and should be incorporated as a whole in the pattern set. However, approaches based on fixed-length patterns use much shorter pattern lengths and would therefore not detect such long patterns.

Using variable-length patterns is also motivated by the fact that, for example, the ftp daemon answers user commands, and that each such command can probably be represented by a (set of) long sequence(s) of audit events. Therefore, the capability to extract variable-length patterns from our data set makes sense (Teng, Chen & Lu, 1990).

We are looking for an automated approach to extract variable-length patterns. Manual pattern extraction might be feasible for small-scale experiments (Kosoresow & Hofmeyr, 1997), but it is very time consuming. A technique based on suffix trees has been proposed to generate variable-length patterns automatically (Debar *et al.*, 1998a). Although preliminary results show the ability of this method to detect real intrusions, detailed analysis indicates that it is also more prone to issue false alarms.

We present a novel method to generate the set of variable-length patterns. This method comprises two steps. First, all variable-length patterns contained in the set of training sequences, i.e. all the process images created in the training system, are determined. Second, a reduction algorithm is applied to prune entries in the pattern table that do not contribute to the coverage. Because patterns may share common sub-sequences, not all patterns may be needed to cover the training sequences. The goal is to end up with a minimal pattern set that can still cover all the training sequences.

3.1. Extracting the maximal variable-length patterns

The input to the pattern extraction module (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 sub-sequence that has a minimal 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 sub-sequence and has the same number of occurrences as pattern *p*. If there are, for example, two patterns “ABC” and “ABCD”, pattern “ABC” is considered maximal only if it occurs more often than pattern “ABCD”.

There are several algorithms to determine variable-length patterns. We use the Teiresias algorithm (Rigoutsos & Floratos, 1998), a novel algorithm developed initially for the discovery of 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.
- Its performance scales quasilinearly with the size of the output.

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

3.2. Reducing the pattern set

We want the pattern set to be as process-specific as possible. This means that the patterns can cover the audit sequences generated on behalf of the process they were created for, but not the audit sequences of any other process.

The set of maximal variable-length patterns usually contains overlapping patterns, i.e. patterns that share common sub-sequences. Let us assume that there is the following simple set of training sequences:

{“ABCDEA”, “BCFDEABCD”, “ABCEADEFDE”}

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

{“ABCD”, “ABC”, “DEA”, “FDE”, “BC”, “DE”, “EA”}

Examples of overlapping patterns are “ABCD" and “ABC" because they share the common sub-sequence “ABC”. The question arises whether all patterns are needed to cover the training sequence. Let us decompose the training sequences such that the resulting sub-sequences correspond to entries in the pattern table. Two possible decompositions of the training sequences are listed below. We use the symbol “-“ to mark the decomposition points.

{“ABCD-EA”, “BC-FDE-ABCD”, “ABC-EA-DE-FDE”}
{“ABC-DEA”, “BC-FDE-ABCD”, “ABC-EA-DE-FDE”}

As we can see, not all patterns are used in the first or in the second decomposition. In the first decomposition, pattern “DEA” does not occur, and pattern “ABCD” does not occur in the second. We conclude that the pattern set as determined by Teiresias can be reduced, and the decomposition (or the coverage) of the training sequences is not unique.

There are various ways to construct the reduced pattern set. First, the reduced pattern set must fulfill the following requirement:

- It must be possible to cover the training sequences with the patterns in the reduced pattern set.

In addition, the following features are desirable:

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

The pattern set preferably contains long patterns, which are assumed to be more process-specific than short patterns. For example, a pattern of length two is expected to be found in the audit sequences of several different processes, but it is quite unlikely that a pattern of length 40 will be found in audit sequences of different processes.

A small pattern table is useful from a practical point of view. The fewer patterns a pattern-matching algorithm has to consider, the more efficiently it can run.

It has to be mentioned that the two preferences are contradictory. If we allowed a minimal pattern length of one in the extreme case, the pattern table would be smallest. It would consist only of all the audit events that can be created on behalf of a process. However, such a pattern set would be useless because single audit events are not process-specific and could be created on behalf of many other processes. Furthermore, attacks that do not trigger an additional audit event cannot be detected. Therefore, we apply some heuristics when constructing the reduced pattern set to overcome this duality and reach a good compromise.

As mentioned above, we are looking for an automated approach to determine the table of variable-length patterns because in many cases the problem is too complex to be solved manually. Table 1 gives some information about the problem size of the ftp experiment we performed.

Training sequences	58
Events	23,302
Patterns	167,187
Maximal patterns	554
Covering patterns	71

Table 1: Problem size of the ftp experiment.

All the subcommands that can be executed by the ftp client result in 58 different audit sequences on the server side. These audit sequences comprise 23,302 audit events. The audit sequences serve as input for our pattern-extracting module (see Fig. 1). If one retrieved all the variable-length patterns from the training sequences, one would obtain a total of 167,187 patterns. Out 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 is a great advantage over other approaches that generate the intermediate patterns as well. Out of the 554 maximal variable-length patterns, a pattern set of only 71 patterns can be constructed that covers all the training sequences. This proves the necessity of reducing the pattern sets generated by Teiresias because a pattern-matching process that has to consider only 71 patterns should clearly run faster than one that has to consider 554

entries.

4. Results

We have set up a test environment to measure the quality of various pattern sets. To train the system, we use the functionality verification test (FVT) suite for the AIX ftp daemon. The test suite is used to exercise all ftp subcommands automatically. Based on the pattern-extraction algorithm in place, various pattern tables may be created. We present the results obtained for three different pattern sets:

- fixed-length patterns of length 3,
- fixed-length patterns of length 4,
- variable-length patterns obtained with the Teiresias algorithm and the pattern reduction.

Fixed-length patterns are created by splitting the training sequences into pieces of the given pattern length. Certain corrections must be made to account for the case that the training sequence length is not a multiple of the pattern length (Debar *et al*., 1998a). In our comparison, we use fixed-length patterns of length 3 and 4. We do not consider patterns of length 2 because they are not sufficiently process-specific, and we do not consider patterns of length 5 and longer because, as initial tests showed, they did not produce good results.

For a proof of concept we use our testbed to simulate user sessions where no attack is performed and user sessions where several attacks are performed against the ftp server. Using the same pattern-matching algorithm for the three different sets, we would like to see that all attacks are detected and no alarm is raised when ordinary user sessions without any attacks are simulated. The results are summarized in Table 2.

Set	Method	Mean pattern length	Number of patterns	Uncovered events (out of 26,000)	False positives	Attacks detected (out of 10)
1	Fixed-length	3	114	185	0	8
2	Fixed-length	4	152	437	4	9
3	Variable-length	10	71	47	0	8

Table 2: Comparison of three pattern tables.

The longer the length of fixed-length patterns, the more patterns are needed to cover

the training sequences. A total of 114 patterns are needed in the case of length 3, and 152 are needed in the case of length 4. However, with the variable-length approach we obtain a mean pattern length of 10, and much fewer patterns are needed for the coverage, namely only 71. We conclude that the table of variable-length patterns constitutes a good compromise between pattern length and table size.

As discussed in Section 2.4, the intrusion-detection system raises an alarm if the length of uncovered sequences exceeds a given threshold. Heuristics have to be applied to determine the ideal threshold. In principle we would like to see a threshold of 0, i.e. any uncovered event results in an alarm being raised. The results we obtained so far show that this is difficult to achieve. However, as a general rule we can state that it is desirable to have as few uncovered events as possible for the normal user sessions. Column 5 of Table 2 shows that the variable-length pattern performs best with respect to this criterion. Only 47 events are not covered as opposed to 185 and 437 uncovered events obtained for the set of fixed-length patterns of length 3 and 4, respectively.

The fixed-length patterns of length 3 and the variable-length patterns raise no false positives, i.e. they do not claim to have detected an attack where actually no attack has occurred. However, the second pattern set results in four false positives.

We have implemented ten attacks against the ftp server. Some of the attacks exploit server misconfigurations, some exploit deficiencies in older versions of the ftp daemon. Of the ten attacks, 8 attacks can be detected with the patterns sets 1 and 3, and 9 with the pattern set 2. It has to be mentioned that the number of false positives and the number of attacks detected are related to the threshold. A high threshold reduces the number of false positives but also the number of attacks detected, whereas a low threshold increases the false positives but also the number of attacks detected.

By taking into account all the five criteria we used to compare the quality of the pattern sets, we can state that the variable-length pattern set performs the best. It contains longer patterns than the other two sets, requires a smaller pattern table, covers nearly all the events of normal user sessions, raises no false positives, and detects eight out of ten attacks.

5. Conclusions

We have presented an 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 sub-sequences 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. We have described a technique to generate tables of

variable-length patterns automatically. To construct the patterns, the Teiresias algorithm, a method initially developed for the discovery of rigid patterns in unaligned biological sequences, is used in combination with a pattern-reduction algorithm.

We showed that the variable-length pattern model has several advantages over the fixed-length model. Among other advantages, the variable-length patterns are longer than the fixed-length patterns, and fewer patterns are needed to cover the training sequences. As a consequence, they are more process-specific and the pattern-matching can be implemented more efficiently.

Further work will concentrate on validating our approach for other network services and on implementing a real-time pattern-matching algorithm.

References

Debar, H., Dacier, M., Nassehi, M., & Wespi, A. (1998a). Fixed vs. variable-length patterns for detecting suspicious process behavior. In J.-J. Quisquater, Y. Deswarte, C. Meadows & D. Gollmann (Eds.), Computer Security - ESORICS 98, Lecture Notes in Computer Science (Vol. 1485, pp. 1-16). Berlin, Heidelberg: Springer.

Debar, H., Dacier, M., & Wespi, A. (1998b). Reference audit information generation for intrusion detection systems. In R. Posch & G. Papp (Eds.), Information Systems Security, Proceedings of the 14th International Information Security Conference IFIP SEC'98 (pp. 405-417). Vienna, Austria, and Budapest, Hungary: Chapman & Hall.

Debar, H., Dacier, M., Wespi, A., & Lampart, S. (1998c). An experimentation workbench for intrusion detection systems. Technical Report RZ 2998. IBM Research Division, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland.

D'haeseleer, P., Forrest, S., & Helman, P. (1996). An immunlogical 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.

Esmaili, M., Safavi-Naini, R., & Pieprzyk, J. (1995). Computer intrusion detection: A comparative survey. Technical Report 95-07. Center for Computer Security Research, University of Wollongong, Wollongong, NSW 2522, Australia.

Forrest, S., Hofmeyr, S.A., & Somayaji, A. (1997). Computer immunology. Communications of the ACM, 40(10): 88-96.

Forrest, S., Hofmeyr, S.A., Somayaji, A., & Longstaff, T.A. (1996). A sense of self for UNIX processes. In Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy (pp. 120-128). IEEE Computer Society Press.

Forrest, S., Perelson, A.S., Allen, L., & Cherukuri, R. (1994). Self-nonself discrimination. In Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy (pp. 202-212). IEEE Computer Society Press.

Kosoresow, A.P., & Hofmeyr, S.A. (1997). Intrusion detection via system call traces. IEEE Software, 35-42.

Rigoutsos, I., & Floratos, A. (1998). Combinatorial pattern discovery in biological sequences. Bioinformatics, 14(1): 55-67.

Teng, H.S., Chen, K., & Lu, S. C.-Y. (1990). Adaptive real-time anomaly detection using inductively generated sequential patterns. In Proceedings of the IEEE Symposium on Research in Security and Privacy (pp. 278-284). IEEE Computer Society Press.