Experiences with Specification-based Intrusion Detection[⋆]

P. Uppuluri and R. Sekar

Department of Computer Science
SUNY at Stony Brook, NY 11794.
E-mail: {prem,sekar}@cs.sunysb.edu

Abstract. Specification-based intrusion detection, where manually specified program behavioral specifications are used as a basis to detect attacks, have been proposed as a promising alternative that combine the strengths of misuse detection (accurate detection of known attacks) and anomaly detection (ability to detect novel attacks). However, the question of whether this promise can be realized in practice has remained open. We answer this question in this paper, based on our experience in building a specification-based intrusion detection system and experimenting with it. Our experiments included the 1999 DARPA/AFRL online evaluation, as well as experiments conducted using 1999 DARPA/ Lincoln Labs offline evaluation data. These experiments show that an effective specification-based IDS can be developed with modest efforts. They also show that the specification-based techniques live up to their promise of detecting known as well as unknown attacks, while maintaining a very low rate of false positives.

1 Introduction

With the growing number of attacks on network infrastructures, the need for techniques to detect and prevent attacks is increasing. Intrusion detection refers to a broad range of techniques that defend against malicious attacks. Intrusion detection techniques generally fall into one of the following categories: *misuse detection, anomaly detection* and *specification-based detection*. The main advantage of misuse detection is that it can accurately detect known attacks, while its drawback is the inability to detect previously unseen attacks. Anomaly detection, on the other hand, is capable of detecting novel attacks, but suffers from a high rate of false alarms . This occurs primarily because previously unseen (yet legitimate) system behaviors are also recognized as anomalies, and hence flagged as potential intrusions.

Specification-based techniques have been proposed as a promising alternative that combine the strengths of misuse and anomaly detection. In this approach, manually developed specifications are used to characterize legitimate program behaviors. As this method is based on legitimate behaviors, it does not generate false alarms when unusual (but legitimate) program behaviors are encountered. Thus, its false positive rate can be comparable to that of misuse detection. Since it detects attacks as deviations from legitimate behaviors, it has the potential to detect previously unknown attacks.

[⋆] This research is supported in part by Defense Advanced Research Agency (DARPA) under contract number F30602-97-C-0244 and by an NSF grant CCR-0098154.

Although the promise of specification-based approach has been argued for some time, the question of whether these benefits can be realized in practice has remained open. Some of the questions that arise in this context are:

– How much effort is required to develop program behavioral specifications? How do these efforts compare with that required for training anomaly detection systems?
– How effective is the approach in detecting novel attacks? Are there classes of attacks that can be detected by specification-based techniques that cannot be detected by anomaly detection or vice-versa?
– Can it achieve false alarm rates that are comparable to misuse detection?

We provide answers to the above questions in this paper, based on our experience in building a specification-based intrusion detection system and experimenting with it. Our experiments included (a) the 1999 DARPA/AFRL online evaluation, (b) the 1999 DARPA/Lincoln Labs offline evaluation data, and (c) several locally-developed experiments. These experiments show that an effective IDS can be developed with modest specification development efforts, of the order of tens of hours for a many security-critical programs on Solaris. Moreover, these efforts need to be undertaken just once for each operating system – further customization on the basis of individual hosts or installations seems to be unnecessary. This contrasts with anomaly detection systems that typically need training/tuning for each host/system installation.

Our experiments show that the specification-based techniques live up to their promise of detecting known as well as unknown attacks, while maintaining a low rate of false positives. In particular, we were able to detect all the attacks without any false positives in the online evaluation. In the offline evaluation, we could detect 80% of the attacks using specifications that characterized legitimate program behaviors. The remaining 20% of the attacks did not cause program (or system) misbehavior, and it is debatable whether they constitute attacks at all. Nevertheless, we were able to easily model these attacks as misuses. Augmented with these misuse specifications, our system was able to achieve 100% detection with no false alarms in the offline evaluation as well.

The rest of this paper is organized as follows. We begin with an overview of our specification-based intrusion detection technique in Section 2. Following this, we develop a methodology for specification development in Section 3. In Section 4, we report our experimental results. Section 5 further analyzes our results. Finally, we summarize our conclusions in Section 6.

2 Background

In [15, 1] we described our approach to specification based intrusion detection. Central to our approach is the observation that intrusions manifest observable events that deviate from the norm. We extend the current state of the art in event-based intrusion detection by developing a domain-specific language called *behavioral monitoring specification language (BMSL)* [15]. BMSL enables concise specifications of event based security-relevant properties. These properties can capture either *normal behavior* of programs and systems, or *misuse behaviors* associated with known exploitations.

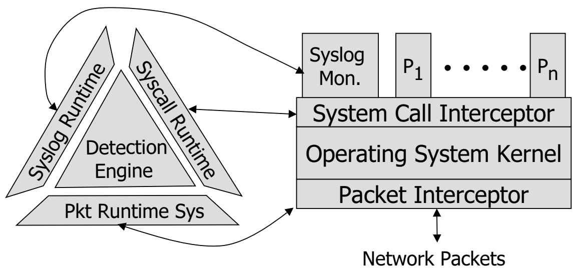

Fig. 1. Runtime view of the system. $P_1, ..., P_n$ are processes; Syslog Mon is a system log watcher.

In our approach, we compile BMSL specifications into efficient *detection engines (DE)* [15]. For network packets, BMSL specifications are based on packet contents, and for system calls, BMSL specifications are based on system calls and the values of system call arguments. In both contexts, BMSL supports a rich set of language constructs that allow reasoning about not only singular events, but also temporally related event sequences. In this paper, we are concerned only with system call events.

The overall architecture of our intrusion detection/response system is shown in Figure 1. For a given event stream such as packets or system calls, an interceptor component placed in the stream provides efficient interception of raw events. The interceptors deliver raw event streams to a runtime environment (RTE) associated with each stream. Typically, a single detection engine monitors each defended process.

Note that in the figure, we show a runtime environment for system calls, one for network packets, and another for log file entries. The log file entries are made by a syslog monitor program (syslog mon). Although our specification based approach is capable of processing event data from such diverse sources, the discussion in the rest of this paper pertains only to system call events.

2.1 Behavior Modelling Specification Language (BMSL).

BMSL is a core language that we have designed for developing security-relevant behavior models. Specifications in BMSL consist of rules of the form $pat \rightarrow action$, where pat is a pattern on event sequences, otherwise known as *histories*; and $action$ specifies the responses to be launched when the observed history *satisfies* pat. Observe that we typically initiate responses when *abnormal behaviors* are witnessed; thus, pat components of rules usually correspond to negations of properties of normal event histories.

System Call Events. In the context of system calls, we define two events corresponding to each system call. The first event, which is given the same name as that of the system call, corresponds to the invocation of the system call. The arguments to the event are exactly the set of arguments to the system call. The second event corresponds to the return from the system call, and its name is obtained by suffixing $\_exit$ to the name of the system call. The arguments to the exit event include all of the arguments to the system call, plus another argument that captures the return value from the system call.

Note that the system call entry and exit events always occur in pairs. To minimize clutter, we use the convention that a system call entry (or exit) event can stand for both the entry and exit events.

The pattern language: Regular Expressions over Events (REE). As a language that captures properties of (event) sequences, our pattern language draws on familiar concepts from regular expressions. It extends these concepts from sequences of characters to events with arguments.

The simplest REE pattern captures single-event occurrences (or non-occurrences):

– *occurrence of single event:* $e(x_1, ..., x_n)|cond$ is satisfied by the (single-event) history $e(v_1, ..., v_n)$ if $cond$ evaluates to $true$ when variables $x_1, ..., x_n$ are replaced by the values $v_1, ..., v_n$.
– *occurrence of one of many events:* $E_1||E_2|| \cdots ||E_n$, where each E_i captures an occurrence of a single event, is satisfied by a history H if any one of $E_1, ..., E_n$ is satisfied by H.
– *event nonoccurrence:* $!(E_1||E_2|| \cdots ||E_n)$ is satisfied by H if none of $E_1, ..., E_n$ are satisfied by H. Here again, $E_1, ..., E_n$ denote an event pattern of the form $e(x_1, ..., x_n)|cond$.

These primitive event patterns can be combined using *temporal* operators to capture properties of event sequences as follows:

– *sequencing:* $pat_1; pat_2$ is satisfied by H_1H_2 if H_1 and H_2 satisfy pat_1 and pat_2 respectively.
– *alternation:* $pat_1 \;||\; pat_2$ is satisfied by H if H satisfies pat_1 or pat_2.
– *repetition:* $pat*$ is satisfied by $H_1H_2 \cdots H_n$ where each H_i satisfies pat.

When a variable occurs multiple times within a pattern, an event history will satisfy the pattern only if the history instantiates all occurrences of the variable with the same value. For instance, the pattern $e_1(x); e_2(x)$ will not be satisfied by the event history $e_1(a)e_2(b)$, but will be satisfied by $e_1(a)e_2(a)$. This ability of REE's to remember event argument values (for later comparison with arguments of subsequent events) makes them much more expressive than regular languages. Their expressive power is in fact comparable to that of attribute grammars rather than regular grammars.

Based on the notion of satisfaction defined above, we define the notion of an event history H *matching* a pattern p: we say that H matches p if any suffix of H matches p. If we need to *anchor* the match (i.e., constrain the match to occur from the beginning of the history H), then we introduce a special event $begin$ at the beginning of the pattern. All histories begin implicitly with the event $begin$.

Actions. As is well known in the context of finite-state automata, some pattern-matching operations are conveniently stated using regular expressions, while there are others that are more amenable to a state-machine formulation. The same holds true in the case of our behavior specifications as well, so our language permits explicit introduction of state variables. These variables may be assigned in the action component, or tested in the pattern component of a rule.

In addition to variable assignments, the action component of a rule may invoke *external functions* that are provided by the RTE. These functions can be used to invoke response actions that are aimed at preventing and/or containing damage due to an attack. External functions can also be used to express computations that are not easily described in BMSL, but can be coded in a general-purpose programming language such as C++.

2.2 Example Specifications

As a first example, we illustrate a pattern that restricts a process from making a certain set of system calls:

```
execve||connect||chmod||chown||creat||sendto||mkdir → term()
```

Note that we have omitted system call arguments, as they are not used elsewhere in the specification. (Sometimes, we will use "..." to denote unused event arguments.) Also note that in this example, an external function `term()` is being used to terminate a process that makes one of these disallowed system calls. Other response actions, such as aborting the disallowed system call, or logging a message, are also possible.

As a second example, consider the following specification that restricts the files that a process may access for reading or writing:

```
admFiles = { "/etc/utmp","/etc/passwd"}
open(f, mode)|(realpath(f)∉admFiles || mode≠O_RDONLY) → term()
```

Here, the auxiliary function `realpath` is used to convert a filename into a canonical form that does not make use of the special characters "`.`" and "`..`," or symbolic links.

To illustrate the use of sequencing operators, consider the following pattern that asserts that a program never opens and closes a file without reading or writing into it. Before defining the pattern, we define abstract events that denote the occurrence of one of many events. Occurrence of an abstract event in a pattern is replaced by its definition, after substitution of parameter names, and renaming of variables that occur only on the right-hand side of the abstract event definition so that the names are unique.

```
    openExit(fd) ::= open_exit(..., fd) || creat_exit(..., fd)
    rwOp(fd) ::= read(fd) || readdir(fd) || write(fd)
    openExit(fd);(!rwOp(fd))*;close(fd) →···
```

Although regular expressions are not expressive enough to capture balanced parenthesis, note that the presence of variables in REE enables us capture the `close` system call matching an `open`. Note that such matching open and close operations cannot be captured even by context-free grammars.

3 Specification development

Our approach for specification development consists of the following steps:

1. *Developing generic specifications:* Generic specification is a specification that is parameterized with respect to system calls as well as their arguments. By appropriately instantiating these parameters, such specifications can be used to monitor almost any program.
2. *Securing program groups:* In this step, we strengthen generic specifications for classes of programs that have similar security implications, e.g., all setuid programs, all daemons, etc.

3. *Developing application-specific specifications:* Some programs, such as servers that can be accessed from remote sites, will likely be attack targets more frequently than other programs. For such applications, we can increase the effectiveness of the intrusion detection system by developing more precise specifications.
4. *Customizing specifications for an operating system/site:* Specifications obtained from the previous steps are customized to accommodate variations in operating systems, and also any site specific security policies.
5. *Adding misuse specifications:* It is likely that the detection of some attacks requires knowledge about attacker behavior. In such cases, a pure specification-based approach (wherein only legitimate program behaviors are specified) may not be sufficient. Therefore, one can augment such pure specifications with *misuse patterns* that can capture features of specific attacks.

We note that in progressing from step 1 to step 5, we are developing more and more precise specifications of the behavior of a program. Less precise specifications mean lower specification development effort, but can negatively impact the effectiveness of the approach in terms of missed attacks as well as increased false alarms. More precise specifications increase the effectiveness of the system at the cost of increased specification development effort. We discuss these tradeoffs further in the next section. Below, we proceed to describe the five steps mentioned above in greater detail.

3.1 Step 1: Developing generic specifications

The first step in the development of generic specifications is to group system calls of similar functionality. This allows us to consider a smaller number of system call groups (few tens) while writing specifications, rather than dealing with a few hundred system calls. Grouping of system calls also helps in developing *portable* specifications, because the role of each of these groups tends to be constant across different operating systems, even though the system calls within the groups may be different.

For the purpose of developing generic specifications, we have identified 23 groups, further organized into 9 categories. Below, we provide a selected subset of these 23 groups, drawn from five categories. The complete listing can be found in [3].

– *File Access Operations*
  - $WriteOperations(path)$: This group includes system calls such as $open$ (with open for write option) and $creat$ that change the contents of a file named $path$.
  - $ReadOperations(path)$: This group includes system calls which perform read operations on a file specified in the parameter $path$.
  - $FileAttributeChangeOps(path)$: This group includes system calls such as $chgrp$ and $chmod$.
  - $FileAttributeCheckOps(path)$: System calls which check file attributes (e.g., permissions and modification times) are included in this group.

– *Process Operations*
  - $ProcessCreation$: This group includes system calls $fork$ and $execve$ which create or execute a new process.
  - $ProcessInterference$: These are system calls such as $kill$ which enable one process to alter the course of execution of another process.

```
1. WriteOperations($path$)|$path \in fileWriteList \rightarrow$ term()
2. ReadOperations($path$)|$path \in fileReadList \rightarrow$ term()
3. (FileAttributeChangeOps($path$)|$path \in fileAttributeChangeList$
    $\rightarrow$ term()
4. execve($prog$)|$prog \in fileExecList \rightarrow$ term()
5. ProcessInterference $\rightarrow$ term()
6. (ConnectCalls || AcceptCalls) $\rightarrow$ term() /* Remove for clients/servers */
7. (Privileged ) $\rightarrow$ term()
8. SetResourceAttributes $\rightarrow$ term()
```

Fig. 2. Sample generic specifications

- *Network Calls*
  - $ConnectCalls$: These are system calls made by a client to connect to a server.
  - $AcceptCalls$: These system calls are made by a server to accept a connection.
- *Setting resource attributes*: These are system calls which set or change resource attributes such as scheduling algorithms and scheduling parameters.
- *Privileged Calls*: Includes system calls such as $mount$ and $reboot$ that require root privileges to run.

Based on the above classification, we have developed a generic specification as shown in Figure 2. It is parameterized with respect to (a) the definition of which system calls are in the each of the system call groups mentioned above, and (b) lists of files that may be read or modified by a program. Of these, the definition of (a) will be provided in step 4. File lists are given certain default values in the generic specification, and will be further refined in steps 2 through 4.

3.2 Step 2: Securing program groups

Certain groups of programs have similar security properties. We can exploit these commonalities to develop specifications that can be reused among programs within the group. For instance all *setuid to root* programs should be restricted from opening or changing attributes on arbitrary files, and executing arbitrary programs. This can be achieved by customizing the file lists used in the generic specification. (The customized lists obtained in this manner for the entire group of setuid programs may have to be further refined for individual programs in Step 4.)

Most setuid programs expect arguments (provided either through the command line or through environment variables) of a bounded size. Hence we add the following rule to the specification for setuid programs:

```
execve(path,argv,envp )|checkSize(path,argv,envp,max)  → term()
```

The function `checkSize` checks if the `path` argument size is less than the system-defined constant `PATH_MAX`, and that each of the command-line and environment arguments are of size less than `max`. The parameter `max` may have to be customized for individual programs.

As another example, consider server programs that perform user-authentication. In such programs, a large number of failed login attempts within a short period of time is considered abnormal behavior. The discussion below is set in the context of the telnet server, but is applicable to other programs such as FTP as well. Note that programs such as `telnet` restrict users to a small number of login attempts (usually three), after which the server terminates. Thus, in order to make a large number of login attempts, a user would have to cause many instances of `telnetd` to be spawned in succession, with each instance restricting the user to a small number of attempts. Therefore, we develop the following rule for the `inetd` superserver, which counts the number of `telnetd` instances spawned by `inetd` within a short period of time.

```
   execve(prog)|prog = "in.telnetd" && tooManyAttempts() →
         log("too many failed logins")
```

We are currently adding better language support for capturing the notion of "too many occurrences within a short time," based on our earlier work in network intrusion detection [14]. Meanwhile, for the purpose of the experiments described in this paper, we have relied on an external function `tooManyAttempts`, whose implementation was provided in C, to capture this notion.

Note that the above rule simply counts the number of `telnetd` instances spawned, without any consideration of whether these instances led to a failed login or not. We therefore add the following rule to the `telnetd` specification, which resets all of the counters involved whenever a successful login is completed, which is signified by the execution of a `setuid` or `setreuid` system call.

```
   (setuid || setreuid) → resetAttempt()
```

Currently, our approach for customizing specifications is based on manual editing. Language support for such customization is a topic of current research.

3.3 Step 3: Developing application-specific specifications

We can further refine specifications for individual applications in order to provide better security, especially for network servers. Figure 3 shows a partial specification for an FTP server (`ftpd`). (See [15] for a more complete model that contains five additional rules.) Our starting point for this model was the documentation provided in FTP manual pages. From there, we identified the principal security-related system calls made by `ftpd` and plausible sequencing orders for their execution to obtain the model. For `ftpd`, completion of user login is signified by the first `setreuid` system call executed, and this userid is stored in `loggedUser` by the first rule. Similarly, the name of a client host is extracted from a return argument when the `getpeername` system call exits, and remembered in `clientIP` by the second rule. The third and fourth rules restrict the behavior of the ftp server based on whether user login has been completed or not. Rules 5 and 7 capture properties that must be generally adhered to by servers such as `ftpd`: they must permanently reset their userid using `setuid` (rule 5), and any file opened with superuser privileges must be explicitly or implicitly closed (rule 7) before executing other programs[1]. Rule 6 captures the property that when `ftpd` resets its userid to 0 (i.e., superuser in UNIX) it must do so for executing a small set of operations that

[1] These two rules could have been made part of the specification for the group of server programs such as telnet and ftp, but we did not do so for the experiments reported in this paper.

```
1.  (!setreuid)*;setreuid(r,e) → loggedUser := e
2.  (!getpeername)*;getpeername_exit(fd,sa,l) →
    clientIP := getIPAddress(sa)
/*  Access limited to certain system calls before user login.  */
3.  (!setreuid())*;ftpInitBadCall() → term()
/*  Access limited to certain other set of system calls after user login is completed.  */
4.  setreuid();any()*;ftpAccessBadCall() → term()
/*  Userid must be set to that of the logged in user before exec.  */
5.  (!setuid(loggedUser))*;execve → term()
/*  Resetting userid to 0 is permitted only for executing a small subset of system calls.  */
6.  setreuid(r,0);ftpPrivilegedCalls*;!(setreuid(r1,loggedUser)
    || setuid(loggedUser)||ftpPrivCalls) → term()
/*  A file opened with root privilege is explicitly closed, or has close-on-exec flag set.  */
7.  (open_exit(f, fl, md, fd)|geteuid()=0);(!close(fd))*;
    (execve|!closeOnExec(fd)) → term()
8.  connect(s, sa)|((getIPAddress(sa) != clientIP)
    && (getPort(sa) ∉ ftpAccessedSvcs)) → term()
```

Fig. 3. A (partial) specification for `ftpd`.

deal with binding to low-numbered ports. Finally, rule 8 restricts `ftpd` from connecting to arbitrary hosts.

This specification was developed using the principle of least privilege, without paying any attention to previously known attacks on `ftpd`. It is interesting to note that in spite of this, the partial model would in fact detect known attacks such as FTP bounce (CERT advisory CA-97.27), FTP Trojan Horse (CERT advisory CA-94.07) and race conditions in signal handling (CERT advisory CA-97.16). None of these attacks would be detected by the generic specification alone.

3.4 Step 4: Customizing specifications for an OS/Site

In this step, we do the following:

- Define system calls that fall under each group determined in Step 1. We will also define system calls that are included in any groups that may have been defined explicitly for the purpose of writing an application-specific specification, e.g., the $ftpPrivilegedCalls$ in the FTP specifications.
- Refine the lists of files that are used in generic or application-specific specifications. We perform this refinement by running the program and logging the list of files it accesses using the specifications. For instance, the programs on Solaris write into the `/devices/pseudo/pts*` files, whereas on Linux they do not. So `/devices/pseudo/pts*` is removed from the $fileWriteList$ mentioned in Step 1.
- Add site specific security policies. These are policies which restrict certain policies that are normally allowed by an application. For instance, users of anonymous `ftp` may not be allowed to write into the `ftp` directory.

3.5 Step 5: Adding Misuse rules to specifications

Certain attacks can be detected only based on knowledge about specific attack behavior. For instance, even a small number of login failures using the user name $guest$ usually indicates an attack. In contrast, small number of failures are not that unusual with other user names, as users frequently mistype or forget their passwords. Knowing that an attacker is likely to try cracking the guest account is thus crucial for detecting such attacks. Note that this knowledge pertains to attacker behavior, rather than the behavior of the program itself. As such, it is hard to develop (normal) behavior specifications that can accurately detect this attack. However, it is possible to encode this knowledge about attacker behavior as *misuse rules* that are designed to capture typical actions of an attacker. With misuse rules in place, we can detect such attacks accurately.

We note that our pattern language BMSL is well-suited for capturing normal as well as misuse rules. However, we note that reliance on such misuse rules should be minimized, as they need to be updated when new attack behaviors are discovered.

4 Experimental Results

In this section we discuss our experimental results obtained on the 1999 DARPA/Lincoln Laboratories offline evaluation [10]. We also briefly summarize the results we obtained in the 1999 DARPA/AFRL online evaluation.

Lincoln Labs recorded program behavior data using the Basic Security Module (BSM) of Solaris, which produces an audit log containing data in BSM format. We used BSM audit records that corresponded to system calls. Each such record provides system call information (such as its name, a subset of arguments and return value), as well as process-related information (such as process id, userid and groupid of the process). In addition, specific records contain source and destination IP-address and port information. The BSM audit records are well documented in [11].

In order to use the BSM data, we had to develop a runtime environment that would parse the BSM audit logs, and feed the events into the detection engine component shown in Figure 1. Development of such an RTE had not been undertaken until the end of year 2000. Therefore, our experiments were conducted after Lincoln Labs had published the results of 1999 evaluation, including information about the attacks contained in the evaluation data. This factor enabled us to focus our initial specification development effort on programs involved in attacks, as opposed to all programs. Specifically, we developed specifications for 13 programs: `in.ftpd`, `in.telnetd`, `eject`, `ffb-config`, `fdformat`, `ps`, `cat`, `cp`, `su`, `netstat`, `mail`, `crontab` and `login`. All other programs were monitored using generic specifications with default values for various file lists. Note, however, that several programs such as `init`, and several daemons such as `crond` were not audited, and hence could not be monitored. We were planning to expand the number of program-specific specifications to include more setuid programs, but found that the above list already included over 50% of the setuid programs present in the evaluation data, with or without attacks on them. The complete list of specifications developed for this experiment can be found at [4].

4.1 Specification development

The specifications were developed using the five step process of Section 3.

- Step 1: We developed generic specifications shown in Figure 2.
- Step 2: We refined the generic specification for two important program groups, namely, setuid programs and daemon processes.
- Step 3: We developed application-specific specifications for FTP and telnet servers. (Actually, our work was one of porting specifications for these programs that were originally written for a Linux environment.)
- Step 4: We added site-specific security policies to these specifications. One such policy was stated explicitly by Lincoln Labs: that files in the directory `secret` should not be accessed using `cp` and `cat`. There were also some implicit policies, e.g., *anonymous ftp users* should not write files into the directory `~ftp`, nor should they read or write files in hidden directories. In step 4, we also populated various file list parameters of the generic and application-specific specifications for the setuid programs, daemons and network servers. Finally, we defined a default value for the file lists so that the generic specification can be used with all programs that did not have a customized specification.
- Step 5:

4.2 Results

According to the result data provided by Lincoln Labs, 11 distinct attacks were repeated to generate a total of 28 attack instances. It turned out that two of these attacks were not actually present in the data due to corruption of the BSM files during the times when these two attack instances were carried out. One other instance (namely, the set up phase of the *HTTPtunnel* attack) was not visible in the BSM data. Offsetting these, there were five additional instances of the attack phase of the *HTTPtunnel* attack. Thus, the total number of attacks present in the data was 30.

Table 4 summarizes the 11 attack types and 30 instances of these attacks that were present in the data. The table specifies the number of instances of the attack in the data and the number of instances detected using the specifications developed till step 4 and step 5. Most attacks were discovered at the end of step 4. A few attacks could not be legitimately characterized as deviations from normal behavior. In these cases, we developed misuse specifications to detect the attacks (step 5 of the specification development effort). At this point, all of the attacks could be detected. The last row of the table specifies the total number of instance of all the attacks and the percentage detected after step 4 and step 5.

The percentages shown in the figure refer to the percentage of attack types detected, rather than attack instances. For every attack type, all instances were identical, so it makes more sense to count the fraction of attack types detected rather than attack instances.

Most of the attacks were relatively simple, and were detected without any problem by our system. This was in spite of the fact that little effort had been put into developing program-specific behavioral specifications. In the next subsection, we discuss some of the noteworthy attacks in the data, and comment on how they were detected by our system.

Attack Name	Number of instances	Percent instances detected	
		after Step 4	after Step 5
Fdformat	3	100%	100%
Ffbconfig	1	100%	100%
Eject	1	100%	100%
Secret	4	100%	100%
Ps	4	100%	100%
Ftpwrite	2	100%	100%
Warez master	1	100%	100%
Warez client	2	100%	100%
Guess telnet	3	100%	100%
HTTP tunnel	7	0%	100%
Guest	2	0%	100%
Total	30	82%	100%

Fig. 4. Attacks detected in BSM data

4.3 Description of selected attacks and their detection

Buffer overflows There were four buffer overflow attacks on *eject*, *fdformat*, *ffbconfig* and *ps* programs. The attacks on the first three programs exploited a buffer overflow condition to execute a shell with root privileges. The specification we used to monitor *setuid to root* programs could easily detect these attacks by detecting oversized arguments and the execution of a shell. In addition, a violation was reported when the buffer overflow attack resulted in execution of an unexpected program (shell).

The ps attack was significantly more complex than the other three buffer overflow attacks. For one thing, it used a buffer overflow in the static area, rather than the more common stack buffer overflow. Thus, the argument size rule does not detect this attack. Second, it used a `chmod` system call to effect damage, rather than the more common `execve` of a shell program. Nevertheless, `chmod` operation is itself unusual, and it is not permitted by our generic specification (except on certain files). Thus, detection of this attack was straightforward.

Site specific property A policy indicating that all files in the directory `/export/home/secret/` are secret and cannot be moved out of the directory by programs such as cp and cat was published by Lincoln labs. By adding this policy in step 4, we were able to detect violations of this policy and hence flag these attacks.

Ftp-write attack The *ftp-write* attack is a remote to local user attack that takes advantage of a common anonymous ftp misconfiguration. The ∼`ftp` directory and its subdirectories should not be owned by the ftp account or be in the same group as the ftp account. If any of these directories are owned by ftp or are in the same group as the ftp account and are not write protected, an intruder will be able to add files (such as a `.rhosts` file) and eventually gain local access to the system. We could detect this attack easily due to the site-specific policy that no file could be written in ∼`ftp` directory.

Warez attacks During this attack, a *warezmaster* logs into an anonymous FTP site and creates a file or a hidden directory. In *warezclient* attack, the file previously downloaded by the *warezmaster* is uploaded. This attack could be easily captured by the specifications which encoded the site-specific policy of disallowing any writes to the $\sim$`ftp` directory.

Guess Telnet This is similar to a dictionary attack, where the attacker makes repeated attempts to login by guessing the password. The behavioral specification for `telnetd`, had a specification which limited the number of login attempts and flagged an attack, when the number exceeded 3.

Guest The *guest* attack is not amenable to detection using a specification of normal behavior because of the fact that the detection of the attack requires the knowledge that attackers commonly try the $user/password$ pairs of `guest/guest` and `guest/anonymous`. The attacks simulated by Lincoln Labs involve only two such attempts, with the second attempt ending in a successful login. We therefore encoded this knowledge about attacker behavior in our specifications, and were then able to detect all instances of the *guest* attack. Note that a related attack, namely the *guess telnet* attack, can be detected by a positive specification, as discussed earlier. To detect the guest attack, we used the system call *chdir* to directory `/export/home/guest` as the marker to indicate a guest login. After 2 or more login attempts a `chdir` to the file `/export/home/guest` was flagged as an attack.

HTTPtunnel The *HTTP tunnel* attack is the most questionable of all attacks. This attack involves a legitimate system user installing a program that periodically connects to a remote attacker site, using a connection initiated by the UNIX `cron` daemon and sends confidential information to the site. As per the configuration of the victim system, the user (i.e., attacker) in this case was allowed to add `cron` jobs, and was also allowed to connect to remote sites. Since legitimate user may have used these facilities to periodically download legitimate information such as news and stocks, and there is no easy way to rule out these as the reasons for installation of the *HTTP tunnel* software, we could not develop a normal behavioral specification. Therefore, we developed a misuse specification that captures the periodic invocation of a program by `cron` such that this invocation results in a connection to a remote server.

Periodic invocation of a program by `cron` can be identified by the marker system call `fork`. However, since `crond` was not audited, the `fork` system call made by `crond` was not present in the BSM data. But all subsequent system calls made by the child process of `crond` were present. However, there is no idntification of where this child process came from, which makes it impossible in our setting, to associate any specifications with the child process. To work around this problem, we observed that the children of `crond` execute a sequence of system calls not seen elsewhere. We used this sequence as a marker, and made it a part of the generic specification that would be used to monitor all processes. Our misuse specification for HTTPtunnel is shown in Figure 5. Function `raiseFlag` sets a global flag to 1, as soon as the cron process starts. The `cron` process then forks and execs. In rule 2, `setFlag` sets the global flag to 2 after the `fork()`. When the `cron` job initiates a connection, rule 3 flags an attack.

The set up phase of the HTTPtunnel attack was not visible in the BSM data, since some of the data that is crucial for the set up phase was not recorded in the BSM logs.

In particular, the setup phase involves installing a new crontab file that contains an entry for period invocation of the HTTPtunnel attack, but the file contents are not recorded in the BSM audit logs.

```
/*  To run a cron job, crond forks children which immediately execute the following system call sequence.  */
1.  open(a1,a2,a3);ioctl(a4,a5,a6);close(a6);setgroups(a7,a8);
    setgroups(a7,a8);open(a9,a10,a11);ioctl(a12,a13,a4);
    close(a6);setgroups(a14,a15);open(a17,a18,a19);
    fcntl(a20,a21,a22); close(a6);close(a6);creat(a23,a24);
    fcntl(a20,a21,a22);close(a6);fcntl(a20,a21,a22);close(a6);
    close(a6);chdir(a30);close(a6);close(a6);
    execve(a25,a26,a27) → setFlag(1);
/*  The child then forks and execve's the process it has to run.  */
2. fork()|checkFlag()=1 → setFlag(2);
/*  If the process spawned by crond connects to a remote host we log an attack.  */
3. connect(a1,a2,a3)|checkFlag()=2 → log(''illegal connect'')
```

Fig. 5. Specifications to detect connection of a cron job to a remote host.

False Alarms No false alarms were reported by our method on the BSM data. This result strengthens our claim that specification-based approaches produce very few false positives. It also shows that the test data did not particularly stress our IDS. When tested against a more sophisticated or comprehensive attack data, our system would likely lead to a higher rate of false positives.

5 Discussion

5.1 Effectiveness of our approach

Without including the misuse specifications, we could detect 80% of the attacks with 0% false positives. After including the misuse specifications we could detect 100% of all attacks with 0% false positives. While this result would surely not hold against a more sophisticated attack data, it does demonstrate the high "signal-to-noise ratio" that can be achieved by our specification-based approach.

It might be argued that our results would not have been as good if we had participated in the evaluation, since we would not have known all of the attack instances present in the data. We point out, however, that we achieved similar results in the 1999 online evaluation, where we entered the evaluation with no knowledge about the attacks that they were going to subject our system to. We did not even have the ability to rerun the attacks if we were to encounter any difficulties or bugs in our IDS. But this result is not without its own caveats: the online evaluation involved only three distinct attack types. The attack types, however, were significantly more complex than the offline evaluation, as they were designed to stress our IDS.

5.2 Development effort

We found that very general specifications seem to be sufficient for detecting a majority of the attacks. In fact, many attacks produce violations of multiple rules, thus suggesting that the attacks may be detectable with even less effort in specification development than we expended. The initial development of generic specifications, which was undertaken for the AFRL online evaluation, was about 12 man hours and it resulted in 9 rules and 12 lines of code in BMSL. Behavioral specifications for applications such as `ftpd` took a longer time. In particular for `ftpd` we developed 14 rules [15] which were 18 lines of code in BMSL. However as we mentioned earlier, the sophistication of this specification was hardly necessary for detecting the attacks in the data. Thus, typical application-specific specifications need not be that exhaustive.

As described earlier in Section 4.1, the specifications for *setuid to root* programs required a few modifications to the generic specification. It took us about 6 man hours to refine and customize the specifications. Here too, a higher degree of effort is needed in the beginning, i.e., the development of specifications for the first one or two setuid programs. Most of the specifications developed were not dependent on any particular implementation of UNIX operating system, so the effort required for subsequent programs was only in making minor changes in the file and process lists. These changes took as low as 10 minutes per program. Projecting from these results we can state that customizing specifications for new programs will not involve much effort. In fact, noting that there are only several tens of daemon programs and setuid programs, and only a handful of server programs, the total amount of effort required for specification development is in the range of a few tens of man hours, which we consider to be quite acceptable.

Another way to look at the development effort is to compare the specification development effort with that needed for training anomaly detection systems. We note that production of training data (by a human) for anomaly detection systems is resource-intensive. It is very important to ensure that the training data encompasses almost all legitimate behaviors, which is difficult, and typically requires manual intervention on a program by program basis. Moreover, it may be necessary to cross check and manually verify that the anomaly detector did not end up learning artifacts in the training data, as opposed to learning features that were pertinent to the normal operation of a program. In addition, trained data is usually specific to a particular installation of an operating system. Considering these factors, we believe that it would be hard to do better than few tens of man hours for training with respect to several tens of programs.

5.3 Portability

We had initially developed our specifications for Linux operating system. However to test attacks in BSM data we had to port them to Solaris. We found that except for Step 5, the specifications did not require any modifications. The effort required for step 5 was also modest: of the order of 10 to 30 minutes per program. Here again, the effort required for the first one or two programs was higher, of the order of one or two hours. Subsequent programs took much lesser effort, of the order of 10 minutes per program.

We point out that accurate measurement of development effort is very hard for the following reasons. First, the development effort will be reduced if we followed the methodology outlined earlier in this paper. However, this methodology evolved as a result of our experiences, and hence, our specification development efforts were higher towards the early part of our experiments. Second, our ability to develop and debug specifications improves gradually with experience. Thus, a task that took us days to complete initially would now take hours.

5.4 Novel attacks and false positive rates

As shown in Table 4, over 80% of the attacks could be detected without encoding any attack-specific information into the specifications. Viewed differently, all of these attacks are "unknown" to our system. The relatively high detection rate supports our claim that specification-based approaches are good at detecting unknown attacks. The low rate of false alarms supports our claim that specification based approaches can detect novel attacks without having to sacrifice on the false alarm front.

Perhaps the notion of "unknown" attacks is better defined in the context of the online evaluation, where we *really* had no knowledge of the attacks that our system was going to be subjected to. In the online evaluation, we did not encode any misuse rules, and hence all detections were based on normal behavior specifications. In this context, our system detected all of the three attacks launched during this evaluation, once again with zero false alarms.

We emphasize once again that neither the offline nor the online evaluation stressed the capabilities of our IDS. If they had, we would have an experimental basis to compare the ability of the specification-based approach for detecting new attacks as compared with anomaly detection approaches. Nevertheless, we wish to point out some differences, based on the way the two approaches operate.

– A learning-based approach learns a subset of all legitimate behaviors of a program – this subset corresponds to those behaviors that were actually exhibited by a program during training. To eliminate false-positives, we must ensure that all legitimate behaviors are exercised during training. In practice, this turns out to be the problem of ensuring that all program paths are exercised. As is well-known in software testing, achieving 100% coverage of all program paths is essentially impossible due to the astronomical number of paths in a program. As such, some rarely traversed paths are never learnt, leading to a fair number of false positives.
  A specification is aimed at capuring a superset of possible behaviors of a program. As such, there is no inherent reason to have false positives, except due to specification errors.
– The superset-subset argument above also indicates that there may be some behaviors that are outside of the legitimate behaviors of a program, but within the superset captured by a specification. Exploits that involve such behaviors will be missed by a specification-based approach, while they can be captured by a learning based approach.
– There exist a significant number of attacks that can detected by specification based approaches that are difficult or impossible to detect using anomaly detection. This occurs for two main reasons:

- For certain latent errors, the learning-based approach essentially learns that execution of a security-relevant error is normal. Viewed alternatively, exploitation of the error to inflict intrusion does not lead to any change in the behavior of the program containing the error. This class of errors is fairly extensive in scope, and includes the following:
  - ∗ race conditions in file access
  - ∗ opening files without appropriate checks
  - ∗ leaving temporary files containing critical information

  A specification can constrain the program execution so that it does not take actions that leave opportunities for exploitation. We routinely add specifications that protect against the above kinds of errors to most programs. (We note, however, that none of these types of attacks were present in the 1999 offline evaluation, although they are reported frequently enough by CERT [2].)
- Existing anomaly detection approaches based on system calls generally ignore system call argument values. This makes it difficult to detect many attacks where normal behavior is distinguishable from attacks only in terms of such argument values. Examples of such attacks include:
  - ∗ executing /bin/sh instead of ls
  - ∗ writing to /etc/passwd instead of some other file

Based on our analysis of the attacks reported by CERT [2], over 20% of the attacks reported in the period 1993 to 1998 fall in this category of attacks not detectable by anomaly detectors.

– Anomaly detectors typically need to be trained/tuned for each host/site. In contrast, our specifications seem to dependent only on an operating system distribution, rarely requiring any customization for individual hosts/sites.

6 Conclusions

1. *Our specification-based approach is very effective.* It was able to detect 80% of the attacks with only positive behavior specifications. Since these specifications did not incorporate any knowledge about attacks or attacker behavior, this experiment demonstrates the effectiveness of the specification-based approaches against unknown attacks.
2. *Very general specifications seem to be sufficient for detecting a majority of the attacks.* In fact, many attacks produce violations of multiple rules, thus suggesting that the attacks may be detectable with even less effort in specification development than what we put into it.
3. *Users of our approach can trade increased specification development efforts against decreased probability of successful attacks.* This is borne by the fact that by investing some additional effort, we were able to detect all of the attacks included in the BSM data.
4. *By combining specifications of legitimate program behaviors with some misuse specifications, 100% detection rate could be achieved with 0% false positive rates.* While this result would likely not hold against a more sophisticated attack data, it does demonstrate the high “signal-to-noise ratio” that can be achieved by our specification-based approach.

5. *Our specification-based approach is typically able to provide additional attack related information that can be used to pinpoint the attack*, e.g., execution of a disallowed program, access to certain privileged operations, access to disallowed files, etc.

References

1. T. Bowen et al, Building Survivable Systems: An Integrated Approach Based on Intrusion Detection and Confinement, DISCEX 2000.
2. CERT Coordination Center Advisories 1988–1998, `http://www.cert.org/advisories/index.html`.
3. Classification of system calls using security based criteria, `http://seclab.cs.sunysb.edu/ prem/classifbody.html`.
4. Specifications used for 1999 DARPA offline evaluation, `http://seclab.cs.sunysb.edu/˜prem/specs.html`.
5. S. Forrest, S. Hofmeyr and A. Somayaji, Computer Immunology, Comm. of ACM 40(10), 1997.
6. T. Fraser, L. Badger, M. Feldman, Hardening COTS software with Generic Software Wrappers, IEEE Symposium on Security and Privacy, 1999.
7. A.K. Ghosh, A. Schwartzbard and M. Schatz, Learning Program Behavior Profiles for Intrusion Detection, 1st USENIX Workshop on Intrusion Detection and Network Monitoring, 1999.
8. A.K. Ghosh, A. Schwartzbard, A study in using Neural Networks for Anamoly and Misuse Detection, USENIX Security Symposium, 1999.
9. C. Ko, Execution Monitoring of Security-Critical Programs in a Distributed System: A Specification-Based Approach, Ph.D. Thesis, Dept. Computer Science, University of California at Davis, 1996.
10. R. Lippmann, J.W. Haines, D. Fried, J. Korba and K. Das, The 1999 DARPA Off-line evaluation Intrusion Detection Evaluation, Computer Networks, 34, 2000.
11. SunSHIELD Basic Security Module Guide, `http://docs.sun.com`.
12. P. Porras and R. Kemmerer, Penetration State Transition Analysis:A Rule based Intrusion Detection Approach, Eighth Annual Computer Security Applications Conference, 1992.
13. F. Schneider, Enforceable Security Policies, TR 98-1664, Department of Computer Science, Cornell University, Ithaca, NY, 1998.
14. R. Sekar, Y. Guang, T. Shanbhag and S. Verma, A High-Performance Network Intrusion Detection System, ACM Computer and Communication Security Conference, 1999.
15. R. Sekar and P. Uppuluri, Synthesizing Fast Intrusion Prevention/Detection Systems from High-Level Specifications, USENIX Security Symposium, 1999.