Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach

Calvin Ko
Trusted Information Systems, Inc.
444 Castro Street, Suite 800
Mountain View, CA 94041
ko@tis.com

Manfred Ruschitzka Karl Levitt
Department of Computer Science
University of California, Davis
Davis, CA 95616
{ruschitzka, levitt}@cs.ucdavis.edu

Abstract

This paper describes a specification-based approach to detect exploitations of vulnerabilities in security-critical programs. The approach utilizes security specifications that describe the intended behavior of programs and scans audit trails for operations that are in violation of the specifications. We developed a formal framework for specifying the security-relevant behavior of programs, on which we based the design and implementation of a real-time intrusion detection system for a distributed system. Also, we wrote security specifications for 15 Unix setuid root programs. Our system detects attacks caused by monitored programs, including security violations caused by improper synchronization in distributed programs. Our approach encompasses attacks that exploit previously unknown vulnerabilities in security-critical programs.

1. Introduction

Many security problems are directly or indirectly related to vulnerabilities in security-critical programs (e.g., privileged programs). Intruders exploit vulnerabilities in these programs to gain unauthorized access or to exceed their privileges in a system. Also, inappropriate uses of these programs could lead to security breaches.

Intrusion detection is an alternative approach to coping with these problems besides testing and verification. Three major approaches to intrusion detection are anomaly detection, misuse detection, and specification-based detection. Anomaly detection [1, 2] assumes that attacks will result in behavior different from that normally observed in a system, and can be detected by comparing the current behavior with the pre-established normal behavior. Statistics-based [11, 5], rule-based [15], and immunology-based [3] methods have been employed in modeling normal behavior. Anomaly detection has the advantage that no specific knowledge about security flaws is required in order to detect penetrations. However, it is difficult to set up the anomaly thresholds so that attacks produce significant anomalies. In addition, anomaly detection alone cannot detect all kinds of intrusions, since not all intrusions produce an identifiable anomaly.

Misuse detection [11, 9, 4] attempts to identify known patterns of intrusions (intrusion signatures) when they occur. A misuse detection system detects intrusions by matching the audit trails with the set of predefined signatures. It can guarantee the detection of an intrusion if a signature of the intrusion is included in the system. However, it cannot detect previously unknown attacks, since it is not possible to specify intrusion signatures for exploiting a vulnerability if the vulnerability is still unknown. Also, it is difficult to write signatures that capture all variants of an intrusion (e.g., different ways to exploit a known weakness).

Specification-based detection [8] relies on program specifications that describe the intended behavior of security-critical programs. The monitoring of executing programs involves detecting deviations of their behavior from these specifications, rather than detecting the occurrence of specific attack patterns. Thus, attacks can be detected even though they may not previously have been encountered. Early efforts focused on sequential programs, and their intended behavior was specified in terms of a static set of allowable operations [8].

This paper introduces a formal intrusion-detection model that makes use of *traces*, ordered sequences of execution events, for specifying the intended behavior of concurrent programs in distributed systems. A formal

1081-6011/97 $10.00 © 1997 IEEE

specification language expresses the set of valid operation sequences of such programs in a general and efficient manner. Moreover, an intrusion detection system for distributed computer systems and networks is discussed. According to the model, a specification describes valid operation sequences of the execution of one or more programs, collectively called a *(monitored) subject*. A sequence of operations performed by the subject that does not conform to the specification is considered a security violation. Since a specification determines whether an execution trace of a subject is valid, it is called a *trace policy*.

The valid operation sequences of subjects are specified in terms of grammars whose alphabets are system operations. We developed a novel type of grammar, *parallel environment grammars* (PE-grammars), for specifying trace policies. PE-grammars can describe many different classes of trace policies that are important to security, including behavior related to synchronization in concurrent or distributed programs. Parsing of audit trails thus becomes the detection mechanism in a specification-based detection system; it detects operations performed by subjects that are in violation of the trace policies. We designed PE-grammars in a way that permits them to be parsed efficiently in many practically important cases.

We developed a prototype intrusion detection system for Unix and wrote trace policies for 15 Unix programs. Our method can be used to detect known attacks on these programs and has the potential to detect previously unknown attacks. It can also detect exploitations of race-conditions in privileged programs and security violations caused by improper synchronization in distributed programs.

The remainder of the paper is organized as follows. Section 2 identifies several aspects of program behavior that are important to security. Section 3 describes the basic concepts and the monitoring model. Section 4 describes the specification language. Section 5 describes the design and implementation of a prototype specification-based detection system for a distributed system. Section 6 presents conclusions and suggests future research.

2. Security-Relevant Aspects of Program Behavior

Our goal is to specify the intended behavior of programs that are security-relevant, as it is not possible to specify in full the behavior of a program. In this section, we discuss and identify aspects of program behavior that are relevant to security.

- *Accesses of system objects:* A simple but important aspect of the behavior a program is the set of objects (e.g., files) it accesses. The executing program is treated as the subject of access control. Specifying the files a program can access permit the detection of many attacks involving privileged programs in Unix. In most cases, the number of files that a privileged program needs to access is very limited, and can be enumerated easily. For example, the finger daemon *fingerd* should execute only the finger program (*/usr/ucb/finger*), and read only some status files (e.g., */etc/utmp*, *.plan*, *.profile*). An attacker who exploits the finger daemon causing it to perform any other operations (e.g., execute programs other than the finger program or modify the password file) will be caught. By specifying the valid accesses of a privileged program, it is possible to detect attacks that cause the privileged program to perform accesses that are not part of the specification. In addition, one can specify the access policy of a suspect program so that any unauthorized accesses of files can be detected even if the program is a Trojan horse.

- *Sequencing:* In some situations, it is not only the set of operations performed by a program that is of concern, but also the order of these operations. For example, the *login* program ought to read the password file before it lets a user enter the system by starting a shell program and passing control to the user. Thus, it should be ascertained that the operation of reading the password file appears prior to the execution of a shell in the trace of a *login* program. Another example concerns locking. When a process locks a file for exclusive access, it needs to be ensured that the process removes the lock on the file before it exits in order to prevent the file from being locked permanently.

- *Synchronization:* In a distributed system, security failures often result from improper synchronization of programs. For example, if a user invokes the *passwd* program to change his or her password while the system administrator is editing the password file, the password file may be left inconsistent [7, Chapter 2]. In addition, synchronization problems occur in concurrent or distributed programs, whose execution consists of multiple processes. Therefore, the order among operations performed by different concurrent processes in a concurrent or distributed program is also of concern, and we need to be able to specify valid synchronization behavior of concurrent programs. Two predominant synchronization activities are mutual exclusion and precedence. Since two processes must not modify a file in the system simultaneously, one process has to wait until the other process finishes. Similarly, there may be a precedence relation between operations of different processes (e.g., process B should not read file C until process A finishes writing it).

- *Race conditions:* A race condition is a special case of the synchronization problem. If a program has a race-condition flaw, an attacker can affect the behavior of the program by performing certain operations during the execution of the program. Monitoring exploitations of a race condition requires monitoring of the operation sequence of the executing program and of all other relevant processes in the system.

3 The Model

A distributed system consists of a number of hosts that are connected by a network. The basic entities that perform operations on objects in the system (e.g., files) are processes. An event denotes an execution of an operation in the system, and is attributed to the process that performs the operation. Events happening in the system can be totally ordered [10], and the history of the system is the sequence of events that occurred since the system started.

Definition 1 (System Traces) The execution of a distributed system S produces a sequence of events

$$v_1, v_2 \cdots, v_i, v_{i+1}, \cdots,$$

which is called a *system trace* of the system. Each event v_i has an occurrence time, denoted by $C(v_i)$. Events are totally ordered, that is, $C(v_i) < C(v_{i+1})$ for all $i \geq 1$. A sequence of events $v_{l_1}, v_{l_2}, \cdots, v_{l_k}$ is a *subtrace* of a system trace if $l_1, l_2, \cdots, l_k$ is a subsequence of $1, 2, \cdots$. Two subtraces $v_1^a, v_2^a, \cdots, v_k^a$ and $v_1^b, v_2^b, \cdots, v_l^b$ are said to be *distinct* if and only if $v_i^a \neq v_j^b$ for all $1 \leq i \leq k$, $1 \leq j \leq l$.

The execution of a sequential process (or simply process) p_l in the system produces a sequence of events

$$v_1^l, v_2^l, \cdots, v_i^l, v_{i+1}^l, \cdots .$$

The sequence of events is called a *process trace*, and denotes the sequence of operations performed by the process from the time it starts to the time it terminates. A process trace is a subtrace of the system trace.

Definition 2 (Merge of Traces) Given two distinct subtraces V_1 and V_2 of V, the *merge* of the two traces is a subtrace of V, denoted by $V_1 \oplus V_2$, and is defined by $V_1 \oplus V_2 = v_1, v_2, v_3, v_4, \cdots, v_{m+n}$ if and only if there exist two subsequences $i_1, \cdots, i_m$ and $j_1, \cdots, j_n$ of the sequence $1, 2, \cdots, m+n$ s.t. $V_1 = v_{i_1}, v_{i_2}, \cdots, v_{i_m}$ and $V_2 = v_{j_1}, v_{j_2}, \cdots, v_{j_n}$.

Definition 3 (Filter of Traces) A filter ∇_p is a function that maps a trace $V = v_1, v_2, \cdots, v_n$ to another trace V_s, a subtrace of V, where p is a predicate on the set of possible event attributes, and V_s is obtained from V by removing all events v_i $(i \geq 0)$ in V s.t. $p(v_i) = false$. For example, given a predicate q s.t. $q(v)$ is *true* if only if the event e describes an operation performed by the user *ko*, $\nabla_q(V)$ is a subtrace of V that consists of events caused by *ko*.

3.1. Monitoring Programs

A program is a passive entity. To monitor a program means to monitor the executions of the program. An execution of a program is a *distributed process* $dp = \{p_1, p_2, \cdots, p_n\}$, $n \geq 1$, which consists of one or more processes. For instance, the execution of a sequential program is a distributed process consisting of a single process, while the execution of a distributed program or a concurrent program is a distributed process which consists of multiple processes.

The trace of a program execution is the sequence of events corresponding to the operations performed by the distributed process, which is the merge of the individual traces of the processes forming the distributed process. With $dp = \{p_1, p_2, \cdots, p_n\}$ denoting the distributed process, the execution trace is

$$V_{dp} = V_{p_1} \oplus V_{p_2} \oplus \cdots \oplus V_{p_n},$$

where V_{p_i} $(1 \leq i \leq n)$ is the execution trace of process p_i. V_{dp} is a subtrace of the system trace V.

A single program can have a number of executions existing at the same time. For example, two users can execute the same program at the same time, resulting in two distributed processes both running the program under the same or different operating systems. In some situations, the executions need to be monitored separately. In other situations, all executions of a program need to be monitored. For example, in order to detect whether a user using *passwd* and an administrator using *vi* are modifying the password file simultaneously, all executions of *vi* by the administrator (i.e., *root*) and all executions of *passwd* need to be monitored. In this case, the input to the monitor is the merge of the filtered traces corresponding to all executions of *vi* (predicate: user *root*) and the traces of all executions of *passwd*.

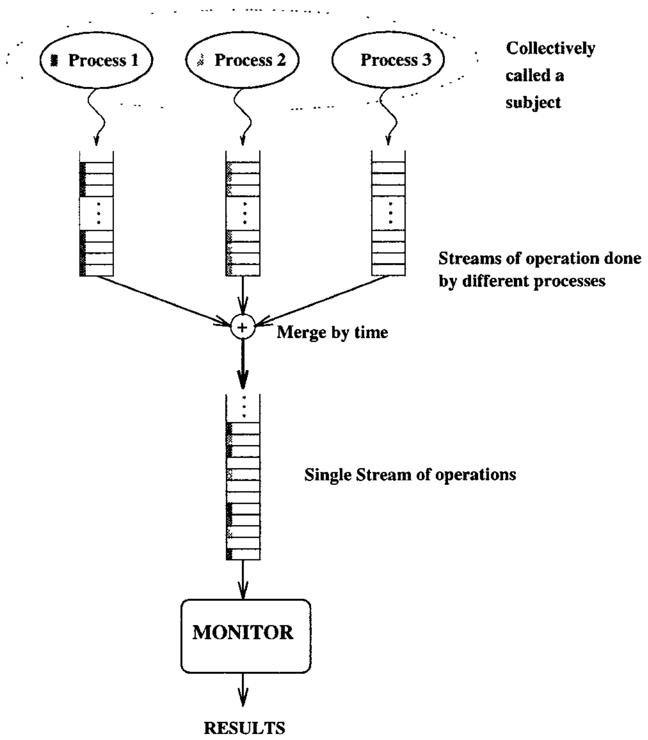
Process 1
Process 2
Process 3
Collectively called a subject
Streams of operation done by different processes
Merge by time
Single Stream of operations
MONITOR
RESULTS

Figure 1. The Monitoring Model.

3.2. The Subject of Monitoring

A trace policy describes the valid operation sequences of a single program execution, multiple program executions, a user, a group of users, a host, etc. The entity or entities are collectively called a *monitored subject*, or simply a *subject.* At the user level, a subject could be one or more program executions, one or more users, and one or more hosts. At the system level, a subject consists of one or more processes. A host refers to all processes running on the host. A user refers to all processes that are owned by the user.

Monitoring a subject means analyzing the sequence of operations performed by the subject. The execution trace of a subject is the time-ordered sequence of operations performed by the processes forming the subject. The execution trace of a subject $sp = \{p_1, p_2, \cdots, p_n\}$ is

$$V_{sp} = V_{p_1} \oplus V_{p_2} \oplus \cdots \oplus V_{p_n},$$

where V_{p_i} $(1 \leq i \leq n)$ is the execution trace of process p_i. V_{sp} is a subtrace of the system trace V. Fig. 1 depicts the situation of monitoring a program execution which involves three processes. Each process produces an execution trace. The process traces are *merged* to form the single trace of the three processes. The latter trace forms the input to the monitor.

3.3. Selection Expression

Each trace policy contains a selection expression, which serves to identify subjects that are to be monitored. The elements in a selection expression are derived from the characteristics of the distributed system, which include the set of programs P, the set of users U, and the set of hosts H. Moreover, the two special symbols $*$ and ? are used to signify "all" and "any", respectively. A selection expression is a list of selectors $(s_1, s_2, \cdots, s_n)$ where s_i $(1 \leq i \leq n)$ is a selector of the form

$$< PID,\ PS,\ US,\ HS >,$$

where

- PID is the distributed process selector, $PID \in \{* , ?\}$,
- PS is the program, $PS \in P \cup \{* , ?\}$,
- US is the user, $US \in U \cup \{* , ?\}$, and
- HS is the host, $HS \in H \cup \{* , ?\}$.

A selection expression identifies one or more subjects. A selection expression without the ? symbol defines a single subject; a selection expression consisting of ? defines a set of subjects. The symbol ? in a selection expression is similar to a variable in a generic template, which is given a value in an instantiation. For instance, $<*,$ *rdist*, *ko*, $?>$ defines a set of subjects which contains the subjects $<*,$ *rdist*, *ko*, $k2>$, $<*,$ *rdist*, *ko*, *blanc*$>$, and so on, with ? replaced by each of the hosts in the distributed system. Several selection expressions and the subjects they identify are listed below.

1. <?, *rdist*, $*$, $*$> defines a set of subjects; a subject here is a distributed process executing the *rdist* program for any user on any host, for example, <234, *rdist*, $*$, $*$>.
2. (<$*$, *passwd*, $*$, *blanc*>, <$*$, *vi*, *root*, *blanc*>) defines one subject: all executions of *passwd* and all executions of *vi* by *root* on the host *blanc.*
3. <$*$, ?, *ko*, $*$> defines a set of subjects; a subject refers to all executions of a program p $(p \in P)$ by user *ko* on any host.
4. <$*$, $*$, *ko*, *blanc*> defines the user *ko* on the host *blanc.*
5. <$*$, $*$, $*$, $k2$> defines the host *k2,* i.e., all processes on *k2.*

The execution of a program is a distributed process, which is identified by a process ID (*pid*), the user it is representing, the program it is executing, and the host on which it is running. Given a selection expression and a distributed process identified by its *pid*, user, program and host, a simple matching operation can determine whether the distributed process belongs to any of the monitored subjects specified by a selection expression.

4. Specification Language

In our approach, a trace policy, which captures the intended behavior of a program, is specified by means of a grammar. This grammar defines a formal language (a set of sentences) whose alphabet consists of program operations. Monitoring a subject amounts to syntax-driven parsing of the sequence of program operations executed by the subject. This sequence of operations (the trace) is obtained from audit trails in real time. An unsuccessful parsing attempt indicates a violation of the trace policy and triggers remedial responses.

4.1. Parallel Environment Grammars

In view of the large variety of possible trace policies and the constraints imposed by real-time parsing, the expressiveness of the grammar and parser efficiency demand special attention. We developed a novel type of grammar, the parallel environment grammar (PE-grammar) [7], to meet these demands. PE-grammars form parallel extensions of environment grammars [12] which have been applied to real-time data translations in heterogeneous relational database systems. Owing to their support of parallel sequences, PE-grammars can be used to specify traces of concurrent processes.

A PE-grammar involves four quantities: a set of terminals (the alphabet), a set of hyperrules, a set of environment variables, and a start expression. Rather than using production rules directly to define a language, PE-grammars use parameterized versions of such rules. These parameterized versions, called hyperrules, serve as templates for the dynamic generation of production rules as the parsing progresses. A production rule is generated by replacing the parameter variables of a hyperrule by their current values. Collectively, the parameter variables are referred to as the environment variables. An environment variable is either global or local to a sub-grammar, analogous to the concept of global and local variables in programming languages. The parallel aspect of a PE-grammar arises from the start expression, which defines several start notions. A PE-grammar can be viewed as consisting of n sub-grammars, each corresponding to one start notion in the start expression. The sentence derived from the start expression is the ordered merge of the sentence derived from the individual start notions. For an in-depth treatment of PE-grammars, see reference [7].

To introduce the general properties of PE-grammars, we will refer to the specific PE-grammar shown in Fig. 2. This PE-grammar contains two environment variables, E and L. E is a global environment variable and L is a local environment variable. Both environment variables are initialized to 0 on lines 1 and 2. Line 3 denotes the start expression, which consists of the two start notions <progA> and <progB>. In general, the start expression has the form $s_1 \parallel s_2 \parallel \cdots \parallel s_n$, where s_i is a start notion ($1 \leq i \leq n$). In a parallel derivation, each start notion s_i derives a sentence x_i, and the sentence derived from the start expression is the *ordered merge* $x_1 \oplus x_2 \oplus \cdots \oplus x_n$.

Lines 4-11 show the hyperrules of the grammar. In general, a hyperrule has the form

$$X_0 \rightarrow X_1 \, X_2 \, \cdots \, X_m \, [B]$$

where X_0 is a hypernotion, X_i, $1 \leq i \leq m$, is either a hypernotion or a terminal, and B is a set of attached actions. A hypernotion is a tuple $< x_1, x_2, \cdots, x_n >$, where x_i, $1 \leq i \leq n$ is a symbol or an environment variable. The set of attached actions may contain assignments to environment variables and semantic actions. For example, the hyperrule on line 5 has an attached environment assignment which decreases the value of E by 1.

```
    Environment Variables
1.  ENV int E = 0;
2.  LOCAL ENV int L = 0;

    Start Expression
3.  SE: <progA> || <progB>

    Hyperrules
4.  <progA> -> <writeA, E>.
5.  <writeA, 0> -> <openA> <closeA> { E = E - 1;}.
6.  <openA> -> open_A { E = E + 1; L = 1;}.
7.  <closeA> -> close_A.

8.  <progB> -> <writeB, E>.
9.  <writeB, 0> -> <openB> <closeB> { E = E - 1;}.
10. <openB> -> open_B { E = E + 1; L = 1;}.
11. <closeB> -> close_B.
```

Figure 2. An Example of a Parallel Environment Grammar.

Given a hyperrule in a PE-grammar, a production rule is obtained from the hyperrule and the current environment by replacing each environment variable e_i in the rule with its current value V_i; the hyperrule is called the *reference* of the generated production rule. A hyperrule serves as a template for the replacement. The left-hand side of the resulting production rule becomes a single protonotion (i.e., a hypernotion without any environment variables) and the right-hand side a combination of protonotions and terminals. The resulting production rules are equivalent to production rules of context-free grammars (with protonotions corresponding to nonterminals) and their applications correspond to derivation steps in context-free grammars.

However, the set of production rules is not constant; it varies with the environment as a derivation progresses. A change in the environment occurs when an environment assignment in B is executed upon successful application of a production rule. For example, the hyperrule on line 4 in Fig. 2 generates the production rule <progA> → <writeA, 0> when $E = 0$, and produces the production rule <progA> → <writeA, 1> when $E = 1$.

The language defined by the grammar in Fig. 2 is { open_A close_A open_B close_B, open_B close_B open_A close_A }. Table 1 shows a parallel derivation of the sentence { open_A close_A open_B close_B }. Sentential forms labeled with † represent intermediate results of derivation steps that involve executions of environment assignments or semantic actions. In general, a parallel derivation can be thought of as n individual derivations where s_i derives x_i for $1 \leq i \leq n$; the n derivations proceed in parallel and each consists of a sequence of (leftmost) derivation steps. The order among the derivation steps is driven by the occurrence times of the tokens. For an individual derivation, parsing corresponds to a (leftmost) traversal of the derivation tree of x_i. We affiliate with each internal node of the derivation tree (which represents a protonotion) a *traversal time*, the occurrence time of the leftmost leaf node (a token) in the subtree rooted at the internal node. In each step, the node with the earliest traversal time is traversed.

The parallel derivation consists of the two individual derivations of open_A close_A from <openA> (steps 1-4) and of open_B close_B from <closeA> (steps 5-8). In the table, L^1 and L^2 denote the local values of L from the perspective of sub-grammar 1 (corresponding to <progA>) and sub-grammar 2 (corresponding to <progB>). Let t_1, t_2, t_3 and t_4 denote the occurrence times of the tokens open_A, close_A, open_B and close_B, respectively, i.e., $t_1 < t_2 < t_3 < t_4$. These times pace the traversals of the two derivation trees (cf. discussion of Fig. 3 below). In derivation step 1, the production rule <progA> → <writeA, 0>, which is obtained from hyperrule 4 in Fig. 2 while the environment has the initial value $E = 0$, is applied to the first sentential form, changing it to <writeA, 0>. Step 2 converts it to <openA> <closeA> $\{E = E - 1\}$. In step 3, <openA> is first changed to open_A $\{E = E + 1; L = 1\}$, and after the execution of the environment assignment (which sets E and L to 1) it becomes open_A. Similarly, step 4 results in open_A close_A after setting E to 0. The derivation of open_B close_B proceeds analogously.

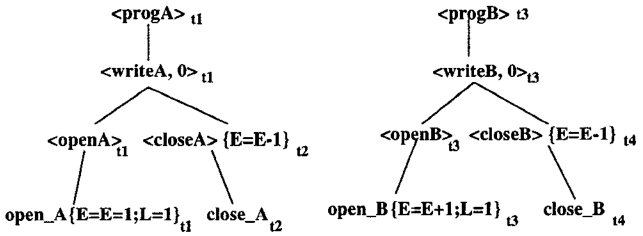

Figure 3. A Derivation Tree of the Example of a Parallel Derivation.

Fig. 3 depicts the derivation trees of the parallel derivation and shows a subscript for the traversal time affiliated with each node. The first derivation step expands <progA> (rather than <progB>) because $t_1 < t_3$. The next two derivation steps expand <writeA, 0> and <openA> for the same reason. Since $t_2 < t_3$, <closeA> is expanded next, followed by <progB>.

Note that open_A open_B close_A close_B is not a sentence of the grammar; after step 3 sets E to 1, <progB> (which would have an earlier traversal time than <closeA>$\{E = E - 1\}$) derives <write, 1>, which does not match any of the left-hand sides of the grammar. Thus, the parser returns a failure indication which signals the detection of a violation of the trace policy.

4.2. A Trace Policy for *Rdist*

The program *rdist* [13] (Remote File Distribution Program) is a Unix utility for maintaining identical copies of files over multiple hosts. It has a race-condition flaw [1], which enables an attacker to acquire root privileges. The flaw relates to the way in which *rdist* updates a file as well as to the semantics of the *chown* and *chmod* system calls regarding symbolic links. Specifically, the flaw enables a nonprivileged user to change the permission mode of any file in the system. It has been exploited by attackers to set the setuid bit of a system shell (e.g., */bin/sh*), resulting in a setuid root shell that is publicly executable. We explain below how our method detects such an attack.

[1] The flaw exists in 4.3 BSD Unix and other variants of Unix.

Step	Sub-grammar 1	Sub-grammar 2	E	L^1	L^2
	<progA>	<progB>	0	0	0
1.	<writeA, 0>		0	0	0
2.	<openA> <closeA> $\{E = E - 1\}$		0	0	0
3.	†*open_A* $\{E=E+1; L=1\}$ <closeA>$\{E=E-1\}$		0	0	0
	open_A <closeA> $\{E = E - 1\}$		1	1	0
4.	†*open_A close_A* $\{E = E - 1\}$		1	1	0
	open_A close_A		0	1	0
5.		<writeB, 0>	0	1	0
6.		<openB> <closeB> $\{E = E - 1\}$	0	1	0
7.		†*open_B* $\{E=E+1; L=1\}$ <closeB> $\{E=E-1\}$	0	1	0
		open_B <closeB> $\{E = E - 1\}$	1	1	1
8.		†*open_B close_B* $\{E = E - 1\}$	1	1	1
		open_B close_B	0	1	1

Table 1. An Example of a Parallel Derivation.

```
1. SPEC rdist  <?, rdist, *, blanc>
2.   ENV User U = getuser();
3.   ENV int  PID = getpid();
4.   ENV int FILECD[int];
5.   ENV int PATHCD[str];
6.   ENV str HOME = "/export/home/U.name";

7.   SE: <rdist>
8.   <rdist>    -> <valid_op> <rdist> |.
9.   <valid_op> ->
         open_r_worldread
     |   open_r_not_worldread
         {  if !Created(F) then
               violation();   fi    }
     |   open_rw
         {  if !(Dev(F)) then violation(); fi;   }
     |   creat_file
         {  if !((Inside(P, "/tmp") || Inside(P, HOME))
            then violation(); fi
            FILECD[F.nodeid] = 1;
            PATHCD[P] = F.nodeid;                      }
     |   creat_dir
         {  if !((Inside(P, "/tmp") || Inside(P, HOME))
            then violation(); fi                       }
     |   symlink
         {  if !((Inside(P, "/tmp") || Inside(P, HOME))
            then violation(); fi                       }
     |   chown
         {  if !(Created(F) and M.newouid = U)) then
            then violation();  fi                      }
     |   chmod
         {  if !(Created(F)) then violation(); fi      }
     |   rename
         {  if !(PtCreated(P) && Inside(M.newpt, HOME))
            then violation()  fi;                      }

10. END;
```

Figure 4. A PE-grammar for Monitoring Rdist.

Fig. 4 shows the trace policy for *rdist*. This PE-grammar describes the valid operations of a single *rdist* execution. Line 1 shows the *selection expression* for the subjects with which the specification is concerned. For this specification, it describes the operation sequence of a single execution of *rdist* on the host *blanc*. Lines 2-6 show the initial environment assignments. `U` is initialized to the user associated with the execution, which is returned from *getuser()*. `PID` contains the process ID (obtained from *getpid()*) of the process corresponding to the execution. `FILECD` (`PATHCD`) is an associative array for storing the *inode* numbers (path names) of the files created by the program execution. They are both initialized to empty and are changed during parsing when a create-file operation is recognized. `HOME` is initialized to the home directory of the invoker, which is */export/home/<Username>*. Its value will not be changed thereafter.

On line 7, the start expression contains only the start notion <`rdist`>, which implies that the input is described by the hypernotion <`rdist`>.

Hyperrule 8 recursively defines the input as a repetition of the valid operations specified by <`valid_op`>; the second alternative represents the termination condition.

Hyperrule 9 describes the operations *rdist* is allowed to perform. It has 9 alternative right-hand sides. The first alternative contains just a terminal, and each of the remaining eight alternatives contains a terminal followed by one or more semantic actions. The tokens recognized by the hyperrule are `open_r_worldread` (any *open_r* operation on a publicly readable file), `open_r_not_worldread`, (any *open_r* operation on a file that is not publicly readable) `open_rw`, `creat_file` (any operation that results in creation of a file), `creat_dir` (any operation that results in creation of a directory), `symlink`, `chown`, `chmod`, and `rename`.

The semantic actions in the various alternatives of the hyperrule check the attributes (such as the path name and the *inode* of the file) of the recognized operation to determine whether the operation is valid. They raise a violation for an invalid operation by calling the function *violation()*. Several attributes of operations are referenced in the semantic actions: F denotes characteristics of the process and the file associated with the recognized operation, P denotes the path name of the file, $M.newouid$ denotes the new owner and $M.newpt$ denotes the new path name. The semantic action in the second alternative of hyperrule 9 in Fig. 4 raises a violation if the file is not created by the process. The semantic action in the third alternative raises a violation if the recognized *open_rw* operation is not associated with a device file. There are three semantic actions following `creat_file` in the fourth alternative. The first semantic action raises a violation if the file associated with the operation is not inside the */tmp* directory or the home directory (specified by the environment variable `HOME`). The second and third semantic actions update the environment variables `FILECD` and `PATHCD` to indicate that a new file and a new path has been created. The semantic action following `symlink` raises a violation if the file is not inside the */tmp* directory or the home directory. The semantic actions following `chown` and `chmod` raise a violation if the file is not created by the process. The semantic action following `rename` raises a violation if the old path name associated with the rename operation is not created by the program or the new path name is not inside the home directory of the invoker.

To summarize, the trace policy specifies that an execution of *rdist* on host *blanc* may (1) open a publicly readable file for reading, (2) open a file that is created for reading, (3) open a device file for both reading and writing, (4) create a new file, directory, or symbolic link in the */tmp* directory or the home directory of the invoker, (5) change the permission mode and the ownership of a file that it created, and (6) rename a file that it created in the host directory. Since the trace policy allows *rdist* to change the permission mode of only those files that are created by the program execution itself, attacks that exploit *rdist* to change the permission mode of other files (e.g., /bin/sh) will be detected.

5. Design and Implementation

The section presents the design and implementation of a specification-based intrusion detection system, the *Distributed Program Execution Monitor* (DPEM), which monitors executions of programs in a distributed system to detect behavior inconsistent with their trace policies.

5.1. Design of DPEM

The target platform is a distributed system which consists of several hosts connected by a local area network. Each host in the system collects audit trails about the system operations that occur in the host, which should include all system calls.

DPEM consists of a *director*, a *specification manager*, *trace dispatchers*, *trace collectors*, and *analyzers* situated in various hosts in the distributed system. Our design combines distributed data collection and data reduction with decentralized analysis. Our system enables audit data to be collected and filtered in individaual hosts and data analysis to be carried out concurrently on multiple hosts. Also, each component is designed to minimize the amount of audit data that needs to be transferred across the network.

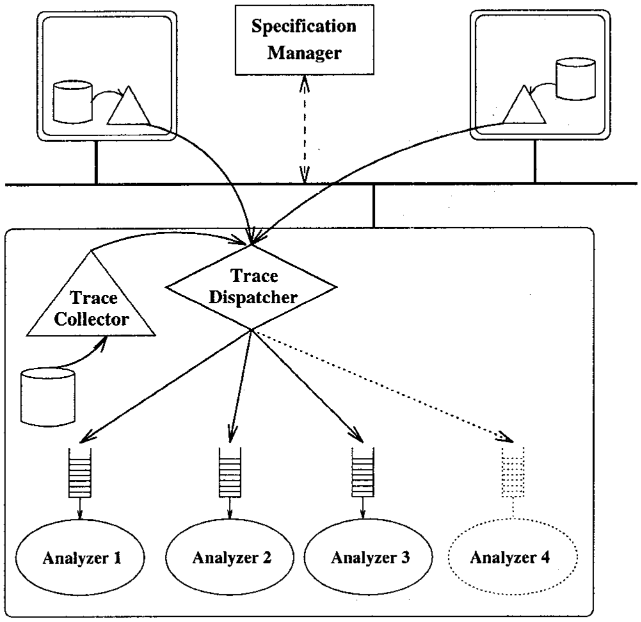

Figure 5. Architecture of DPEM from the Perspective of a Host.

Fig. 5 depicts the architecture of DPEM from the perspective of a host in the distributed system. It also shows the data flow among its components. Analyzers are the components that perform the monitoring. An analyzer checks the execution trace of a subject for violations with respect to a given trace policy. Thus, an analyzer can be thought of as the runtime counterpart of a policy specification. The number of analyzers running in the system may change dynamically. In Fig. 5, three analyzers are running, and each of them is associated with a different trace policy. The first analyzer may be associated with a trace policy for a single execution of *rdist*. The second analyzer may be associated with a trace policy concerning the behavior of a single execution of *fingerd*. The third analyzer may monitor all executions of *passwd* and *vi*. Additional analyzers will be started when new subjects need to be monitored. Analyzers terminate when the subjects they monitor exit. For instance, if a user executes *rdist*, an analyzer associated with the trace policy concerned with *rdist* will be executed to monitor that execution of *rdist* (Analyzer 4 in Fig. 5). An analyzer reports any erroneous behavior of its monitored subject to the director, which carries out the appropriate response for the incident, such as notifying the system security officers or starting up additional analyzers. An analyzer can run on any host in the system. Therefore, the analysis of audit data is distributed among multiple hosts in the distributed system.

A trace dispatcher must be present on each host on which analyzers are running. It is responsible for sending the execution traces of the subjects to the analyzers running on the host. Conceptually, it reads the audit trace from each of the hosts in the system, merges them to form the system trace V of the whole distributed system, and filters (pred: *pid* $\in$ *sp*, see Sec. 3) it to obtain the trace of the subject *sp* required by an analyzer. A trace dispatcher gets audit records from the trace collectors situated in various hosts. Depending on the subjects monitored by the attached analyzers, a trace dispatcher may or may not request audit records from the trace collector in a particular host. It identifies the trace collectors from which audit records are needed for monitoring, and it requests audit records from them only as long as they are needed.

A trace collector runs on each host on which an audit trail resides. It fetches the audit records directly from the audit repository and sends the records to the trace dispatchers (possibly situated on different hosts) that request the records. It filters the records such that only the records requested by a dispatcher will be sent, thus minimizing the network bandwidth used by the monitoring system.

The specification manager enables the system administrator to manage the security specifications. An administrator can add, modify, or delete the security specifications in the system through the specification manager interface. The specification manager starts up analyzers to monitor program executions when programs that need to be monitored are executed.

5.2. The Specification Manager

The specification manger keeps all trace policies in a specification database. Each trace policy is associated with a selection expression indicating the subjects with which the policy is concerned. At a high level, a subject can be one or more program executions, one or more users, and one or more hosts. At the system level, a subject consists of one or more distributed processes. When a new distributed process is created, i.e., a program is started, the specification manager checks the selection expressions to determine if the distributed process belongs to any of the monitored subjects. If so, it invokes a corresponding analyzer to monitor the new process.

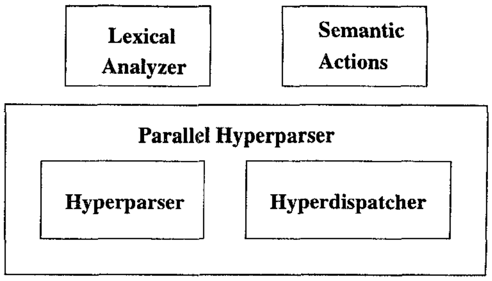

Figure 6. The Structure of an Analyzer.

Recall from Section 3.1 that the execution of a program is a distributed process. A distributed process is identified by a process ID (*pid*), the user it is representing, the program it is executing, and the host on which it is running. Given a selection expression and a distributed process identified by its *pid*, user, program and host, a simple matching can determine whether the distributed process belongs to any of the monitored subjects specified by a selection expression.

When the specification manager receives an audit record indicating a process *pid* associated with a user u who executes a program p on a host h, it matches $< pid, p, u, h >$ with the selection expression of each trace policy. If the selection expression of a trace policy matches the audit record, a new subject is instantiated. For example, the selection expression $<?, rdist, *, * >$ of a trace policy P causes the instantiation of the subject $<456, rdist, *, * >$ when a user executes *rdist*, and the *pid* of the process created is 456. If there is no analyzer associated with P, an analyzer will be started to monitor the subject. In general, no two analyzers monitor the same trace policy and the same subject.

5.3. Analyzers

An analyzer monitors the execution of a subject with respect to a trace policy. A different analyzer is constructed for each trace policy and is invoked when a subject that is to be monitored starts. It determines whether the execution of the monitored subject violates the trace policy by parsing the execution trace.

Fig. 6 shows the structure of an analyzer, which consists of a parallel hyperparser, a lexical analyzer, and a semantic-action module. The parallel hyperparser forms the core of an analyzer. It is constructed based on the PE-grammar representing the trace policy.

Fig. 7 depicts the structure of a parallel hyperparser. A parallel hyperparser consists of a hyperdispatcher, one or more (sequential) hyperparsers, a set of global environment variables shared by all hyperparsers, and an initialization procedure for performing the initial environment assignments to the global environment variables. Each hyperparser is a top-down parser, similar to the hyperparser of an environment grammar [12].

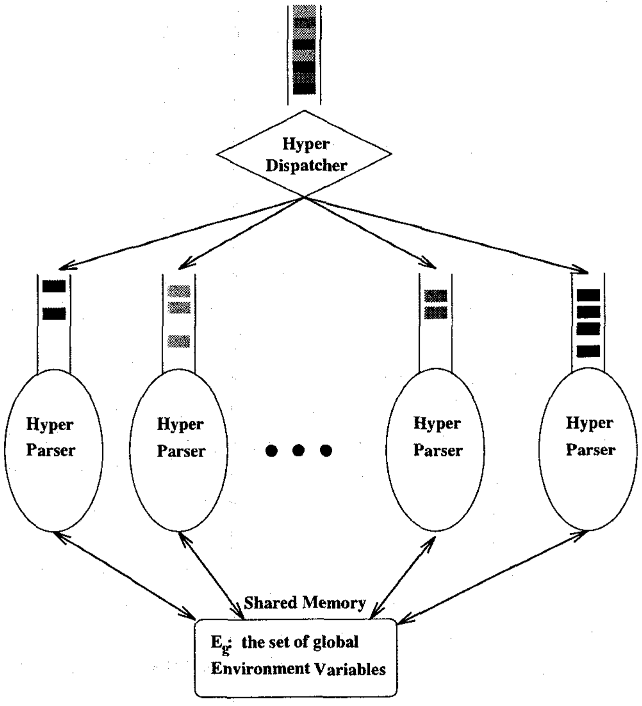

Figure 7. A Parallel Hyperparser.

5.3.1. Hyperparsers

There is one hyperparser per start notion s_i in the start expression. Each hyperparser contains a set of local environment variables, a local initialization procedure for performing the initial environment assignments to the local environment variables, a set of lexical procedures for reading and recognizing the terminals, and a set of (possibly recursive) hyperprocedures for generating and applying production rules. There is one hyperprocedure per hyperrule, where a hyperrule may contain alternative right-hand sides. Also, each hyperparser has a local variable *current_token* which holds the token it is processing.

5.3.2. Hyperprocedures

The hyperprocedures form the core of a hyperparser. Let $HP\_X_0()$ be the hyperprocedure corresponding to the hyperrule $X_0 \rightarrow X_1X_2\cdots X_m[B]$. It first calls the function *inspectQueue()* to inspect the next token in the input queue and sets *current_token* to that token. The function *inspectQueue()* waits and returns when the next token is available. The hyperprocedure subsequently uses the environment variables to generate the current production rule by obtaining the protonotions (or terminals) Y_i from the hypernotions (or terminals) X_i on the right-hand side of the hyperrule. The hyperprocedures for the references (see Sec. 4.1) of the protonotions Y_i (or lexical procedures in case the latter are terminals) are first identified and subsequently called in sequence. If all m procedures return successfully, any attached environment assignments or semantic actions B are performed and the hyperprocedure returns indicating success.

If $HP\_X_0()$ represents a set of alternative hyperrules, the order in which they are processed is determined by consulting the next input token. Processing is as above, but a failure indication by a called procedure causes the hyperparser to backtrack and start the processing of the next alternative rather than to immediately return with a failure indication.

5.3.3. Parsing

When an analyzer starts, it first calls the initialization procedure of the parallel hyperparser with information on the monitored subject to initialize the environment variables. It also calls the initialization procedure of the lexical analyzer, the dispatcher module, and the semantic action module with information on the subject it monitors. After the initialization, it passes control to the parallel hyperparser.

Within the parallel hyperparser, the hyperdispatcher first performs the global initialization procedures to initialize the global environment variables, and subsequently starts all hyperparsers. It then reads the tokens one by one and dispatches them to the input queue of the appropriate hyperparsers. After dispatching a token to a hyperparser, it waits until the queue of the hyperparser is empty (i.e., the token is accepted) and the hyperparser pauses by calling *inspectQueue()* for the next token (i.e., any actions following the recognition of the previous token have been performed) before it feeds the next token to another hyperparser [2]. Therefore, only one hyperparser is active at any single time, and all others are blocked waiting for the next token.

When a hyperparser starts, it first initializes its local environment variables. It then calls the hyperprocedure $HP\_s_i()$ for the hyperrule with the start notion s_i on the left-hand side. The execution of each hyperparser is driven by the input tokens. A hyperparser either

[2] A more efficient version of the parallel hyperparser permits the hyperparsers to run in parallel (with some synchronization constraints) without affecting parsing correctness [7].

pauses in a lexical procedure or at the beginning of a hyperprocedure when the next token is not available, i.e., the input queue is empty. In particular, the hyperprocedure $HP\_s_i()$ stops if the next input token is not available when the hyperparser starts.

The purpose of a hyperparser is to parse the sequence of tokens in its input queue and report any violations. When each hyperparser parses the sequence of tokens in its input queue successfully, the sentence is recognized by the parallel hyperparser. Otherwise, the input sentence is not a sentence of the PE-grammar, signaling the detection of a violation.

The lexical analyzer scans the source input for tokens of the PE-grammar. The hyperparser invokes the lexical analyzer with a specification of the expected token type, and the lexical analyzer returns a success/failure indication together with the token value in case of success. The semantic-action module contains the necessary procedures for performing the semantic actions. It includes a procedure *violation()* which reports a violation and pertinent information to the security officer. This procedure does not terminate the parser; it allows parsing to continue. In general, users can code up their own procedures for semantic actions.

5.4. A Unix Prototype DPEM

We built a prototype DPEM for a single host based on the design above. It serves as a proof-of-concept implementation for our approach.

The prototype is written in the C programming language [6]. The C programming language was chosen because of its wide-spread use and portability across different Unix platforms. The prototype runs under the Solaris 2.4 operating system and uses the auditing services provided by the Sun BSM audit subsystem [14].

The BSM audit subsystem provides a log of the activities that occur in the system. It records the sequence of system events in the order of occurrence. Thus, the audit trails contain a trace of the system. An audit record contains information such as the process ID and the user ID of the process involved as well as the path name, the *inode*, and the permission mode of the files being accessed. However, it does not contain information about the program the process is running. Therefore, we added an audit record preprocessor to associate a program identification with each audit record.

The audit record preprocessor actually serves two purposes. First, it filters audit records that are irrelevant to the monitoring system. Second, it translates the BSM audit records into the format required by the monitoring system. It keeps an array *pg* that holds the identification of the program each process is currently executing. When it reads an audit record associated with process ID x, it associates the program in $pg[x]$ with that record. It monitors all *exec* and *fork* calls and updates the array accordingly. When an *exec* call invokes a program P, the program associated with the new process becomes P. When a *fork* call creates a new process, the program associated with the new process is that of its parent.

Our prototype maintains a table of registered programs. Each entry contains the name of a program, as well as the pathname and the *inode* of the physical file that contains the program. For the identification of a program that is started by an *execve* event, the *inode* number in the record is used as the key because the pathname is not a unique identifier of a program. For example, an attacker can create a hard link to a setuid root program and execute the hard link, in effect executing the setuid root program. Yet the pathname of the *execve* record would not indicate an execution of a setuid root program.

When the prototype is started, several processes are typically executing in the system. It is generally impossible to trace back how the processes got created from the audit trail as the audit records could have been modified or deleted. To obtain identifications of the programs that existing processes are executing, the audit record preprocessor inspects the kernel memory and initializes the array *pg* when it starts up.

5.5. Experimental Efforts

We have written trace policies for 15 *setuid root* programs and servers in Unix, including *fingerd*, *rdist*, *sendmail*, *binmail*, *passwd* and *vi*. Using these trace policies, we tested the prototype with three different kinds of intrusions. These intrusions exploit vulnerabilities in *rdist*, *sendmail*, and *binmail*.

The experiments described below were performed on a Sun SPARCstation 5 with 32MB of memory running Solaris 2.4. We configured the audit system to log all successful events for all users, and for all network daemons including *inetd*, *fingerd*, *rlogind*, and *telnetd*. The original version of *rdist* and *sendmail* were replaced by the SUN 4.3 versions, as their vulnerabilities have been removed in Solaris 2.4. The prototype DPEM runs continuously and analyzes the audit data generated by the audit system in real time and reports any violations to the security specification.

The intrusions were realized using combinations of a Perl [16] scripts and C programs. One intrusion exploited a vulnerability in *rdist*; another intrusion exploited a vulnerability in the sendmail program that causes *sendmail* to create a setuid shell in the */tmp* directory that is owned by root and is publicly executable. A third intrusion caused *binmail* to overwrite the shadow password file with values that the attacker desires. The prototype detected these intrusions within 0.1 seconds. Also, no noticeable degradation of performance was noted.

Fig. 8 shows the report generated by the prototype DPEM for the first intrusion. The trace policy in Fig 4 was used for *rdist*. This intrusion produced two violations. The first was that *rdist* changed the ownership of a file not created by the program. The second was that *rdist* changed the permission mode of */usr/bin/exsh*. The time elapsed between the occurrence of the violation and the detection was approximately 0.06 seconds. The time of detection was obtained by the *gettimeofday()* library call when the hyperparser executed a semantics action that called the *violation()* function. The time of the operation was obtained from the audit records.

```
% rdistattack /bin/sh
% /bin/sh

VIOLATION -- Tue May 14 16:52:20 1996 + 0.894236000 sec
--------------------------------------------------------
Tue May 14 16:52:20 1996 + 0.820003000 sec
lchown, (5456, 5000, 0, rdist), nodeid: 27, path: /tmp/r
dista05456

VIOLATION -- Tue May 14 16:52:20 1996 + 0.972195000 sec
--------------------------------------------------------
Tue May 14 16:52:20 1996 + 0.830008500 sec
chmod, (5456, 5000, 0, rdist), nodeid: 4149, path: /usr/
bin/exsh, mode: 4777
```

Figure 8. A Report Generated by the Prototype DPEM.

We also tested the prototype with a scenario involving simultaneous modifications of the password file. This scenario was simulated manually. In one window, one author logged on as the superuser and executed the command *vi /etc/shadow* to modify the shadow password file. Subsequently, this author logged on as a normal user in another window and executed *passwd* to change the password. We used a trace policy which requires that only one execution of *passwd* or *vi* can modify the password file at any time. The *passwd* program obtained the old password, but when it started to modify the password file, the prototype detected and reported the violation. The violation was detected approximately 0.05 seconds after *passwd* opened the shadow password file. [3]

6. Conclusions

We presented advancements on specification-based intrusion detection [8]. We identified aspects of program behavior that are security-relevant, and presented a detection model based on traces and a formal notion of monitored subjects. The model uses a formal language for describing the intended behavior of programs. In addition, we developed a prototype of a specification-based intrusion detection system that detects attacks exploiting the vulnerabilities of privileged programs in Unix.

Our approach lends itself to a decentralized analysis system, which enhances the scalability of the intrusion detection system. In particular, data analysis is carried out by various analyzers situated on different hosts. Also, audit data that is relevant to an analyzer is precisely defined, thus enabling effective audit filtering at the trace collectors. Moreover, our approach explicitly addresses security problems due to the lack of synchronization in concurrent programs.

An innovation of our approach is the use of grammars for specifying valid traces of programs. It has the advantage that many results of the mature discipline of formal languages can be readily applied. We developed a language framework, parallel environment grammars, for specifying trace policies. These parameterized grammars can express a large variety of trace policies in a compact manner. Yet they can also be parsed efficiently in many cases of practical importance. We described the principal features of such a parsing method, and illustrated its application to the detection of intrusions that exploit vulnerabilities in security-critical programs.

Our detection system is not without limitations. A trace policy deals mainly with operations, e.g., file accesses that a subject is authorized to perform. It does not consider the particular data values that a subject reads or writes. The detection of reading or writing unauthorized data values would require the inclusion of a description of authorized data values in the specifications. Moreover, contemporary audit records would have to be expanded to capture data values, resulting in a significant increase in overhead costs.

In addition to monitoring privileged programs, the specification-based approach can be used to monitor network components or network services that are relevant to security, including domain name services (DNS), network file systems, and routers. The systematic methodology for developing trace policies for programs may also be useful in future research on developing overall security policies for computer systems and networks.

[3] In Solaris 2.4, the occurrence time associated with an audit record denotes the finishing time of the system call.

Acknowledgments

This work was supported in part by the Defense Advanced Research Projects Agency under Contract DOD/DABT63-93-C-0045 (DARPA Order A785) and by the National Security Agency University Research Program under Contract DOD-MDA904-93-C4083. The authors thank Heather Hinton and the anonymous referees for their helpful comments on revising the paper.

References

[1] J. P. Anderson, "Computer security threat monitoring and surveillance," Technical report, James P. Anderson Co., Fort Washington, PA, April 1980.

[2] D. E. Denning, "An intrusion-detection model," *IEEE Transactions on Software Engineering*, vol. 13, no. 2, pp. 222–232, 1987.

[3] S. Forrest *et al.*, "A sense of self for unix processes," in *Proceedings of the 1996 Symposium on Security and Privacy*, (Oakland, CA), May 6-8 1996, pp. 120–128.

[4] K. Ilgun, "USTAT: A real-time intrusion detection system for Unix," in *Proceedings of the 1993 Symposium on Security and Privacy*, (Oakland, CA), May 24-26, 1993, pp. 16–28.

[5] H. S. Javitz and A. Valdes, "The NIDES statistical component description and justification," Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 1994.

[6] B. Kernighan and D. Ritchie, *The C Programming Language.* Englewood Cliffs, NJ: Prentice Hall, 1978.

[7] C. Ko, *Execution Monitoring of Security-Critical Programs in a Distributed System: A Specification-based Approach.* PhD thesis, Department of Computer Science, University of California, Davis, September 1996. Also available at http://seclab.cs.ucdavis.edu/~ko/thesis/paper.ps.

[8] C. Ko, G. Fink, and K. Levitt, "Automated Detection of Vulnerabilities in Privileged Programs Using Execution Monitoring," in *Proceedings of the 10th Computer Security Application Conference*, (Orlando, FL), December 5-9, 1994.

[9] S. Kumar, *Classification and Detection of Computer Intrusions.* PhD thesis, Department of Computer Science, Purdue University, August 1995.

[10] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[11] T. Lunt *et al.*, "A real-time intrusion detection expert system (IDES) - final technical report," Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, Feburary 1992.

[12] M. Ruschitzka and J. L. Clevenger, "Heterogeneous data translations based on environment grammars," *IEEE Transactions on Software Engineering*, vol. 15, no. 10, pp. 1236–1251, 1989.

[13] Sun Microsystem, *Man Pages: Rdist - remote file distribution program*, Novermber 1993.

[14] SunSoft, Mountain View, California, *Solaris SHIELD Basic Security Module*, August 1994.

[15] H. Vaccaro and G. Liepins, "Detection of anomalous computer session activity," in *Proceedings of the 1989 Symposium on Security and Privacy*, (Oakland, CA), May 1-3, 1989, pp. 280–289.

[16] L. Wall and R. L. Schwartz, *Programming Perl.* Sepastopol, CA: O'Reilly and Associates, Inc., 1992.