

Formal Reasoning About Intrusion Detection Systems

Tao Song¹, Calvin Ko², Jim Alves-Foss³, Cui Zhang⁴, and Karl Levitt¹

¹ Computer Security Laboratory, University of California, Davis
{tsong, knlevitt}@ucdavis.edu

² NAI Labs, Network Associates Inc., Santa Clara, CA
calvin_ko@nai.com

³ Center for secure and dependable system, University of Idaho
jimaf@cs.uidaho.edu

⁴ Computer Science Department, California State University, Sacramento
zhangc@ecs.csus.edu

Abstract. We present a formal framework for the analysis of intrusion detection systems (IDS) that employ declarative rules for attack recognition, e.g. specification-based intrusion detection. Our approach allows reasoning about the effectiveness of an IDS. A formal framework is built with the theorem prover ACL2 to analyze and improve detection rules of IDSs. SHIM (System Health and Intrusion Monitoring) is used as an exemplary specification-based IDS to validate our approach. We have formalized all specifications of a host-based IDS in SHIM which together with a trusted file policy enabled us to reason about the soundness and completeness of the specifications by proving that the specifications satisfy the policy under various assumptions. These assumptions are properties of the system that are not checked by the IDS. Analysis of these assumptions shows the beneficial role of SHIM in improving the security of the system. The formal framework and analysis methodology will provide a scientific basis for one to argue that an IDS can detect known and unknown attacks by arguing that the IDS detects all attacks that would violate a policy.

Keywords: Intrusion detection, verification, formal method, security policy

1 Introduction

Intrusion detection is an effective technology to supplement traditional security mechanisms, such as access control, to improve the security of computer systems. To date, over 100 commercial and research products have been developed and deployed on operational computer systems and networks. While IDS can improve the security of a system, it is difficult to evaluate and predict the effectiveness of an IDS with respect to the primary objective users have for the deployment of such a system: the ability to detect large classes of attacks (including variants of known attacks and unknown attacks) with a low false alarm rate. In addition,

it is difficult to assess, in a scientific manner, the security posture of a system with an IDS deployed. So far, experimental evaluation and testing have been the only approaches that have been attempted. There is a critical need to establish a scientific foundation for evaluating and analyzing the effectiveness of IDSs.

This paper presents an approach to formal analysis of IDSs. Our approach is primarily applicable to IDSs that employ declarative rules for intrusion detection, including signature-based detection and specification-based detection [16] [18] [3] [8] [4] [5] [22]. The former matches the current system or network activities against a set of predefined attack signatures that represent known attacks and potential intrusive activities. The latter recognizes attacks as activities of critical objects that violate their specifications. Testing is currently being used to evaluate the effectiveness of the rules. Nevertheless, testing is usually performed according to the tester's understanding of known attacks. It is difficult to verify the effectiveness of an IDS in detecting unknown attacks.

Our approach is inspired by the significant body of formal methods research in designing and building trusted computer systems. Briefly, the process of designing and building a trusted system involves the development of a security model, which consists of a specification of a security policy (the security requirements or what is meant by security) and an abstract behavioral model of the system. Usually, the security policy can be stated as a mapping from system states to *authorized* (secure) and *unauthorized* (insecure) states [14] or as a property (often stated as an invariant) of the system (e.g., noninterference). The model is an abstraction of the actual system that provides a high level description of the major entities of the system and operations on those entities. There may be layers of abstractions within the model, each a refinement of the higher level abstraction. Given the security policy and model, one should be able to prove that the model satisfies the security policy, assuming some restrictions on the state transition functions (e.g., the classical Bell and LaPadula model).

Our framework consists of an abstract behavioral model, specifications of high-level security properties, and specifications of intrusion-detection rules. The abstract behavioral model captures the real behavior of the targeted system. In addition to common abstractions such as access control lists, processes, and files, the abstract behavioral model will capture the auditing capabilities of the targeted system (i.e., given an operation, it will be decided whether or not an audit event will be generated and what information about the operation will be visible). The specifications of intrusion-detection rules describe formally when the rules will produce IDS alarms given the sequence of audit events generated. Intrusion detection rules can be viewed as constraints on the audit trace of the system (e.g., the sequence of observable state changes).

We employ the formal framework to analyze the properties of SHIM, a specification-based IDS that focuses on the behaviors of privileged programs. In SHIM, specifications are developed to constrain the behaviors of privileged programs to the least privilege that is necessary for the functionality of the program.

ACL2 [15] is employed to describe an abstract system model that can be used as the basis for different IDSs. A hierarchical model is built to generalize

the verification of specifications. As an example, we formalize specifications of SHIM and a security policy (e.g, a trusted file access policy). And we prove that these specifications can satisfy the policy with various assumptions. Again, the assumptions represent activities that the SHIM IDS does not monitor, although it could if the IDS designer believes an attacker could cause the assumptions to be violated.

The rest of the paper is structured as follows: Section 2 introduces and analyzes intrusion detection rules, primarily used in a specification-based IDS such as SHIM. Section 3 describes a hierarchical framework of verification. Section 4 shows an example of our verification approach. We formalize specifications of SHIM and prove that these specifications together with assumptions satisfy trusted file access policies. In Section 5 we discuss our results and the limitations of the verification method we developed. We conclude and provide recommendations for future work in Section 6.

2 Analysis of Intrusion-Detection Rules

Development of correct intrusion-detection rules is a very difficult and error-prone task: it involves extensive knowledge engineering on attacks and most components of the system; it requires a deep and correct understanding of most of the components in a system and how they work together; it requires the rule developers to be cautious and careful to avoid mistakes and gaps in coverage. Often, crafting of intrusion-detection rules is performed by human security experts based on their knowledge and insights on attacks and security aspects of a system. Therefore, it is very difficult to assess whether a given set of intrusion-detection rules is correct (they detect the attacks). Furthermore, the complexity and subtlety of systems and attacks make it a challenging task to judge whether changes to the rules actually improve or degrade their efficacy with respect to their ability to detect new attacks.

We discuss the subtleties involved in writing valid behavior specifications for a program. Traditionally, in specification-based IDSs, a valid behavior specification for a program declares what operations and system calls are allowed for the program. Whether an operation is allowed or not depends on the attributes of the process and the object reference, and attributes of the system calls. In SHIM, a specification for a program is a list of rules describing all the operations valid for the program. For example, the following rule in the line printer daemon (lpd) specification allows the program to open any file in the */var/spool/hp* directory to read and write.

```
(open, $flag == O_RDWR&&InDir($F.path, "/var/spool/hp"))
```

The expression formally describes a set of valid operations: any *open()* system call with the flag argument equal to *O_RDWR* (open the file for read and write) with an absolute path name subordinate to the */var/spool/hp* directory.

One way to develop a specification for a program is to first identify what operations and accesses the program needs to support its functionality. Based on

an examination of the code or its behaviors, one writes rules in the specification to cover the valid operations of the program. The “draft” specification will be tested against the actual execution of the program. Often, the draft specification, when used to monitor the program execution, will produce false positives (i.e., valid operations performed by the program reported as erroneous because they are *not* included in the specification). Then, one augments the specification to include rules to cover these operations. In general, one needs to be very careful in writing the specification for a program to avoid errors.

For example, given the above rule, if `/var/spool/hp` somehow is writable by attackers, they can create a link from `/var/spool/hp/file` to the `/etc/passwd` file. A specification-based IDS with this rule in the specification of `lpd` will permit this operation and the attack will go undetected. Therefore, we augment the rule to check the number of links to the file and to generate a warning if the number of links to the file is greater than one, thus preventing this attack from using hard links. This also works for soft links in our system because the audit record for an `open()` operation will provide the absolute pathname of the file being opened, if the path is a symbolic link. Based on our experience, writing specifications for a program is subtle and tricky, thus demanding an approach to rule validation.

Little research has been done on analyzing intrusion-detection rules. Different approaches have been taken to specify and analyze the intrusion signatures and detection rules [12] [19] [17], primarily for signature-based IDSs. A declarative language, MuSigs, is proposed in [12] to describe the known attacks. Temporal logic formulas with variables are used to express specifications of attack scenarios [19]. Pouzol and Ducasse formally specified attack signatures and proved the soundness and completeness of their detection rules. In addition, data mining techniques and other AI techniques such as neural network are used to refine and improve intrusion signatures [6] [13] [21].

Our approach is different from these approaches in various ways. First of all, we developed a framework to evaluate detection rules of different IDSs. We formalized security-relevant entities of an UNIX-like system as well as access logs. Detection rules including intrusion signatures and specifications can be formalized and reasoned about in the framework.

Second, we proposed a method to verify security properties of IDSs together with assumptions, with respect to security policies. Security policies are always satisfied with sufficiently strong assumptions. So the key is to identify assumptions that are strong enough but not too strong. An attack can violate a security policy by breaking its assumptions. So it is possible to verify the improvement of security by proving the weakening of assumptions. For example, assuming a policy P is satisfied with assumption A and with the deployment of the mechanism m , and P is satisfied with assumption B where A implies B , then we can say m improves the security because attacks violating assumption B will also violate A , but attacks that violate assumption A may not violate B .

As our preliminary results, we have verified a significant property of specification-based IDSs: the capability to detect unknown attacks. In our verification, the specifications of SHIM satisfy a `passwd` file access policy with assumptions.

This means *any* attacks, including known attacks and unknown attacks, that violate the policy can be detected by SHIM.

3 Framework

We present a framework for analyzing detection rules in IDSs. Our goal is to answer the question of whether a given set of intrusion detection rules can satisfy the security requirements of the system. Security polices and properties of attacks are used to describe the security requirements of the system. The satisfaction of the security requirements determines whether violations of security policies or instances of attacks can be detected by the detection rules.

3.1 Hierarchical Framework of Verification

Figure 1 depicts the verification model, which consists of an abstract system model, an auditing model, detection rules, assumptions, and security requirements. The basis of the model is the abstract system model (S) in which security-critical entities of the system are formalized. The auditing model (L) is necessary for the model because almost all IDSs are based on the analysis of the audit trails from operating systems, applications and network components. Detection rules (R) vary dependent on the IDS. In SHIM, detection rules are specifications of normal behaviors of privileged programs. Security Requirements (SR) define properties that should be satisfied to guarantee the security of the system. Assumptions (H) are necessary for the verification. Security properties that we are not sure of and more important, properties that cannot be efficiently monitored will be declared as assumptions (e.g, kernel of the system is not subject to attack). Note that all of our assumptions could be checked by monitoring but at a substantial performance penalty to the IDS.

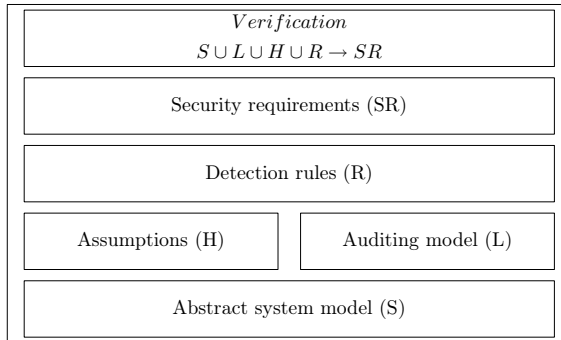


Fig. 1. Verification Hierachy

3.2 Formalization of the Model

In this section, we describe how to construct the components of the framework. We start with the abstract system model.

The abstract system model plays an important role in the framework. It provides a general basis to formalize security requirements, detection rules of IDSs and assumptions, and makes it possible to verify security properties of detection rules. To develop a simplified abstract model, we only formalize security-critical parts of UNIX-like systems. The model can be defined as a tuple (F, U, E, P, S) where F describes a file system, U shows user and group mechanisms, E corresponds to environment variables, P describes a list of processes, and S describes system call interfaces of the kernel. Our preliminary experiment focuses on the access control mechanism, so we define access permissions of file objects and privileges of subjects.

Because of the importance of the auditing component in IDSs, we formalize it separately from the abstract system model. We model the auditing component at the system call level. Assume Σ is a set of all system calls, let $B \subseteq \Sigma^*$ be all sequences of operations of a program A . A trace $b \in B$ presents a sequence of operations of A . For each operation of a program, we use a tuple (p, f, c, n) to indicate that a process p invokes a system call c on object f and assigns a new property n to f (e.g. a new owner for a file).

Detection rules vary according to different IDSs. In a specification-based IDS, detection rules are specifications which are used to describe normal behaviors of systems. Conversely, in a signature-based IDS, detection rules are signatures that identify different attacks. In this paper, we focus on the specification-based approach. Suppose the set of all possible behaviors of a program is defined as B , a specification $spec()$ can identify a set of valid behaviors VB where $VB \subseteq B$ and for any trace $b \in VB$, $spec(b) = true$.

Security requirements are used to describe properties necessary to satisfy the security of the system. There are basically two ways to present the security requirements: one is to define security policies, the other is to describe attack scenarios. Security policies can map the behavior of a system into two states: *authorized* or *unauthorized*. In this way, a security policy $security-policy()$ separates the behavior of a system into an authorized behavior set AB and an unauthorized behavior set UB where $AB \subseteq B$, $UB \subseteq B$. For any trace b , $b \in AB$ iff $security-policy(b) = "authorized"$. Attacks are behaviors that violate the security policy. We can use functions to define characterizations of attacks. An attack function $attack()$ can define a set of dangerous behaviors DB where $DB \subseteq B$ and for any trace b , $b \in DB$ iff $attack(b) = true$.

In the verification, we try to answer two questions: Can some security policies be satisfied by IDSs? And can some attacks remain undetected by IDSs? The first question can be formalized as follows. Given specification s and security policy p , is the valid behavior set VB that is defined by s a subset of the "authorized" behavior set AB defined by p . We describe this relation as $VB \subseteq AB$ or for any trace b , $spec(b) = true$ implies $security-policy(b) = "authorized"$. In some cases, assumptions are introduced in proving whether a security policy is satisfied by a specification. The verification can be described as: for any trace

$$b \in B, (assumption(b) = true) \wedge (spec(b) = true) \vdash (sr(b) = "authorized").$$

The second question can be formalized as follows. Given an attack ab or a set of attacks DB , is ab a member of the valid behavior set VB or does DB share elements with VB . It can be described as $ab \notin VB$ or $DB \cap VB = \phi$.

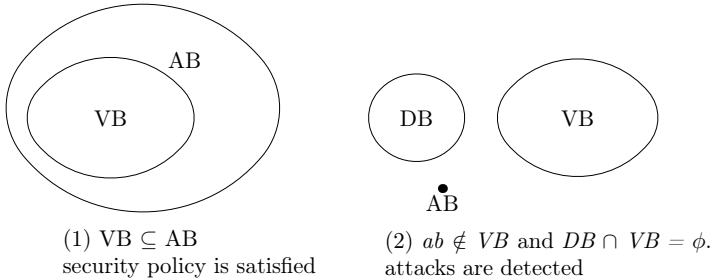


Fig. 2. Relationship among security policy, specifications and attacks

3.3 Mechanization of the Model

ACL2 is used in the mechanization of the framework. Structures and functions in ACL2 are used to formalize declarative components of the framework, including an abstract system model, audit data, detection rules of IDSs, assumptions, and security requirements. To perform the verifications we define the appropriate theorems in ACL2 and prove them using mathematical induction and the other proof mechanism of ACL2.

Introduction of ACL2. ACL2 is a significant extension of Nqthm [1], intended for large scale verification projects. The ACL2 system consists of a programming language based on Common Lisp, a logic of total recursive functions, and a mechanical theorem prover [15]. The syntax of ACL2 is that of Common Lisp. In addition, ACL2 supports the macro facility of Common Lisp. The following data types are axiomatized: rational and complex numbers, character objects, strings, symbols and lists. Common Lisp functions on these data types are axiomatized or defined as functions or macros in ACL2 [9]. Several functions that are used in our verification are listed in table 1. The ACL2 logic is a first order logic of recursive functions providing mathematical induction on the ordinals up to ϵ_0 and two extension principles: one for recursive definition and one for the constrained introduction of new function symbols. Each preserves the consistency of the extended logic.

The ACL2 theorem prover has powerful and effective heuristics controlling the application of the following proof techniques: preprocessing including tautology checking using ordered binary decision diagrams (OBDDs) under user direction; simplification by primitive type checking and rewriting; destructor elimination; cross-fertilization using equivalence reasoning; generalization; elimination of irrelevant hypotheses; and mathematical induction.

Table 1. Important functions of ACL2

<i>Functions</i>	<i>Descriptions</i>
Nil	The empty list or False in Boolean contexts
T	True
(if x y z)	If x not equal to nil, return y, otherwise z
(equal x y)	If x and y have the same value, return t, otherwise nil
(and x y)	“And” operation in Boolean contexts
(car l)	First element of list l
(cdr l)	All but the first element of list l
(consp x l)	Add x onto the front of list l
(implies x y)	If x is nil or y is t, return t; otherwise return f

Abstract System Model. The abstract system model is formalized as a structure *sys* in ACL2. Security-critical components are defined as fields of the structure. For each field, *asserts* are defined to check the integrity of values of the fields. A predicate *sys-p* is defined to recognize values that have the required structural form and whose fields satisfy the assertions. The predicate *weak-sys-p* is defined to recognize values that have the required structural form, but does not require the field assertions to be satisfied. Functions are defined to get, put or check values from specific fields. In our verification, we can use instances of the structure to tell whether a statement is true in a system with specific settings. On the other hand, the system model can appear in a theorem without specific values to indicate a general condition in which the statement is held.

```
(defstructure sys
  (proglisp (:assert (and (not (endp proglisp))(proglisp proglisp))))
  ;list of programs, e.g. privileged programs
  (calllist (:assert (and (not (endp calllist))(calllist calllist))))
  ;list of system calls, e.g. open, read, write etc.
  (filelist (:assert (and (not (endp filelist))(filelist filelist))))
  ;list of files, e.g. /etc/passwd file
  (userlist (:assert (and (not (endp userlist))(userlist userlist))))
  ;list of system users
  (envlist (:assert (and (not (endp envlist))(envlist envlist))))
  ;list of environment variables, e.g. home directories in a UNIX system
```

Audit Trail. The auditing capability of a system is formalized as a list of audit records and an audit record is formalized as a structure *logrec* in ACL2. We reference Sun Solaris BSM audit subsystem and simplify the audit record structure to four fields: process, file object, system call and new properties to the file object.

```
(defstructure logrec
  (pobj (:assert (and (consp pObj)(proc-obj-p pObj))))
  ;object of the process
  (fobj (:assert (and (consp fobj)(file-obj-p fobj))))
  ;object of the target file
```

```
(callobj (:assert (and (consp callobj)(syscall-obj-p callobj))))
;object of the system call
(newattrobj (:assert (newattr-obj-p newattrobj))))
;new properties of the target file
```

Security Requirements. In our verification, different classes of attacks and security policies are formalized to analyze detection rules of IDSs.

There are two ways to verify whether an attack can be detected by a specific IDS. The first method is to formalize possible audit trails, which include the attack scenarios, and then analyze the audit data according to the specification of the program for the violation. Such verification can be used to prove the capabilities of the specifications to detect *known* attacks. A more general way is to describe the security property that will be violated by the attacks instead of particular audit trails. Then we develop a proof based on the property that the formalized specifications will always result in the system being monitored for that property. For example, in an ftp-write attack, an attacker takes advantage of a normal anonymous *ftp* misconfiguration. If the *ftp* directory and its subdirectories are owned by the *ftp* account or in the same group as the *ftp* account, the attacker will be able to add files (such as the *.rhosts* file) and eventually gain local access to the system.

Security policies are also formalized to allow reasoning about the security properties of specifications. Trusted file access policies are security policies that we developed to keep trusted files from unauthorized access. In UNIX systems, a discretionary access control(DAC) mechanism defines whether a subject can access an object or not depending on the privilege of the subject and the access permission of the object. Some files are intended to be accessed by specific users or using specific programs. For example, the *passwd* file of a UNIX system should be editable by root using any program or by an ordinary user using the *passwd* program. Thus, file access policies are defined in our format as: (*trusted file, authorized user, program, access*) where *trusted file* is the file to be protected, *authorized user* defines the user that can access the file with any programs and *program* defines the program that can be used by other users to access the file.

As an example, the *passwd* file access policy is defined as: (*/etc/passwd, root, passwd, (open-wr,create, chmod, chown, rename)*). This policy is used in the verifications of the next section. The policy is formalized as a function in ACL2.

Assumptions. Our verification methodology rests on assumptions. A system specification will have assumptions on how the system and programs behave. The specifications cannot be declared as complete before all assumptions of the specifications are identified. In some cases, once the assumptions are declared as required by the verification approach, an IDS does not have to monitor the properties asserted in these assumptions. There are two different kinds of assumptions in our verification: general assumptions of the system and specific assumptions of verifications.

System assumptions are very important although they are not formalized in our verification. Some general assumptions of the system model are listed as follows:

– **System kernel is not vulnerable to attack**

The security of system kernel is beyond the scope of this paper. We simply assume that system kernel is not vulnerable to attack.

– **DAC mechanism of the system is correctly implemented**

Access control is a concern of our verification. So correct implementation of the DAC mechanism is an assumption for the security of the system. If the access control mechanism is not well implemented and a user can access objects for which he is not authorized, it is impossible to protect these objects by only constraining behaviors of privileged programs.

– **Completeness on log data**

As a hypothesis of the IDS, audit logs should record the trace of attacks so that analysis of the audit logs may detect such attacks. Therefore, log data should include all important operations with their correct sequence. If an attacker can successfully eliminate his traces before an IDS analyzes them, it is impossible to detect this activity by an IDS.

The specific assumption of verification will be discussed in section 4, in the context of the verifications of specific IDS and security policies.

4 Specification and Verification of SHIM

We formalized the specifications of a specification-based IDS, SHIM, and analyzed them according to different security policies and attacks.

4.1 Introduction of SHIM

SHIM is a specification-based IDS. Specification-based IDSs are based on the creation of specifications that describe desired functionality for security-critical entities [7] [8] [20] [10] [23]. The security specifications in SHIM mainly focus on the valid operations of a UNIX privileged program. Privileged programs are analyzed because of their significant impact on system security. The effective user of a privileged program has root privileges, and attacks against a privileged program often exploit the privilege to access security-critical objects that are not intended to be accessed by the victim program. For example, in a *ftp* buffer overflow attack, an attack can invoke a shell with root privilege and use it to access any files of the system [2]; that the attack is a buffer overflow is incidental to the specification.

During program operation, the system accesses associated with the operation of a program are recorded in audit logs and matched against the specifications by SHIM. Mismatches are reported and almost always indicate an attack. Theoretically, SHIM is capable of detecting unknown attacks or variants of known attacks, and a report is issued as soon as a specification violation occurs. If the program was compromised by an attack (e.g. buffer overflow) and attempted to invoke any system calls that violated the specifications, an alert would be raised.

4.2 Formalization of Specifications

In SHIM, a language, Parallel Environment Grammar (PE grammar), is developed to define specifications that describe all valid operations of a program. The language permits the parameterization of the language syntax and environment variables that aid in parsing efficiency. PE grammar can be used to specify the valid execution traces of programs. The specification of *ftp* daemon is listed to show how the language works:

```

SE: <prog>
    <prog> -> <validop> *;
    <validop>
    -> (OPEN_RD, WorldReadable($F.mode))
;the program can read a file that is world-readable
    | (OPEN_RD, CreatedByProc($P.pid, &$F))
;the process can read a file that is created by itself
    | (OPEN_RD, $F.ouid == $S.uid)
;the process can read a file whose owner is the current user
    | (OPEN_WR, CreatedByProc($P.pid, &$F))
    | (OPEN_WR, $F.path == "/var/log/wtmp")
;the process can write to a file a specific path
    | (OPEN_WR, $F.path == "/var/log/xferlog")
    | (OPEN_RW, $F.path == "/var/run/ftp.pids-all")
    | (open, $F.path == "/dev/null")
    | (unlink, CreatedByProc($P.pid, &$F))
    | (CHMOD, CreatedByProc($P.pid, &$F))
    | (CHOWN, CreatedByProc($P.pid, &$F))
    | (fork||vfork)
    | (OPEN_RD, InDir($F.path, getHomeDir($S.uid)))
;the process can read a file situated on a specific directory
    | (OPEN_WR, InDir($F.path, getHomeDir($S.uid)))
    | (read, IsSocket($F.mode) && $K.lport == 21)
;the process can get information from specific port
    | (write, IsSocket($F.mode) && $K.lport == 21)
    | (CREAT, InDir($F.path, getHomeDir($S.uid)))
    | (EXEC, $path == "/bin/tar" || $path == "/bin/compress" ||
        $path == "/bin/ls" || $path == "/bin/gzip") ;END;

```

In this specification, valid operations are defined with a term *validop*. Eighteen valid operations are included in this specification and each valid operation is a function of system calls and environment variables. For example the operation (*OPEN_RD, WorldReadable(\$F.mode)*) means this programs can open a file in read mode when the file is readable by all users. PE grammar is capable of defining a multi-state specification, but in this specification, only one state is used, namely the state associated with the invocation of the program. This specification shows a balance between expressiveness and detection efficiency.

In our verification, a function is defined to check whether audit trails of a *ftp* daemon process violates the specification. The function accepts an audit trail as a parameter. If any operation of the audit trail violates the specification, the function will return *false* otherwise *true*. All valid operations are defined with

two functions: operation and property restriction. The operation function defines the operation on an object and the property restriction function defines the condition in which the operation will be performed. For example, the valid operation (*OPEN_RD*, *WorldReadable(\$F.mode)*) can be formalized as (*and (operate 'openrd logrec) (WorldReadable (logrec-fobj logrec))*). In the definition, the function *operate* gets the correct operation, and function *WorldReadable* determines whether the permission of the file is *world readable*.

4.3 Verifications

Our verification focuses on the effectiveness of specifications of SHIM in satisfying security requirements, including attacks and security policies. We attempt to address the issue whether, with specific detection rules, SHIM can detect specific attacks or detect attacks that cause specific security policies to be validated.

Detection of Attacks. Attacks are modeled as sequences of operations. We use two ways to describe attack scenarios: an audit trail that contains an attack or a characterization of attacks. According to the characteristic of specification-based IDS, SHIM cannot detect any attacks that do not change the behavior of victim programs. So here we introduce an assumption about attack:

Assumption: an attack cannot cause any damage without changing the behavior of a victim program.

Then we can claim that SHIM is capable of detecting attacks before or at least “as” they cause damage to the system.

For known attacks, we can always simulate their audit trails. Considering the buffer overflow attack against *wuftpd* 2.4.2-beta-18. The program can be compromised by overflowing a buffer in *strcat()*. We simulated an audit trail which invoked a shell after penetration of *strcat()*. Then we used the specification of *ftp* daemon to check this audit trail. A violation is reported and this indicates that the attack can be detected by SHIM. A further analysis shows that the violation is revealed by the audit record that indicates invocation of the shell. The call to the library function *strcat()* does not reveal a violation. This result proves that the specification of SHIM can detect this buffer overflow attack if the attacker tries to invoke a shell after penetration.

For unknown attacks, we can consider a group of similar attacks that invoke shells after they successfully compromise an *ftp* daemon program. We indicate a theorem which shows any audit trail with an operation invoking a shell will be detected by the specification of *ftp* daemon. The theorem is defined as :

```
(defthm attack-ftp
  (implies
    (member 'exec "/bin/bash" log sys)
    ;any operation invoking a shell
    (not(spec_ftp sys log nil))
    ;violate the specification of ftp program
  ))
```

This theorem demonstrates an important feature of SHIM: detection of unknown attacks. Any unknown attacks against the *ftp* daemon will be detected if an operation of invoking shells is observed. The proof of the theorem is straightforward because */bin/bash* is not a valid path for the *exec* system call in the specification.

Proving a Specification Satisfies a Security Policy. In this section, we carry out a verification that indicates the trusted file access policy is satisfied by specifications of SHIM with some assumptions.

We use the *passwd* file access policy as an example in this verification. As we defined in section 3, the *passwd* file access policy defines how the *passwd* file should be accessed. According to the DAC mechanism of UNIX systems, any user without root privilege cannot modify the *passwd* file except through a privileged program. If the DAC mechanism is well implemented, no user except root can use unprivileged programs, like *vi*, to change the *passwd* file. So, we only focus on the behavior of privileged programs in verifying that the system satisfies the policy.

Given an audit trail of a specific privileged program, we try to prove that any audit trail that passes the check of the specification will satisfy the *passwd* file access policy. We use *ftp* daemon as an example to show how it works.

The proof is defined as a theorem which is indicated below. The formalization of the abstract system model *sys* and audit data *log* are used in this theorem. We may notice that some assumptions are added to complete the proof. Two important verification assumptions are made in this proof.

The first assumption is about the access permissions of the *passwd* file. The *passwd* file can only be protected when it has proper access permission. If the *passwd* file is set world writeable, the integrity of the file cannot be protected because any user has the privilege to change the *passwd* file.

The other assumption is concerned with the setting of the home directory of the user who attempts to access the *passwd* file. A user can access the *passwd* file if his home directory is set as */etc*. The reason is that the specification of the *ftp* daemon allows the user to access the files under his home directory. In fact, this assumption can be guaranteed by deploying some configuration checking tools such as KUANG [24]. But in SHIM, such a property of the system is not monitored. With these assumptions, any audit data that passes the specification check of the *ftp* daemon will satisfy the *passwd* file access policy.

```
(defthm passwd-ftp
  (implies
    (and(not (member '(/ etc passwd) created))
      ;passwd file was not created by the process
      (consp log)(consp sys)(logp log)(consp created)(sys-p sys)
      ;format checking
      (validuser sys log)
      ;assumption: no invalid user as determined by the audit data
      (passwdsafe log))
```

```

;assumption: passwd file has proper permissions
(homedirsafe sys)
;assumption: home diretory settings are correct
(spec.ftpd sys log created))
;the specification is not violated by any operations
(not-access-passwd log)
;then, thepasswd access policy is satisfied
))

```

Using a similar method, we have proved that the specification of the *lpd* program satisfies the *passwd* file access policy with the assumption that the environment variable *printerspool* is not misconfigured. Changes to environment variables are not monitored by SHIM, so this assumption clearly covers a property that SHIM cannot check.

Composition of Specifications Satisfies the Policy. A further question is whether the composition of different specifications will satisfy the *passwd* file access policy. In this section, we consider concurrent execution of different privileged programs. We use *ftp* daemon and *lpd* as examples to show that the composition of specifications of these two programs satisfying the policy.

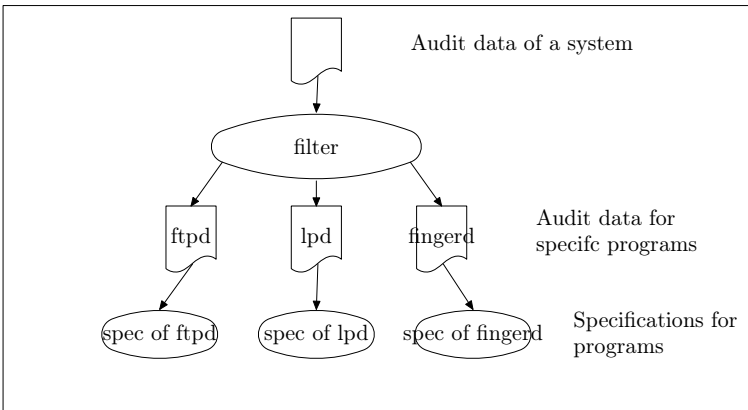


Fig. 3. Mechanism of SHIM to filter concurrent execution audit log

As shown in figure 3, in SHIM, the audit filter is used to separate the audit trail of individual programs from the audit data of the system. We simulate the filter using a function $filter(prog, log)$ in ACL2, where *prog* is the name of the program and *log* is the audit trail of the system. A question is whether the filter will change the security property of the audit trail. If the filter maps the data trail of a few privileged programs to the audit trail of each program and all subsets of the data trail satisfy the *passwd* file access policy, does this means the audit trail satisfies the policy?

Suppose *log* is the audit trail of *ftpd* and *lpd*. We have proved that if the audit trail of *ftpd*, *filter('ftpd, log)*, can pass the specification check of *ftpd* and if the audit trail of *lpd*, *filter('lpd, log)*, can pass the specification check of *lpd*, the audit trails of *ftpd* and *lpd* satisfy the *passwd* file access policy.

```
(defthm passwd-specs
  (implies
    (not (member '/ etc passwd) created))
    ;passwd file was not created by the process
    (implies
      (and (logp log) (consp log) (consp sys) (sys-p sys) (procsafe log)
        ;format checking
        (passwdsafe log) (homedirsafe sys) (validuser sys log)
        ;assumptions for ftpd program
        (validenv sys 'printerspool)
        ;assumptions for lpd program
        (spec.ftpd sys (filter 'ftpd log) created)
        ;the specification of ftpd is not violated by any operations
        (spec.lpd sys (filter 'lpd log) created))
      ;the specification of lpd is not violated by any operations
      (not-access-passwd log)))
  )
;then, the passwd access policy is satisfied
```

We notice that the assumptions in this verification are the union of assumptions in these two verifications that have been proved previously. All theorems appearing in this section have been proved automatically by the ACL2 theorem prover using rewriting and mathematical induction.

4.4 Performance

We measure the performance of ACL2 in carrying out the proofs described above. We formalize the abstract system model, detection rules and security policies with 174 functions and 13 data structures. We defined and proved 56 lemmas and theorems to complete the verification. It took three weeks to develop all the functions and complete the verification. On a 450MHZ Pentium machine with 384 MB memory, ACL2 spent 15.21 minutes to complete the verification. This suggests that using ACL2 to formalize and verify security properties of IDSs is a feasible approach.

5 Discussion

The assumptions of the verification process may, in some cases, be guaranteed through other tools. In the exemplary verification, we introduced assumptions needed to satisfy the *passwd* file access policy. These assumptions relate to access permissions of target objects (e.g., the *passwd* file cannot be world-writable), proper configurations (e.g., home directories of users cannot be */etc/*), etc. SHIM

is not capable of monitoring these static properties of the system. But these assumptions can be checked by deploying other security tools such as Tripwire [11] [25] .

In our verification, the soundness and completeness of detection rules of IDSs are not yet completely proved. If the soundness of the detection rules could be verified, the false positive rate of IDSs would theoretically be proved to be zero. In SHIM, the detection rules are specifications of the system. It is feasible, in principle, to prove the soundness of specifications by comparing the specifications with the implementation of the system. Automatic generation and verification of specifications can be achieved by associating formal methods with code analysis. As an extreme but practically useless example, it is easy to prove a specification rejecting all possible behaviors is sound. Considering the huge false negative rate, this specification is clearly not an acceptable solution even with a zero false positive rate. If the completeness of detection rules can be verified, the false negative rate of IDSs will be zero. Similarly, a specification accepting all behaviors can be proved complete.

The ACL2 theorem prover is used in our verification. It provides reliable verification by using well-accepted deduction rules, e.g., mathematical induction. By describing properties of attacks, we can prove that all the attacks (including known attacks and unknown attacks) with specific operations (e.g. invoking shell) can be detected by SHIM. This verifies an important and often-cited claim of specification-based intrusion detection: detection of unknown attacks. There are a few limitations about mechanical theorem provers. First, proof creation of almost any practical properties correct in theorem provers is not totally automatic. Although theorem provers help find missing steps in proofs, it is still impossible for a theorem prover to create proofs without human interaction. Second, even if a proposition cannot be proved by a theorem prover, it doesn't indicate the proposition is wrong. Also it is difficult to find a counter-example to show conditions under which a property is incorrect.

6 Conclusions and Future Work

In this paper, we present a formal framework that can be used to evaluate detection rules of IDSs. ACL2 is used to formalize declarative components of the framework and to carry out the verifications. An abstract system model is built as the basis for verifications. Trusted file access policies are developed to define authorized access on security-critical objects of a system. We also report on our experience with a preliminary implementation of this framework in reasoning about security properties of SHIM, a specification-based IDS. We have formalized all detection rules of SHIM, specifications for privileged programs, and addressed two important issues about SHIM (and specification-based IDS, in general): what attacks can be detected by SHIM and whether abstract security policies can be satisfied by SHIM. An important feature of SHIM, its ability to detect unknown attacks, is actually verified by specifying properties of attacks.

Potential future work includes analyzing misuse detection systems (i.e. signature-based IDSs) and network IDSs; generating specification using code analysis;

verifying soundness of specifications; and developing realistic security policies for network protocols.

Acknowledgements

We thank Steven Cheung, Jeff Rowe, Poornima Balasubramanyam, Tye Stallard and Marcus Tylutki for helpful discussion on security policy, verification and intrusion detection. We are grateful to Patty Graves for her valuable help. This material is based upon work supported by the National Science Foundation under Grant No 0341734. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. R. S. Boyer and J S. Moore, *A computational logic*, Academic Press, New York, 1979.
2. Cert coordination center, advisory ca-1999-03, <http://www.cert.org/advisories/CA-99-03.html>
3. C.C.W. Ko , “Execution Monitoring of Security-Critical Programs in a Distributed System: A Specification-Based Approach”, Ph.D. Thesis, August 1996
4. C. Ko, “Logic induction of valid behavior specifications for intrusion detection”, Proc. of IEEE Symposium on Security and Privacy 2000
5. C. Ko, J. Rowe, P. Brutch, K. Levitt, “System Health and Intrusion Monitoring Using a hierarchy of Constraints”, Proceeding of 4th International Symposium, RAID, 2001
6. Anup K. Ghosh and Aaron Schwartzbard , “A Study in Using Neural Networks for Anomaly and Misuse Detection”, Proc. of USENIX Security Symposium, 1999
7. C. Ko, G. Fink, and K. Levitt. “Automated detection of vulnerabilities in privileged programs by execution monitoring”. In Proceedings of the Tenth Computer Security Applications Conference, pages 134-144, Orlando, FL, Dec. 1994. IEEE Computer Society Press.
8. C. Ko, M. Ruschitzka, and K. Levitt, “Execution Monitoring of Security-critical Programs in Distributed Systems: A Specification-based Approach,” Proc. of the 1997 IEEE Symposium on Security and Privacy, Oakland, California, May 1997, pp. 134-144.
9. M. Kaufmann, P. Manolios, J S. Moore, “Computer-Aided Reasoning : An Approach”, Kluwer Academic Publishers, 2000
10. C. Ko, J. Rowe, P. Brutch, K. Levitt, “System Health and Intrusion Monitoring Using a hierarchy of Constraints,” Proceeding of 4th International Symposium, RAID, 2001
11. G. Kim, E. H. Spafford, “The design of a system integrity monitor: Tripwire,” Technical report CSD-TR-93-071, Purdue University, November 1993
12. Jia-Ling Lin; Wang, X.S.; Jajodia, S., “Abstraction-based misuse detection: high-level specifications and adaptable strategies”, Proc. of IEEE Computer Security Foundations Workshop, 2002.

13. Wenke Lee; Stolfo, S.J.; Mok, K.W., "A data mining framework for building intrusion detection models", Proc. of IEEE Symposium on Security and Privacy, 1999
14. Matthew A. Bishop, Computer Security: Art and Science, Addison Wesley Longman 2002
15. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, "Computer-Aided Reasoning: An Approach", Kluwer Academic Publishers, June, 2000
16. M. Roesch, "Snort: Lightweight Intrusion Detection for Networks", Proc. of USENIX LISA '99, Seattle, Washington, November 1999, pp. 229-238.
17. J.P. Pouzol, M. Ducasse, "Formal specification of intrusion signatures and detection rules", Proc. of IEEE Computer Security Foundations Workshop, 2002.
18. P.A. Porras and P.G. Neumann, "EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances", Proc. of the 20th National Information Systems Security Conference, Baltimore, Maryland, October 1997, pp. 353-365.
19. Roger, M.; Goubault-Larrecq, J., "Log auditing through model-checking", Page(s): 220-234, proc.of 14th IEEE Computer Security Foundations Workshop, 2001.
20. R. Sekar, Yong Cai, Mark Segal, "A Specification-Based Approach for Building Survivable Systems," Proc. 21st NIST-NCSC National Information Systems Security Conference 1998
21. Schultz, M.G.; Eskin, E.; Zadok, F.; Stolfo, S.J., "Data mining methods for detection of new malicious executables", Proc. of IEEE Symposium on Security and Privacy, 2001
22. P. Uppuluri, R. Sekar, "Experiences with Specification-based intrusion detection," Proc of Recent Advances in Intrusion detection, 2001
23. David Wagner, Drew Dean: Intrusion Detection via Static Analysis. IEEE Symposium on Security and Privacy 2001.
24. D. Zerkle, K. Levitt, "NetKuang-A Multi-host Configuration Vulnerability Checker," Proc of Sixth USENIX Security Symposium, 1996
25. A. Mounji, B. Le Charlier, "Continuous Assessment of a Unix Configuration: Integrating Intrusion Detection and Configuration Analysis," Proc.of the ISOC' 97 Symposium on Network and Distributed System Security. 1997.