PREPRINT

Submitted to the *14th IFIP Int'l Information Security Conf. "SEC 98"*
Vienna, Austria, and Budapest, Hungary, Aug. 31–Sept. 4, 1998
(Proc. to be published by Chapman and Hall)

Our file:
6593

Reference Audit Information Generation for Intrusion Detection Systems

Hervé Debar, Marc Dacier and Andreas Wespi

IBM Research Division, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

Abstract

This paper addresses the problem of generating reference audit information used in the intrusion detection technique proposed by S. Forrest *et al.* [1]. This technique uses a model of normal behavior of the information system being monitored to detect attacks against it. We present a novel approach to collect the reference behavior information used by the intrusion detection system to solve the problem identified in [1]. The model of normal behavior is extracted from this reference information. This model is then tested against real user activity and attacks.

[1] S. Forrest, S.A. Hofmeyr, and A. Somayaji, "Computer immunology," *Commun. ACM*, vol. 40, no. 10, October 1997.

1 Introduction

We are working on an intrusion detection prototype using as one of its components the anomaly detection method developed by S. Forrest *et al.* [11]. One of the requirements of the method is to obtain **reference information** that adequately contains all the possible behaviors exhibited by the system. It is essential that this reference information not contain any attacks, as it is used to train the intrusion detection system. We propose a method to meet this requirement in the context of the reference information required by this specific intrusion detection technique.

We also focus our efforts on a specific environment, which is an ftp server providing service to both registered and anonymous users. We have chosen ftp as our first target because it is a widespread service in the Internet environment, and one with well-known vulnerabilities. Of course, other services (e.g. http, sendmail, finger) will be considered later.

The paper is organized as follows. Section 2 presents general concepts concerning intrusion detection. Section 3 describes the intrusion detection prototype that we have developed. Section 4 presents our initial approach for generating the reference information for the intrusion detection system, namely using audit events recorded during real user activity, and some results obtained. Section 5 presents our current method to generate this reference, and Section 6 experimentally validates this method.

2 Intrusion detection

Since the seminal work by D. Denning in 1981 [2], many intrusion detection prototypes have emerged. There are two complementary trends in intrusion detection: (1) using the knowledge accumulated about attacks and looking for evidence of the exploitation of these attacks, and (2) building a reference model of the usual behavior of the information system being monitored and looking for deviations from the observed usage. The first trend is often referred to as misuse detection [3], but also as detection by appearance [4]. In this paper, we use the term "knowledge-based" intrusion detection, which we feel describes more precisely the technique being used. Conversely, the second trend is referred to as anomaly detection [3] or detection by behavior [4], and we use the term "behavior-based" intrusion detection.

- Knowledge-based intrusion detection techniques exploit the knowledge accumulated about specific attacks and system vulnerabilities. The intrusion detection system contains information about these vulnerabilities and even looks for an attempt to exploit these vulnerabilities. When such an attempt is detected, an alarm is triggered; otherwise the actions of the user are deemed appropriate from a security point of view.

  In other words, any action that is not explicitly recognized as an attack is considered acceptable.

  Advantages of the knowledge-based approaches are that they have the potential for a very low false alarm rate, and the contextual analysis proposed by the intrusion detection system is detailed, making it easier for the security officer using this intrusion detection system to take preventive or corrective action.

  Drawbacks include the difficulty of gathering the required information on the known attacks and keeping it up to date with new vulnerabilities and environments. We are maintaining a vulnerability database to which we add five or six new vulnerabilities weekly. Maintenance of the knowledge base of the intrusion detection system requires careful analysis of these vulnerabilities and is therefore a time-consuming task.

- Behavior-based intrusion detection techniques assume that an intrusion can be detected by observing a deviation in the behavior of the system. The model of normal behavior of the system is extracted from reference information collected by various means. The intrusion detection system later compares this model with the current activity. When a deviation is observed, it generates an alarm.

  In other words, anything that does not correspond to a previously learned behavior is considered unacceptable.

  Advantages of the behavior-based approaches are that they can detect attempts to exploit new and unforeseen vulnerabilities. They can even contribute to the (partially) automatic discovery of these new attacks. They are less dependent on operating-system-specific mechanisms. They also help detect "abuses of privileges" types of attacks that do not actually involve exploiting any security vulnerability.

  The high false alarm rate is generally cited as the main drawback of behavior-based techniques, because the entire scope of the behavior of the information system may not be covered by the learning phase. Also, behavior changes with time, introducing the need for periodical on-line retraining of the behavior profile, resulting in either unavailability of the intrusion detection system or additional false alarms. The information system can undergo attacks at the same time the intrusion detection system is learning the behavior. As a result, the behavior profile may contain intrusive behavior, which would not be detected as anomalous.

Various tools can be used to implement both approaches. For example, statistics [3], expert systems [5], signature analysis [6], neural networks [7], user intention identification [8] and model-based reasoning [9] have been used to implement various research prototypes and commercial products.

We have implemented a prototype of a behavior-based intrusion detection system, which is described in the following section. This tool monitors the behavior of UNIX services, not users. The changes in behavior are thus much smaller, easier to learn, and not subject to frequent updates. The problem addressed in this paper concerns the generation of the initial reference information from which the model of the service will be extracted.

3 The intrusion detection system

The intrusion detection approach that we are following has been described in [10, 1, 11]. UNIX processes are described by the sequence of audit events 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) sequences of audit events will appear.

The upper part of Fig. 1 creates the model off-line. Audit events (represented by letters here, imagine that A=fork, B=open, C=read, and so on) are recorded from the ftp daemon, which has been triggered by some experiment process. This recording runs through a filtering and reduction process, whose purpose is explained later. When the experiment is completed, the entire audit information is used to generate the patterns of the model. The patterns are generated as presented in [1], using a simple pattern extraction algorithm to extract all the possible fixed-length patterns. This model is then used as input for the on-line intrusion detection part.

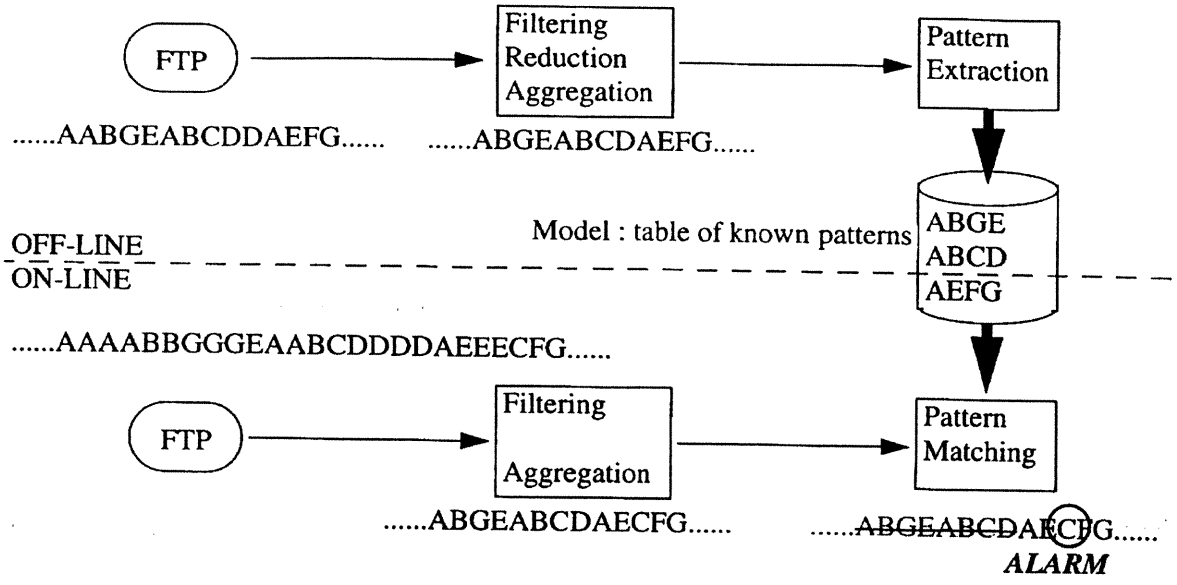

Figure 1: Intrusion detection process

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 filtering and reduction mechanism, in real time. Then, the resulting sequence is matched on the fly to the patterns in the model. We also say that this sequence is **covered** by the patterns, and the resulting **coverage** is the percentage of audit events in the sequence covered by the patterns. When no known pattern matches the current stream of audit events, an alarm is generated (as shown in Fig. 1). We apply this technique to AIX, using C2 audit records to record the activity of the process (ftpd) that we are monitoring.

The purpose of the Filtering/Reduction/Aggregation box is (1) to filter the incoming audit stream into processes, (2) to remove processes that have generated exactly the same stream of audit events from start (*fork*) to finish (*exit*) (this is not used in the on-line version, because keeping all known process images would become too costly in terms of memory and search after some time), and (3) to remove consecutive occurrences of system calls. The filtering prevents the introduction in the audit stream of arbitrary context switches by the operating system. The reduction keeps only "unique" process images for model extraction. The aggregation comes from the observation that strings of n consecutive audit events are quite frequent, with n exhibiting small variations. A good example is the ftp login session, where the ftp daemon closes several file handles inherited from inetd, sometimes 17, sometimes 20, sometimes 21, with no obvious cause. Therefore, we have decided to simplify the audit trail in order to obtain a model containing fewer and shorter patterns. There is no claim of equivalence between the reduced (simplified) audit trail and the original one, the new one has possibly less semantic content. This is an experimental choice, and we will remove the aggregation part if the true/false alarm performance of the intrusion detection system is not satisfactory.

The most obvious aggregation replaces 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. The tables in the remainder of the paper include not only level 1 aggregation (A=A+), but also level 2 and 3 aggregation (AA=AA+, AAA=AAA+). We do

not use the last two currently, but will use them if more precision is required while building models for other detection methods (e.g. statistics, neural networks).

4 Designing an environment for user activity simulations

The most commonly practiced method for training behavior-based intrusion detection systems is to record the activity of the users on a normal workday. This recording is used as the reference information for the training phase that will extract the model. Our first prototype improves this approach by creating a workbench with which we can reproduce the reference information and record it under varying external conditions. The recording procedure ensures that this reference information is attack-free.

4.1 Description of the simulation environment

The first approach that we present to create this sample is to simulate user sessions. However, we wish to do the recording in a closed environment to ensure that no attacks against the ftp server take place. We achieve this goal by asking several participants in our laboratory to record ftp sessions against an internal ftp server that supports both users and anonymous connections, and to dump their terminal interaction at the shell level using *script* [12]. The ftp server is then reconfigured inside a closed network to prevent interactions with the outside world.

Once this recording process is complete, we automatically transform these recordings into simple *expect* [13] scripts, which recreate the interactive session. User-related information such as username, password, and target ftp server, are variable to allow the parallelization of scripts (removal of the user-script dependency that allows the same script to be run simultaneously several times against the same server). These *expect* scripts contain activities performed by the users either as themselves (having an account on the server) or as "anonymous" (anonymous ftp service). Analysis of their content ensures that no user-controlled attack script is included in our collection. Attacks scripts exploiting ftp vulnerabilities are included in the workbench, but are used for testing purposes only.

Additional scripts are added to the database to perform specific actions, such as using *tar* and *compress* to transfer files, which were not in the original configuration of the recordings. This need comes from the attack simulator. Some attacks involve the use of *tar* and compression programs to create setuid shells, therefore our normal user activity also has to invoke these commands.

To handle these scripts, a workbench controller has been developed. The design requirements for this workbench controller are (1) automatic setup of the environment, both on the server (ftp configuration) and on the client, (2) creation of normal user activity allowing variable load and targets, and (3) automatic launching of attacks. The workbench consists of several setup scripts and a controller for running the user sessions. The controller is split between the server and the client, to allocate resources and pass information between them. The protocol is described as follows:

1. The client looks for a free user on its side and locks it. It then requests a user identity from the server side.
2. The server receives the request and forks a subserver to handle the request. The subserver looks for a free user on its side and locks it. It answers with the name of this user.

3. The client then picks a script in the script database (randomly, sequentially, according to a configuration file) and starts it. When the script is completed, the client prompts the server.
4. The subserver cleans the user environment on the server, removes the lock, and terminates.
5. The client cleans the user environment on the client machine, removes the lock, and terminates.

The controller server acts as a standard daemon, which can process several requests simultaneously. An atomic locking mechanism is used to prevent collisions. The controller client is encapsulated into a wrapper that allows one to configure the way scripts are to be run. The parameters are the total number of scripts to run, the time to wait between two script executions, and the selection of scripts to run (e.g. all scripts, scripts involving anonymous users, script involving normal users, some portion of the scripts database).

This workbench is a very effective environment for evaluating intrusion detection systems. It supports four operating systems, but is currently limited to ftp activity. We are working on installing intrusion detection systems using different techniques, and extending the scope of the workbench beyond the ftp service to evaluate the effectiveness of these different techniques [14].

4.2 Recordings of user activity

We performed a number of experiments to record the reference behavior from which the model is extracted. These experiments involve (1) varying the load on the server (number of simultaneous connections), (2) changing the network interface (Ethernet or token ring), and (3) picking out different subsets of scripts.

We separate our user scripts into four groups:

- all the scripts that we have in the database (set 1);
- the scripts involving normal user behavior (set 2);
- the scripts involving anonymous user behavior (set 3);
- and the scripts involving normal user behavior but without those that have been added to include the usage of tar and compress (via the site exec command) (set 4, subset of set 2).

Unfortunately, the small number of ftp sessions (55) does not allow a complete separation between a representative training set and a complete false alarm evaluation set.

The experimentation results are presented in Table 1. The first column describes the experiment, the second gives the total number of scripts run, the third the total number of audits generated, the fourth the total number of processes created (output of the filtering), the fifth the number of unique processes generated (output of the reduction), and the sixth, seventh and eighth the number of unique processes generated with aggregation levels of 1, 2, and 3, respectively.

This table shows that the results of the experimentation are very homogeneous. The number of processes generated is in direct proportion to the number of scripts run. When we look for unique processes, we always find the same number of different processes (70). Aggregation does not reduce this number significantly. This is radically different from observations (in the LINUX environment) carried out using *strace*, monitoring actually at the system call level. There, we observed variances in the number of different processes generated from experiment

Table 1: Collection of user data

Experiment description	No. of scripts	Number of audit events in the reference set	Number of Processes				
			After filtering	After filtering and reduction	After filtering, reduction and aggregation—		
					Level 1	Level 2	Level 3
Sequence of all scripts run on Ethernet interface	55	111521	430	70	69	70	70
Sequence of all scripts run on token-ring interface	55	105064	430	70	69	70	70
Sequence of all scripts run simultaneously two network interfaces (token ring & Ethernet)	110	201357	860	70	69	70	70
Simulation (randomly selecting scripts), in parallel one on each network interface, light load	200	494126	1604	70	69	70	70
Simulation (randomly selecting scripts), in parallel three on each network interface, semi-heavy load	600	1082548	4583	70	69	70	70
Simulation of five users in parallel, using set 1	500	849095	3931	72	71	72	72
Simulation of five users in parallel, using set 2	500	299048	4008	25	24	25	25
Simulation of five users in parallel, using set 3	500	1038129	3764	49	49	49	49
Simulation of five users in parallel, using set 4	500	1118403	4178	40	40	40	40

to experiment, mainly because of operating system context switches and signals. This shows that even a low-level information source such as the C2 audit trail gives some abstraction to the data, such as eliminating context switches, signals, and interrupts.

However, as the process for creating the scripts and running the experiments is very time consuming, we started looking for another method for generating this reference information.

5 Systematic generation

Using user "controlled" manipulations to generate the reference behavior of the service is not optimal. Our requirements are that in addition to being attack-free and reproducible, this reference behavior must contain all the possible legitimate commands supported by the ftp service, and these commands must be received by the ftp server from the network interface. The second requirement is that these commands are either generated by an ftp client or directly "manufactured", for example, by a packet-spoofing tool.

5.1 Software engineering and source code analysis

The first possibility to generate the reference is to extract the complete call graph of the code considered. This process is very heavy, and is only possible by acquiring source code for the service considered. However, with the increasing complexity of programs and the fact that many system calls would actually happen inside the libraries for which source code may not be available, we have eliminated this course of action because it is too cumbersome. Moreover, this technique does not fit our client-side requirement.

5.2 Macroscopic simulation as a finite state automaton

This second approach is based on a finite state automaton that implements all possible actions that can be performed by a user. Generating normal behavior means simply running the automaton. An example of this is given in Fig. 2, which shows a partial representation of the required finite state machine to simulate the client of an ftp daemon. The one in Fig. 2 depicts only a very limited subset of the some 70 different commands (without options!) that the ftp daemon can analyze, and not all paths are represented (e.g. one may issue the ls command after a put command).

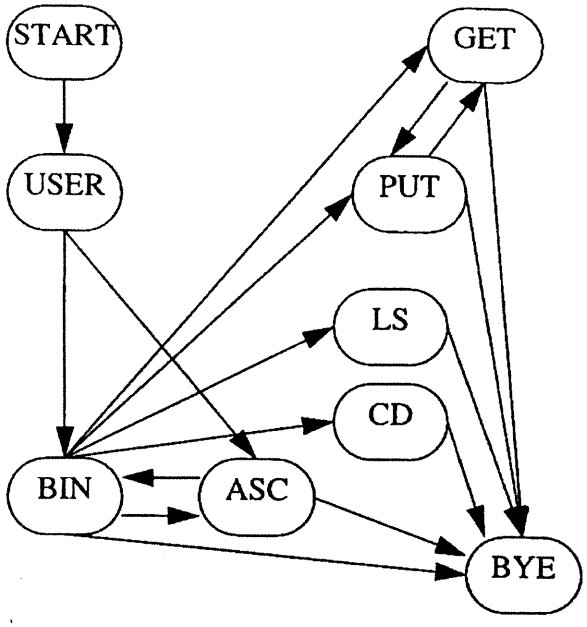

Figure 2: Over-simplified transition graph for simulating ftp client

However, only a very limited subset of these commands is actually known to users and used by them. Therefore, it would make sense to enrich these arcs with transition probabilities. As we are interested in the logic of this process, a natural solution for this kind of modelization would be to use stochastic Petri nets [15], whose coverage graphs are Markov graphs [16]. We could then use those Markov graphs to generate test suites using, for example, classical statistical software testing methods [17].

This approach is very difficult to implement for three main reasons:

1. The values of the transition probabilities are difficult to determine and are specific to the experimental environment.
2. This approach is only suitable when the set of commands is limited. It can simulate an ftp client but not a normal user shell session.
3. Even in simple cases such as ftp clients, implementation problems can arise. Indeed, ftp clients on different platforms sometimes have a different syntax, and the commands they generate make the ftp server react in different ways. Also, maintaining the coherence of the environment is quite difficult, as the automaton must analyze the return information given by the ftp daemon to decide which parameters it can insert in the next command. For the automaton to generate meaningful sequences of commands, it has to analyze the information returned by the ftp server and create the subsequent command accordingly, otherwise the number of error messages generated by the ftp daemon would be much higher than what the normal usage pattern shows.

These difficulties encountered in modeling ftp activity are true for other services as well. Therefore, this approach was dropped from the current implementation of the workbench.

5.3 Approximating user behavior with test suites

A simpler approach consists of using the test suites provided by the development laboratories that develop operating systems. In our case, we obtained access to the test suite used by the

AIX development team. This test suite, called functional verification tests (FVT for short), tests the compliance of the implementation of the TCP/IP services with their specification from the functional point of view, i.e. it tests the server functionality using the client in the case of ftp. These FVT are implemented within the DejaGnu [18] testing framework as a set of expect scripts, which exercise all the possible actions that a user can send to an ftp server. Owing to the portability requirements of the workbench environment, we modified these FVT not only to run on a pair of AIX boxes, but also to use LINUX and SUN boxes as clients or servers.

These FVT do not offer a realistic view of what users usually do, but at least they let us see how the system should behave when responding to normal requests. Actually, these test cases implement very simple scripts to test one command at a time. In that sense, they can be considered as traveling a subgraph of the one presented in Fig. 2, where the length of the path would not exceed one or two commands between **start** and **bye** (see Fig. 2).

This approach is also interesting to evaluate the false alarm rate of knowledge-based intrusion detection systems. Because the FVT will make the network daemons generate all sorts of events that do not (or very infrequently) occur in real-life systems, it can confuse these events with attacks scenarios. In this case, every alarm generated by a knowledge-based intrusion detection system is a false alarm.

If the intrusion detection system has rules governing the normal behavior of the network service, running such a test suite allows us to check the completeness of the normal behavior model and also to interpret alarms generated by the intrusion detection system as false alarms. However, the capability to fully train a behavior-based intrusion detection system on this kind of data is still an open issue, as the profiles may contain many "borderline" behaviors not actually exercised by the users. The purpose of the next section is to show results for the usage of this FVT data to record different reference information sets, which in turn are used to generate models for the intrusion detection system.

6 Experimentations with the functional verification tests

6.1 Recording of FVT data

Recording the FVT sessions was done quite simply by running all the available tests. Table 2 presents the same results that have been presented in Table 1, but for the functional verification tests data collection. Three separate runs of the FVT were made to compare the resulting reference information. Data collection took place on an isolated network, with as little interference as possible from other applications.

The first remark on the experimentations with the FVT is that even though all runs generated the same number of processes (401), the number of unique processes varies from run to run. We do not have a definite explanation for this, as the network on which the experimentations were performed was isolated and did not support additional load. We suspect that this variation depends on one of the tests in the FVT, the ftp proxy setup test, but have not been able to verify it.

The second observation is that 77 unique process images for 487 tests is very small. Our explanation is that many of these tests are redundant, i.e. they transfer exactly the same file in different modes (e.g. ascii, ebcdic, struct, record). The basic sequence (login, set parameters, transfer, logout) is repeated n times, but the parameter changes affect only the network level, and not the system. Therefore, we do not observe changes in the audit trail.

The third observation is that, contrary to the user simulation case, aggregation reduces

Table 2: Collection of FVT data

Experiment description	Number of scripts	Number of audit events in the reference set	Number of Processes				
			After filtering	After filtering and reduction	After filtering reduction and aggregation		
					Level 1	Level 2	Level 3
Sequential run of all available FVT tests (1)	487	87793	401	77	57	65	73
Sequential run of all available FVT tests (2)	487	878137	401	73	58	62	68
Sequential run of all available FVT tests (3)	487	89128	401	71	56	61	67

the number of different processes in a significant way. The explanation we have found is that even with these very similar processes, the recording of audits is sometimes still not completely identical, particularly on the login sequence depending on the number of open file handles handed by the inetd daemon to the ftpd daemon. Aggregation further reduces these differences, making it clear that the ftp daemon activity is quite limited.

We extract the model of behavior from these process images, which we consider as our reference information.

6.2 Generation of the model

Of these experiments, we use the first one to build the model (highlighted value in Table 2). We validate this model against the two other FVT runs. Of course, generating the model for an intrusion detection product would require several FVT runs.

There are two variables that we can vary to generate the fixed length patterns for the model [11]. The first one is the length of the patterns considered. This ranges from two consecutive audit events up to the length of our shortest sequence. The second parameter is the number of times the sequence occurs in the training set. This gives us an evaluation of the representativity of the sequence, or its contribution to the coverage.

The results obtained are summed up in Table 3. Rows represent pattern sizes from 2 to 6, columns the number of occurrences required. Each cell contains four lines, the number of patterns generated, the coverage for the training set (experiment 1), and the coverage for the two test sets (experiments 2 and 3).

The results obtained are very good, even with long patterns, which potentially carry more semantic information. The coverage is well above 98% of all the data sets, which leads us to the conclusion that our method represents well the data we wish to model.

6.3 Evaluation of the false alarm rate of the model

We have applied this model to our user simulation data set to evaluate how well this model actually represents live data. Results in Table 4 include only models with patterns of length 2, 4, and 6, and repetitions, respectively, of 2, 4, and 1, out of the complete picture shown in Table 3. For this test, we reused the experiments presented in Table 1.

First, the percentage of audit events covered is large as well. It is difficult to draw conclusions from global figures, and therefore we took a closer look, computing the same statistics on a per-process basis. This closer look shows that some processes are very poorly covered. After analysis, these processes are related to the execution of the *tar* command, used by some of our users to aggregate directories for downloading. This is a service offered by our ftp configuration, but it does not belong to the standard set of services offered by a basic ftp daemon, and it is not part of the FVT. Therefore, we had to generate the patterns for the *tar* command and add them to our model.

Table 3: Generation of reference tables

Size of Patterns	Minimum of occurrence=1	Minimum of occurrence=2	Minimum of occurrence=3	Minimum of occurrence=4	Minimum of occurrence=5
2	71[a] 99.567% 99.534% 99.556%	59 99.509% 99.477% 99.497%	57 99.480% 99.421% 99.467%	56 99.480% 99.421% 99.467%	52 99.422% 99.364% 99.408%
3	122 99.466% 99.449% 99.452%	92 99.350% 99.336% 99.334%	88 99.321% 99.266% 99.304%	85 99.321% 99.266% 99.304%	80 99.148% 99.096% 99.127%
4	178 99.422% 99.378% 99.408%	127 99.177% 99.124% 99.157%	119 99.134% 99.040% 99.112%	114 99.105% 99.011% 99.083%	107 98.975% 98.884% 98.950%
5	239 99.307% 99.280% 99.304%	168 99.119% 99.082% 99.112%	152 99.018% 98.955% 99.009%	145 98.874% 98.814% 98.861%	137 98.744% 98.687% 98.728%
6	307 99.321% 99.264% 99.304%	205 99.073% 99.026% 99.053%	182 98.975% 98.913% 98.950%	172 98.803% 98.743% 98.772%	157 98.657% 99.408% 98.624%

[a]Not 100% coverage because the number of events in the process is not necessarily a multiple of the pattern length.

Table 4: Evaluation of the reference tables with user data

Experiment description	Pattern size = 2 repetition = 2	Pattern size = 4 repetition = 4	Pattern size = 6 repetition = 6
Sequence of all scripts run on Ethernet interface	190 (99.246%)	405 (98.431%)	410 (98.411%)
Sequence of all scripts run on token ring interface	190 (99.246%)	405 (98.431%)	410 (98.411%)
Sequence of all scripts run simultaneously on the two network interfaces	155 (99.400%)	347 (98.684%)	353 (98.635%)
Simulation (randomly selecting scripts) in parallel, one on each network interface, light load	155 (99.400%)	347 (98.684%)	353 (98.635%)
Simulation (randomly selecting scripts) in parallel, three on each network interface, semi-heavy load	155 (99.400%)	347 (98.684%)	353 (98.635%)
Simulation of five users in parallel, using all available scripts, running 100 scripts each	158 (99.401%)	350 (98.675%)	358 (98.644%)
Simulation of five users in parallel, using anonymous scripts, running 100 script each	34 (99.604%)	81 (98.643%)	100 (98.352%)
Simulation of five users in parallel, using normal user scripts, running 100 scripts each	126 (99.368%)	276 (98.615%)	259 (98.701%)
Simulation of five users in parallel, using normal user scripts without the ones using "site exec", running 100 scripts each	50 (99.750%)	189 (99.019%)	174 (99.097%)

Now, how can we reliably detect attacks? We imagine that for very long processes, we might get a significant number of non-matching audit events, which might lead to some false alarms. Therefore, we have decided to attach our detection process to the length of the sequence of consecutive uncovered audit events. We observe occurrences of one non-matching audit event, and sometimes two consecutive non-matching audit events, but not more. Therefore, we decide that any non-matching sequence of more than two audit events will be flagged as an attack. This sounds reasonable when we recall that we are observing actions of users at a very low level, and that it is absolutely impossible for an attack attempt to be carried out with a single or even two consecutive audit events.

6.4 Evaluation of the true alarm rate of the model

We have also tested our intrusion detection prototype by carrying out attacks against our isolated ftp server. Our attacks all involve getting root access on the server using a vulnerability of the ftp daemon. These vulnerabilities fall into two categories, configuration errors of the ftp service (such as letting the anonymous ftp directory world writable), and exploitation of bugs in the ftp code (after compiling and installing vulnerable versions). Moreover, our attacks are as stealthy as possible, making sure that no new entries are created in the table listing all possible system calls. For each attack and model, two figures are given: the percentage of covered audit events, and the length of the longest uncovered sequence. The table contains only a subset of the attacks that we have carried out against our server. The attack scripts have been developed by S. Lampart as a part of his master's thesis [19] while at IBM.

Table 5: Evaluation of the true alarm rate of the model

Attack code	Model (2,2)		Model (4,4)		Model (6,1)	
put forward	78.285%	34	76.571%	35	77.714%	34
put rhost	99.099%	1	97.297%	1	97.297%	2
site copy	85.039%	15	81.889%	18	81.889%	15
site exec copy ftpd	85.826%	15	83.464%	15	82.677%	15
site exec copy ls	85.039%	15	81.889%	18	81.889%	15

Using the threshold method described earlier, only the second attack would not be detected in our model. Actually, the first two attacks are very similar, because they involve putting a file to the server (legitimate action) and triggering its execution (illegitimate action). In the first case, the illegitimate action is carried out by the attack script and is detected. In the second case, the illegitimate action is not carried out and therefore the "potential" attack (as no intrusion has actually been carried out) is not discovered.

The second conclusion drawn from Table 5 is that the models perform differently, both in terms of coverage and length of uncovered sequence. We are currently looking at this issue to propose a method for selecting the best model.

7 Conclusion and future work

We have presented a method to generate reference information for intrusion detection systems that allows us to train a behavior-based model. The intrusion detection technique we use,

trained on this reference information, detects all attacks with potentially no false alarms. Our method requires the existence of test suites for the applications we wish to monitor.

We shall address two remaining issues in upcoming work. The first issue is the generation of patterns. We think that better strategies can be envisioned for creating the patterns, which would be more restrictive in what they tolerate as good behavior, and yet have the same coverage level. Also, we will test our intrusion detection prototype in a real-life environment to measure the actual false alarm rate over long periods and the efficiency in terms of real-time processing. We believe that our method will allow a very efficient anomaly detection technique. We can then improve the prototype for either real-time low overhead intrusion detection on the machine considered or for batch processing of large sets of audit files.

References

[1] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff, "A sense of self for Unix processes," in: *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1996.

[2] D. Denning, "An intrusion-detection model," *IEEE Transactions on Software Engineering*, vol. SE-13, February 1987.

[3] T.F. Lunt, R. Jagannathan, D. Anderson, C. Dodd, F. Gilham, C. Jalali, H. Javitz, P. Neuman, A. Tamaru, and A. Valdez, "System Design Document: Next-Generation Intrusion Detection Expert System (NIDES)," SRI International Technical Report, March 1993.

[4] P. Spirakis, S. Katsikas, D. Gritzalis, F. Allegre, J. Darzentas, C. Gigante, D. Karagiannis, P. Kess, H. Putkonen and T. Spyrou, "SECURENET: A network-oriented intelligent intrusion prevention and detection system," *Network Security Journal*, vol. 1 no. 1, November 1994.

[5] M. Esmaili, R. Safavi-Naini, and J. Pieprzyk, "Computer Intrusion Detection: A Comparative Survey," Center for Computer Security Research, University of Wollongong, Australia, Technical Report, Ref. 95-07, May 1995.

[6] S. Kumar and E.H. Spafford, "A pattern matching model for misuse intrusion detection," in: *Proceedings of the 17th National Computer Security Conference*, Baltimore, MD, October 1994.

[7] H. Debar, M. Becker and D. Siboni, "A neural network component for an intrusion detection system," in: *Proceedings of the 1992 IEEE Symposium on Research in Computer Security and Privacy*, Oakland, CA, May 1992.

[8] T. Spyrou and J. Darzentas, "Intention modelling: Approximating computer user intentions for detection and prediction of intrusions," in: *Proceedings of IFIP SEC'96*, Samos, Greece, May 1996.

[9] T.D. Garvey and T.F. Lunt, "Model-based intrusion detection," in: *Proceedings of the 14th National Computer Security Conference*, Washington DC, October 1991.

[10] P. D'haeseleer, S. Forrest, and P. Helman, "An immunological approach to change detection: algorithms, analysis, and implications," in: *Proceedings of the 1996 IEEE Symposium on Computer Security and Privacy*, Oakland, CA, May 1996.

[11] S. Forrest, S.A. Hofmeyr, and A. Somayaji, "Computer immunology," *Communications of the ACM*, vol. 40, no. 10, October 1997.

[12] "Script", *UNIX manual page.*

[13] Don Libes, *Exploring Expect – A TCL-based Toolkit for Automating Interactive Programs*, O'Reilly and Associates, Sebastopol, CA (1995).

[14] H. Debar, M. Dacier, A. Wespi and S. Lampart, "An experimentation workbench for intrusion detection systems" IBM Research Report, RZ 2998 March 9, 1998.

[15] S. Natkin, *Les réseaux de Petri stochastiques*, Thèse de Docteur Ingénieur, CNAM, Paris 1980.

[16] M. Molloy, "Performance analysis using stochastic Petri nets", *IEEE Transactions on Computers*, vol. 39 no. 9, 1982.

[17] P. Thevenod-Fosse, "Software validation by means of statistical testing: Retrospect and future direction," in: *Proceedings 1st IFIP Working Conference on Dependable Computing for Critical Applications*, Santa Barbara, CA, 1989.

[18] Rob Savoye, "The solution: DejaGnu," *free software report*, vol. 3 no. 1, 1994.

[19] S. Lampart, *Creation of an Intrusion Detection Testbed*, Professional Thesis, Eurecom Institute (France), July 1997.