On a Pattern-Oriented Model for Intrusion Detection

Shiuh-Pyng Shieh, *Member, IEEE*,
and Virgil D. Gligor, *Member, IEEE*

Abstract—Operational security problems, which are often the result of access authorization misuse, can lead to intrusion in secure computer systems. We motivate the need for pattern-oriented intrusion detection, and present a model that tracks both data and privilege flows within secure systems to detect context-dependent intrusions caused by operational security problems. The model allows the uniform representation of various types of intrusion patterns, such as those caused by unintended use of foreign programs and input data, imprudent choice of default privileges, and use of weak protection mechanisms. As with all pattern-oriented models, this model cannot be used to detect new, unanticipated intrusion patterns that could be detected by statistical models. For this reason, we expect that this model will complement, not replace, statistical models for intrusion detection.

Index Terms—Access misuse, audit analysis, context-dependent intrusion, intrusion detection, operational security problems, statistical methods, rule-based methods, secure systems.

INTRODUCTION

COMPUTER system intrusion typically occurs as a result of either system penetration or misuse of access authorization. In principle, access control and authentication mechanisms can provide the penetration resistance necessary to *prevent* illegitimate access by unauthorized users. In contrast, intrusion resulting from misuse of access authorization can be prevented by neither authentication nor access control since unauthorized access need not be attempted. Instead, this type of intrusion must be *detected* by after-the-fact analysis of audit trails, that is, by intrusion detection. However, real-time analysis of audit trails is often impractical because of the substantial overhead added by audit-trail analysis to system performance [1].

Most intrusion-detection methods proposed to date can be categorized as either statistical or rule-based methods. Statistical methods define intrusion patterns as statistical deviations from an activity profile that characterizes "typical" access behavior of a given subject (i.e., user or process) or group of subjects with respect to a given object (e.g., a file, a directory) or group of objects [7], [8], [9], [12], [13], [14], [20]. Activity profiles attempt to capture statistically the normal access activity of subjects during a given period of system operation. A key advantage of these methods is that intrusion patterns need not be defined explicitly, thereby enabling detection of new, unanticipated intrusion patterns. A key disadvantage of these methods is that they are generally unable to adapt easily to changes of subject behavior; e.g., changes legitimately caused by the use of new applications.

Rule-based methods detect intrusion patterns based on sequence of events [3], [16], [17], [20]. These methods share the disadvantage of statistical methods as they define intrusion patterns in terms of audit records that cannot be easily adapted to changes of subject behavior. To mitigate this disadvantage, Teng, Chen, and Lu proposed a method which uses sequential rules that are dynamically generated and modified by a time-based, inductive engine to define expected user-activity profiles [19]. More recently, Ilgun, Porras, and Kemmerer proposed a method which defines intrusion as a sequence of state changes that take the computer system from some initial state to a target compromised state, and requires the analyst to identify the minimum number of actions to reach this state [11]. Although both these methods improve adaptability to changing subject behavior, they do not track data and privilege flows between subjects and objects. As a consequence, intrusion occurrences may remain undetected when they include sequences of events in which both the size of the subject and object sets, and the length of the sequence itself, vary in time.

In general, both rule-based and statistical methods are rather insensitive to occurrences of context-dependent intrusions with illegal data and privilege flows, such as those characterizing unintended use of foreign programs or virus propagation. Insensitivity to context-dependent intrusion may lead to failure to detect some intrusion instances and may cause false alarms to be raised. This limitation is shared by all intrusion-detection methods that ignore the meaning and significance of event sequences in defining intrusion patterns.

In this paper, we present a pattern-oriented, intrusion-detection model, discuss some of its salient properties, and illustrate its applications with an example. (Further examples and analysis are provided elsewhere [15].) The model presented here is inspired by the use of the Take-Grant model [6], [18] and the U-KUANG rule-based tool [2] for static analysis of system protection states, where a system protection state is defined by the current setting of the access control matrix; e.g., for the static determination of whether a subject can potentially acquire extra object privileges or data. However, in contrast to the Take-Grant model and the U-KUANG tool, our model captures the dynamic, potential flows of privileges and data, thereby enabling the definition and discovery of specific intrusion patterns in audit trails. In particular, our model helps uniformly define patterns of data and privilege flows that characterize operational security problems caused by misuse of access authorization in otherwise secure systems. These problems include those of

1) unintended use of foreign programs,
2) unintended use of foreign input data,
3) imprudent choice of default privileges, and
4) use of weak protection mechanisms.

However, as is the case with any model that requires explicit definition of intrusion patterns, our approach detects only intrusions that can be anticipated prior to their occurrence. For this reason, pattern-oriented intrusion detection is intended to complement, not replace, statistical approaches for intrusion detection.

The balance of this paper is organized as follows. In Section 2, we present several examples of operational security problems of secure systems that help motivate the pattern-oriented approach to intrusion detection. Section 3 contains a succinct definition of the model and its salient properties. Section 4 illustrates an application of this model for the detection of unintended use of foreign programs.

2 OPERATIONAL SECURITY PROBLEMS

Operational security problems arise from complex interactions between the pieces of a computer's protection system and users, and can be eliminated neither by use of access control and authentication mechanisms nor by administrative measures; e.g., problems caused by imprudent use of object privileges, inadequate

• *S.-P. Shieh is with the Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan 30010, ROC. E-mail: ssp@csie.nctu.edu.tw.*
• *V.D. Gligor is with the Department of Electrical Engineering, University of Maryland, College Park, MD 20742.*

Manuscript received 9 Aug. 1995.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 104037.

setting of name-resolution variables, or execution of foreign programs that may contain Trojan Horse or virus codes. These problems arise independent of the level of security assurance of, and policy employed by, a computer system, and may be caused by both privileged, administrative users and unprivileged, casual users. We present four of the many types of operational problems leading to context-dependent intrusions that suggest the use of a pattern-oriented approach to intrusion detection. These problems are illustrated using the Unix system mechanisms and services. However, it should be clear that these problems are not specific to Unix; instead, they are fairly typical of all current multiuser operating systems.

2.1 Unintended Use of Foreign Programs

The unintended use of foreign programs is a common operational problem in many operating systems, such as Unix. This problem can be caused by failure to set and reset the environment variables [4], [5]. The following two examples illustrate the problem.

2.1.1 Command Search Paths—PATH

In Unix systems, an internal variable **PATH** for *command search path* in the shell (command interpreter) defines the order of directories being searched to find a command [4]. The unintended use of foreign programs can be caused by inadequate setting of the user's command search paths. Two types of problems may arise:

1) a user may execute an intruder's version of a system command instead of the intended system command whenever the user's command search paths start from the current directory; and / or
2) a user may execute a system command instead of the intended user program if the user's search paths start from a system command directory.

These two problems cannot be easily resolved because:

1) it is undesirable to require users to type full path names for every program execution, and
2) it is unrealistic to expect all users understand the effects of the search-path setting on their executed programs.

2.1.2 Word Separators for Commands—IFS

Similar problems appear when environment variables **IFS**s are not carefully set or reset by users. In Unix systems, an environment variable **IFS** is used to define a list of characters representing word separators for commands whenever the Bourne shell *sh* (a command interpreter) is used. If **IFS** is set to "o," the shell command *show* (which prints mail messages on the screen) is treated as the command *sh* with one argument *w* (since the "o" is treated as a blank). Thus, whenever an unsuspecting user fails to set or reset **IFS**, another user may set up a file *w*, which can include command programs under the control of the latter user.

Similar problems appear if aliases are not carefully set or reset by users when used within Unix commands.

2.2 Unintended Use of Foreign Input Data

Misuse of environment variables that causes unintended use of foreign programs can also cause unintended use of foreign input data. However, more subtle instances of unintended use of foreign data also appear in Unix systems. For example, a miscreant user may exploit the fact that directories */tmp* and */usr/tmp* provide a writable repository for the creation of temporary files by system programs like the C compiler, *cc(1)*. Since these directories must be readable and writable by everyone, a miscreant user can remove any or all files saved there and replace them with his own files. *cc(1)* may take these intermediate files as unintended input and generate (unintended) executable files. The resulting operational problem is that an unsuspecting user may end up using bogus input data and programs with the same effects as those of unintended use of foreign programs.

2.3 Imprudent Choice of Default Privileges

Operating systems, such as Unix, allow users to define default object privileges. For example, a user may set his default-privilege mask to allow:

1) all privileges for the owner,
2) read-write privileges for a specified group of users, and
3) read privilege for all other users, whenever he or she creates an object.

Object privileges can be subsequently modified to accommodate the needs of different sharing patterns. For example, the user editing a sensitive document stored in a file may modify the file's privileges to allow read-only privileges for the group and no-privileges for other users. Suppose, however, that an editor running with the user identity, creates a new temporary file (which has the default privileges set by the user mask), copies the contents of the original file in it, edits it, destroys the original file, and renames the temporary file with the name of the original file. The unsuspecting user appears now to have granted read access to all other users, perhaps contrary to the wishes of the group members; also, the group members gain inadvertently write access to the document file, perhaps contrary to the unsuspecting user's wishes. In this context, subsequent read access to the document file exercised by the other users and write access exercised by the group represents an intrusion. The use of unsafe object-privilege defaults is a very common operational security problem in most (secure) systems.

2.4 Use of Weak Protection Mechanisms

Compatibility with existing system interfaces requires retention of protection mechanisms visible at a system interface even when these mechanisms are demonstrably weak or unsafe. The *setuid/setgid* system calls of Unix systems, which are intended to help implement protected subsystems, represent a good example of this requirement. Hypothetically, a secure system would not use such mechanisms within its trusted programs but, nevertheless, would retain them for discretionary use by various applications. In such a system, a user may (un)intentionally call a foreign program that creates a *setuid* file with the user's identity, unbeknownst to the user. The effect of this seemingly innocuous create-file call is that the owner of the foreign program, which would be granted read and write privileges on the file, could copy to, and run programs from, that file under the user's identity. The resulting operational problem, namely the ability of a user to masquerade as another user, cannot be avoided easily because casual users often lack the ability or permission necessary to discover the potential use of *setuid* like calls by untrusted application code. Additional operational problems caused by the use of the *setuid* mechanism and their solutions are described elsewhere [4], [5], [10].

The above examples show that seemingly innocuous access patterns may, in fact, represent intrusion patterns whenever they appear in *the context of specific sets of subjects and objects.*

3 THE INTRUSION-DETECTION MODEL

In this section, we provide a succinct definition of the intrusion detection model by formally defining system states and state transitions in terms of direct relations, and rules that establish indirect relations between subjects and objects. System states are captured in audit trails and are represented by a *protection graph*. A protection graph has two distinct types of vertices: *subjects* and *objects*. Subjects are the active vertices that represent processes and users; they initiate operations that cause data and privileges to flow among subjects and object. Objects, on the other hand, are completely passive; they represent data containers such as files and directories, and cannot initiate operations nor data and privilege flows. Because all system events can be interpreted as different types of data and privilege flows, intrusion events can be formally defined in terms of data and privilege flows. However, some data and privilege flows represent only potential, not actual, flows. Whether an actual flow takes place is undecidable because flows are both call-argument and system-state dependent, not just operation-dependent. Hence, some intrusions may represent only potential, rather than actual, intrusions.

3.1 System States

A system state is defined by a directed protection graph $G(V, E, C, F)$, which is a structure consisting of a set of vertices, V, a set of labeled edges, E, a set of protection sets, C, and a legal-flow matrix, F.

The set of vertices V consists of subjects and objects. A subject s_i is graphically represented by the symbol

○

and an object o_i is represented by the symbol

□

in the protection graph. A vertex v_i, $v_i \in V$ that can be either a subject or an object is represented by

☒

Each directed edge $e_i \in E$ connects elements of an ordered pair of vertices (u, v), and is labeled with subsets of a finite relation set R, where $R = \{r, w, cb, d, I_r, I_w, I_{cb}, I_d, r^*, w^*, cb^*, d^*, I_r^*, I_w^*, I_{cb}^*, I_d^*\}$. (When written as labels on a graph, the set braces of relations are normally omitted.) Symbols r, w, d, and c represent present *read*, *write*, *drive*, and *control* relations, respectively, whereas symbols *Ir*, *Iw*, *Ic*, *Id* represent indirect relations originally derived from direct relations r, w, d, and c. Alternatively, we use symbols *rb*, *wb*, *db*, and *cb* to denote *read-by*, *written-by*, *driven-by*, and *controlled-by*, respectively. Symbol * denotes the *past* unary operator, which is used to distinguish the relations of two epoches, namely, the *past* and the *present*. The "present relation" is provided by the access operation under current consideration, and can be interpreted as the latest event (e.g., write event) read from the audit trail or as an continuing event (e.g., execute event) that was invoked before the latest event. All access operations performed before the present access operations are past operations.

The basic subject accesses to objects can be defined as a set of *direct* and *indirect* relations between subjects and objects, and divided into two classes of accesses depending on whether the accesses cause a potential flow of object data or of privileges. *Read* and *write* relations are introduced to capture the potential flow of object data. A subject has a (*direct*) *read* relation with every object it reads, and a (*direct*) *write* relation with every object it writes. A subject s_i that *reads*/*read* an object o_m is represented by the relation $r(s_i, o_m)/r^*(s_i, o_m)$, or graphically,

$$\begin{tikzcd} s_i \bigcirc \arrow[r, "r/r^*"] & \square\, o_m \end{tikzcd}$$

Similarly, a subject s_m that *writes*/*wrote* an object o_n is represented by the relation $w(s_m, o_n)/w^*(s_m, o_n)$, or graphically,

$$\begin{tikzcd} s_m \bigcirc \arrow[r, "w/w^*"] & \square\, o_n \end{tikzcd}$$

Two other basic relations, namely *control* and *drive*, are introduced to capture the potential flow of object privileges. The control relation is used to detect the execution of malicious programs (e.g., programs containing Trojan Horses) by unsuspecting subjects, whereas the drive relation is used to detect instances whereby malicious subjects (e.g., the creators of malicious programs) can force unsuspecting subjects to execute malicious programs. A subject is (*directly*) *controlled by* an object whenever the subject executes the object; e.g., a process that executes a Unix command **ls** is controlled by **ls**. A subject s_i that is/was *controlled by* an object o_m is represented by relation $cb(s_i, o_m)/cb^*(s_i\ o_m)$, or graphically,

$$\begin{tikzcd} s_i \bigcirc \arrow[r, "cb/cb^*"] & \square\, o_m \end{tikzcd}$$

A subject (*directly*) *drives* another subject whenever the former provides a command (or program) to be executed by the latter; e.g., whenever a parent process creates a child process to execute command **ls**, the child process is driven by its parent process (and is controlled by **ls**). A subject s_m that *drives*/*drove* a subject s_i is represented by relation $d(s_m, s_i)/d^*(s_m, s_i)$, or graphically,

$$\begin{tikzcd} s_m \bigcirc \arrow[r, "d/d^*"] & \bigcirc\, s_i \end{tikzcd}$$

We note that the drive relation differs from the control relation because the controlling object provides the controlled subject with a sequence of instructions for execution and, thus, the control relation usually lasts for a period of time. In contrast, the the drive relation is an instantaneous operation, as the driving subject only provides the driven subject with the command or program to be executed.

A *protection set* $C_i \in C$ defines the set of all subjects and objects that require protection from intrusion by outside subjects, and is associated with each user. Object data and privileges are allowed to flow freely within a protection set at its owner discretion, however, inward and outward flows of a protection set should be restricted. For example, all processes and files owned by the "superuser" of a Unix system should form a protection set, and all processes and files of an unprivileged user should also form a protection set. That is, these subjects and objects should be included in their owner's protection set so that they can be protected from other users. Similarly, all subjects and objects of a protected subsystem should be protected from other users and, therefore, should be included in that subsystem's protection set.

The *legal-flow matrix*, F, with rows and columns corresponding to protection sets, defines the flow policy (e.g., restrictions) of the secure system. An entry $F(c_1, c_2)$ lists the legal data and privilege flows from c_1 to c_2, thereby helping distinguish between intrusions and legal accesses. An access is legal if all the incurred potential flows are permissible in F. If any of the potential flows incurred are illegal in F, the access represents an intrusion. For example, an indirect data flow from an unprivileged user to a system command (owned by the superuser) violates F and thus is illegal.

The legal-flow matrix can capture the data and privilege flows of a variety of security policies [15]. However, the representation of these policies via a two-dimensional matrix is generally impractical due to the sparseness of the matrix.[1] Instead, the policy captured by the matrix flows can be represented as a set of rules that define either legal or illegal flows between subjects and objects. For example, a rule may prohibit any "superuser" process from directly or indirectly executing a file owned by ordinary users; another rule may prohibit the direct or indirect writing of a user's sensitive files (e.g., setup files*.login*, *.cshrc* in Unix) by other users; and, yet another rule, may prohibit data flows (read or write) from a high security level to a low security level.

1. Efficient representation of the legal-flow matrix may also be hampered by the fact that the flow policy might be neither transitive nor symmetric. For example, it is common that a flow policy allows the flows from a to b and b to c but not a to c or b to a, respectively.

In summary, a system state is defined by the present data and privilege flows and, whether a state is intrusion-free, is determined by the protection set and legal-flow matrix.

3.2 State Transitions

In our model, state transitions are caused by present access operations. A present access operation that occurs between two vertices v_1 and v_2 is denoted by *present_relation*(v_1, v_2), where *present_relation* $\in$ {*r*, *w*, *d*, *cb*}. We define the present relations in terms of *r*, *w*, *d*, and *cb* since most known intrusion patterns can be characterized by these four types of data and privilege flows.

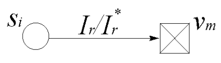
In our model, a state transition of the system is formally defined as

$$T: \mathcal{L}c \times G \rightarrow G \times D,$$

where $\mathcal{L}c$ is the set of present relations $V \times A \times V$, $A = \{r, w, cb, d\}$, and D is the set of exception-condition results {yes, no}. A "yes" indicates that intrusion patterns have been identified.

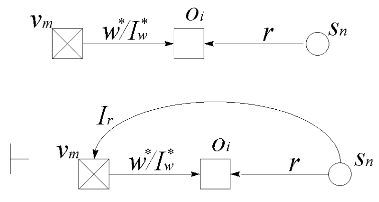
To determine whether an intrusion pattern occurred during a state transition, all indirect relations of the system state must be derived. This derivation is effected using Rules 1-4 defined below. Each of these rules includes two relations, at least one of which is a present direct relation; e.g., r(s_n, o_i) in Rule 1, w(s_i, o_n) in Rule 2, cb(s_n, o_i) in Rule 3, and d(s_i, s_n) in Rule 4 are present direct relations. Rules 1 to 4 provide a complete set of inference rules that provide all indirect relations for the four types of indirect potential data or privilege flows. These rules form a minimal set because each rule covers a single type of indirect relation and none of these rules can be removed. In the graphic definition of these rules, the symbol $\vdash$ is used to represent that the graph following this symbol is produced by all the graph-rewriting rules on the graph preceding it.

Given a subject and an object, the subject has an *indirect read* relation with all subjects and objects that provided input to a write operation on that object. *Indirect read* is represented by the relation I_r / I_r^*(s_i, v_m), or graphically,

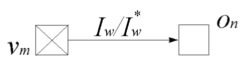


if a subject s_i *indirectly reads / read* an object (or a subject) v_m. Indirect read is achieved through the following sequences of relations:

RULE 1. w^* / I_w^*(v_m, o_i) $\wedge$ r(s_n, o_i) $\Rightarrow$ Ir(s_n, v_m), or graphically,

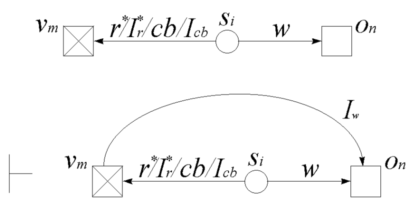


A given subject / object has an *indirect write* relation with all objects that are written by other subjects which executed, or took input from, that subject / object. *Indirect write* is represented by the relation I_w / I_w^*(v_m, o_n), or graphically,

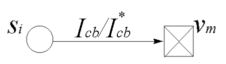


if a subject (or object) v_m *indirectly writes / wrote* an object o_n. Indirect write is achieved through the following sequences of relations:

RULE 2. r^* / I_r^* / cb / I_{cb}(s_i, v_m) $\wedge$ w(s_i, o_n) $\Rightarrow$ I_w(v_m, o_n), or graphically,

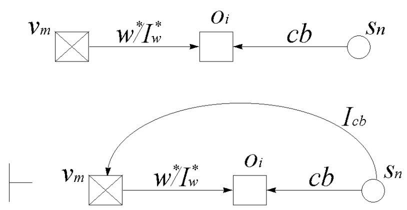


A subject is *indirectly controlled* by another subject / object, if the latter directly or indirectly wrote an object that directly controls the former. *Indirect controlled-by* is represented by the relation I_{cb} / I_{cb}^*(s_i, v_m), or graphically,



if a subject s_i *is / was indirectly controlled by* an object (or subject) v_m. Indirect controlled-by is achieved through the following sequences of relations:

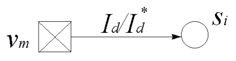
RULE 3. w^* / I_w^*(v_m, o_i) $\wedge$ cb(s_n, o_i) $\Rightarrow$ I_{cb}(s_n, v_m), or graphically,



A given subject is *indirectly driven*

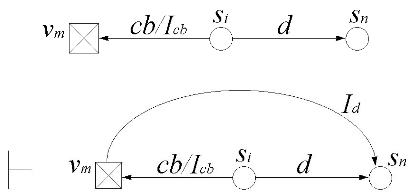
1) by another subject / object, if the latter directly or indirectly controls a subject that directly drives the former, or
2) by another subject / object, if the latter directly or indirectly drove a subject that directly drives the given subject.

Indirect drive is represented by the relation I_d / I_d^*(v_m, s_i), or graphically,

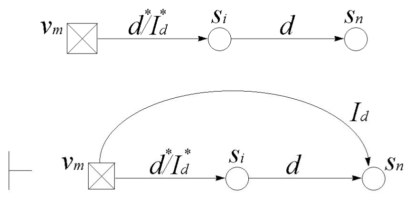


if a subject (or object) v_m *indirectly drives / drove* a subject s_i, that is, if v_m provides the next command to be executed by s_i. Indirect drive is achieved through the following sequences of relations:

RULE 4-1. cb / Icb(s_i, v_m) $\wedge$ d(s_i, s_n) $\Rightarrow$ Id(v_m, s_n), or graphically,



RULE 4-2. d^* / Id^*(v_m, s_i) $\wedge$ d(s_i, s_n) $\Rightarrow$ Id(v_m, s_n), or graphically,



Rules 1-4 above include specific sequencing of the present direct relations to capture data and privilege flows among subjects and objects. Relation sequencing is based on relation ordering and concurrency. Two access relations are ordered if one occurs before the other. For example, if indirect relation L1 occurs before relation L2, then L1 finishes before L2 starts; i.e., every single direct access relation that causes L1 occurs before the any access of L2 starts. Two access relations are concurrent, or simultaneous, if neither occurs before the other. For example, the read and control relations are concurrent if a user reads a file while executing a program.

To understand the significance of relations sequencing, consider the application of Rules 1 and 3 to two processes that share a file. If process 1 writes the file *before* process 2 reads it, or is controlled by it, process 2 has an indirect read, or controlled-by, relation with process 1; and, also, process 1 has an indirect write relation with any object written subsequently by process 2. However, if process 1 writes the file *after* process 2 reads it, an indirect relation (i.e., a data or privilege flow) cannot be established between the two processes. For similar reasons, the indirect write and drive relations of Rules 2 and 4 require that

1) the *controlled-by* and *write* relations occur concurrently (viz., Rule 2), and
2) the *controlled-by* and *drive* relations also occur concurrently (viz., Rule 4); e.g., an indirect write/drive relation exists between a program and a file if a process writes/drives the file when executing the program.

Rules 1-4 above apply only to a present direct relation and its adjacent relations (i.e., relations with which it shares a vertex). Also, in any state, the past direct and indirect relations are distinguished from the present direct relations (viz., Section 3.1). Hence, to discover all new indirect relations, and thereby to determine the next system state, we only need to

1) apply Rules 1-4 to the relations that are adjacent to present direct relations, and
2) to update the past/present status of each direct and indirect relation when transiting to each new system state.

4 An Application of the Intrusion-Detection Model

In this section, we illustrate the detection of unintended use of a foreign program by using our model. Other applications, such as detection of virus propagation, can found elsewhere [15]. This example illustrates the effect of the execution of an intruder's program, instead of the intended system command, by an unsuspecting user.

In Unix, user applications often access the */tmp* directory, which can be read, written and executed by every user in the system. This implies that, for example, a user (identified below as the Intruder) may deliberately leave a program named *ls* in directory */tmp*, and then change the owner of */tmp/ls* to another identity (denoted by *ssp* below) so that his real identity cannot be discovered later. Subsequently, an unsuspecting user (identified below as the Victim) may change his current directory to */tmp* and attempt to list current directory contents by using (what he believes to be the system-provided command) *ls*. If the Victim's first search path is set to the current directory, then the bogus version of *ls* gets executed instead of the system command *ls*. Thus, the bogus *ls* can acquire all the Victim's privileges, thereby controlling all his activities and allowing the Intruder to (albeit indirectly) write over the unsuspecting user's sensitive objects; e.g., setup files *.login* and *.cshrc*. In this context, these seemingly normal actions of the bogus *ls* program represent intrusions. Note that the mere execution of the bogus program **ls** cannot be treated as an intrusion pattern independent of its context of use. As long as this execution does not cause any harm to the invoker's sensitive objects, it cannot be considered to be an intrusion. However, an alarm must be raised when the Intruder indirectly writes the Victim's setup file *.login*, which would undoubtedly represent a violation of the legal flows in *F*.

The Intruder's actions may be summarized as follows:

```
$cp I_write /tmp/ls
$chmod 0111 /tmp/ls
$chown ssp /tmp/ls
```

The Victim's actions may be summarized as follows:

```
$cd /tmp
$ ls
```

The contents of the */tmp/ls* file created by the Intruder are shown in the Appendix. The command

```
"set path =
(.$HOME/bin /bin /usr/bin /usr/security/bin /etc)"
```

is a typical command found in users' *.cshrc* files on Unix-like system to set up a user's command search paths. After the execution of program */tmp/ls*, the Victim's command search paths in his *$HOME/.cshrc* is changed to

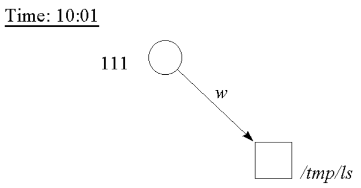
"set path =
(/**tmp**.$HOME/bin/bin/usr/bin/usr/security/bin/etc)"

and a new line "/tmp/ls" is appended to the end of $HOME/.login. Thus, both .cshrc and .login files of the Victim can be affected by the Intruder.

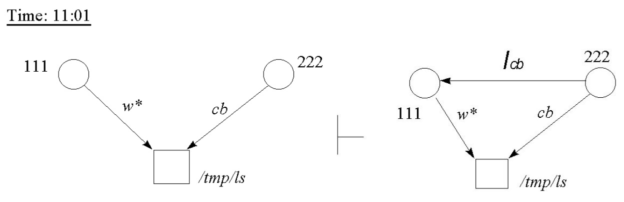
As the consequence of the commands invoked by the Intruder and Victim, the following events can be selected from audit trails:

process	user	primitive	object	time
111	Indruder	write	/tmp/ls	10:01
222	Victim	exec	/tmp/ls	11:01
222	Victim	write	Victim/.login	11:02
222	Victim	write	Victim/.cshrc	11:03

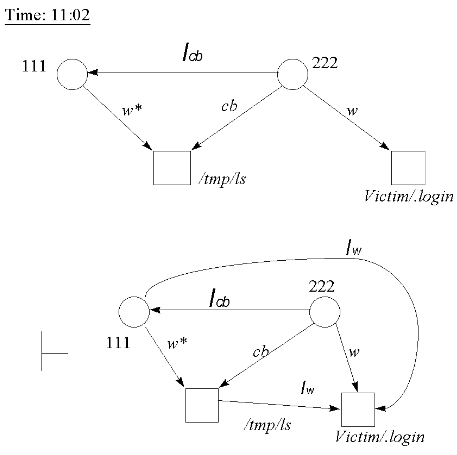
This sequence of events is interpreted by our model as the sequence of protection graphs shown below. At time 10:01, process 111 (owner Intruder) writes into /tmp/ls, which is interpreted as *w*(*111*, */tmp/ls*), shown in the following graph.



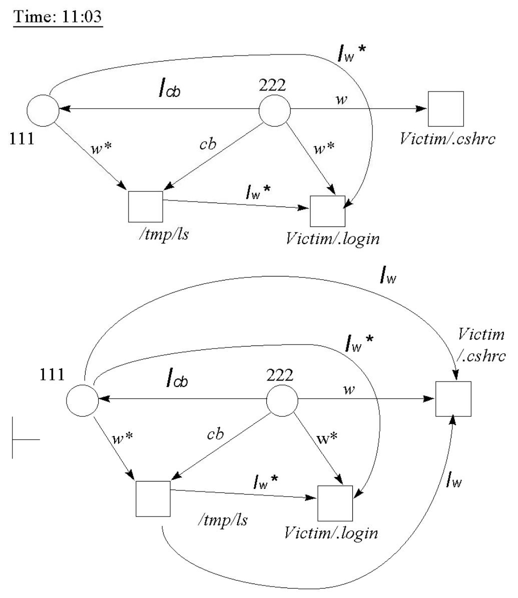
At time 11:01, process 222 (owner Victim) starts being controlled by */tmp/ls*. For simplicity, the previous object executed by process 222 is not shown in the graph. The direct-write relation between process 111 and */tmp/ls* is a past relation because the operation occurred in the past. By Rule 3, a new indirect controlled-by relation between processes 111 and 222 can be derived from the direct controlled-by relation between process 222 and */tmp/ls*, as shown in the following graphs. The indirect controlled-by relation cannot be treated as an intrusion pattern, because it is normal for a user to execute a shared program as long as the execution does no harm to the invoker's sensitive objects.



At time 11:02, process 222 writes data into $HOME/.login. By Rule 2, a new indirect-write relation between process 111 and $HOME/.login and a new indirect-write relation between /tmp/ls and $HOME/.login can be derived. The new indirect write relation between process 111 and the Victim/.login represents a data flow from a malicious user to a victim's sensitive files. This flow would be defined to be illegal in F, since it is reasonable to prohibit other users to directly or indirectly write a user's sensitive file. Therefore, an intrusion instance occurs:



At time 11:03, process 222 writes data into $HOME/.cshrc. By Rule 2, a new indirect-write relation between process 111 and $HOME/.cshrc and a new indirect-write relation between /tmp/ls and $HOME/.cshrc can be derived. The new indirect write relation between process 111 and the Victim/.cshrc represents another intrusion instance:



5 Conclusion

In this paper, we presented a pattern-oriented, intrusion-detection model that can track both data and privilege flows within secure systems. The key advantage of this model is its ability to characterize context-dependent patterns of intrusion, such as those present in unintended use of foreign programs. The use of this model has the potential to give better performance than the statistical approaches for detecting context-dependent intrusions because it focuses on specific patterns and, consequently, avoids analysis of all audit data for a large number of statistical profiles. However, as with all pattern-oriented models, it has the limitation that it can detect neither new, unanticipated intrusion patterns, nor patterns that can only have statistical characteristics.

The open problems of statistical approaches pointed out by Denning [8] are also relevant for pattern-oriented approaches; i.e., generation of false alarms and missed real intrusions. Although our model can be used to detect all intrusions whose patterns are defined, it is possible for an intruder to escape detection whenever the intruder uses an unknown pattern. It is also possible that a potential data or privilege flow may not cause an actual flow. However, because pattern-oriented approaches, such as ours, complement statistical approaches, we anticipate that practical intrusion-detection systems will rely on both the pattern-oriented and the statistical approaches thereby combining the advantages of both.

Appendix

The intrusion program /tmp/ls may have the following contents:

```
/**************************************************************
*PROGRAM NAME: ls
* PATH: /tmp/ls
* FUNCTION:
* 1. indirectly write to $HOME/.login & $HOME/.cshrc
* 2. list the names of files in a directory
**************************************************************/
#include <stdio.h>
#include <fcntl.h>
#define FAIL –1
#define LENGTH              20
char temp1[6] = ”/tmp ”, temp2[6];
char token[11]=”set path=(”;
int fd1;
main(argc, argv)
int argc;
char *argv[];
{
    int fd, n, i;
    char buff[BUFSIZ], path[LENGTH], *ptr, *getenv();
/*get owner’s $HOME dir*/
  if((ptr = getenv(”$HOME”)) == NULL)exit(–1);
    strcpy(path, ptr);
/*get pathname of owner’s.login*/
    strcat(path, ”/.login”);
    if((fd = open(path, O_RDWR | O_CREAT |
        O_APPEND, 0666)) == FAIL)exit( –1);
    chmod(path, 0666);
/*change the search path for commands in $HOME/.cshrc*/
    strcpy(path, ptr);
    strcat(path, ”/.cshrc”);
    if((fd1 = open(path, O_RDWR | O_CREAT,
        0666)) == FAIL)exit( –1);
    chmod(path, 0666);
    change_path();
```

```
/*append a command to $HOME/.login*/
    write(fd, "/tmp/ls", 7);
/*execute the intended ls command*/
    execv("/bin/ls", argv);
}


change_path()
{
    int i, n;
    lseek(fd1, 0L, 0);
    if(search_token() == 0){
        while((n = read(fd1, temp2, 5)) == 5){
            lseek(fd1, –5L, 1);
        write(fd1, temp1, 5);
            for(i = 0; i < 5; i ++)temp1[i] = temp2[i];
        }
        lseek(fd1, –1L*(long)n, 1);
        write(fd1, temp1, 5);
        write(fd1, temp2, n);
        }
        else write(fd1, "set path = (/tmp. /bin)", 17);
    }
    /*search token "set path = ("*/
    search_token()
    {
        int i = 0;
        char c;
        while(read(fd1, &c, 1)! = 0){
            if(c == token[i ++]){
                if(i == 10)return(0); /*size of token*/
            }
            else i = 0;
        }
        return(–1);
    }
    /*********************************************************/
```

ACKNOWLEDGMENTS

This work was supported, in part, by the National Science Council of Taiwan, Republic of China, under Contract No. NSC-85-2213-E-009-032. Some of the results presented in this paper appeared in the *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*. The authors are grateful to Prof. David Spooner and the reviewers, whose suggestions helped improve the paper.

REFERENCES

[1] *A Guide to Understanding Audit in Trusted Systems*, NCSC-TG-001 Version 2, June 1988.
[2] R.W. Baldwin, "Rule Based Analysis of Computer Security," technical report, Massachusetts Inst. of Technology, Mar. 1988.
[3] D.S. Bauer and M.E. Koblentz, "NIDX—A Real-Time Intrusion Detection Expert System," *Proc. Summer Usenix*, San Francisco, pp. 261-273, June 1988.
[4] M. Bishop, "Security Problems with the Unix Operating System," technical report, Dept. of Computer Sciences, Purdue Univ., West Lafayette, Ind., pp. 1-28, 1983.
[5] M. Bishop, "How To Write a Setuid Program," *Proc. Winter Usenix*, vol. 12, no. 1, pp. 5-11, 1987
[6] M. Bishop, "Theft of Information in the Take-Grant Protection Model," *Proc. Computer Security Foundations Workshop*, Franconia, N.H., pp. 194-218, June 1988.
[7] A.R. Clyde, "Insider Threat Identification Systems," *Proc. 10th National Computer Security Conf.*, Baltimore, pp. 343-356, Oct. 1987.
[8] D.E. Denning, "An Intrusion Detection Model," *IEEE Trans. Software Eng.*, vol 13, no. 2, pp. 222-226, Feb. 1987.
[9] T.D. Garvey and T.F. Lunt, "Model-Based Intrusion Detection," *Proc. 14th Nat'l Computer Security Conf.*, Washington D.C., Oct. 1991.
[10] V.D. Gligor et al., "Design and Implementation of Secure XENIX," *IEEE Trans. Software Eng.*, vol. 13, no. 2, pp. 208-221, Feb. 1987.
[11] K. Ilgun, P.A. Porras, and R.A. Kemmerer, "State Transition Analysis: A Rule-Based Intrusion Detection Approach," *IEEE Trans. Software Eng.*, vol. 21, no. 3, pp. 181-199, Mar. 1995. (This paper is an extended version of two previous conference papers: 1) P.A. Porras and R.A. Kemmerer, "Penetration State Transition Analysis: A Rule-Based Intrusion Detection Approach," *Proc. Eighth Ann. Computer Security Applications Conf.*, San Antonio, Texas, pp. 220-229, Dec. 1992; 2) K. Ilgun, "A Real-Time Intrusion Detection System for Unix," *Proc. 1993 IEEE Symp. Research in Security and Privacy*, Oakland, Calif., pp. 16-28, May 1993.)
[12] H.S. Javitz and A. Valdes, "The SRI Statistical Anomaly Detector," *Proc. IEEE Symp. Research in Security and Privacy*, Oakland, Calif., pp. 316-326, May 1991.
[13] T.F. Lunt and R. Jagannathan, "A Prototype Real Time Intrusion Detection Expert System," *Proc. IEEE Symp. Security and Privacy*, Oakland, Calif., pp. 59-66, Apr. 1988.
[14] T.F. Lunt, "Automated Audit Trail Analysis and Intrusion Detection: A Survey," *Proc. 11th Nat'l Computer Security Conf.*, Baltimore, pp. 65-73, Oct. 1988.
[15] S.P. Shieh and V.D. Gligor, "A Pattern-Oriented Intrusion Detection System and Its Applications," *Proc. IEEE Symp. Research in Security and Privacy*, Oakland, Calif., pp. 327-342, May 1991.
[16] S.E. Smaha, "Haystack: An Intrusion Detection System," *Proc. Fourth Aerospace Computer Security Application Conf.*, Orlando, Fla., pp. 37-44, Dec. 1988.
[17] M. Sebring, E. Shellhouse, and M. Hanna, "Expert Systems in Intrusion Detection: A Case Study," *Proc. 11th Nat'l Computer Security Conf.*, Baltimore, pp. 74-81, Oct. 1988.
[18] L. Snyder, "Theft and Conspiracy in the Take-Grant Protection Model," *J. Computer and System Sciences*, vol. 23, pp. 333-347, Dec. 1981.
[19] S.H. Teng, K. Chen, and S.C. Lu, "Adaptive Real-time Anomaly Detection Using Inductively Generated Sequential Patterns," *Proc. IEEE Symp. Research in Security and Privacy*, Oakland, Calif., pp. 278-284, May 1990.
[20] H.S. Vaccaro and G.E. Liepins, "Detection of Anomalous Computer Session Activity," *Proc. IEEE Symp. Security and Privacy*, Oakland, Calif., pp. 280-289, May 1989.