A Specification-Based Approach for Building Survivable Systems*

R. Sekar Yong Cai Mark Segal

{sekar,ycai}@cs.iastate.edu ms@bellcore.com

Department of Computer Science
Iowa State University
Ames, IA 50011.

Bellcore
445, South Street
Morristown, NJ 07960.

Abstract

Survivable information systems continue to perform their mission in the face of spontaneous faults, as well as malicious attacks. To build such systems, it is necessary to detect and isolate problems *before* they impact system performance, functionality or security. Previous research efforts in this area focussed primarily on detecting intrusions, rather than preventing them. Moreover, the topic of automated responses to attacks has not received much attention. In this paper, we propose a new approach that combines *prevention, detection* and *isolation* techniques to recover from a variety of threats and malicious attacks. Our method is based on high-level specifications of legitimate interactions between programs and their host operating systems. We generate optimized detectors that can monitor the runtime behavior of programs to identify violations from specified behavior. Programs that violate the specifications can be placed in an isolated environment where they cannot cause any further damage to the system. This approach leads an attacker to believe that they are succeeding in their attack, whereas in reality they are simply wasting their time and resource, meanwhile giving the administrators of the attacked system an opportunity to observe and document their actions. The isolation and corrective actions can also be specified uniformly within our specification-based framework.

Keywords: intrusion detection, survivable systems, network security, computer security.

1 Introduction

Information and networking technologies are playing increasingly important roles in our infrastructures for such critical services as power generation and distribution, telecommunication, commerce and banking, and transportation. Along with the potential benefits brought about by this change, several new dangers arise, as individuals or organizations can wreak havoc on these critical services (for financial, political or military gains) by mounting carefully orchestrated attacks on the underlying computing and networking infrastructures. It is thus critical to make these computing and networking infrastructures highly robust and resilient, so that they can continue to perform their critical functions even in the face of large-scale failures or coordinated, malicious attacks. Unfortunately, the recent spate of medium and large-scale failures in the telecommunication network and the Internet has highlighted the potential weaknesses of these systems. Existing technology relies heavily on highly skilled professionals to configure and protect the infrastructures against attacks. Such manual approaches are error-prone, costly, do not respond in a timely fashion, and are not scalable. Consequently, a lot of research efforts have focussed on *automating* the tasks related to securing such systems.

A majority of attacks on modern computer systems are based on exploiting errors in various application and system programs to gain unauthorized privileges in the system. For instance, the well-known Internet worm [Spafford89] exploited a buffer-overflow error in the UNIX `fingerd` program, and also an inadequate authentication error in the `sendmail` program involving the use of a debug option. In spite of extensive use and several years of bug-fixes, the continuing stream of advisories from organizations such as CERT(TM) coordination center indicates that similar errors will continue to persist in many application and system processes in the foreseeable future. Thus, techniques for securing computer systems must focus on approaches that can detect exploitation of such errors, rather than being based on eliminating the underlying errors themselves. Several such techniques for *intrusion detection* have been developed recently [Anderson95, Forrest97, Ilgun93, Kumar94, Ko96, Lunt93]. However, in order to build a reliable and survivable system, it is necessary to detect and isolate the problems *before* they impact system performance or functionality. Previous research efforts in this area focussed primarily on detecting intrusions, rather than preventing them. Moreover, the topic of

*This project is supported by Defense Advanced Research Project Agency's Information Technology Office (DARPA-ITO) under the Information System Survivability program, under contract number F30602-97-C-0244. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U. S. Government.

automated responses to attacks has not received much attention. These two factors mean that a fast-progressing attack can compromise system security and possibly disable the intrusion detection mechanisms themselves before the problem is detected. Even in cases where the problem can be detected, an attacker may be able to cause considerable amount of damage before adequate responsive action is taken. We therefore propose a new approach that combines *prevention, detection* and *isolation* techniques aimed at automatically responding to a variety of threats and malicious attacks, and recovering quickly from the damage caused by them. Our approach is inspired by *software auditing* and *formal specifications,* two of the techniques that have been used with significant success in telecommunication systems.

1.1 Overview of Approach

Software auditing is a technique that has been developed in the context of telecommunication switching systems, and has been very successful in making these systems highly reliable. Software auditing does not attempt to tackle the high-level behaviors of processes and systems, which are far too complex to make a precise distinction between correct (or safe) and incorrect behaviors. Instead, it is based on constant runtime monitoring of several integrity or consistency conditions that must hold in a system that is operating correctly. For instance, dangling pointers, corrupted data structures, lost resources, etc. indicate potential problems in the system. By continuously checking the data structures used by switching system software, this technique identifies corruption and inconsistencies in data before they lead to system failures. Use of this technique resulted in an impressive two to three orders of magnitude improvement in the MTBF (mean time between failures) of switching software. Our approach seeks to extend this success to large-scale information systems.

Whereas software auditing was focussed mainly on data-structures, in the context of survivability, we are more interested in auditing *security-related behaviors* of processes. To simplify development of correct and efficient auditing functions that can be readily customized to different applications and operating environments, we propose to specify security-related behaviors declaratively in a high-level language called *auditing specification language (ASL).* ASL is designed to be powerful enough to express a wide range of integrity constraints and behaviors over time. The constraints can span multiple processes and/or hosts. The code for auditing is generated by a compiler from these specifications. Our approach is similar in motivation and spirit to the approach of Ko et al [Ko94, Ko96], although the specification language and detection methods are quite different. We describe this relationship in more detail in the next section.

We employ runtime enforcement techniques designed to ensure that a program satisfies the behavior specified in ASL. When enforcement is impractical or undesirable, we employ detection techniques to identify deviations from specified behavior, at which point automatic corrective actions can be taken. A simple corrective action is to terminate a process that deviates from specified behavior, but this may not be desirable since it may alert an attacker that he/she has been detected. Instead, we develop *isolation techniques* that can isolate the compromised software process in such a way that it cannot cause any further damage to the system. The process is allowed to continue operation in this isolated environment, thus leading an attacker to believe that they are succeeding in their attack, whereas in reality they are simply wasting their time and resource, while giving the administrators of the attacked site an opportunity to observe and document the action of the attacker. We refer to software that can be run in this manner as *ablative software.* These corrective actions are also described within the specification-based framework using ASL.

For auditing-based detection to be successful, we need the ability to obtain the low-level information to be monitored from the large base of existing software that does not provide any support for such monitoring. To deal with this difficult problem, we decompose the detection system into two components. The primary detection system (PDS) is designed to observe and intercept the system calls made by a process, which provide the only mechanism through which a process can inflict damage on a system. In particular, operations for manipulating files or network connections are all administered through system calls. Based on this observation, our approach in PDS is to intercept and validate the system calls, their parameters and the return values. Note that this approach gives us the ability to detect problems before they happen, and can thus be preventive.

Obtaining information on a more global scale is important to enlarge the class of problems that can be detected. However, global information is associated with a higher cost than can be afforded in the performance-sensitive PDS. We therefore propose to develop a secondary detection system (SDS) that trades off real-time performance in favor of a larger class of problems that can be detected. One of the key issue in SDS is the ability to obtain information about systems and applications from such heterogeneous sources as configuration and log files, and SNMP[†] agents that provide information about the network and the other hosts.

The rest of this paper is organized as follows. We compare our work with previous research in intrusion detection in Section 1.2, followed by a discussion of the salient features of our approach in Section 1.3. Our specification language is described in Section 2 and illustrated with several examples in Section 3. Our techniques for isolation are described in Section 4. A preliminary study of the effectiveness of our approach is presented in Section 5. Finally, implementation issues are described in Section 6.

1.2 Related Work

Intrusion detection techniques can be broadly divided into *anomaly* detection and *misuse* detection techniques. Anomaly detection based approaches first compile a profile that defines *normal behaviors* and then detect deviations from this profile. Several such techniques have been developed, based on statistical methods, expert-systems, neural networks, or a combination of these methods [Fox90, Lunt88, Lunt92, Anderson95]. One of the main advantages of anomaly-based intrusion detection is that the system can be trained to identify *normal behavior,* and it can then automatically detect when observed behavior deviates significantly from this. The downside is that an attacker can evade detection by changing behavior slowly over time. For this reason, most systems combine anomaly detection with *misuse detection,* where we define and look for precise sequences of events that result in compromising the security of a system. Intrusion can be flagged as soon as these events occur. Techniques for misuse detection have been based on expert systems, state-transition systems [Porras92, Ilgun93] and pattern-matching [Kumar94]. While it is relatively easy to deal with known vulnerabilities using misuse detection, it is difficult to cope with unknown vulnerabilities.

A *specification-based approach,* first proposed by Ko et al [Ko94, Ko96], is aimed at overcoming the drawbacks of misuse detection by describing *intended* behaviors of programs, which does not require us to be aware of all the vulnerabilities in the program that could be misused. Our approach, although inspired by software-auditing, is in fact similar in spirit to their approach in that we also specify intended behaviors of processes. An important improvement in our approach is that we can *enforce* the specified behaviors at runtime so as to prevent large classes of attacks, whereas their approach uses offline analysis of audit logs. Another important distinction arises in terms of the specification language used. [Ko96] uses a specification language based on context-free grammars augmented with state variables, while our specification language is closer to regular languages augmented with state variables. While regular grammars are less expressive than context-free grammars, the difference is much less pronounced when these grammars have been augmented with state variables. Moreover, use of regular grammars affords the ability to compile the specifications into an extended finite-state automaton (i.e., an FSA augmented with state variables). Such an EFSA would enable very efficient runtime checking with bounded resources (CPU or memory) that can be determined apriori. These factors are particularly important in the context of an online approach such as ours.

Forrest et al [Forrest97, Kosoresow97] have developed intrusion detection techniques inspired by immune systems in animals. They characterize "self" for a UNIX processes in terms of (short) sequences of system calls that are made by the process in course of normal operation. Intrusion is detected when we observe "foreign" system call sequences that have not been observed under normal operation. Their research results are largely complementary to ours, in that their main focus is on *learning* normal behaviors of processes, where as our focus is on *specifying* and *enforcing* these behaviors efficiently. In particular, the finite-state automaton learnt by the technique of [Kosoresow97] can be fed as input to our runtime monitoring and isolation system.

Goldberg et al [Goldberg96] have developed the Janus environment designed for confining helper applications (such as those launched by web-browsers) so that they are restricted in their use of system calls. Like our techniques, they can also prevent unauthorized operations, such as attempts to modify a user's `.login` file. But their approach is designed more as a finer-grained access-control mechanism rather than as an intrusion detection mechanism. The key distinction we make in this context is as follows. Access control mechanisms enable us to provide the minimum set of access rights to each process that will enable them to get their job done, while intrusion detection techniques are aimed at determining whether a process uses its access rights in the intended fashion. For instance, problems such as race conditions and unexpected interactions among multiple processes all manifest themselves as unintended use of access rights. Consequently, it is necessary for us to support a more expressive specification language that can capture sequencing relationships among system calls, whereas Janus permits restriction of access to individual system calls only. Moreover, they base their

[†]SNMP stands for simple network management protocol. It provides the mechanisms for obtaining network related information through *management information bases* (MIBs).

techniques on specific operating system features provided by Solaris, whereas we are interested in *detection* and flexible *isolation* techniques that can be applied to a wide range of operating systems.

Our approach to isolation has some similarities with the approach proposed in *Deception Toolkit (DTK)* [Cohen98]. In particular, when an intrusion is detected, our approach aims to isolate the offending process to a restricted environment in such a way that further actions taken by the process can no longer damage the system. From the perspective of the attackers, this gives the impression that they are succeeding in their effort, while in reality they are simply wasting their effort and resources. The DTK also employs a similar strategy. DTK replaces each network service such as sendmail and telnet by a "fake" server that can carry on a dialogue with a remote user much like the real server. However, the real service is no longer available when a deception server is running on a machine. This contrasts with our approach, where standard server functionality is still present.

1.3 Salient Features of Approach

- *Prevention and/or early detection.* One of the central features of our approach is that it enables early detection of large classes of attacks *before* they compromise system security. This preventive ability enlarges the class of threats we can deal with, e.g., we can protect against intrusions that exploit bugs in programs that are otherwise trustworthy, as well as malicious programs (e.g., Trojan horses) from untrusted sources. It also brings additional flexibility to the system, e.g., it enables a system administrator to respond quickly to a newly discovered vulnerability, without having to wait for a vendor-supplied patch.
- *Degree of transparency.* It is desirable to achieve survivability without inconveniencing normal users. The key to do this is to develop a highly transparent solution that is (a) application transparent so that existing applications can run on a survivable system without requiring them to be rewritten, (b) user-transparent, so that users continue to perceive the system as "open" and "user-friendly," and (c) misuser-transparent, so that a misuser or attacker can be fooled into believing that he/she is succeeding in their attack.
- *Specification-based detection and response.* Previous research in specification-based intrusion detection focussed on specifying misuse or intended behavior. In contrast, our approach integrates detection as well as the actions to be taken in response to intrusion attempts.
- *Optimization techniques to minimize runtime overhead.* We use the specifications as a basis for performing optimizations necessary to meet the stringent efficiency requirements imposed by the fact that all system calls have to be intercepted and monitored at runtime in our approach.
- *Dynamically tunable monitoring.* Our technique allows the granularity of monitoring to be changed on the fly at runtime. We can use a low-level of monitoring under normal conditions, but can quickly increase the level of monitoring when errors or suspicious activity is detected

2 Overview of ASL

In ASL, we model behaviors of processes and systems in terms of sequences of (security-related) events. In general, events may be internal to a single process, e.g., a function call made by the process; or they may be externally observable, e.g., a message delivered to a machine, or information that could be obtained from a system utility or a remote machine. We represent an event in the form $EventName(Arg_1, ..., Arg_n)$, where $Arg_1, ..., Arg_n$ are expressions denoting parameters of the event. An *event history* is a sequence of events (internal or external) generated by a system. Security-related behaviors are captured using patterns on the event history. Thus, our specifications are structured in the form of rules that consist of a pattern and reaction components. The reaction component is executed whenever the pattern component matches the event history.

One of the classes of events in ASL includes network-level events, such as network management data obtained using standardized management protocols such as SNMP. A more important class of events consists of system calls made by processes to obtain services from the operating system kernel. Our approach is to enforce behaviors of processes at runtime by intercepting the system calls made by each of them, and comparing them against the specifications. When enforcement is infeasible or impractical, our monitor is designed to *detect* violations as early as possible. At the system call level, we focus primarily on behaviors of individual processes that can be observed without any regard to the actions being taken by other processes in the system. This means that the runtime detection engines for different processes do not need to interact, which is desirable for their efficient implementation. This does not always limit our approach from detecting intrusions involving undesirable interactions among multiple processes. This is because we can often characterize such interactions in terms of the actions performed by one process and the state of the system. (See Section 3 for an example involving race conditions.) A more systematic approach for specifying behaviors that involve multi-process interactions is currently being developed.

Modules At the top level, each monitoring specification will be structured as a collection of parameterized modules. A module is characterized by a name and a set of parameters. It may define a set of *state variables* and also instantiate other modules by providing appropriate parameters. The last component of a module is a sequence of pattern-matching rules further described in the following sections.

The module construct is aimed at minimizing the need to duplicate similar or identical behaviors many times. As is well-known in software engineering, such duplication leads to inconsistencies and obscure bugs over the long term, and is to be avoided in a good language or system design. Modules capture generic behaviors that are parameterized with respect to a set of module parameters. For instance, we can capture a behavior involving the use of a sequence or repetition of certain system calls, and have the module parameterized with respect to the system call arguments, time intervals between the calls, or the number of repetitions. Specific behaviors can be obtained by instantiating the modules by providing the values of the parameters. A module may also have substructure, i.e., it may instantiate other modules. Thus it provides for a grouping of related properties, and more generally, a way to organize specifications in a hierarchical fashion. This grouping capability is particularly important for dynamically altering the degree of monitoring in response to suspicious activities. The same capability can also be used for isolating compromised processes as described in Section 4.

Patterns ASL *patterns* are designed to capture valid or invalid behaviors. The patterns consist of *event patterns* composed using *temporal operators.* The event patterns describe specific events of interest, while the temporal operators capture the sequencing and timing relationships that must hold between these events. The event patterns are of the form $event|cond$, where $event$ is of the form $EventName(Arg_1, ..., Arg_n)$, and $cond$ is an expression on the variables in $Arg_1, ..., Arg_n$ that evaluates to $true$ or $false$. The condition component can make use of standard arithmetic, comparison and logical operations. It will also support regular expression matching, which is particularly useful for file-name-related operations. Finally, for events that represent system calls, C-style array and structure access operations may be used on the system call arguments.

Event patterns allow us to describe conditions on the events that are permissible and those that are not. For instance, we can capture allowable processes that can be executed by the `fingerd` as follows:

$$exec(filename, ...)|realpath(filename) = \text{“}/usr/ucb/finger\text{”}$$

Here, the event $exec$ refers to any of the possible exec-related system calls.[‡]. In the condition component, we use a predefined function in ASL called *realpath* to resolve all links, symbolic links, and references to "." and ".." contained in $filename$. We then check that the resolved name is `/usr/ucb/finger`. This simple pattern can prevent the exploitation of the vulnerability that was used in the Internet worm attack on the finger daemon.

To capture sequencing or timing relationships among events, we need to use temporal operators. In order to define the meaning of these operators, we develop the notion of an event history satisfying a pattern.

- $e(pa_1, ..., pa_n)$, where e is an event with argument patterns $pa_1, ..., pa_n$, is satisfied by any event history of the form $..., e(a_1, ..., a_n)$ where a_i is an instance of pa_i.
- $pat_1 \circ pat_2$ is satisfied by an event history H of the form $H_0H_1H_3H_2$ provided H_1 satisfies pat_1, H_2 satisfies pat_2, and H_3 does not satisfy either pat_1 or pat_2. Intuitively, the operator '$\circ$' denotes the notion of one event following another, with other unrelated events possibly occurring between the two. Note here that $H_0, ..., H_3$ are event histories in themselves such that their concatenation $H_0H_1H_3H_2$ yields the entire history H.
- pat^n is satisfied by an event history $H_0H_1H_2 \cdots H_n$ provided that H_i satisfies pat, for every $1 \leq i \leq n$. This operator is used for repetition of patterns.
- $pat_1 \,||\, pat_2$ is satisfied by an event history H if either pat_1 or pat_2 is satisfied by H.

[‡]The relationships among the different variations of this system call are described separately to the ASL compiler, and we skip the details here due to space considerations.

- *pat* **within** t is satisfied by an event history $H_0 H_1$ if H_1 satisfies *pat* and the time interval between the first and last events in H_1 is less than or equal to t.

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 $e_1(x) \circ e_2(x)$ will not be satisfied by the event history $e_1(a)e_2(b)$, but will be satisfied by $e_1(a)e_2(a)$.

We illustrate the use of some of these temporal operators below. Race conditions in file access, which often lead to errors, can be captured using patterns such as:

$$[access(name, ...)|realpath(name) = f1] \circ [open(name, ...)|realpath(name) \neq f1] \text{ within } 3\ sec$$

This pattern can capture several attacks that exploit the small window of time between the time access permissions are checked and the file actually opened by a privileged program on behalf of a normal user. This vulnerability existed in many programs, including `passwd`, `rdist` and `xterm`. The three-second "time-out" is used so that unrelated system calls dispersed widely in time are not accidentally identified as intrusions.

Repetitive actions, such as repeated login failures, can be captured as shown below. Here, the event *login*(*username*) is an *synthesized event* that is generated by some component of the monitoring system, possibly based on an examination of system log files or audit logs. This event is then fed into the detection system.

$$[login(name, ...)]^3 \textbf{ within } 5\ min$$

As another example, consider a TCP SYN attack. One way to detect this attack is to monitor an SNMP MIB variable called `tcpAttemptFails` and look for sudden increases in this value. We can model periodic polling for the value of this variable using an event of the form *tcpAttemptFails*(n), where n denotes the value of this MIB variable. With this model, a pattern to identify a (warning sign of) SYN attack is given by:

$$[tcpAttemptFails(n1) \circ (tcpAttemptFails(...))^0 \circ tcpAttemptFails(n2)|(n2 - n1 > max)] \textbf{ within } t$$

Here max and t are constants defined elsewhere in the specification.

Rules A rule is of the form $pat \rightarrow actions$, where *pat* is a pattern of the form described above, and *actions* is a sequence of responsive steps to be initiated as soon as we identify a system behavior that satisfies the pattern. In the simplest case, the sequence could be empty, in which case no action is undertaken.

An action can be an assignment to state variable belonging to the module that contains the rule. It may also be an invocation of any pre-defined ASL function. Two important class of functions are important in the primary detection system. The first class consists of all system calls. The second class consists of the single action *fail* which prevents the system call from being completed. An optional argument to *fail* permits "fake" return values (and error codes) to be transmitted back to the calling program. These actions are particularly useful in the context of isolation mechanisms described later on.

Finally, two other forms of action are permitted to enable us to change the level of monitoring. The *switch* action enables switching to a completely different specification for monitoring. The *activate* action enables additional rules in the specified module to be activated for monitoring, while the rules in the current module are still active. The use of the reaction component is further detailed in Section 4.

3 Example Specifications for Intrusion Prevention/Detection

Finger Daemon The following specification restricts the finger daemon so that it can open only a select set of files for reading, cannot open any file for writing, cannot execute any file, and cannot initiate connection to any machine. When any of these conditions are violated, the offending system call is not allowed to execute, and an error code is returned. The following specification pertains to the GNU finger program, and in particular, the finger daemon running as the master server. Note that the implementation of GNU finger daemon is quite different from the Berkeley implementation mentioned earlier.

```
Module fingerd(datadir)
    open(file, mode)|(realpath(file) ∉ {``/etc/utmp'', ``/etc/passwd'', ``$datadir/*''})
            ∨(mode ≠ O_RDONLY) → fail
    disallowed : exec, connect, chmod, chown, chgrp, create, truncate, sendto, mkdir
end
```

Here, *exec* stands for all variations of the system call for executing programs. We also use a shorthand notation to list all disallowed calls, rather then writing a pattern that disallows each specific call. (The example shows only a subset of disallowed system calls rather than providing a complete list.) With this specification, we can prevent a variety of attacks, including the buffer overflow attack used by the Internet worm.

FTP Daemon With FTP, we may be interested in ensuring that the files accessed by the FTP server are accessible to the user making use of the FTP service. We can accomplish this using the following specification:

$$getpwname(name) \circ open(file, mode, ...)|\neg accessible(realpath(file), mode, name)) \rightarrow fail$$

Since the name of the user is not an argument to any system call (except possibly a setuid call that may be executed after the ftp user login procedure is completed by some ftp servers), we need to observe a library call such as *getpwnam*. If this is not possible, then we should rely on information logged into system and ftp server log files to detect this problem.

Race conditions in xterm and passwd Old versions of xterm on certain UNIX implementations needed to run as super-user. When xterm was invoked with a logging option, it needed to ensure that the invoking user had the permissions to write to the log file. Between the time xterm checks the access permissions on the file and the time it actually opens the file (as a superuser), there is a small window of time that can be exploited by an attacker to overwrite arbitrary files. The attacker can create a file that is a symbolic link to one of his/her files at the time xterm checks access permissions, and then relink it to a file such as /etc/passwd by the time xterm opens the file. A specification for this problem may look as follows:

$$[access(name, ...)|name1 = realpath(name) \circ open(name, ...)|realpath(name) \neq name1] \rightarrow fail$$

A Utility Program from Untrusted Source In this case, we may want to ensure that the program can only read world readable files, it can write only within `/tmp` directory, cannot execute any programs, and cannot perform network operations. This is easily ensured by the following specification:

```
$Module\ fingerd(datadir)$
    $open(file, mode)|[realpath(file) \notin \{\text{"/tmp/*"}\}$
                                    $\wedge (mode$ & (O_WRONLY | O_APPEND | O_CREAT | O_TRUNC))]
            $\vee(\neg accessible(realpath(file), mode, \text{"nobody"})) \rightarrow fail$
    $disallowed : exec, connect, bind, chmod, chown, chgrp, sendto, mkdir$
$end$
```

4 Isolation

Once a compromised process has been identified, the system must ensure that this process can no longer affect the system integrity. The simplest policy is to kill the compromised process and restart it afresh. However, this is not satisfactory for several reasons. It does not give an indication of the nature of the problem, and it may alert the attacker too quickly that the system is counter-reacting. Instead, we reconfigure the environment of the compromised process to isolate it from the rest of the system. Others parts of the system will not be affected by the isolation and continue to operate normally. This approach can fool the attacker into believing that the attack is succeeding, when in reality they are merely wasting their own resources as well as providing the owner of the attacked system an opportunity to track down the attacker. We refer to software that can be isolated from a system and continue running as *ablative software.*

The isolation component is integrated seamlessly into our specification based framework for detection. When we detect an intrusion, we use the `switch` action to switch to a new specification that contains ASL rules to isolate the compromised process. These rules modify the behavior of system calls in such a way that compromised processes are prevented from executing operations that can damage the rest of the system. Specifically, the reaction component can be used to perform one or more of the following:

- *return error code or bogus return value.* When a system call that can potentially damage the system is invoked by the compromised process, we can prevent the system call from being completed, and instead return an error code or a bogus return value.

- *log the activity for later analysis.*
- *reduce resources consumed by the rogue process.*
- *restrict access to files.* We can use the `setuid` system call to change the effective userid of the process to that of a user with very few access rights. Alternatively, we could use the `chroot` system call to change the root directory of the compromised process.

To illustrate this idea, consider the finger daemon with a slightly modified specification to implement isolation. In particular, we introduce the rule:

$$exec(...) \rightarrow chroot(\text{“/altroot”}); setuid(-1); nice(100); switch\ genericIsolate;$$

This rule changes the root of the calling process to a decoy file system (called `altroot`), changes the userid to `nobody`, reduces the priority of the process, and finally switches to a new monitoring specification called *genericIsolate*, which may be specified as:

```
Module genericIsolate
    connect(...) → sleep(60); fail ETIMEDOUT
    bind(...) → sleep(5); fail EADDRINUSE
    recv(...) → sleep(1); continue
    open(...) → sleep(1); continue
end
```

There may be several other rules in this module, but the ones given above are illustrative. Since the process is operating in a decoy file system, file system operations are allowed go through. However, network operations are restricted. Most operations are slowed down using `sleep`, so that the CPU and resource usage on the attacked system are minimized, while the intruder will likely perceive a slow system and/or congested network.

5 On the Effectiveness of our Approach

In this section we present a preliminary evaluation of the effectiveness of the our approach. For this purpose, we studied the advisories put out by CERT(TM) coordination center [CERT98] from 1993 through May of 1998. Our results are shown in the following table.

Category	Count	Detectability
Buffer overflow	29	Most
Insecure filename handling	14	Most
Other inadequate argument checking	7	Majority
Trojan Horses in privileged programs	3	Moderate
Insecure program features	7	Moderate
Configuration errors	10	Minority
Kernel-level attacks	3	None
Weak authentication/encryption protocols	7	None

Since the CERT advisories do not provide detailed description of vulnerabilities, some of the attacks in the table may be misclassified. Moreover, the table indicates only whether a vulnerability can be captured in an ASL specification; we have not yet experimentally verified that we can indeed capture the attacks. Nevertheless, the table gives an idea of the categories of vulnerabilities that have a low or high likelihood of being identified by our technique. It also provides preliminary indication that our technique will likely be able to capture a large fraction of intrusions known to date.

The number of UNIX related advisories over this period was 96, out of which we could not determine the underlying vulnerability for 16. We have classified the remaining 80 into eight categories as shown above. We provide a brief description of some of these categories below, together with a justification of the detectability figure. Buffer overflows typically involve a privileged program forking a shell connected to an attacker, or otherwise performing actions that are well out of the norm for the attacked process. As such, our technique has a high likelihood of capturing such attacks. Insecure filename handling refers to the fact that a program fails to perform adequate checking on the target file (especially when a file happens to be a symbolic link). Since the result is typically that the program overwrites a file that it is not normally supposed to touch, this class of attacks can also be generally captured by us. The third category also involves operations on objects typically not accessed by a program, and hence can be largely identified by our technique.

Configuration errors, kernel-level attacks and weak protocols all generally fall outside the scope of our technique. For instance, kernel-level attacks (such as ping-of-death) are not even visible at the system call level, and hence cannot be captured. Weakness in authentication or cryptographic protocols are also completely outside our model. A minority of configuration errors can be captured by our technique, to the extent that the error results in a process accessing data that it does not typically access.

6 Implementation Issues

6.1 Compilation of ASL for Fast Intrusion Detection

Our specification-based approach enables us to focus on the relationships, constraints and behaviors that must hold in a system that operates correctly, as opposed to the details of implementing the checks for these conditions. Efficient implementation of auditing functions are derived automatically from the ASL specifications by our ASL compiler. This approach enables us to analyze the ASL specifications and optimize them, using techniques such as sharing of common computations across multiple auditing functions. Such optimization is especially critical for PDS, where potentially every system call is subjected to auditing.

Note that the behavior of a single process is typically constrained by several ASL rules. A naive implementation of pattern-matching for these rules would require us to test each system call against each of these patterns. This will lead to unacceptable overheads for runtime monitoring. Thus, it is critical to speed up the checking by (a) avoiding sequential search through all ASL patterns to be checked, and (b) enabling incremental checking of temporal patterns, without having to explicitly sequence through the history of system calls made in the past. The key idea here is to *factor* the pattern-matching operations into a sequence of test operations so that we can share expensive tests across multiple patterns. Moreover, a single test may enable us to rule out many candidate patterns, so that we can quickly zoom in on the applicable patterns. We remark that optimizations such as sharing and factoring come for "free" in our specification-based approach. On the other hand, if the auditing functions were hand-implemented, considerable additional effort would be required to obtain these benefits. More importantly, hand-coding of such optimizations typically leads to intertwining of different auditing functions, leading to code that is difficult to understand or evolve.

The ASL compiler will translate the ASL specifications into an *extended finite-state automaton* (EFSA). An EFSA extends traditional finite-state machine by adding auxiliary variables, in addition to the states of the automaton. Each system call made by a monitored process is passed to the EFSA, which makes transitions based on the name and arguments of this system call. If a transition takes it to a final state, then any action associated with the final state is performed. The action would correspond to the reaction component of an ASL specification. Otherwise, the system call is passed onto the kernel. EFSA-based representations are crucial for efficient runtime monitoring of multiple ASL assertions; analogous FSA representations form the basis for efficient techniques for similar problems in other domains, including regular-expression matching and tree-pattern matching.

6.2 Runtime Monitoring and Isolation

We propose to implement the runtime monitoring and isolation mechanisms within the system call interface layer of the operating system kernel. As compared to some of the other approaches mentioned below, this approach lends itself to a more secure implementation, since a user process can only access the OS through well-defined entry points and cannot modify kernel code in a well-designed operating system. Moreover, the response actions need not be constrained by the privileges of the userid corresponding to the monitored process.

The monitoring system will be implemented as a loadable module in the operating system kernel. If the operating system does not support loadable modules, then we would need to integrate an interpreter for EFSA permanently into the OS kernel, which would read in compiled EFSA code. In either case, the ASL specifications and the compiled EFSA code can all be held within a secure repository, with access to the repository carefully controlled.

An alternative to the kernel-based approach is to implement the monitoring within the process space of the monitored process, by using a modified version of the system call library. This modified library will perform the checking operations before passing the call onto the kernel. This approach has the benefit that it does not require any changes to the kernel. But the downside is that it can be defeated by malicious attacks that exploit the memory-unsafety of the C-language (or machine code, for that matter) to corrupt the data or code. For instance, the approach cannot prevent buffer-overflow attacks.

A third approach makes use of user-level system-call tracing facility provided by System V style UNIX operating system through the `/proc` interface. This approach is more robust than the library-based approach, but a limiting factor is that such mechanisms may not be available on all operating systems. Even where they are available, they typically do not provide the flexibility needed for implementing the isolation mechanism, which requires us to be able to execute arbitrary system calls on behalf of the calling process.

Acknowledgments

The authors would like to thank Thomas Bowen with whom we have had many valuable discussions regarding the subject of this article. These discussions have significantly improved the content and presentation of the article.

7 Bibliography

[*8lgm*] 8lgm Security Advisories, `http://www.8lgm.org/advisories-f.html`.

[*Anderson95*] D. Anderson, T. Lunt, H. Javitz, A. Tamaru, and A. Valdes, Next-generation Intrusion Detection Expert System (NIDES): A Summary, SRI-CSL-95-07, SRI International, 1995.

[*Aslam96*] T. Aslam, I. Krsul and E. Spafford, A Taxonomy of Security Faults, Nat'l Computer Security Conference, 1996.

[*Bellovin89*] S. Bellovin, Security Problems in TCP/IP Protocol Suite, ACM Computer Comm. Review, 19(2), 1989.

[*CERT98*] CERT Coordination Center Advisories 1988–1998, `http://www.cert.org/advisories/index.html`.

[*Cheswick92*] W.R. Cheswick, An evening with berferd, in which a cracker is lured, endured and studied, Winter USENIX Conference, 1992.

[*Cohen98*] Fred Cohen and Associates, The Deception Toolkit Home Page, `http://www.all.net/dtk/dtk.html`.

[*Connet72*] J. Connet et al, Software Defenses in Real-Time Control Systems, IEEE Fault-Tolerant Comp. Sys., 1972.

[*Denning87*] D. Denning, An Intrusion Detection Model, IEEE Trans. on Software Engineering, Feb 1987.

[*Forrest97*] S. Forrest, S. Hofmeyr and A. Somayaji, Computer Immunology, Comm. of ACM 40(10), 1997.

[*Fox90*] K. Fox, R. Henning, J. Reed and R. Simonian, A Neural Network Approach Towards Intrusion Detection, National Computer Security Conference, 1990.

[*Goldberg96*] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer, A Secure Environment for Untrusted Helper Applications, USENIX Security Symposium, 1996.

[*Ilgun93*] K. Ilgun, A real-time intrusion detection system for UNIX, IEEE Symp. on Security and Privacy, 1993.

[*Ko96*] 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.

[*Ko94*] C. Ko, G. Fink and K. Levitt, Automated detection of vulnerabilities in privileged programs by execution monitoring, Computer Security Application Conference, 1994.

[*Kosoresow97*] A. Kosoresow and S. Hofmeyr, Intrusion detection via system call traces, IEEE Software '97.

[*Kumar94*] S. Kumar and E. Spafford, A Pattern-Matching Model for Intrusion Detection, Nat'l Computer Security Conference, 1994.

[*Landwehr94*] C. Landwehr, A. Bull, J. McDermott and W. Choi, A Taxonomy of Computer Program Security Flaws, ACM Computing Surveys 26(3), 1994.

[*Lindqvist97*] U. Lindqvist and E. Jonsson, How to Systematically Classify Computer Security Intrusions, IEEE Symposium on Security and Privacy, 1997.

[*Lunt88*] T. Lunt and R. Jagannathan, A prototype real-time intrusion detection system, IEEE Symp. on Computer Security and Privacy, 1988.

[*Lunt92*] T. Lunt et al, A Real-Time Intrusion Detection Expert System (IDES) - Final Report, SRI-CSL-92-05, SRI International, 1992.

[*Lunt93*] T. Lunt, A survey of Intrusion Detection Techniques, Computers and Security, 12(4), June 1993.

[*Porras92*] P. Porras and R. Kemmerer, Penetration State Transition Analysis - A Rule Based Intrusion Detection Approach, Computer Security Applications Conference, 1992.

[*Spafford89*] E. Spafford, The Internet Worm Program: An Analysis, ACM Computer Comm. Review, 19(1), 1989.