# Logic Induction of Valid Behavior Specifications for Intrusion Detection

Calvin Ko
NAI Labs, Network Associates, Inc.
calvin_ko@nai.com

## Abstract

*This paper introduces an automated technique for constructing valid behavior specifications of programs (at the system call level) that are independent of system vulnerabilities and are highly effective in identifying intrusions. The technique employs a machine learning method, Inductive Logic Programming (ILP), for synthesizing first order logic formulas that describe the valid operations of a program from the normal runs of the program. ILP, backed by theories and techniques extended from computational logic, allows the use of complex domain-specific background knowledge in the learning process to produce sound and consistent knowledge. A specification induction engine has been developed by extending an existing ILP tool and has been used to construct specifications for several ($> 10$) privileged programs in Unix. Coupling with rich background knowledge in systems and security, the prototype induction engine generates human understandable and analyzable specifications that are as good as those specified by a human. Preliminary experiments with existing attacks show that the generated specifications are highly effective in detecting attacks that subvert privileged programs to gain unauthorized accesses to resources.*

## 1. Introduction

After more than a decade of research, intrusion detection has been widely adopted as a retrofit solution to the increasingly important problem of computer security. More than 10 commercial intrusion detection systems (IDS) have been developed and pushed onto the market. All these IDS can at least reliably detect penetrations that employ known attack methods, which account for the majority of the attack incidents. Nevertheless, attackers are getting more advanced and sophisticated. Attackers increasingly make use of automated scripts to attack systems from different locations in a short period of time. In addition, they attempt to escape IDS detection by using new attack methods (e.g., exploiting a new vulnerability) that are not modeled by the signature database of the IDS. Real-time detection of previously unseen attacks with high accuracy and a low false alarm rate is needed but remains a challenge.

Current intrusion detection approaches—anomaly detection, misuse detection, and specification-based detection—have different strengths towards detecting unknown attacks. Anomaly detection, which identifies intrusive activities based on deviations from a normal behavior profile, is able to detect unknown attacks as the normal profile is independent of the system vulnerability. Many different techniques [8, 24, 4, 3, 16] have been employed to establish normal behavior profiles from historical behavior. The major difficulty remains to detect intrusions accurately and minimize the false alarm rate. Also, most techniques identify a procedure for detecting attacks without explaining why the detected incident is an attack, what went wrong, and how to fix the problem. Nevertheless, anomaly detection remains a viable approach to detecting unanticipated attacks.

Misuse detection [7, 12], though widely employed to detect known attacks, also has the potential to detect unknown attacks [17]. The capability lies in the fact that generic signatures/rules can be written to detect classes of attacks that have similar manifestations (e.g., buffer-overflow attacks). In principle, one might be able to hypothesize attacks based on models of attacks and vulnerabilities (e.g., [13]) and develop generic signatures to detect the attacks. However, little research has been done on how to write generic signatures and there is no systematic methodology for developing generic signatures.

Specification-based techniques [10, 11, 23], which detect deviation of executing programs from their valid program behavior, have shown early promise for detecting previously unseen attacks. Specification-based detection approaches the problem from a human-reasoning perspective, trying to develop "formally" what is valid based on the functionality of the program, its usage, and the system security policy. The premise is that penetrations often cause privileged programs to behave differently from their intended behavior, which, for most programs, are fairly regular and can be written concisely. It can achieve a very low false alarm rate and be able to explain why the deviation

0-7695-0665-8(C) 2000 IEEE

is an intrusion. Nevertheless, specifications have to be written by system and security experts for every security-critical program in a system. The specification-based approach will benefit from techniques that automate the development of specifications.

In this paper, we present an approach for developing security specifications of programs in a highly automated fashion. Our approach employs a machine learning method, Inductive Logic Programming (ILP) [19], to synthesize first order clausal theory describing valid operations of a program from its historical behavior and background security knowledge.

ILP, a well-established discipline with a good theoretical foundation, allows generation of sound and consistent knowledge that is amenable to formal reasoning. Using logic as the representation language, ILP can make use of complex domain-specific background knowledge to produce intelligent knowledge. Based on the Mode-Directed Inverse Entailment theory[20], we derived an induction algorithm that is tailored for security and protection. We implemented the algorithm by extending an existing ILP tool, Progol [20], which was developed for general purpose learning. The prototype inductive engine, sProgol, was used to construct specifications for several Unix programs from their sample runs. The generated specifications are of very high quality and are effective in detecting attacks, as confirmed by preliminary experiments with existing attacks.

The rest of the paper is organized as follows. Section 2 describes our approach to learning valid behavior specifications. Section 3 formally defines the learning problem. Section 4 presents the learning algorithm together with some background and theories of ILP that are being applied. Section 5 shows the experiments we performed and the results obtained. Section 6 discusses related research. Section 7 provides conclusions and suggests future work.

## 2. Learning Specifications from Example Executions

Generally speaking, a valid behavior specification of a program constrains the sequence of operations that can be performed by the program in execution. To make the problem tractable, we confine ourselves in this paper to consider learning of the set of valid operations (or accesses) of a program, ignoring the ordering of the accesses. In particular, we consider the operations of a program at the kernel boundary, at which system calls are invoked to perform access to objects. Valid access specifications have been found to be effective in discerning attacks for many security-critical programs [10].

Informally, a valid access specification of a program classifies individual operations of the program as either good or bad. Given example runs of a program consisting of a set of valid execution traces and a set of possibly empty invalid execution traces, collected by running the program on a securely configured host and running it under attacks, our goal is to construct a valid access specification that satisfies the following criteria.

**Completeness:** All operations in a valid trace should be classified as good (or valid).

**Consistency:** For every invalid trace (or intrusion trace), a valid access specification should classify at least one operation as bad (or invalid).

**Compactness:** The specification should be concise so that it can be inspected by a human and be able to use for real-time detection. One simple compactness measure is the number of rules (or clauses) in a specification.

**Predictability:** The specification should be able to explain future execution traces, not producing a high false alarm rate.

**Detectability:** The specification should fit closely to the actual valid behavior and reject future execution traces which are intrusions.

Completeness and consistency are verifiable criteria which can be formulated (See Section 3) and checked against a set of example traces. However, it is more difficult to definitely evaluate predictability and detectability of a specification as there could be infinite number of future valid traces and intrusion traces. Developing a good specification is not a easy process. It requires significant insight into the internals of complex applications (e.g, sendmail) and continual re-analysis as program revisions are released. Below we discuss some techniques or guidelines for developing valid access specifications based on our experience, with an attempt to codify the knowledge in our learning algorithm to achieve better predictablity and detectability.

**Least Access** To maximize the chance of detecting intrusions, a valid access specification should tightly restrict the resources that are accessible by a program, where the resources could include concrete file objects or abstract network services. One important guideline is to allow a program to access to only resources that are needed by the program for accomplishing its jobs. This is actually a variant of the "least privileged principle" [22]. In general, we can identify the valid accesses from the functionality of the program (e.g., described in the manual pages) and the normal runs of the program. In addition, the allowable operations of a program may be site-specific, depending on the usage, as a site may not use all the functions of a program.

**Attributes of Operations** The validity of an operation usually depends not only on its type, but also other parameters such as the attributes of the process which performs the operation (e.g., process ID, user ID of the owner of the process), attributes of the object being accessed (e.g., permission modes), and other abstract state information. It is essential to identify the attributes which values are needed for distinguishing intrusions from legitimate behavior and include the relevant attributes in the model so that the attributes will be considered by the learning algorithm. In general, attributes about the process, system call arguments, the path name, and the object being accessed are necessary. Also important are the creator of an object, and the user on behalf of which the program is running. Attributes that can uniquely identify an object are also needed.

**Generalization** In the development of a valid access specification, an expert usually identifies commonality of the operations (e.g., having the world-readable permission bit on) of a program and produces a rule that explain many operations. Such generalization process could enhance the conciseness of a specification. Also, the operations of a program may vary across executions, depending on the input, configurations, and environment. Generalization is needed to predict operations in future executions of programs. For example, in Unix, a program may use a file-name generation function to obtain a name for a temporary file, which is different in different executions. It is essential to observe some characteristics of the temporary files (e.g., they all share the same prefix) to produce a general rule that can explain the operations related to the temporary file.

On the other hand, a general rule that explains many valid operations of a program may also cover many unnecessary operations. Therefore, it is very important to strike a good balance between generalization and least access and to develop general rules in an intelligent manner. For example, even observing that a program writes to many files in the */etc* directory, one would not use a rule to allow the program to write to all the files in the */etc* directory; it is because such rule allows the program to write to many security-critical files that are not needed to be accessed by the program. A good rule should explain many valid operations but no invalid operations, and should not cover too many other high-privileged operations. Therefore, substantial reasoning is involved and extensive knowledge in system and security is required to produce intelligent specifications.

# 3. Problem Definition

In this section we first present a brief background of Inductive Logic Programming (ILP). We then define the specification learning problem formally as an ILP problem. The terminology commonly used in first order logic is described in Appendix A.

## 3.1. Inductive Logic Programming

Inductive Logic Programming has been defined as the intersection of Machine Learning and Logic Programming [19]. Loosely speaking, induction logic programming constructs knowledge from examples, both represented in first order logic, by reversing the process of deductive inference.

Deductive inference derives consequences $E$ from a prior theory $C$. For example, if $C$ says that all kinds of birds have wings, $E$ might state that a particular kind of bird (e.g. a pigeon) has wings. Inductive inference derives a general belief $C$ from specific beliefs $E$. After observing that eagles, pigeons, and sea gulls have wings, $C$ might be conjecture that all kinds of birds have wings. Inductive inference is a very common form of everyday reasoning.

Formally, the general problem of ILP is, given background knowledge $B$ and examples $E(E^+ \cup E^-)$, find the simplest consistent hypothesis $H$ such that

$$B \wedge H \models E$$

As an example, imagine yourself as trying to learn about the concept of *father*. You are given some background knowledge $B$ about particular parents and their children as well as with their genders.

$$B = \begin{cases} male(calvin). \\ male(timothy). \\ female(mandy). \\ female(victoria). \\ parent(calvin, timothy). \\ parent(calvin, victoria). \\ parent(mandy, timothy). \\ parent(mandy, victoria). \end{cases}$$

You are now given the following facts, which include the relationships between particular fathers and their children (positive examples $E^+$) and the relationships that do not hold (negative examples $E^-$).

$$E^+ = \begin{cases} father(calvin, timothy). \\ father(calvin, victoria). \end{cases}$$

$$E^- = \begin{cases} \overline{father(mandy, timothy).} \\ \overline{father(mandy, victoria).} \\ \overline{father(calvin, mandy).} \end{cases}$$

The goal of ILP is to learn the target relation *father* from the given background knowledge $B$ and example $E$ ($E^+ \cup E^-$). For example, the Progol system is able to learn the following concept of father.

$$father(X, Y) \leftarrow parent(X, Y) \wedge male(X)$$

### 3.2. Specification Learning Problem

In our problem domain, we model the valid operations of a program using a set of system-call predicates, each describing the valid operations associated with one particular system call (e.g., read, write, execve). In general, an operation performed by a program denotes an execution of a system call, and is represented by a ground literal (i.e., an atomic formula without any variables)

$$scall(t_1, t_2, t_3, \cdots, t_n),$$

where $scall \in SC$, the set of system-call predicate symbols, and $t_i$, $1 \leq i \leq t_n$, are terms describing various attributes (e.g., pathname of the object, ID of the process performing the operation) of the operation. (See Figure 1(a) for examples of operations) A trace of a program execution is an ordered set of operations performed by the program execution. If we ignore the order of the operations, we can view a trace as a set of operations and represent a trace as a set of ground literals.

Note that the representation of an operation is crucial to the success of our approach. As mentioned in Section 2, it is important to include essential attributes of operations which values can affect the validity of the operations.

Formally, our problem is: given example runs of a program consisting of a set of valid trace $T_V$ and a set of possibly empty intrusion trace $T_I$, construct a set of Horn clauses $S$ of the form

$$scall(a_1, a_2, \cdots, a_i) \leftarrow L_1, L_2, \cdots, L_j$$

where $scall \in SC$ and $L_1$, $L_2$, $\cdots$, $L_i$ are literals defined in the background knowledge $B$ (See Section 4.3 for examples of background knowledge), such that the following constraints are satisfied.

$$B \wedge S \models t \quad \forall t \in T_V \quad (Completeness) \qquad (1)$$
$$B \wedge S \not\models t \quad \forall t \in T_I \quad (Consistency) \qquad (2)$$

The set of Horn clauses $S$ is called a *valid access specification*. The set of clauses $B \wedge P$ forms a logic program which defines the relationships among the attributes in a valid operation of a program. Note that this definition is slightly different from the ILP problem definition described in Section 3.1. To cast our problem as an ILP problem, an example set of valid operations ($E^+$) should be obtained by including all operations in all valid traces into the set. In addition, an example set of invalid operations ($E^-$) should be constructed by selecting at least one invalid operation from each intrusion trace. Note that not all the operations in the intrusion traces are invalid; in fact, operations in a valid trace and operations in an intrusion trace could overlap. Therefore, it is important to extract operations that are truly invalid for inclusion in $E^-$. In general, obtaining a good set of invalid operations from intrusion traces of a program is not an easy process. The lack of attack data further hinders the problem of getting invalid examples. In view of this problem, the learning algorithm is designed to work well with only valid example operations.

## 4. Induction Algorithm

This section describes the ILP algorithm for constructing a valid access specification from background knowledge and examples. Muggleton [19] regards ILP as a search problem in which a space of candidate solutions is searched to find the best solution with respect to some acceptance criterion characterizing solutions to the ILP problem. In principle, the problem can be solved using a naive generate and test algorithm. However, such algorithm is computationally too expensive to be of practical interest. Therefore, different techniques based on different theories [26, 21, 14, 20] are employed to structure the solution space to allow for pruning of the search so that a good solution can be found efficiently. We employ the Mode Directed Inverse Entailment (MDIE) [20] approach to confine and structure the search space of the solution specifications. We choose this approach because it matches what we need and the tools exist on which to base our engine.

### 4.1. Confining Solution Space using MDIE

This subsection briefly describes the MDIE theory originally presented in [20] as well as how the theory helps structure and confine the solutions to an ILP problem. Recall the general problem of ILP that given background knowledge $B$ and examples $E$ find the simplest consistent hypothesis $H$ such that $B \wedge H \models E$. If we rearrange the equation using the law of contraposition we get the more suitable form

$$B \wedge \overline{E} \models \overline{H}$$

Restricting $H$ and $E$ to being single Horn clauses, $\overline{H}$ and $\overline{E}$ will be ground skolemised unit clauses. Let $\overline{\perp}$ be the conjunction of ground literals which are true in all models of $B \wedge \overline{E}$, we have

$$B \wedge \overline{E} \models \overline{\perp}$$

```
(a) An example list of operations produced by the program ping

    read(s(657,joe), p(657,0,1), /etc/mtab, f(0,0,19232), m(6,4,4)).
    read(s(657,joe), p(657,0,1), /etc/networks, f(0,0,14382), m(6,4,4)).
    read(s(657,joe), p(657,0,1), /etc/protocols, f(0,0,1530), m(6,4,4))).
    read(s(657,joe), p(657,0,1), /etc/hosts, f(0,0,2534), m(6,4,4)).
    read(s(657,joe), p(657,0,1), /etc/spwd.db, f(0,0,1933), m(6,0,0)).

(b) Knowledge synthesized by the Progol system

    read(S,P,N,F,M) :- indir(N, /etc).

(c) The desired knowledge constructed by the author with considerations for security

    read(S,P,N,F,M) :- indir(N,/etc), worldreadable(M).
    read(S,P,N,F,M) :- isfile(N,/etc/spwd.db).
```

**Figure 1. List of Operations Performed by Ping**

Since $\overline{H}$ must be true in every model of $B \wedge \overline{E}$ it must contain a subset of the ground literals in $\overline{\bot}$. Hence

$$B \wedge \overline{E} \models \overline{\bot} \models \overline{H}$$

and so for all $H$

$$H \models \bot \tag{3}$$

From (3), the complete set of candidate solutions for $H$ could in theory be found from those clauses that imply the most specific clause $\bot$. Restricting $H$ to be nonrecursive clauses, the complete set of solutions could be found from those clauses which $\theta$-subsume $\bot$[1]. (See Appendix A for the definition of $\theta$-subsume.) Also, the clauses which $\theta$-subsume $\bot$ form a sub-lattice with a most general element $\square$ (empty or false clause) and a least general element $\bot$ [19]. In general, the number of ground literals in $\bot$ are infinite, but it can be restricted by using mode declarations of the head and body of the hypothesis clause. (A mode for a predicate is a description of the possible arguments and their types of a predicate when it is called [20].)

Given the MDIE results, the search for a suitable hypothesis for an example $i$ is reduced to the bounded sub-lattice

$$\square \preceq H \preceq \bot_i,$$

where $\preceq$ denotes $\theta$-subsumption, $\bot_i$ is the most specific clause that can explain the example $i$, and $\square$ is the empty (false) clause. With appropriate head and body mode declarations, the set of possible hypotheses is further confined syntactically to rule out nonsense rules to speed up the search process. Algorithms for generating the most specific clause $\bot_i$ from an example $i$ and for enumerating the lattice of clause that $\theta$-subsume $\bot_i$ were developed and proved [20].

[1]It was illustrated that $x \models y$ not necessary imply $x$ $\theta$-subsume $y$ for self-recursive clauses [19].

## 4.2. Finding the Best Solution

Besides confining and structuring the search space to allow for efficient searching, another aspect of ILP is how to evaluate the candidate solutions. In particular, there could be many candidate solutions that satisfy the completeness, consistency, and compactness criteria. We need to identify from them the best solution with respect to predictablity and detectability.

Most ILP tools use similar evaluation criteria, which identify the best clause as a clause that achieves the highest compression, that is, the simplest clause that can explain the largest number of positive examples, without incorrectly explaining any negative examples. For learning with positive only data, many ILP algorithms estimate the probability distribution of positive and negative examples and assume the positive examples are gathered by drawing randomly from all the examples and discarding negative examples drawn. Existing ILP evaluation methods do not apply to our problem well. We illustrate the limitation of existing ILP algorithms using the example shown in Figure 1. Figure 1(a) shows an example list of operations produced by the *ping* program. The first parameter denotes the identity of the program execution, the second parameter denotes the process which performs the operation, the third parameter denotes the path name of the object, the fourth parameter denotes the attributes of the object (e.g., user ID of the owner), and the last parameter denotes the owner, group and the permission mode of the object. The Progol system synthesizes the knowledge shown in Figure 1(b), which allows reading of any files inside the `/etc` directory. (The predicate `indir(F, D)` means that the path $F$ is inside the directory $D$).

However, from a security perspective, the clause in Figure 1(b) is not a good rule for explaining the valid operations because it allows *ping* to access many unnecessary

privileged files in the `/etc` directory. An attacker gaining controls of the program could perform these privileged operations (e.g., read `/etc/master.passwd`, which contains the passwords of users in plain text) without being detected by the specification. Therefore, the author chooses other clauses to explain the examples (Figure 1(c)), which allow the *ping* program to read the `/etc/spwd.db` file and only files inside the `/etc` directory that are also publicly readable.

#### 4.2.1. Generality and Privilege

We introduce two quantitative measures of a hypothesis clause: *generality* and *privilege* for evaluation of the quality an hypothesis with respect to security.

The *generality* of a hypothesis clause measures how broad are the objects that are allowed to be accessed by the hypothesis. For example, the generality of `read(S, P, N, F, M) :- worldreadable(M)` is high as there are many files that are publicly readable. On the other hand, the generality of `read(S, P, N, F, M) :- isfile(N, "/etc/passwd")` is very low since there is only one file whose name is *"/etc/passwd"*.

The *privilege* of a hypothesis clause is a measure of the privileges of the operations allowed by the hypothesis. For example, `write(S, P, N, F, M) :- own(root, F)` is high as it encompasses many privileged operations (e.g., write operations to the password file).

Obviously, the more general a hypothesis is, the greater is its generality and privilege. Any generality function and privilege function should satisfy this criterion, as described below.

**Definition 1:** A generality function $g$ over a set of well-formed formula (*wffs*) is well defined if and only if for any clauses $C1$ and $C2$,

$$C1 \models C2 \quad implies \quad g(C1) \geq g(C2)$$

**Definition 2:** A privilege function $p$ over a set of *wffs* is well defined if and only if for any clauses $C1$ and $C2$,

$$C1 \models C2 \quad implies \quad p(C1) \geq p(C2)$$

**Definition 3: Generality measure.** Given a snapshot of a system consisting of a set of objects $O$, and let $H$ be a *wff*, the generality g of $H$ is

$$g(H) = \frac{|O_H|}{|O|}, \qquad O_H = \{\mathcal{O}_{op} \mid B \wedge H \models op\},$$

where $\mathcal{O}_{op} \in O$ denotes the object associated with the operation *op*. The function $g$ is well defined because if $H \models H_1$, then $O_{H_1} \subseteq O_H$ by the definitions of $O_H$ and $O_{H_1}$, hence $g(H) \geq g(H_1)$.

**Definition 4: Privilege measure.** Let $H$ be a *wff* and $\mathcal{P}$ be a mapping over the set $X$ of possible operations which assigns privilege value from 0 to 1 to every possible operation. The *privilege* $p$ of $H$ is defined as

$$p(H) = \max_{B \wedge H \models op, op \in X} \mathcal{P}(op)$$

The functions $p$ are well defined. Suppose $p(H_1)$ is $v$, there exist an operation $op$ such that $B \wedge H_1 \models op$ and $\mathcal{P}(op) = v$. If $H \models H_1$, then $B \wedge H \models op$, hence $p(H) \geq p(H_1)$.

#### 4.2.2. Algorithm

Figure 2 shows the algorithm for construction of a valid access specification from the background knowledge $B$ and a set of examples $E$ ($= E^+ \cup E^-$), which is constructed from a set of example traces ($T_V \cup T_I$) using the method described in Section 3.

1. Set $S = \emptyset$
2. if $E^+ = \emptyset$ return $S$
3. Pick an example $e$ from $E^+$
4. Construct clause $\perp$ for $e$
5. Construct clause $H$ from $\perp$
   (a) Iterate through the subsumption lattice starting from the most general clause
   (b) find the clause $h$ with highest quality $f_h$ and without covering any negative examples, i.e., $B \wedge H \not\models e'$ for any negative example $e'$
6. Set $E' = \{e | B \wedge H \models e\}$
7. Let $S = S \cup H$
8. Set $E^+ = E^+ - E'$,
9. Goto 2

**Figure 2. Specification Construction Algorithm**

The algorithm associates the following values to each of the candidate hypothesis $h$.

$g_h$ = the generality of $h$

$p_h$ = the privilege of $h$

$c_h$ = the length of the clause $h$

$e_h$ = the explanation power—number of valid traces which can be partially explained by the clause h. (A clause partially explains a trace means that the clause explains at least one operation in the trace)

$f_h = e_h - (g_h + p_h + c_h)$

In the computation, the values $g_h$ and $p_h$ are normalized to range from 1 to the total number of valid traces. The value $f$ is set to favor short, low-privilege, and low-generality clauses but to be able to explain examples in many traces (high-explanation power). Our goal is to minimize the privilege of the hypothesis and to allow just the operations needed by a program while not making the specification too complex. In addition, we set the explanation power $e_h$ to be the number of valid traces that can be partially explained by the clause h. That is, $e_h = |W_h|$ where

$$W_h = \{T \mid \exists\ op \in T\ s.t.\ B \wedge h \models op, T \in T_V\}.$$

In other general ILP tools, $e_h$ is set to the number of positive examples explained by h. Such setting does not work well for our purpose since a clause may explain a large number of operations in one very long trace but may not explain any operations in other traces.

The algorithm generates a complete and consistent valid access specification with respect to the given set of example traces $T$.

**Justification:** First, the algorithm will terminate because examples are taken out from $E^+$ in step 8 in each iteration and eventually $E^+$ will become empty; note that $E'$ in step 6 must at least contain the example $e$ selected in step 3 and at least $e$ will be taken out from $E^+$ in step 8. Next, for any valid trace $t$, all operations in $t$ will be in $E^+$ by the way $E^+$ is constructed. Each operation in $t$ must be taken out from $E^+$ in step 8 since $E^+$ is empty when the algorithm terminates. Therefore, there must be a clause $H$ in $S$ such that $B \wedge H \models op$ for each $op$ in $t$. Hence, the specification is complete. Last, for each invalid trace $t'$, there is an invalid operation $op^- \in E^-$ by the way we construct $E^-$ from invalid traces. Every $H$ added to $B$ must satisfy $B \wedge H \not\models op^-$. Therefore, $B \wedge H \not\models t'$ for each $H$ added in step 7, hence the consistency of the specification.

### 4.3. Implementation

We implemented the induction algorithm by modifying the Progol tool [20], an existing implementation of MDIE. Progol contains code for generating the most specific clause $\perp_e$ from an example $e$ and for iterating through the lattice of possible solutions which $\theta$-subsume $\perp_e$. In addition, Progol is able to check for contradiction when a new clause is added as background knowledge to the system.

The availability of a tool like Progol allows us to implement our algorithm very quickly. We have modified the code for evaluating the quality of a hypothesis $f$ as described in the Section 4.2. In addition, we have modified the searching algorithm for finding the solution clause from the lattice of solutions. The prototype (sProgol) accepts an input file containing head and body mode declarations, background knowledge, and a set of examples to synthesize a set of Horn clauses that can explain the examples.

**Mode Declarations**

We use sensible head and body mode declarations to restrict the terms that appear in the solution. The head and body mode declarations that are used for learning valid read operations are shown below.

```
1.  :- modeh(*, readop(+subj, +proc, +path,
                       +fattr, +pmode))?
2.  :- modeb(1, group(+subj, +fattr))?
3.  :- modeb(1, createdby(+proc, +fattr))?
4.  :- modeb(1, createdby(+subj, +fattr))?
5.  :- modeb(1, worldreadable(+pmode))?
6.  :- modeb(1, groupreadable(+pmode))?
7.  :- modeb(1, ownerreadable(+pmode))?
8.  :- modeb(1, worldwritable(+pmode))?
9.  :- modeb(1, groupwritable(+pmode))?
10. :- modeb(1, ownerwritable(+pmode))?
11. :- modeb(1, isfile(+path, #path))?
12. :- modeb(8, indir(+path, #path))?
13. :- modeb(1, own(+subj, +fattr))?
14. :- modeb(1, own1(#id, +fattr))?

15. id(U) :- int(U), U =< 65535, U >= 0.
16. subj(s(Sid, U)) :- id(U), id(Sid).
17. pmode(m(W,G,O)) :- int(W),int(G),int(O),
        W=<7, W>=0, G=<7, G>=0, O=<7, O>=0.
```

Line 1 declares the mode of *readop*, which can appear in the head of the hypothesis. The predicate *readop* takes five compound arguments of type *subj*, *proc*, *path*, *fattr*, and *pmode*, describing the program execution, the acting process, the path to the object, the attributes of the object, and the permission mode of the object. Defined on line 16, *subj* identifies a program execution, denoted by a functor $s(pid, uid)$, where $pid$ is the head process ID of the program execution and $uid$ is the ID of the user who executes the program. *proc* describes a process, identified by $p(pid, euid, egid)$, which performs the operations.

Lines 2-14 declare the predicates that can appear in the body of the hypothesis. Most of them are self-explanatory. The predicate *own(S, F)* is true if the file $F$ is owned by the user of $S$. The predicate *own1(I, F)* is true if the file $F$ is owned by the user ID $I$. We do not include a mode declaration for *worldexecutable*, *groupexecutable*, and *ownerexecutable* because whether a program can read a file should have no relationship with the executable bits of the file. Line 15 defines an (process or user) ID as an integer between 0 and 65535. Line 17 defines *pmode*, which denotes the permission mode of a Unix file represented as a triple of octets. (e.g., The permission mode 644, which indicates read/writeable by owner and readable by group and world, is represented as `m(6, 4, 4)`).

**Background Knowledge**

The background knowledge contains the definitions of the predicates that can appear in the body of the hypothesis and

```
(a) readop(s(3012,215,20), p(3012,0,20), ['ld.so',libexec,usr], f(3,7,184332),m(5,5,5)).

(b) readop(A, B, C, D, E) :- worldreadable(E), groupreadable(E), ownerreadable(E),
      isfile(C,['ld.so',libexec,usr]), indir(C,[libexec,usr]),
      indir(C,[usr]), indir(C,[]), own1(3,D).

(c) readop(A, B, C, D, E) :- worldreadable(E).
```

**Figure 3. An Illustration: (a) An example operation, (b) The most specific clause, (c) The selected hypothesis clause**

other knowledge concerned with the state of the system. For example, the meanings of the predicates *worldreadable*, *groupreadable*, and *ownerreadable* are given as follows.

```
worldreadable(m(O, G, W)) :- W >= 4.
groupreadable(m(O, G, W)) :- G >= 4.
ownerreadable(m(O, G, W)) :- O >= 4.
```

A world/group/ownereadable predicate takes a permission mode, represented by a functor $m(O, G, W)$, as an argument and is true if the permission mode is world/group/ownerreadable.

As another example, knowledge about the creator of objects is also included.

```
created(762, 7624, 1123).
created(762, 123, 1345).
created(762, 123, 1346).
```

The predicate `created(A, B, C)` specifies that process $A$ has created a file with *inode* number $B$ and generation number $C$. In Unix, every file has an unique inode containing information necessary for a process to access the file. The example indicates that process 762 has created three files. Note that the files corresponding to the second and third statements are different files from a user's perspective, but they correspond to an inode with the same inode number. The inode number alone cannot uniquely identify a file because inodes can be reused; Systems often choose the most recently deallocated inode for a new file. Using both the inode number and the generation number, which is incremented every time the inode is reused, we can identify a file uniquely over the lifetime of a system.

Besides the meanings of body predicates and state information, another important type of background knowledge is the generality and privilege of a hypothesis clause. From Definition 4, the privilege of a hypothesis depends on a privilege mapping that gives a valid privilege value to each operation. We use a set of `priv` statements to define a privilege mapping that assigns a privilege value from 1 to 10 to each operation for calculation of privilege values of hypotheses. Examples of `priv` statements are:

```
priv(1) :- readop(S,P,N,F,m(O, G, W)), W >= 4.
priv(9) :- readop(S,P,N,f(0,G,I),m(O,G,W)), W < 4.
priv(10) :- readop(S,P,[spwd, etc],F,m(6,0,0)).
```

The body of a `priv` statement denotes a set of operations, for example, the first `priv` statement denotes all read operations on a publicly readable file. Given an operation $op$, the privilege of $op$ is the largest $i$ such that `priv(i)` is true for the operation $op$. sProgol computes $p(H)$ based on the privilege mapping defined by a set of `priv` statements. In this example, the privilege of the clause `readop(A, B, C, D, E) :- isfile(C, [spwd, etc])` is 10.

To calculate the generality of a clause, we use a very simple scheme that assigns a generality value to a clause based on the terms that appear in the body of the clause. We associate a generality value from 1-10 to each predicate symbol that can appear in the body of an hypothesis. Table 1 shows the value of each symbol. The generality of a clause is calculated based on the terms in the body of the clause. The generality of a hypothesis get the smallest value of the terms that appear in the body of the hypothesis. For example, the generality value of *readop(A, B, C, D, E) :- isfile(C, [passwd,etc]), worldreadable(M)* is 1.

| Clause | Generality Value |
|---|---|
| $indir(X, D), \|D\| < 5$ | $9 - \|D\|$ |
| $indir(X, D), \|D\| >= 5$ | 4 |
| $worldreadable(M)$ | 5 |
| $groupreadable(M)$ | 5 |
| $ownerreadable(M)$ | 5 |
| $owner(X, Y)$ | 4 |
| $isfile(X, Y)$ | 1 |
| $createdby(X, Y)$ | 1 |

**Table 1. Generality Value**

### An Illustration

Figure 3(a) shows an example positive operation $e$ and 3(b) the most specfic clause $\perp_e$ computed by sProgol. sProgol searches for the best solution starting from the most general clause `readop(A,B,C,D,E)` to find the solution with the highest $g$ value (Figure 3(c)). sProgol prunes the solutions that do not make sense or are redundant. For example, at most one *indir* term in the clause is allowed.

In constructing the valid access specification of a program, the hypothesis for each relevant system call is generated separately. Currently, we have developed background knowledge for learning the security specifications for seven system calls: read, write, chmod, chown, creat, unlink, execve. A rename operation to an existing destination file is treated as an unlink operation on the source file followed by a write operation to the destination file.

## 5. Experimental Results

To test the validity of our approach, we have used sProgol to synthesize the valid access specifications for over 10 privileged programs in FreeBSD Unix, version 2.2.2. The traces of normal program execution were obtained using the Generic Software Wrapper Toolkit [5], which allows us to intercept the operations of processes at the kernel boundary and collect the relevant information about the operations. As a result, we are able to obtain valuable information that is not available from traditional audit trails (e.g., read and write operations, generation number of an inode). We have collected normal data for the programs and used the induction engine to construct specifications for the programs. The collected data is first preprocessed to 1) generate background knowledge concerned with what files do processes create and the parent child relationship between processes and 2) to translate the operations into a form as described in Section 3. Figures 4 and 5 show the generated specifications for *lpr* and *passwd*.

```
readop(A,B,C,D,E) :- worldreadable(E).
readop(A,B,C,D,E) :- isfile(C,['spwd.db',etc]).
readop(A,B,C,D,E) :-
        isfile(C,['.seq',lpd,output,spool,var]).
readop(A,B,C,D,E) :- createdby(A,D).

writeop(A,B,C,D,E) :-
        isfile(C,['.seq',lpd,output,spool,var]).
writeop(A,B,C,D,E) :- createdby(A,D).

creatop(A,B,C,D,E) :-
        indir(C,[lpd,output,spool,var]).
chownop(A,B,C,D,E) :- createdby(A,D).
unlinkop(A,B,C,D,E) :- createdby(A,D).
```

**Figure 4. Generated Specification for lpr**

We have tested the specifications using some existing attacks, in particular, an attack to *imapd* [1] and an attack to *lpr* [2]. The generated specifications were translated into wrappers that intercept the system calls made by the programs to check for deviations. The translation was done manually, but it can be automated very easily. The generated specifications of *imapd* and *lpr* are able to detect both attacks. Table 2 summarizes the information about the generated specifications. The generated specifications are as good as the specifications developed by a human expert. In addition, they can be inspected and are understandable by a human.

```
readop(A,B,C,D,E) :- worldreadable(E).
readop(A,B,C,D,E) :- isfile(C,['spwd.db',etc]).
readop(A,B,C,D,E) :-
        isfile(C,['master.passwd',etc]).
readop(A,B,C,D,E) :- createdby(A,D).

writeop(A,B,C,D,E) :- own(A,D).
writeop(A,B,C,D,E) :- isfile(C,[tty,dev]).
writeop(A,B,C,D,E) :- createdby(A,D).
writeop(A,B,C,D,E) :- isfile(C,['pwd.db',etc]).
writeop(A,B,C,D,E) :-
        isfile(C,['spwd.db',etc]), own1(0,D).
writeop(A,B,C,D,E) :- isfile(C,[passwd,etc]).
writeop(A,B,C,D,E) :-
        isfile(C,['master.passwd',etc]).
creatop(A,B,C,D,E) :- indir(C,[etc]).
chownop(A,B,C,D,E) :- createdby(A,D).
```

**Figure 5. Generated Specification for passwd**

## 6. Related Work

Substantial amount of research has been done trying to construct profiles or rules from normal examples that can distinguish normal and intrusion behavior. Early work focuses mostly on modeling behavior of users, whose behaviors are highly irregular and depend greatly on the environment. IDES [18] employs a statistical approach to profile the normal behavior of users and treats behavior that significantly deviates from the normal profiles as intrusions. The Time-based Inductive Machine (TIM) [24] inductively generates time-based rules for characterizing normal behavior patterns of users. It focuses on the sequential relationship between operations and incorporates some probabilistic measurements into the algorithm (For example, after observing $E1$ followed by $E2$, the probability of seeing $E3$ is $90\%$). Wisdom and Sense (W&S) [25] attempts to generate a tree-structured rule forest for describing historical behavior patterns of users. The rules specify normal feature values conditioned on the values of other features. However, the rule base tends to be very large ($10^4$ to $10^6$), as opposed to the size of our specifications, which is on the order of 10 to 20 clauses.

Recently, researchers focus more on modeling the normal behavior of programs, which is more regular than behavior of users. In [4, 3], short sequences of system calls are used as discriminators to differentiate legitimate behavior and intrusions. In addition, data mining techniques [15] and other Artificial Intelligence techniques such as neural networks [6] are used to search for the best set of rules or procedures, from a given set of normal data and a set of intrusion data, that can best distinguish legitimate behavior and intrusions. In [9], a simple file name translation scheme is used to model normal file access patterns of programs for

| Program | No. of Executions | No. of operations | Specification Length |
|---|---|---|---|
| fingerd | 10 | 826 | 3 |
| lpr | 9 | 498 | 9 |
| ftpd | 8 | 2181 | 7 |
| passwd | 11 | 1653 | 14 |
| sendmail | 8 | 5483 | 10 |
| imapd | 6 | 10957 | 11 |
| at | 6 | 346 | 9 |
| atq | 10 | 415 | 2 |

**Table 2. Data about Generated Specifications.**

detecting Trojan Horses.

Our approach is different from these approaches in several ways. First, our approach employs techniques in Inductive Logic Programming to construct declarative knowledge instead of a set of procedures that does not have any contextual meaning. The constructed knowledge is represented in a simple logic notation that is understandable by humans and amenable to formal analysis. Also, the generated specifications are concise and can be checked by humans.

Second, we monitor executions of programs at the system call level and include many security-relevant attributes pertaining to the operations in the analysis. The generated specification describes the relationship among attributes of the operations in a valid execution. At the system call level, the set of operations and their attributes are well defined and accesses to resources must be done by making system calls. In addition, we do not start off restricted by the information available in the audit trails. It is very difficult to achieve high detection accuracy with insufficient data. Instead, we identify attributes of a program execution that are useful in discerning attacks and include them in our learning model (e.g., we use both inode and generation number to uniquely identify a file).

Last, we incorporate substantial background knowledge on systems and security (e.g., privilege and generality values) in the learning process to produce intelligent specifications. We strongly believe that extensive security knowledge is needed for learning algorithms to produce good specifications or procedures for discerning attacks. The best procedures for detecting attacks with respect to a set of examples may not be the best detection procedures in real life, if the examples is not comprehensive, which is currently the case for intrusion data. By incorporating extensive background knowledge in our algorithm, we highly believe that our approach generates specifications that are of high quality with respect to the examples, but also to real situations.

## 7. Conclusions and Future Work

This paper presents an innovative approach to constructing valid behavior specifications that can be used to reliably detect intrusions related to misuse of privileged programs. Our approach employs Inductive Logic Programming to construct valid access specifications of programs from normal runs for detecting intrusions. The constructed specifications are represented in first order logic. Our approach inherits the benefits of both specification-based detection and anomaly detection. First, formal and analyzable specifications are used for discerning attacks accurately and efficiently. Second, rules for detecting intrusions that are independent of system vulnerabilities are generated automatically from the normal runs of a program with little manual intervention.

Using the theories and techniques in ILP, the specification construction problem is transformed into a search problem and a quality measurement problem. We derived an induction algorithm based on an existing MDIE algorithm that intelligently confines and structures the solution space for efficient searching. We have developed a new way to evaluate the quality of a solution hypothesis that incorporates knowledge on system security. We developed an induction engine and synthesized many specifications of programs. Guided by a rich set of background knowledge, the induction engine generalizes program runs into Horn clauses that are guaranteed to explain all existing examples and reject all attack examples. In addition, the synthesized specifications are concised and are as good as those specified by human experts. As illustrated by the preliminary experiments, the generated specifications are effective in accurately detecting attacks, with a low false alarm rate. Our approach is effective in detecting Trojan Horses and program subversions such as buffer overflows or exploitations of race conditions without specific knowledge about the vulnerabilities. Nevertheless, our approach is not designed to detect probing attacks as well as denial-of-service attacks.

Our experiments and testing of our approach are preliminary and by no means comprehensive. Therefore, more

comprehensive testing, with more programs, extensive examples, and more attacks is needed in the future to truly evaluate the effectiveness of our approach. In addition, layered ILP learning techniques can also be applied to improve initial specifications as more examples are given. Furthermore, as the generated specifications are expressed in an analyzable form, more reasoning about the specifications with respect to their ability to protect a system can be done to choose the best set of specifications. For example, these questions can be studied: 1) is the specification tight enough to detect majority of attacks, 2) how to compare two specifications, and 3) given the specifications of many programs, what implication do they have at the high level. Also, we need to study how to generate specifications for some complex programs (e.g., Microsoft Word) that are designed to accomplish a wide variety of functions. In particular, the specifications should model the ordering of important operations of programs. Last, our approach can be applied to learn valid behavior of network protocols or services.

## Acknowledgement

I would like to thank the Defense Advanced Research Projects Agency for supporting this work. Also, I would like to thank Professor Steven Muggleton for allowing me to use the Progol tool. I would like to thank Timothy Redmond, Steven Cheung, Nancy Garman, and anonymous referees for their helpful comments on this paper. Last, I would like to thank my Heavenly Father for inspiring the idea and the continuous encouragement during the writing of this paper.

## References

[1] Computer Emergency Response Team (CERT), Pittsburgh, PA. *CA:97-19 lpr Buffer Overrun Vulnerability*, June 1997. Available from http://www.cert.org/advisories/CA-97.19.bsdlp.html.

[2] Computer Emergency Response Team (CERT), Pittsburgh, PA. *CA:98-09 Buffer Overflow in Some Implementations of IMAP Servers*, July 1998. Available from http://www.cert.org/advisories/CA-98.09.imapd.html.

[3] H Debar et al. Fixed vs. variable-length patterns for detecting suspicious process behavior. In *Proceedings of the 5th European Symposium on Research in Computer Security*, Louvain-la-Neuve, 1998.

[4] S. Forrest et al. A sense of self for unix processes. In *Proceedings of the 1996 Symposium on Security and Privacy*, pages 120–128, Oakland, CA, May 6-8 1996.

[5] T. Fraser, L. Badger, and Feldman M. Hardening COTS software with generic software wappers. *Proceedings of the 1999 Symposium on Security and Privacy*, 1999.

[6] A. Ghosh and A. Schwartzbard. A study in using neural networks for anomaly and misuse detection. In *8th Usenix Security Symposium*, Washington D.C., August 1999.

[7] K. Ilgun, R. Kermmerer, and P. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, 1995.

[8] H. Javitz and A. Valdez. The SRI IDES statistical anomaly detector. *Proceedings of the 1991 Symposium on Security and Privacy*, 1991.

[9] P. A. Karger. Limiting the damage potential of discretionary trojan horse. *Proceedings of the 1987 Symposium on Security and Privacy*, April 1987.

[10] 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.

[11] C. Ko, R. Ruschitzka, and K. Levitt. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach. In *Proceedings of the 1997 Symposium on Security and Privacy*, Oakland, CA, May 5-7, 1997.

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

[13] C. Landwehr et al. A taxonomy of computer program security flaws. *ACM Computing Surveys*, 26(3), September 1994.

[14] S. Lapointe and S. Matwin. Sub-unification: A tool for efficient induction of recursive programs. *Proceedings of the Ninth International Machine Learning Conference*, 1992.

[15] W. Lee and S Stolfo. Data mining approach for intrusion detection. *Proceedings of the 7th USENIX Security Symposium*, 1998.

[16] W. Lee, S. Stolfo, and Mok K. A data mining framework for building intrusion detection models. *Proceedings of the 1999 Symposium on Security and Privacy*, pages 120–132, May 9-12, 1999.

[17] U. Lindqvist and Porras P. Detecting computer and network misuse through the production-based expect system toolset (P-BEST). In *Proceedings of the 1999 Symposium on Security and Privacy*, pages 146–161, Oakland, CA, May 9-12, 2000.

[18] 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.

[19] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[20] S. Muggleton. Inverse entailment and progol. *New Generation Computing*, 13:245–286, 1995.

[21] S. Muggleton and C. Feng. Efficient induction of logic programs. *Proceedings of the First Conference on Algorithmic Learning Theory*, 1990.

[22] J. D. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[23] P. Sekar, T. Bowen, and Segal M. On preventing intrusions by process behavior monitoring. In *USENIX Intrusion Detection Workshop*, 1999.

[24] H. Teng, K. Chen, and S Lu. Adaptive real-time anomaly detection using inductively generated sequential patterns. In *Proceedings of the 1990 Symposium on Security and Privacy*, pages 278–284, Oakland, CA, May 7-9, 1990.

[25] H. Vaccaro and G. Liepins. Detection of anomalous computer session activity. In *Proceedings of the 1989 Symposium on Security and Privacy*, pages 280–289, Oakland, CA, May 1-3, 1989.

[26] R. Writh. Completing logic programs by inverse resolution. *EWSL-89*, pages 239–250, 1989.

## A. Definitions from logic

A variable is represented by an upper case letter followed by a string of lower case letters and digits. A function symbol is a lower case letter followed by a string of lower case letters and digits. A predicate symbol is a lower case letter followed by a string of lower case letters and digits. A variable is a term, and a function symbol immediately followed by a bracketed n-tuple of terms is a term. Thus $f(g(X), h)$ is a term when $f$, $g$ and $h$ are function symbols and $X$ is a variable. As in Prolog, integers, '[]' and '.' are function symbols and if t1, t2, ... are terms then '.' (t1, t2) can equivalently be denoted [t1, t2] and '.'(t1,'.'(t2,..'.'(tn,[])..)) can equivalently be denoted [t1, t2, .., tn]. A predicate symbol immediately followed by a bracketed n-tuple of terms is called an atomic formula, or atom. Every atom is a well-formed formula (wff). If $W$ and $W'$ are wffs then $\overline{W}$ (not $W$), $W \wedge W'$ (W and W'), $W \vee W'$ (W or W'), and $W \leftarrow W'$ ($W$ implied by $W'$) are wffs. If $v$ is a variable and $W$ is a $wff$ then $\forall v.W$ (for all v W) and $\exists v.W$ (there exists a $v$ such that $W$) are wffs. The wff $W$ is said to be function-free if and only if W contains no function symbols. Both $A$ and $\overline{A}$ are literals whenever $A$ is an atom. In this case $A$ is called a positive literal and $\overline{A}$ is called a negative literal. A set of literals is called a clause. The empty clause is represented by $\Box$. A clause represents the disjunction of its literals. Thus the clause $\{a_1, a_2, \cdots, \overline{a_i}, \overline{a_{i+1}}, \cdots, \overline{a_n}\}$ can be equivalently represented as $(a_1 \vee a_2 \vee ..\overline{a_i} \vee \overline{a_{i+1}} \vee \cdots \vee \overline{a_n})$ or $a_1; a_2; \cdots \leftarrow a_i, a_{i+1}, \cdots a_n$. All the variables in a clause are implicitly universally quantified. A Horn clause is a clause which contains at most one positive literal. A definite clause is a clause which contains exactly one positive literal. A positive literal in either a Horn clause or definite clause is called the head of the clause while the negative literals are collectively called the body of the clause. A set of clauses in which no pair of clauses share a common variable is called a clausal theory.

A clausal theory represents the conjunction of its clauses. Thus the clausal theory $\{C_1, C_2, \cdots, C_n\}$ can be equivalently represented as $(C_1 \vee C_2 \vee \cdots \vee C_n)$. Every clausal theory is said to be in clause-normal form. The process of replacing (existential) variables by constants is called skolemisation. The unique constants are called skolem constants. A set of Horn clauses is called a logic program. Let $E$ be a wff or a term. $vars(E)$ denotes the set of variables in $E$. $E$ is said to be ground if and only if $vars(E) = \emptyset$.

Let $\theta = \{v_1/t_1, \cdots, v_n/t_n\}$. $\theta$ is said to be a substitution when each $v_i$ is a variable and each $t_i$ is a term, and for no distinct $i$ and $j$ is $v_i$ the same as $v_j$. The set $\{v_1, \cdots, v_n\}$ is called the domain of $\theta$, or $dom(\theta)$, and $\{t_1, \cdots, t_n\}$ the range of $\theta$, or $rng(\theta)$. Lower-case Greek letters are used to denote substitutions. Let $E$ be a wff or a term and Let $\theta = \{v_1/t_1, \cdots, v_n/t_n\}$ be a substitution. The instantiation of $E$ by $\theta$, written $E\theta$, is formed by replacing every occurrence of $v_i$ in $E$ by $t_i$. Atom a $\theta$-subsumes atom $b$, or $a \preceq b$, if and only if there exists a substitution $\theta$ such that $a\theta = b$. Clause $C$ $\theta$-subsumes clause $D$, or $C \preceq D$, if and only if there exists a substitution $\theta$ such that $C\theta \subseteq D$. The Herbrand universe of the wff $W$ is the set of all ground terms composed of function symbols found in $W$. The Herbrand base of the wff $W$ is the set of all ground atoms composed of predicate and function symbols found in $W$. An interpretation is a total function from ground atoms to $\{T, F\}$. Interpretation $M$ is a model of wff $W$ if and only if $W$ is true in $M$. We say that $W$ semantically entails $W'$, or $W \models W'$ if and only if every model of $W$ is a model of $W'$.