AFRL-IF-RS-TR-2001-145
Final Technical Report
July 2001

SURVIVABLE ACTIVE NETWORKS

Telcordia Technologies, Inc.

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. F245

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2001-145 has been reviewed and is approved for publication.

APPROVED: Glen E. Bahr

GLEN E. BAHR
Project Engineer

FOR THE DIRECTOR:

WARREN H. DEBANY, Technical Advisor
Information Grid Division
Information Directorate

SURVIVABLE ACTIVE NETWORKS

Thomas F. Bowen,
Mark E. Segal, and
R. C. Sekar

Contractor:	Telcordia Technologies
Contract Number:	F30602-97-C-0244
Effective Date of Contract:	01 July 1997
Contract Expiration Date:	31 January 2001
Program Code Number:	5L10
Short Title of Work:	Survivable Active Networks
Period of Work Covered:	Jul 97 - Jan 01

Principal Investigator:	Thomas F. Bowen
Phone:	(973) 829-4550
AFRL Project Engineer:	Glen E. Bahr
Phone:	(315) 330-3515

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Glen E. Bahr, AFRL/IFGB, 525 Brooks Road, Rome, NY.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	JULY 2001	Final Jul 97 - Jan 01

4. TITLE AND SUBTITLE

SURVIVABLE ACTIVE NETWORKS

5. FUNDING NUMBERS

C - F30602-97-C-0244
PE - 62301E
PR - F254
TA - 40
WU - 01

6. AUTHOR(S)

Thomas F. Bowen, Mark E. Segal, and R. C. Sekar

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Telcordia Technologies, Inc.
445 South Street
Morristown New Jersey 07960-6438

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

Air Force Research Laboratory/IFGB
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2001-145

11. SUPPLEMENTARY NOTES

Air Force Research Laboratory Project Engineer: Glen E. Bahr/IFGB/(315) 330-3515

12a. DISTRIBUTION AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A total solution to the computer security problem requires commitment to security issues at every stage of an application's lifecycle. The Survivable Active Networks (SAN) project applies to the last lifecycle stage-application operation. While only one piece of the solution, as a solution of last resort it is vital.

SAN technology is a programming environment for creating solutions. SAN research included using this programming environment to create actual security solutions that provide protection against real-world attacks. SAN technology remediates latent software errors that enable popular and powerful exploits, including stack and buffer overflows, race conditions, ping-of-death, neptune, port scanning, and syn-flooding. It allows end-users who have software development expertise to customize defenses for the applications they use. SAN also allow third-party vendors to offer customized add-ons for applications so that end-users without programming abilities can still obtain SAN benefits, even in the absence of application source code.

The primary focus of SAN was understanding whether the underlying principles of interception and interposition were valuable for empowering end user security remediation. Through building a prototype for Linux and using the prototype in evaluations, the value of basic concepts of interception and interposition have been confirmed.

14. SUBJECT TERMS

Computer Security, Survivable Active Networks, Stack Overflow, Buffer Overflow, Race Conditions, Pint-of-Death, Neptune, Port Scan, SYN-Flood, Real Time Interception, Interposition, System Call, Incoming Packets, CIDF

15. NUMBER OF PAGES

120

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

List of Figures

List of Tables

Executive Summary

Latent errors in software applications pose a great threat to computer security and that threat will escalate as the need for interoperability fosters greater application homogeneity and as applications become more complex. Already exploitation of latent errors has enabled many successful and costly computer intrusions. Current software development methodology is unable to produce error-free applications. The situation will deteriorate in the face of increasingly complex and distributed applications produced under time-to-market pressures, and increasingly skilled and well-funded attackers. Application homogeneity exacerbates the problem by making it easier for attackers to develop attacks that can easily propagate to a large number of computers. A total solution to the computer security problem requires a commitment to security issues at every stage of an application's lifecycle, from requirement, design, implementation, and testing, through installation, configuration, and operation. The Survivable Active Network (SAN) project is investigating technology that applies to the last of these: application operation. While this is only one piece of the total solution, it is a vital piece for two reasons: it is a solution of last resort, and it is a solution that empowers those who have the ultimate security responsibility, the application users, to take corrective actions to remediate the deficiencies introduced at earlier phases. SAN technology is not a security solution in itself; rather it is a general-purpose programming environment for creating security solutions. Nevertheless, SAN research has been driven by using the SAN programming environment to create actual security solutions that are firmly grounded in realism, that is, that provide protection against real-world attacks.

SAN technology has experimentally demonstrated effectiveness in remediating latent software errors that enable currently popular and powerful exploit methods, including stack and buffer overflows, and race conditions, and for detecting and deflecting attacks such as ping-of-death, neptune, port scanning, and syn-flooding. The principles underlying SAN technology are real time interception and interposition. SAN intercepts two events types; system calls and incoming packets. System calls are intercepted as they cross the boundary between the operating system and the application, since it is only at this boundary that a latent application error can actually cause damage to occur. Incoming packets are intercepted as they cross the boundary between the network interface and the operating system. By intercepting system call events application-specific defensive programs can decide whether or not the events are consistent with proper application usage and if they are not, interposition allows normal event processing to be changed to prevent it from causing damage. Interposition supports a wide range of actions from merely shutting down the application to isolating it in a safe environment and deceiving the attacker for evidence collection purposes. By intercepting incoming packets, defensive programs can check for packets that violate protocol constraints or that match known attack patterns. While interposition is in theory possible for dealing with incoming packet events, we have not implemented packet interposition. Through integration with emerging active network technology, SAN allows hosts that are under attack to request assistance from active network routers in order to trace the source of the attack, and isolate the attacker from the network close to the source, thereby protecting not only the victim host, but also reducing the impact of the attack on network resources.

SAN technology allows end-users who have software development expertise to customize defenses for the applications they use to remediate actual errors, to enforce site-specific policy requirements that are not actually supported by those applications, and to use generic defenses for safe execution of untrusted (e.g. shareware) applications. SAN also allow third-party vendors to offer customized add-ons for applications so that end-users without their own programming abilities can still obtain the SAN benefits. SAN technology allows such remediation in the absence of application source code, thereby freeing the end-user from unresponsive application vendors.

SAN has completed three years of DARPA-funded research in which the primary focus was on understanding whether or not the underlying principles of interception and interposition were indeed valuable for empowering end user security remediation. Through the building of a prototype for the Linux operating system and the use of the prototype in internal experiments and three external DARPA evaluations, the basic concepts of interception and interposition have been refined and their value confirmed. The prototype is available for technology transfer and uses the DARPA funded Common Intrusion Detection Framework (CIDF) to allow inter-operation with other security components. SAN will continue to be refined, evaluated, and moved from prototype towards product in a new DARPA funded project called Programmable Adaptive Rapid Reactors (PARR), which will focus more on SAN's autonomous reaction capabilities.

1 Introduction

In the war between security-conscience computer users and malicious hackers, the hackers appear to be winning. Every day, newspapers report intrusions into allegedly secure computers, such as computers operated by the FBI, CIA, or large financial institutions. How can malicious hackers obtain access to secure computers? The answer is through social and technical means. Social means involve inducing trusted individuals to reveal secrets through bribery, blackmail, or trickery. Technical means include exploiting vulnerabilities stemming from misconfiguration, and latent application and operating system bugs. Telcordia Technologies Inc. and the State University of New York at Stony Brook are researching an approach that gives computer users new capabilities for defending against exploitation of application security vulnerabilities by allowing rapid development and deployment of real-time defenses.

Application security vulnerabilities are a serious technical problem. While honest developers try to create secure applications, time-to-market pressures leave them struggling to implement basic functional requirements with little time left for security. Even worse, malicious developers add security vulnerabilities for future exploitation. An army of malicious hackers is ready to study each application to find all vulnerabilities, and develop and disseminate exploitations [NIT98]. Many software vendors have a reactionary approach to security and issue corrections only after successful attacks [MF98] and many system administrators have neither the time, skills, nor inclination to monitor security-related advisories and to apply patches in a timely manner.

To illustrate how little attention vendors give security, consider that the stack (or buffer) overflow attack [ALE96] still occurs in new applications more than a decade after its first exploitation [SPA89], even though vendors can easily avoid it. The stack overflow vulnerability is merely failure to check an input's length before copying it to a fixed-sized buffer. Attackers can exploit the vulnerability to trick applications into executing arbitrary programs by supplying an input that exceeds the buffer size and contains machine operation codes. Stack overflow vulnerabilities should be eradicated by now, but Computer Emergency Response Team (CERT®) advisories show that stack overflow vulnerabilities account for many new exploitations [CERT98]. Even after vendors issue corrections for discovered stack overflow vulnerabilities, hackers can still find vulnerable applications because administrators do not apply the corrections. Failure to eliminate simple vulnerabilities like stack overflow suggests that more difficult vulnerabilities, like race conditions [Bishop96] that are based on incorrect assumptions about concurrency, will also persist.

While the stack overflow vulnerability is an application bug, intruders cannot exploit it to gain an advantage unless configuration problems compound it. Assuming an intruder achieves access as a normal user (perhaps through cracking weak user passwords), the intruder's damage-causing potential is limited by normal user access control. But, if the intruder can escalate from normal user to super-user, then the intruder wins, since super-user has unlimited access to all computer resources. This is where failure to adhere to security principles [LIN76] in particular principles of least-privilege, and small, switchable protection domains compound application vulnerabilities. Security requires that an application have no more privilege at any time than that required for its functionality at that time, but applications are often implemented to run at only one privilege level, and installed to run at the highest privilege level, since this simplifies development and installation. If an intruder can find one privileged application containing a vulnerability then the intruder can achieve unlimited access to the victim computer. In the homogenous environment of most Internet-connected computers, numerous applications with the two necessary attributes exist.

Current practice for preventing exploitation of application vulnerabilities comprises several defensive lines. The first line eliminates vulnerabilities through better development practices. Developer education coupled with tools that detect security vulnerabilities can eliminate some application vulnerabilities. Two such tools are a source code

CERT® is a registered trademark and service mark of Carnegie Mellon University.

scanner that detects race condition vulnerabilities [BISHOP96] and compilers that render stack overflow vulnerabilities unexploitable [COWAN et al]. The first defensive line only applies to applications for which source code is available. Since there are legacy and proprietary applications that must meet security requirements, but for which source code is not available, additional defensive lines are needed. A second line of defense is to monitor an application's input, since many exploitations require input that deviates from input encountered during normal use. For example, the lengthy and binary nature of exploiting a stack overflow vulnerability is usually atypical. Firewalls are one example of the second line of defense. The difficulty with input checking is that distinguishing legitimate and attacking input is not always possible or efficient. The third line of defense—the main topic of this report—detects and prevents exploits by monitoring an application's runtime behavior [SBS98]. A fourth defensive line looks for the residue of successful attacks by periodically examining file systems, log files, etc. [KS94].

The Survivable Active Networks (SAN) project combines host and network based intrusion detection and reaction technologies to enable real-time defenses against intrusions regardless of the intruders' points of origin. Intrusion detection is based upon specifications of acceptable and unacceptable events, which, in contrast to signature-based approaches, increases the likelihood that novel attacks are detected, and in contrast to anomaly based approaches, decreases the probability of false positives. Intrusion reaction is achieved by interposing compensating events to prevent intrusions from causing damage.

The concepts of the SAN project are embodied in a research prototype comprising three major components: the System Call Monitoring System (SMS), the Network Monitoring System (NMS), and the Active Network Reaction System (ANRS). The SMS, NMS, and ANRS collaborate with each other (and with other security systems) using the Common Intrusion Detection Framework (CIDF). While SAN concepts generalize to any computing platform, the research prototype is currently limited to the Linux OS.

2 Overview of Approach

The primary goal of the SAN project is to develop techniques for making systems *survivable*. Such systems continue to provide their critical functions even when they are subject to coordinated attacks launched by skilled adversaries. To build survivable systems, attacks must be detected and reacted to *before* they impact system performance or functionality. Previous research on survivable systems was directed primarily on detecting intrusions, rather than on preventing or containing damage due to intrusions. In contrast, the approach developed in SAN is aimed at *early* detection of attacks, and developing *automated response actions* that prevent or contain potential damage due to the attacks.

The SAN intrusion-detection component uses a *specification-based approach* that is characterized by (positive) specifications of intended system behaviors. Deviations from specified behavior are deemed to indicate attacks. Being based on intended behaviors, a specification-based approach enjoys the main benefit of anomaly detection, namely, the ability to detect novel attacks. At the same time, this approach is able to avoid a pitfall of anomaly detection, namely, a high rate of false positives. This is because the SAN specifications characterize *legitimate* system behaviors rather than typically observed system behavior. Anomaly detection techniques, on the other hand, can trigger false alarms when legitimate but previously unseen behavior is exhibited by the system.

SAN incorporates a multi-tiered response mechanism that addresses the conflicting requirements of speed and coordination in intrusion response. The host-based response mechanisms of SAN constitute that first tier, and they enable responses to be launched even before attacks cause any damage. These responses are typically aimed at preventing an attacker from compromising resources on a host, but do not address problems such as loss of network bandwidth due to an attack. A second tier in the SAN response mechanism is located at the external gateways to the local network of the protected site. The responses in this tier are aimed at filtering out packets (suspected of) carrying attacks, and thus provides a suitable place for countering repetitive attacks, e.g., many denial-of-service attacks. Responses in this tier are slower than host-based responses, since they require an accurate characterization of attack carrying packets. The third tier of response in SAN supports network-wide coordination of responses. This tier uses active-networking-enabled routers to successively trace back and filter out attacks closer and closer to their source(s).

The detection and response mechanisms of SAN are currently implemented as three independent subsystems:

- At the level of individual hosts, intrusion detection and response is embodied in the System-call Monitoring System (SMS) that monitors the system calls made by processes on the host with application-specific intrusion detection programs and interposes corrective functionality for detected intrusions.

- At the level of the local area or enterprise network, intrusion detection and response is embodied in the Network Monitoring System (NMS) that monitors network packets[1].

- Network-wide response is coordinated by the Active networking based Response System (ANRS).

Intrusion reports and response actions of these three components are coordinated by a higher-level manager application. The interface between the manager and the above three components is based on the Common Intrusion Detection Framework (CIDF). At this time, only a rudimentary manager application has been implemented, since it was not the focus of SAN effort.

The rest of this report is organized as follows. In Section 3, we present the SAN event model, and describe our Behavior Monitoring Specification Language (BMSL) for characterizing system behaviors in terms of patterns over event sequences. We illustrate BMSL through several examples. We then develop the semantics of BMSL and describe an efficient algorithm that we have developed for matching system behaviors against patterns specified in BMSL. Sections 4, 5 and 6 describe the three main components of SAN, namely, the SMS, NMS and ANRS. Section 7 details our experimental efforts to evaluate the effectiveness and performance of SAN. Relationship to prior work is discussed in Section 8. We conclude the report in Section 9 by highlighting SAN project accomplishments, technology transfer efforts and directions for future work.

3 SAN Behavior Specification Framework

In SAN, we characterize behaviors in terms of *events* that can be observed at well-defined interfaces within the system. Examples of such events include the following:

- System calls made by a process to access OS facilities,

- Network packets received or transmitted by a host or a gateway,

- Alarms produced by a network or system management system,

- Error messages produced by applications, and

- Intrusion reports sent by one component of an intrusion detection system to another

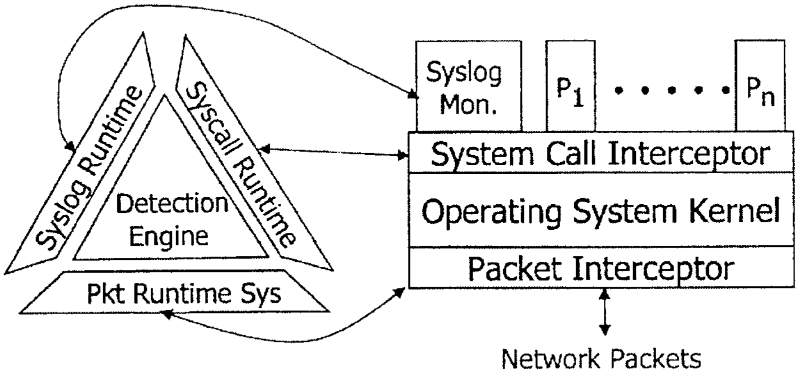
All these events are characterized by an *event name* and zero or more *event parameters*. Moreover, such events are generated by one component of the system (e.g., a process running on a host) and consumed by other system components (e.g., the OS running on a host).

Under normal operation, events are generated and consumed on a continual basis, leading to the notion that operational behaviors of systems can be characterized in terms of sequences of such events. Based on this notion, we specify normal system behaviors in our BMSL language using patterns that are satisfied by event sequences that correspond to normal execution. Real-time responses to intrusions are incorporated in this framework via *event interposition*. This view also enables SAN to modify events after they are generated but before they are consumed. The main benefit of our event-based behavior model is that it does not require rewriting of software running on the

[1] Although the NMS design is intended to support modification or dropping of packets, this capability is not supported in our current prototype.

systems to be protected. Since we are observing interactions at well-defined interfaces, and modifying them as the events cross these interfaces, it is possible to implement our system without making any changes to the systems that produce or consume the events.

In our approach, we compile BMSL specifications into a *detection engine*. Efficiency of this detection engine is critical in order to support real-time intrusion response. The following figure shows a detection engine compiled from a BMSL specification, and its relationship to runtime environments that are responsible for event interposition. Three such runtime environments are shown in the figure, for system calls, network packets, and for messages logged by system applications into the UNIX® syslog file. Note that, in general, there may be multiple detection engines within the system to be protected, and each such detection engine may interact with one or more runtime environments. For instance, the SMS system in our implementation consists of a detection engine that interacts with a runtime environment for intercepting system calls. There is one instance of SMS for each host within the system to be protected. The NMS system consists of a detection engine together with a runtime environment for intercepting network packets. While it is possible to deploy one instance of NMS on each host, a more typical configuration is one that deploys NMS at the host(s) that act as the external gateway(s) to the protected system.



The rest of this section is organized as follows. We first describe the language features of BMSL and illustrate it with examples. We then describe our runtime model for monitoring behaviors, and present our algorithm for translating BMSL specifications into efficient detection engines.

3.1 BMSL Overview

The principal goals in the design of BMSL are:

- *Extensibility* to support multiple event types such as system calls and network packets, and to support data types corresponding to event arguments,
- *Robustness and type-safety*, to reduce specification errors, as well as the scope of damage that may result due to such errors,
- *Simplicity*, to control language and compiler complexity,
- *Amenability to efficient monitoring*, to reduce overheads for runtime behavior monitoring, and
- *Ability to specify responses*, to allow automatic initiation of reactions to prevent and/or contain damage.

The primary mechanism for achieving the first two goals is the development of a suitable type system for the language. The next two goals are achieved by using a simple but expressive pattern language. The final goal is

UNIX is a registered trademark licensed exclusively through X/Open Company Ltd.

achieved by associating each security property with the reaction to be taken when the property is violated. We describe the components of BMSL below.

3.1.1 BMSL Data Types

BMSL types consist of primitive types such as integer, Boolean and float, event types that capture the structure of events, and aggregate types to capture the structure of system call arguments and network packets. BMSL also supports various utility data structures such as lists, arrays and tuples, although these types are yet to be supported in our prototype compiler. We confine the description below to the types that are unique to BMSL.

3.1.1.1 Events

Events may be *primitive* or *user-defined.* Primitive events are those events that are captured by a runtime environment and fed to the detection engine. Primitive event declarations are of the form

```
        event eventName(parameterDecls)
```

where *parameterDecls* is a list of declarations specifying the types of the parameters to the event *eventName.* A primitive event may correspond to a system call or the transmission or reception of a network packet. It is also possible to inject higher-level information into the detection engine by building the appropriate runtime system to provide such information. For instance, the declaration

```
        event telnetConn(unsigned int client, unsigned int server, String
username)
```

may denote an event that is generated by a telnet server on completion of a telnet connection.

User-defined events are *abstract* events that correspond to the occurrence of (potentially complex) sequences of primitive events. They have the form

```
        event eventName(params) = pat
```

where *pat* is an event pattern described in Section 3.1.2, and *params* is a list of parameters to the abstract event. No type information needs to be provided for these parameters, since it is inferred automatically from the manner in which they are used in *pat.* In order to ensure that the values of *params* are well defined at runtime, we require that all of these variables appear in *pat.*

An event abstraction is a convenience mechanism allowing programmer definition of abstract events comprising arbitrary event patterns. Event abstractions allow the programmer to name and treat complex event patterns as if they were primitive events. To illustrate the use of event abstractions, note that many UNIX system calls have overlapping functionality. When we write behavioral specifications, it is cumbersome to write several variants of the specification based on the exact system calls used by a particular program. For convenience, we group similar system calls so that all of the calls in one group can be viewed as implementations of a higher level abstract system call. For instance, the `creat()` and `open()` system calls can both be used to open new files, so we define the abstract event `writeOpen`, which captures this commonality. Then, a single behavioral specification using `writeOpen` can be used to monitor processes that open new files using either `creat()` or `open()`.

```
event writeOpen(path) =
  open(path, flags) |
    flags&(O_WRONLY|O_APPEND|O_TRUNC)||
  open(path, flags, mode) |
    flags&(O_WRONLY|O_APPEND|O_TRUNC)||
  creat(path, mode);
```

3.1.1.2 Packet Types

Type systems in existing languages are not sufficiently expressive to model network packets. In particular, the following problems arise in describing network packet structures:

- the compiler or runtime system for the language does not have the freedom to choose a runtime representation for the types; rather, the representations are prespecified as part of protocol standards, and
- the complete type of a network packet can be determined only at runtime, so type checking cannot be completed at compile-time.

One way to address the problem is to treat the packet as a sequence of bytes. For instance, a reference to the protocol field of an Ethernet header in a packet p may be expressed using C-like syntax as (short)p[12]. However, identifying packet fields using offsets is inherently more error-prone than one based on naming the fields. Moreover, we lose the benefits provided by a strong type system, such as protection against memory access errors or misinterpretation of the contents of a field, e.g., if an integer field is accidentally treated as a short or a float.

We have developed a new type system that can capture complex packet structures, while providing the capabilities to dynamically identify packet types at runtime and perform all relevant type checks before the packet fields are accessed. An Ethernet header is defined in BMSL as follows:

```
ether_hdr {
    byte  e_dst[ETH_LEN]; /*Ethernet destination*/
    byte  e_src[ETH_LEN]; /*and source addresses*/
    short e_type;   /*protocol of carried packet*/
}
```

To capture the nested structure of protocol headers, we employ a notion of inheritance. For instance, an IP header can be defined as follows:

```
ip_hdr: ether_hdr {
   bit   version[4]; /* ip version */
   bit   ihl[4];     /* header length */
   ...               /* several fields skipped */
   unsigned saddr, daddr; /* Src and Dst IP addr */
}
```

Similarly, a TCP header inherits all of the data members from IP header (and Ethernet header). However, simple inheritance is neither powerful nor flexible enough to satisfy our needs. In particular, the structure describing a lower layer protocol data unit typically has a field identifying the higher layer data that is carried over the lower layer protocol. For instance, the field e_type specifies whether the upper layer protocol is IP, ARP, or some other protocol. To capture such conditions, we augment inheritance with constraints. The structure for IP header with the constraint information is as follows.

```
ip_hdr: ether_hdr with e_type=ETHER_IP {
   ... /* other fields same as before */
}
```

Finally, we need to deal with the fact that the same higher layer data may be carried in different lower layer protocols. For this purpose, we develop a notion of *disjunctive inheritance* as follows. To capture the fact that IP may be carried within either an Ethernet or a token ring packet, we modify the constraint associated with ip_hdr into:

```
      (ether_hdr with e_type=ETHER_IP) or
        (tr_hdr with tr_type=TOKRING_IP)
```

Disjunctive inheritance asserts that the derived class inherits properties from exactly one of many base classes. This contrasts with traditional notions of single inheritance (where a derived class inherits properties from exactly one base class) and multiple inheritance (where a derived class inherits the properties of multiple base classes). Viewed alternatively, multiple inheritance would correspond to a conjunction of constraints, whereas disjunctive inheritance corresponds to an exclusive-or operation.

3.1.1.3 Class Types

It is not appropriate to describe and manipulate some of the data that is exchanged over the detection engine's interfaces using built-in or record types, because the concrete representation may be unknown or hidden. For instance, in the case of system calls, an argument may be a pointer that resides in the virtual address space of the process being monitored, and thus may not even be accessible within the detection engine. For these reasons, we introduce the concept of class types (defined by the keyword class) that are essentially abstract data types. The representation of class data is completely encapsulated and invisible to BMSL, and can be manipulated only via operations defined as part of the data type. Sample declaration for a class that corresponds to C-style strings and another class that corresponds to the argument to the `stat` system call in UNIX are shown below:

```
 class CString {
             string getVal() const;
             void setVal(string s);
 }
 class StatBuf {
             int getDev()const;
             int getIno()const;
             int getMode()const;
              ⋮
             int getAtime()const;
             int getMtime()const;
             int getCtime()const;
}
```

Note that the return type of a member function could itself be a class type. Whether a member function changes the value of the object or not is given by the `const` declaration associated with the function. This plays an important role in type checking of BMSL patterns.

3.1.1.4 External Functions

External functions are functions that are defined outside of the detection engines, but can be accessed from the detection engines. Semantically, they are no different from member functions associated with foreign types. In other words, member functions are simply external functions that use a different syntax.

The primary purpose of external functions is to invoke support functions needed by the detection engine or reaction operations provided by the system call interceptors. For instance, when an event for opening a file is received by a detection engine, the detection engine may need to resolve the symbolic links and references to "." and ".." in the file name to obtain a canonical name for file. The detection engine may use a support function declared as follows to find the canonical file name:

```
string realpath(CString s);
```

The detection engine may also need to check the file's access permissions, which may be done using a support function declared as follows:

```
StatBuf stat(const Cstring s);
```

In BMSL system call names either represent an event (i.e., invocation of a system call by a monitored process) or are a component of a reaction taken by the detection engine (i.e., a statement in the a reaction program). We use the same syntax for system calls in both cases, since the context resolves any ambiguity.

3.1.2 Patterns

Security-relevant properties of programs are captured as patterns over sequences of events such as system calls and network packets. An *atomic pattern* is of the form $e(a_1,...,a_n)\,|\,C$, where *e* denotes an event and *C* is a boolean-valued expression on $a_1,...,a_n$. *C* may contain standard arithmetic, comparison and logical operations. C may also contain comparisons of the form *x* = *expr* where *x* is new variable, with the semantics being that of binding the value of *expr* to *x*. A *primitive pattern* is obtained by combining atomic patterns with the disjunction operator `||`, and possibly preceding the entire expression with the complement operator '`!`'. As an example of a primitive pattern, consider:

```
!((open(f)|realpath(f)="/home/*/.plan") || (close(f)) ||(exit(f)))
```

In this pattern, a shorthand notation `/home/*/` is used to refer to any directory that is immediately contained within `/home`. The above primitive event pattern captures all system calls other than those for opening ".plan" files, closing files or terminating processes.

General event patterns are obtained by combining primitive patterns using temporal operators. Such operators enable us to capture sequencing or timing relationships among system calls. Sequencing operators are similar to those used in regular expressions, but operate on events with arguments. We refer to our pattern language as regular expressions over events (REE) to indicate this relationship.

- *Sequential composition:* $p_1; p_2$ denotes the event pattern p_1 immediately followed by pattern p_2.
- *Alternation:* $p_1 \,||\, p_2$ denotes the occurrence of either p_1 or p_2.
- *Repetition:* $p\{n_1,n_2\}$ denotes at least n_1 repetitions and at most n_2 repetitions of p. $p\{n_1\}$ and $p\{n_2\}$ are shorthand for $p\{n_1,\infty\}$ and $p\{0,n_2\}$ respectively. The notation $p*$ is shorthand for $p\{0,\infty\}$.
- *Real-time constraints:* p *within* $[t_1,t_2]$denotes the occurrence of events corresponding to pattern p occurring over a time interval. The shorthand for $[0,t]$ is $[t]$, whereas the shorthand for $[t_1,\infty]$ is $[t_1,]$.
- *Atomicity: nonatomic* d *in* p corresponds to an occurrence of pattern p within which the data item d is not accessed atomically.

Note that most of these operators are similar to those used in regular expressions—the only difference is that we are trying to capture patterns on sequences of events with arguments, whereas regular expressions capture patterns on sequences of symbols. For this reason, we call our pattern language as regular expressions over events (REE).

To avoid excessive use of parenthesis, we define the following associativity and precedence for the temporal operators. The operator || and ';' associate to the left. The operator ! has the highest precedence, '*' has the next lower precedence, ';' has the next lower precedence and || has the lowest precedence.

For convenience, we define the operator ".." that can be applied only to primitive patterns. $p_1..p_2$ is equivalent to $p_1;(!(p_1 \| p_2)*);p_2$, i.e., p_1 followed by p_2 with possibly other events occurring in between. The restriction that ".." be applied only to primitive patterns is imposed since the operator has unintuitive semantics on general event patterns.

We illustrate the use of temporal operators using several simple examples below. Note that in general, we wish to take reactive action when the behavior of a monitored process fails to satisfy certain properties. Hence, we typically develop patterns that are the negation of assertions describing normal behaviors.

e1;!e2*;e1 asserts that e1 must occur twice with no intervening e2. This corresponds to the negation of the property that e1 must always be followed by e2 before a second occurrence of e1.

e1;!e2*) within [t,] captures violation of property that e1 is followed by e2 within time t

e1;!e2*;e3 captures violation of property where e2 must always occur between e1 and e3

e{k} within [t] captures violation of property that e occurs less than k times within time t.

A precise treatment of BMSL pattern semantics can be found in [SU99] and [Uppuluri00].

3.1.3 Rules

To enable response actions to be launched, BMSL enables actions to be associated with BMSL patterns. We use the syntax *pat* → *action* to denote a response *action* that is launched when the pattern *pat* matches the event sequence observed at runtime. The reaction component consists of a sequence of statements, each of which is either an assignment to a state variable or invocation of an external function provided by the runtime environment. The important classes of response actions are specified below.

3.1.3.1 Data Aggregation Operations

To identify some classes of attacks it is often necessary to aggregate information across many events, and act on the basis of the aggregate information. Such aggregation needs to be more sensitive to recent events than older ones. BMSL supports two principal abstractions for such aggregation, decay counters and most-frequently-used (MFU) tables.

Decay counters are characterized by a *time window* and a *decay rate*. Increments of the decay counter that occurred beyond a time window are not included in the output of the counter. Moreover, increments that occurred in the past within the time window are weighted using an exponentially decay function that assigns lesser weight to events that occurred in the past.

MFU tables are hash tables where each entry has a decay counter that keeps track of the number of times the entry has been accessed in the past. The table is of a fixed size, with overflows handled by deleting the entries with the lowest counts. Being based on decay counters, the notion of most-frequently-used is tilted in favor of entries that have been accessed recently over those accessed a long time in the past. MFU table entries can be associated with functions that are to be invoked when the entry's count increases above (or falls below) a threshold, or when the entry is purged from the table.

Key features of the decay counter (and MFU table design) are that it uses constant memory per counter (one per MFU table entry), and operations to increment or decrement the counter (insert or delete entries into the table) take constant time. Thus the data aggregation operation is both time and space efficient. For a more detailed presentation of data aggregation operations, the reader is referred to [SGSV99].

3.1.3.2 Event Modification Operations

Event modification operations are realized as a set of external functions provided by runtime environments. Thus the event modification capabilities will differ for different runtime environments. For instance, a packet runtime system may provide operations to drop, generate or modify packets. (Our current packet runtime implementation, however, does not support response operation.) Similarly, our system call runtime system provides operations to prevent a system call from executing and/or to return a fake return value in response to a requested system call.

3.1.3.3 Interactions Among Multiple Rules

If multiple patterns match at the same time, the associated reactions of each matching pattern are launched, leading to a problem if some of launched reactions conflict. We could solve the conflicting reactions problem by (a) defining a notion of *conflict* among operations contained in the reaction components of rules, whether they be assignments to variables or invocation of support functions provided by the runtime system, and (b) stipulating that there must not exist two patterns with conflicting operations such that for some sequence of system calls, they can match at the same point. Potential conflicts can be identified by the automaton construction algorithms developed in [SU99]—if there is any state in the automaton that corresponds to a final state for two such patterns, then there is a potential conflict. However, we have not implemented this solution yet, and currently rely on the specification writer to deal with conflicts.

3.1.4 Modules

BMSL specifications are structured as a collection of modules, each of which consists of a collection of *state variables* and *rules*. State information can be retained across multiple rules within a module via the state variables. As an aid to programmability, modules may be parameterized. Parameterization enables specification of abstract behaviors that can be customized by providing values for these parameters. A typical use of parameterization is to allow a general-purpose module to be used in similar situations that differ only in a few minor details. The process of generating a compilable module from a parameterized module is known as module *instantiation*.

Another important role of modules is that they provide a mechanism for dynamically altering the degree of monitoring, possibly in response to suspicious events. In particular, the action `switch ModuleName` can be used to start monitoring with respect to a module named `ModuleName`. It is also useful when a process uses the `execve()` system call to overlay itself with a new program. The `switch` action can then be used to perform monitoring that is appropriate for the new program. Finally, if a process is discovered to be compromised, we can alter the behavior of future system calls made by the process in such a fashion as to isolate the process from the system. This may also be accomplished by switching to a new specification.

3.2 Example Behavior Specifications

To restrict a process from making a set of system calls, we create a rule whose pattern matches any of the disallowed system calls and whose reaction causes the disallowed system calls to fail. For instance, we may wish to prevent the finger service program from executing any program, modifying file permissions, creating files or directories or initiating network connections. We may also want to restrict the files that may be opened by the program. The following example shows such a specification. We use the shorthand notation of omitting some of the arguments of a system call (or replacing them with "...") when we are not interested in their values.

```
open(file, mode)|
  ((f = realpath(file)) &&
   ((f != “/etc/utmp”) &&
    (f != “/etc/passwd”) &&
    !inTree(f,”/usr/spool/finger”)) ||
    (mode != O_RDONLY))
  -> fail(-1,EACCESS)

  execve || connect || chmod || chown
    || chgrp || create || truncate
    || sendto || mkdir
  -> exit(-1);
```

The following example illustrates sequencing restrictions by specifying that a process should never open a file and close the file without reading or writing the file. 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, the presence of variables in REE enables us to capture the close system call matching an open.

The example below illustrates the use of atomic sequence patterns. A popular attack uses race conditions in setuid programs as follows. Since a setuid process runs with effective user root, any open system call by the process succeeds or fails base on the file privilege with respect to `root`. If the setuid process wishes to open a file with the respect to permissions of the real user, it first uses the `access` system call to determines if the real user has access to the file, and if so, it opens the file. The attacker exploits the time window between the access and open system calls by creating a symbolic link as the name of the file in question, and changing the target of the link between the access and open system calls, to be a file that is inaccessible to the real user. To prevent race attack, we ensure that the file referred by the `access` and `open` system calls is accessed atomically:

```
nonatomic(f.target) in (access(f);(!open(f))*; open(f)) → fake(EACCES)
```

3.3 Compilation of BMSL

3.3.1 Type Checking

BMSL is designed with the idea that code generated from BMSL specifications may run within operating system kernel space. This means that the code generated from BMSL (and hence BMSL itself) must be robust and guard against serious errors such as invalid memory accesses or other exceptions that could contribute to failures of individual hosts or legitimate processes running on them. Another factor is that a hacker planning to attack a host is likely to first try to cripple the survivability components on the host, and hence it is important to make these components very robust.

3.3.1.1 Packet Types

The semantics of the constraints is that they must hold before we access fields corresponding to a derived type. In particular, note that at compile time, we will not know the actual type of a packet received on a network interface, except for the lowest layer protocol. For instance, all packets received on an Ethernet interface must have the header given by `ether_hdr`, but we do not know whether they carry an ARP or IP packet. To ensure type safety, the constraint associated with the `ip_hdr` must be checked (at runtime) before treating the packet as an IP packet and accessing the relevant fields. Similarly, the condition protocol == IP_TCP must be checked before treating an IP packet as a TCP packet and accessing the relevant fields. More generally, before a field in a structure of a particular type *T* is accessed, all constraints associated with all of the base types of *T* need to be checked. Based on the type declarations, our implementation automatically introduces these checks into the specifications, thus relieving the programmer of the burden to check these constraints explicitly.

Type checking tasks that are specific to packet types involve name resolution and constraint enforcement. Name resolution refers to the problem of identifying the entity referred by an expression such as "a.b." Name resolution is complicated in the case of packet types by the fact that in an expression of the form a.b, we may not know the exact type of a, but only the base class to which a belongs. To illustrate the problem, suppose that we have declarations of the form

```
event ethRx(ether_hdr p); event tokRx(tr_hdr p);
```

This declaration defines an event to denote packet reception on each type of network interface. Reception of a packet will result in the generation of this event, with the packet contents passed as a parameter to the event. Now consider the rule:

```
ethRx(p) | (p.tot_len < 20) && (p.tcp_sport = 80) -> ...
```

At compile time, based on the declaration of the event ethRx, we only know that the type of p is ether_hdr. With this information, if we attempt to resolve p.tot_len, there will be an error, since ether_hdr contains no such field. Reporting this fact as an error is undesirable. It would be preferable to check if the packet is in fact an IP packet at runtime, and if so, proceed to examine the p.tot_len field. We extend the name resolution process in BMSL so that it uses the field name information to infer the runtime type of p. Specifically, when we see an expression of the form a.b, we search for the field b in the (declared) type of a and all its subtypes. Using this process on the expression

p.tot_len, we can infer that the type of p is ip_hdr. On encountering the expression p.tcp_sport, we will further refine the type of a to be a tcp_hdr. When the type of p cannot be determined uniquely using this process, the type checker will return an error. This would happen only when the declared type T of the packet is such that two descendent classes of T use a field with the same name. To disambiguate such cases, the event arguments can be further qualified to indicate the runtime type of an argument:

```
ethRx(tcp_hdr p) | (p.tot_len < 20) && (p.tcp_sport = 80) -> ...
```

Finally, although a pattern may implicitly assert that packets being processed by that pattern will be of a certain type, we need to verify this fact at runtime. More generally, before accessing a field f in a packet, we need to check all constraints associated with the type T containing f and all its ancestor types. For instance, in the above pattern, before we access the field p.tot_len, which is a field in the class ip_hdr, we need to verify the constraint p.e_type = ETHER_IP associated with the ip_hdr type. Similarly, before accessing the field p.tcp_sport which is a field in the tcp_hdr type, we need to verify the constraints associated with this type, i.e., p.protocol==TCP. As part of type checking, we explicitly add these constraints to the event patterns, so that they would be checked at runtime. We also introduce checks to ensure that the packet length is large enough that all offset accesses fall within the packet. To ensure that the preconditions are indeed checked at runtime before accessing a particular field, the ordering among the newly introduced conditions and the original conditions in the pattern have to be maintained. We do this by introducing a new *ordered conjunction* operation &&& as follows. The semantics of a&&&b is that the condition a needs to be checked first, and only if it is true, b will be checked. (Note that the ordinary conjunction operator does not impose an ordering on the way its operands are evaluated. The reordering permitted by && plays an important role in improving the performance of pattern-matching algorithms.)

After explicit addition of constraints during type checking, we obtain the following pattern:

```
ethRx(p)| ((length(p)>=offset(p.tcp_data))&&&(p.e_type==ETHER_IP)&&&(p.tot_len<20)) &&
          ((p.e_type = ETHER_IP)&&&(p.protocol = IP_TCP)&&&(p.tcp_sport = 80)) -> ...
```

Note that the same constraint may appear multiple times in the pattern at this point, but later stages of the compiler will ensure that no constraint is checked more than once.

In the presence of disjunctions in the constraint, such as

```
tcp_hdr: ether_hdr with e_type=ETHER_IP or tr_hdr with tr_type=TOKRING_IP{

   ...

}
```

recall that the alternatives in the condition are mutually exclusive. For instance, in the above example of `ethRx`, we will be able to determine statically that the packet type is `ether_hdr` and not `tr_hdr`. Based on this, the constraint regarding `tr_hdr` is not applicable and can be discarded.

3.3.1.2 Class Types

As mentioned earlier, BMSL class types may correspond to data that resides outside the detection engine. References to such data have to be represented in BMSL as pointers or handles into the memory space of a runtime system or a process being monitored. This means that class data referenced by BMSL may get overwritten, or become invalid in between two events due to operations taking place in the runtime system or the monitored process. Moreover, this happens without the detection engine's knowledge, and may lead to memory access errors, or at the least, have unexpected effects on the specifications due to unanticipated changes. We therefore impose the restriction that class data cannot be stored in BMSL variables across the delivery of multiple events. We also require that any external functions applied to class data should assure that this data would not be changed by the function. If we want to store one or more components of some foreign object in BMSL, we need to convert the components of interest to native BMSL types by copying them from the foreign address space using the appropriate accessor functions on the foreign object.

3.3.2 Compilation of Pattern-Matching

Efficient pattern matching is key to the performance of our detection engines. Our approach to pattern matching is based on compiling the patterns into a kind of automaton in a manner analogous to compiling regular expressions into finite-state automata. We call these automata *extended finite-state automata* (EFSA). An EFSA is similar to a finite-state automaton, with the following differences:

In addition to the control state of an FSA, an EFSA can make use of a fixed set of state variables. The EFSA makes transitions based on events, event arguments and conditions on event arguments and state variables. The transitions may assign new values to state variables.

An EFSA may be deterministic (DEFA) or nondeterministic (NEFA). For the sake of efficiency, we always prefer to generate a DEFA rather than a NEFA. However, this is not always possible as conversion of NEFA into a DEFA can cause unacceptable explosion in space requirements. For traditional FSA, every nondeterministic automaton can be converted into an equivalent deterministic automaton with at most an exponential increase in the number of (control) states. For performance critical applications (e.g., lexical analysis phase of a compiler), this increase in state space is quite acceptable, especially because the worst-case behavior is unusual. For EFSA, the explosion in size is exponential in the product of the number of control states and the range of values that can be assumed by each of the auxiliary state variables. For instance, a deterministic EFSA that is equivalent to a nondeterministic

EFSA with one integer (32-bit) state variable and N control states can have $2^{N*2^{32}}$ states! This problem leaves us with two choices:

- Restrict the class of BMSL patterns so that they can be compiled into DEFA, or
- Do not convert an NEFA into an EFSA, and simulate the NEFA at runtime.

Note that at runtime, the transitions of an EFSA are represented in code, whereas its current state (which includes the control state and the state variables) is stored in a data structure. Since we plan to combine all patterns in one ASL specification into a single EFSA, there is only one instance of the transition relation at runtime. To support nondeterminism, we permit multiple instances of the dynamic state of the EFSA. These multiple instances capture all of the states the NEFA could have reached after examining its input up to this point.

If an EFSA needs to make a two-way nondeterministic transition on an event e, we perform a "fork" operation on the EFSA, i.e., replicate its current state. The replica follows one of the non-deterministic choices, while the parent follows the other choice. This approach can lead to an unbounded increase in the number of instances of EFSA, but unbounded growth should happen only when certain unusually repetitive sequences of system calls are observed at runtime, and hence is not a serious issue in practice. We are currently working on techniques that can avoid unbounded growth by restricting the class of patterns permitted in BMSL.

The starting points for our algorithm for generating EFSA from BMSL patterns are the seminal papers by Brzozowski [Brzozowski64] and Berry and Sethi [Berry86]. However, these papers address regular expressions and classical FSA, whereas we must address conditions on event arguments and state variables that can be complex data structures. Our earlier work on first-order term matching [Sekar95] provides the starting point for addressing this aspect. By combining and extending these two techniques, we developed an algorithm for generating EFSA from BMSL patterns.

To address the problem of size explosion in DEFA, we have devised a new class of EFSA called quasi-deterministic extended finite state automata (QEFA). QEFA eliminate most of the sources of nondeterminism that are present in the NEFA, while still ensuring that their sizes are acceptable. A complete treatment of QEFA and the compilation algorithm can be found in [SU99].

The QEFA is then translated into C++ class in by the BMSL compiler as follows. Specifically, one class is generated from each BMSL specification. This class has one member function for each event. These member functions have the same number and types of arguments as the event. A detection engine is simply an object belonging to this class. When the runtime infrastructure intercepts an event, it delivers it to the appropriate detection engine by invoking the corresponding member. For instance, the runtime infrastructure invokes the open_entry method when a monitored program enters an open system call, and the open_exit method when the process is about to exit this system call.

The transitions in the EFSA are translated into code as follows. We maintain a list of active EFSA instances at runtime. When an event is delivered, we go through the list of EFSA instances and for each of them, make a transition based on its current state and the newly delivered event. If multiple transitions exist out of the current EFSA state for this event, then copies of the EFSA are made (using the fork operation mentioned earlier), so that there is one EFSA to make each of these transitions. If there is no transition for an EFSA instance, then it is "killed" and any resources used for the instance are released.

4 Host Based Intrusion Detection/Reaction: The System Call Monitoring System

4.1 Theory Of Operation

The events which the System Call Monitoring System (SMS) deals with are system calls and their arguments. SMS allows the execution of process monitoring specifications that describe acceptable and unacceptable application behavior in terms of system calls, their arguments and the system state that exists when the system calls are requested. The specifications also allow the interposition of defensive programs of arbitrary functionality in response to requested system calls.

4.1.1 Rationale for System Call Interception/Interposition

The key insight motivating the system call interception/interposition approach is that in modern computer systems any damage that an intrusion does to the victim computer occurs only through execution of system calls. The victim computer's Operating System (OS) is responsible for safeguarding system call execution to prevent execution of damage-causing system calls; therefore one of an intruder's goals is to trick the OS into executing damage-causing system calls. The criteria popular OSs use to decide whether or not a system call should execute are one-dimensional—the decision is based on the statically defined privileges of the application requesting the system call. Many popular and powerful intrusion techniques in use today are those which exploit latent errors in a highly privileged application to trick the OS into executing damage-causing system calls. Through application-specific system call interception/interposition, SMS expands the criteria used to safeguard system call execution. Instead of just considering whether or not the application has the required privilege for the requested system call, SMS system call interception allows consideration of additional application-specific criteria such as:

- Is the requested system call in the set of system calls that appears in the application's source code,
- Is the requested system call in the set of system calls that are typically requested by the application when it is being used as intended,
- Is the requested system call consistent with the overall purpose of the application,
- Is the requested system call consistent with the objectives of the current temporal state of the application,
- Will execution of the system call violate any site-specific security policies,
- Is the requested system call operating on resources whose attributes have been changed since the last time the application checked those resources, and
- Does a sequence of system calls being requested by an application match the sequence produced by exploitation of known application vulnerabilities.

Intercepting system calls to change an application's behavior is not new [RS95] and we, and others [MLO97][FBF99] are researching application of the general technique to the specific problem of application-security enhancement. By intercepting system calls, SMS augments the kernel's general-purpose security functionality with application-specific functionality allowing for exploitation detection. In addition, SMS interposition allows a wide variety of damage preventing reactions to requested system calls that fail to meet the application-specific acceptance criteria, such as:

- Immediate application termination,
- Rejection of system call using a legitimate, but faked, error status code,

- Isolation of the application to a specialized environment previously established for containment of compromised applications,
- Reporting of suspicious system calls to security officer and/or higher level security applications, and
- Substitution of corrective actions crafted to allow the application to provide its intended functionality without causing damage.

Neither the list of criteria or reactions should be considered complete. Rather, SMS should be thought of as a general-purpose programming environment and runtime system that enables a programmer to develop system call interception/interposition programs of arbitrary functionality. As such, the possible criteria and reactions are limited more by the programmer's imagination than by preconceived notions embedded in SMS. Likewise, although we have explored the system call interception/interposition concept in only the UNIX operating system derivatives, and have built an experimental prototype for only the LINUX operating system, we believe that the concept generalizes to any modern operating system.

As a programming environment, the SMS programming language, called BMSL, and the BMSL compiler simplify the programming task by providing a rule-based language well-suited to event-based programming, and a compiler well-suited to production of safe, efficient executable programs. As a runtime environment, SMS provides for simple installation of BMSL programs and efficient interception of system calls needed by the BMSL programs.

The SAN project has been conducted as a systems project rather than a theory project, in that we have attempted to prove that the SMS approach is viable by constructing a working SMS prototype and using that prototype to experiment with defense against real-world intrusions. Our experiments have had two effects. First, by learning about known attack methods our original assumption that many attacks are the result of exploitation of application vulnerabilities that system call interception/interposition can detect, remediate and/or respond to has been confirmed. The results of concurrent researchers also confirm the efficacy of system call interception/interposition. Exploration of the known attack space has also led us to realize that our notion of "latent application errors" was actually insufficient. We have discovered that application vulnerabilities stem from a complicated combination of factors including: unintentional application errors, maliciously inserted application errors, application features, application misconfiguration, and site-specific policy. As a programming and runtime environment, SMS empowers programmers to write defensive programs which deal with these factors, and is not restricted to merely remediating latent application errors. Second, we discovered a special case of vulnerability that is important, but not completely addressed by SMS. This special case is errors in OS software, as opposed to errors in application software. Since SMS allows system call interposition, some instances of OS errors can be handled by SMS, but since SMS is driven by system call events, SMS can only handle OS errors if those errors are in system calls. An OS contains software that is not associated with system calls, for example, the software which implements device drivers. SMS can therefore not deal with errors in this non-system call OS software. Consequently, we developed a separate system, called the Network Monitoring System (NMS) to deal with exploitation of some vulnerabilities in non-system call OS software.

While other research directions are exploring similar techniques, SMS research is unique in four ways. The *first difference* is that SMS supports the use of both system calls, **and their arguments**, in specifying acceptable and unacceptable behaviors. While at first this difference may seem to be a minor detail, it is actually not a minor detail at all. The computation complexity associated with allowing behaviors to be specified in terms of both system calls and their arguments required the development of new algorithms to ensure low overhead execution of the specifications. The *second difference* is that our research is primarily aimed towards high-level specifications of acceptable and unacceptable behaviors of specific applications. Other researchers are using the techniques of misuse-specification, in which the signatures of known exploitations are specified, or anomaly detection, in which a pre-deployment training period determines the acceptable behaviors of an application. Misuse specifications tend to be highly accurate at detecting those exploits that exactly match the specified signature, but miss exploits with minor variations. Anomaly detectors tend do better on exploit variations, but have a high false positive rate because real world use may result in many legitimate deviations that never occurred during training. The specification approach relies on a competent programmer to make decisions about the high level acceptable behavior of an application (by examining manual pages, source code, etc), and translate that behavior into a specification based on system calls. Our initial belief that specification based approaches offer an improvement over the detection accuracy of the other approaches has begun to be justified through the evaluations we participated in during the SAN project, however, as the evaluations are themselves not yet mature, our belief cannot yet be considered proven. The *third difference* is that from the start, our approach is intended as a **real-time** approach. Unlike other approaches in which a post-execution [FHS97] examination of system call logs is used to detect exploitations, our approach uses real-time system call interception. The obvious advantage of real-time interception is that a defense can be mounted before damage is done. A less obvious advantage is that the criteria for detection of an intrusion are expanded to include the instantaneous system state, thereby improving the accuracy of detection. (As a simple example, consider vulnerabilities that are exploited through a momentary change of a file's attributes; since the change persists only for a short window of time before and after the intrusion, it may no longer be evident during post-exploit evaluation of log files.) The *fourth difference* is that our approach does not require access to the source code of the application being protected. Some researchers are using source code scanning techniques [BISHOP96] to look for known vulnerabilities in application software. While source code is one asset that can aid a programmer in developing an application-specific defense in the SMS, it is by no means a requirement, as a competent programmer can usually deduce acceptable behavior from user level application documentation and then reduce this behavior to its application-specific system call signature by conduction a few system call tracing experiments. Even if source code is available, end-users may prefer the SMS approach for defending against vulnerabilities to the more obvious choice of source code correction, because it simplifies deployment, especially in emergency situations.

4.1.2 Overall Organization of SMS

Figure 1 shows the overall organization of SMS. A programmer, using knowledge from many sources such as application manual pages and user documentation, application source code (if available), reports of known intrusions, site specific security policies, etc., develops system call signature based algorithms for application-specific intrusion detection and reaction. The programmer implements the algorithms in the BMSL language and uses the BMSL compiler to translate the BMSL programs into C++ programs. A collection of C++ programs is linked together and with other C++ programs that provide system call interception and runtime support to produce a single detection engine, which is installed on the computer to be protected. While the end result is a single detection engine, it is important to separate two distinct components of the detection engine: the defensive programs, and the SMS infrastructure. The defensive programs are those programs which a programmer writes to defend applications, and over which the programmer has control. The SMS infrastructure is that portion which exists to allow execution of the defensive programs and which cannot be changed by the programmer.

Prototype development phases included an early phase, called the `libc` version, in which the detection engine was instantiated as a user-level process, and a latter phase in which the detection engine was instantiated as a Dynamically Loadable Kernel Module (DLKM), called `imod.o`. Both versions have nearly identical functionality, but the user-level version was easier to construct, and allowed early experimentation with the research concepts. In contrast, the kernel level version was harder to construct, but offers more realistic non-functional characteristics. In particular, the user level version is easily bypassable by an attacker, whereas `imod.o` is not easily bypassed. This document describes both versions although it should be noted that we have not kept the older `libc` version up-to-date and that it may not have all of the functionality described.

Another component may also be installed on the protected computer; the CIDF manager, which receives intrusion reports from the SMS and the NMS. The CIDF manager primarily displays intrusion reports in a human readable form, but we have also implemented some elementary correlation capabilities into the CIDF manager, which enable it to improve accuracy by correlating intrusion reports from both the SMS and NMS. In addition, the CIDF manager provides the interface between the NMS and the SMS, and the ANRS, enabling active network reactions to be initiated in response to SMS and NMS intrusion reports.

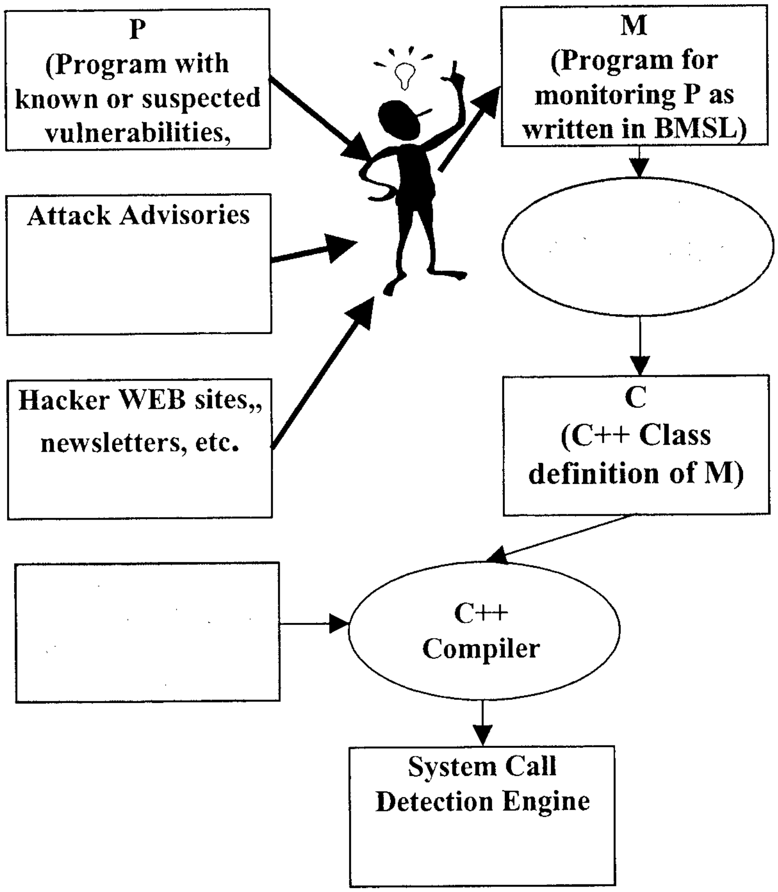

P
(Program with known or suspected vulnerabilities,
Attack Advisories
Hacker WEB sites,, newsletters, etc.
M
(Program for monitoring P as written in BMSL)
C
(C++ Class definition of M)
C++ Compiler
System Call Detection Engine

Figure 1

4.1.3 SMS Inputs

4.1.3.1 System Call Events

Defensive programs written in BMSL for SMS execution are driven by input events. The input events that can be delivered to a BMSL program are system call events. For every system call in the OS system call Application Programming Interface (API) (i.e. section 2 of the Unix user manual) SMS supports two events, system call entry and system call exit. A system call entry event is the interception of a system call after it has been requested by an application, but before the OS functionality associated with the system call has begun to be executed. A system call exit event is the interception of a system call after the OS functionality associated with it has been executed, but before the results of the system call have been returned to the application that requested it.

The need for having two events—exit and entry—associated with each system call, instead of just a single event, may not be immediately obvious, although in latter sections of this report it will be illustrated by example. At a high level, the need for both entry and exit events can be summarized as follows:

- System call entry events are needed since some reactions can only prevent damage if they can prevent the execution of a damaging system call. If it is known that a system call is about to cause damage that cannot be later undone (for example, release of secure information), then the reaction must be inserted prior to system call execution.

System call exit events are needed since some intrusion detection algorithms require the status code of one or a sequence of several system calls in order to detect intrusions.

A single BMSL program written to defend an application has the ability to see all system call entry and exit events that occur during execution of the application. Typically, the functionality of a BMSL program requires it to see only a few system call entry and exit events. The organization of the runtime system recognizes that most BMSL programs need to see only a few system call entry and exit events, and provides a structure that eliminates the programmer's need to deal with uninteresting events. Also, since a slight processing overhead is incurred every time a system call entry or exit event is delivered to a BMSL program, a compile time analysis eliminates interception of those system call entry and exit events that no BMSL program in the detection engine being built needs.

4.1.3.2 System Call Event Data

When system call entry and exit events are relayed to BMSL programs, the events are tagged with data relevant to the system call. The intercepted system call data comprises identity of the intercepted system call event and arguments of intercepted system call. Additionally, for system call exit events, the return value that execution of system call produced is included.

In addition to the data which tags the system call events, as described later, the SMS infrastructure maintains a small collection of data which many BMSL programs need. This data is available to BMSL programs at any time. BMSL programs can also define storage for program-specific data storage.

4.1.4 SMS Outputs

4.1.4.1 Event Disposition

Defensive programs written in BMSL for SMS execution produce output that controls the *disposition* of an intercepted system call. For each of the two input events (system call entry and system call exit), two dispositions, called *normal* and *faked,* are permitted. The semantics are as follows:

1) System call entry event with normal disposition: the intercepted system call is to be executed normally.
2) System call entry event with faked disposition: the intercepted system call is not to be executed, and instead, the return value and output arguments supplied by the defensive program are to be returned to the application.
3) System call exit event with normal disposition: the return value and output arguments produced by the execution of the intercepted system call are to be returned to the application.
4) System call exit event with faked disposition: the return value and output arguments produced by execution of the intercepted system call are not to be returned to the application, instead a return value and output arguments supplied by the defensive program are to be returned.

4.1.4.2 System Call Data Modification

Event disposition, which allows system call interceptors to return normal or faked return codes, provides a basic level of interposition capability. The general-purpose nature of the interceptors augments event disposition with a more powerful capability, that is, the interception programs can be written to actually *change* the input and output arguments of intercepted system calls. The interceptor of a system call entry event may change an input argument (e.g., the file name argument of an open system call), and then use the normal disposition, thus causing the system call to be executed, but with a parameter other than the one supplied by the application. Likewise, output arguments can be changed by system call exit event interceptors, and even though they use normal disposition, the application will see output arguments other than those actually produced by executing the system call. The only constraint pertains to the return value: for a system call exit or entry event to return a value different from that produced by actual execution of the intercepted system call, the faked disposition code must be used. While the constraints on disposition with respect to input and output arguments could be more restrictive (perhaps with the addition of more dispositions) thus far we have found that the less constrained approach is acceptable.

4.1.4.3 Intrusion Reporting

BMSL programs that detect suspicious events report those events to upper layers of the security control hierarchy by issuing intrusion reports. The intrusion reports contain fields such as: type of intrusion, name of program being attacked, process identifier of process running the program, time attack started, time attack ended, estimated severity and estimated certainty. Infrastructure modules in the detection engine provide a convenient API allowing defensive programs to produce intrusion reports. Issued intrusion reports are logged in a text file. For compatibility with other ID systems, a user level process translates the text in the intrusion report file to CIDF format and transfers it to a CIDF manager.

4.1.5 SMS Infrastructure Processing

The programs written to protect an application have the form of system call exit and entry event interceptors written in BMSL (or C++), which can be written to perform arbitrary functions. In support of the defensive programs, the SMS infrastructure provides two discrete support functions. The first support function, called interception support, comprises the processing that precedes delivery of system call entry and exit events, and that implements the disposition decisions of defensive programs. The second support function, called utility methods, deals with the category of necessary functionality that for reasons of utility is better implemented in a standardized manner.

4.1.5.1 Interception Support Function

The interception support function has the following responsibilities:

- Intercepting system calls made by an application being protected,
- For the first intercepted system call made by an application, initiate an instance of the BMSL program that is to protect the application,
- Pass the intercepted system call to the BMSL program that is protecting that application as a system call entry or exit event augmented with the intercepted system call data,
- Process the system call event disposition requested by the BMSL program,
- Collect and log operational status data,
- Maintain data commonly needed by BMSL programs, and
- For the last system call made by an application, terminate the BMSL program protecting the application.

The detailed implementation of these responsibilities is described in a later section.

Of the listed responsibilities, the need for all except for maintaining commonly needed data should be self-evident. There are a few data items which, while not included in the system call event data, are frequently needed by many BMSL programs. Also, there are other data items which, although maybe not used frequently, are sufficiently difficult to obtain that we prefer a centralized approach to collecting them, rather than forcing all BMSL programs to implement separate collection algorithms. In the first category are items such as program name, process identify, and parent process identifier. In the second category are items such as the originating address of an incoming socket connection, and a map between open file descriptors and file names. The need for the originator is obvious, when an intrusion is reported, we wish to identify the source of the attack, and likewise, to initiate an active network reaction, the originating address is sometimes needed. The need for the open file descriptor mapping arises because some defenses need the capability to reason about the name of a file during processing of system call events at which the file name is no longer easily available, because pervious system calls have associated an open file descriptor with the name, and latter system calls use the file descriptor rather than the name. Because there are multiple system calls that can alter the mapping between open file descriptors and names, we decided to centralize this function and perform it for all protected applications. All of the commonly needed data is instantiated by the interception support function in a memory area uniquely associated with a single instance of a BMSL program.

It should also be noted that initiation and termination are actually more complicated than they might seem, as they require special treatment of the `fork`, `clone`, `execve`, and `exit` system calls. Furthermore, if an application terminates abnormally, deducing its last system call is impossible, and a garbage collection strategy must be used. The garbage collection strategy is to monitor the `/proc` file system, which Linux and many other Unix derivatives use to store transient information about running processes, to notice that a process previously known about no longer exists.

4.1.5.2 Utility Methods

A number of functions are expected to be commonly needed by many different defensive programs. The previously mentioned intrusion reporting function is a simple example of this common functionality. For the most part, the selection of functionality that should be implemented once as a utility function as opposed to multiple times as program-specific functions is merely an exercise in software decomposition. But, two classes of common functionality which provide essential functionality are somewhat unique to the intrusion detection/reaction application and are therefore highlighted. These two classes are the forward and backward system call methods.

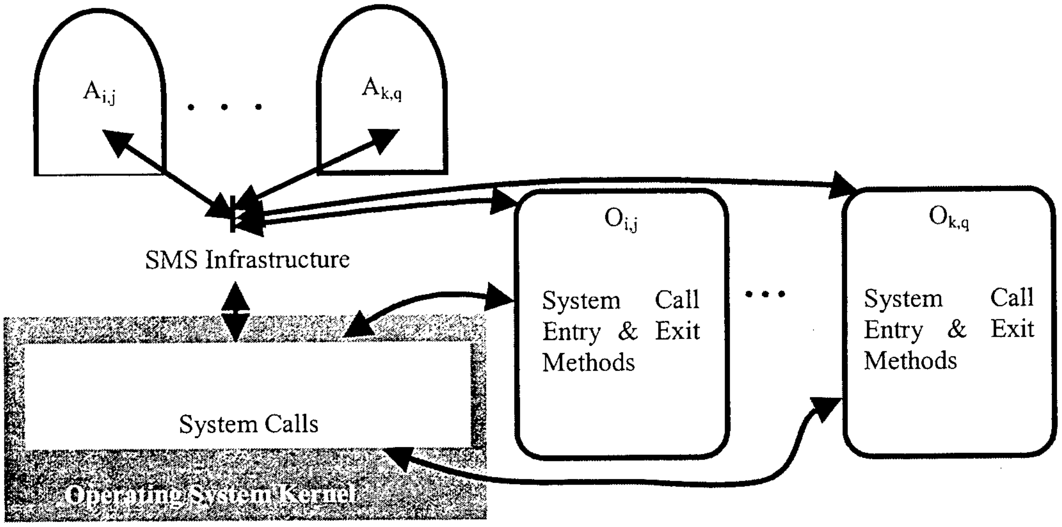
Obviously, to meet the goal of allowing defensive programs to be written to arbitrary functionality, a mechanism is needed by which the defensive programs can themselves request system calls. More explicitly, when a system call entry or exit event is reported to a BMSL program, the BMSL program may need to use system calls to collect additional information useful in determining whether or not the event is characteristic of an intrusion, and similarly to implement a reaction to a detected intrusion, the BMSL program may need to invoke system calls. This need can be illustrated with two simple examples. For the first example, assume that a certain exploit consists of tricking the application into opening a secure file by linking that file to a non-secure file and opening the non-secure file. To detect the exploit, the open system call is intercepted and a check of the name of the file being opened is made to determine whether the file is a link, and if so, if the link points to a secure file. The most convenient way to check these conditions is to use existing system calls. For the second example, assume that in order to prevent a compromised process from causing damage we desire to move the process to a specially constructed file system in which it may continue to execute, but will be unable to access secure files. To implement this reaction, the BMSL program processing the intercepted system call that confirms the compromise needs to insert a `chroot` system call. Thus as the examples illustrate, BMSL programs need access to system calls. However, as we began to experiment with defensive programs to implement detection and reaction, we discovered a subtlety with respect to system calls. The subtlety is that in some cases, the BMSL program needs the system calls to behave as though they were actually issued by the process being defended, while in other cases, the system calls must behave as though they were issued by the highly privileged detection engine. To accommodate this need, the detection engine infrastructure includes two interface functions for every system call. The forward system call interface allows system calls to be executed in the context of the detection engine, while the backward system call interface allows system calls to be executed in the context of the process being defended.

4.2 C++ Interface Between BMSL and SMS Infrastructure

The interface between defensive programs written in BMSL and the SMS infrastructure is the C++ programming language. BMSL programs compile into C++ programs that are then compiled and link edited with the SMS infrastructure to produce the detection engine. The C++ programs produced by compiling BMSL programs must conform to the interface and structure supported by the SMS infrastructure, as shown in Figure 2. Note that manual production of C++ programs conforming to the interface is also possible, and sometimes valuable to an expert user. The interface structure imposes constraints on how the C++ classes are organized, on their inheritance requirements, and on the methods they must implement. Other than these constraints, defined in detail below, no other constraints are imposed on the C++ class definitions. In particular, class definitions can include definition of data members for long-term storage of whatever data items are needed by the defensive program. For example, the BMSL compiler produces C++ classes that define extended finite state machines, and all of the long-term storage required for their operation is provided through data members defined on the C++ classes.

4.2.1 Organization of Interceptor Classes

For each application, A_i to be defended, a class, $C_{i,}$ is defined by compilation of the BMSL program written for A_i. (While it is convenient to describe the mapping from A_i to C_i as one-to-one, many-to-one mappings are also possible, and useful for general-purpose defenses such as sandboxing.) When process $A_{i,j}$, running an instance of A_i makes its first intercepted system call, the SMS infrastructure intercepts the system call and creates object $O_{i,j}$, which is an instance of $C_{i.}$ From that time forward, until $A_{i,j}$ terminates, all system calls requested by $A_{i,j}$ are intercepted and passed to $O_{i,j}$.



$A_{i,j}$ - Process executing application A_i
$O_{i,j}$ - Instance of C_i, the class defined for defending application A_i, instantiated for real-time defense of $A_{i,j}$
System Calls - the existing kernel functions providing normal system call functionality
System Call Interceptor - new functions for intercepting all applications system calls as they enter the Operating System Kernel
System Call Entry & Exit - methods on C_i for receiving A_i's intercepted system call entry and exit events

Figure 2

4.2.2 System Call Entry and Exit Event Methods

The mechanism for passing intercepted system call events to $O_{i,j}$ relies upon the object-oriented concept of inheritance and the C++ concept of virtual functions and also upon a naming convention which we defined. Every C_i must inherit from a class named baseProg that implements virtual methods for all system call events. The system call event methods are named according to the following convention:

```
    int className::sysCallName_entry(args...) and
    int className::sysCallName_exit(args...),
```

where className is the name of the class being defined, sysCallName is the name of a system call, and args are the arguments for that system call. The naming convention precisely defines the name of the method to be invoked when an intercepted system call entry or exit event is to be reported to an object of type C_i. The class baseProg defines entry and exit methods for all system calls, while each C_i defines methods for only those events which it is interested in. The baseProg methods merely provide default treatment of specifying normal disposition, while the C_i methods can be written to perform arbitrary functions. Since baseProg defines the methods as virtual, the C++ rule for resolving virtual methods to those defined for the child class, if they exist, or to those defined by the parent class, if the child class methods do not exist, means that by merely not defining methods for system call events that are not of interest, C_i automatically obtains normal disposition for them. For convenience, the notations:

$O_{i,j}$::X__entry(...), and
$O_{i,j}$::X__exit(...),

are used to name the methods that are invoked when system call entry and exit events for the system call named X are reported to an object of type C_i that has been instantiated to monitor a process whose process identifier is j.

As a convenience to programmers in dealing with system call arguments, classes appropriate for storing some commonly used system call arguments have been defined. For example, many system calls have char * arguments to accommodate null terminated strings, but programmers find the more modern approach of using a string class, with methods supporting typical string operations, more convenient. For this reason two versions of each system call entry and exit method are implemented by baseProg, one using the exact same argument signature as the system call, and the other using the class versions of the arguments. Programmers can use either version of the system call entry and exit methods at their own preference.

When the SMS infrastructure intercepts a system call from $A_{i,j}$ and finds $O_{i,j}$ it casts $O_{i,j}$ to type baseProg, and then uses the naming convention to invoke the appropriate method on the cast of $O_{i,j}$. If the BMSL program has specified the event as one it wishes to see, then the C_i method produced by compiling the BMSL action associated with that event will be invoked. In this case $O_{i,j}$ ::X_entry() or $O_{i,j}$ ::X_exit() can return either normal or faked, and may modify the input and output arguments. If the BMSL program has not specified the event as one it wishes to see, then the intercepted system call will proceed as though it had not been intercepted. In this case $O_{i,j}$ ::X_entry() or $O_{i,j}$ ::X_exit() can only return normal, and no modifications to input or output arguments are possible. Code Example 4-1 gives a trivial example of a C++ class definition that conforms to the SMS interface specification. The example might be used to protect an application that is prohibited from doing any execve system calls. The definition of a single method vulnerableProg::execve_entry() for the class that inherits from baseProg means that if the application vulnerableProg is assigned to monitor (as defined in the config file) ever requests an excve, then vulnerableProg::execve_entry() will be invoked before execve is actually executed. The example method reports the requested execve as an intrusion, and furthermore protects the application from causing damage by a faked return value indicating that the victim computer has insufficient memory for the request. Since the error return is actually legitimate, the attacker may be deceived into believing that the attack is sound, but coincidental conditions are interfering with it. Therefore the attacker may wasted time by continuing to pursue the same non-productive attack before switching to another attack. More examples of BMSL programs and the C++ programs resulting from their compilation are found in section 4.5.

Code Example 4-1 C++ Interface Conformance

```
#include <errno.h>
#include <idip_attack_codes.h>
class vulnerableProg : public baseProg
{
   public:
      int execve_entry(CString filename,const char *argv [],const char
*envp[])
      {

logIntrusion(time(0),time(0),IDIP_ATTACK_CODE_BUFFER_OVERFLOW,100,100);
         fakedRC(-ENOMEM);
         return(FAKED);
      };
}
```

4.3 Detailed Design of SMS Infrastructure Prototype

As previously described, the SMS infrastructure provides interception support functions and utility methods to enable the execution of defensive programs. The interception support functions are implemented in three components:

1) **System call interceptor**, which generates system call entry and exit events from the system calls requested by $A_{i,j}$, and implements the disposition requested after the events are processed by $O_{i,j}$,
2) **Event delivery**, which handles communication between the system call interceptor and the object manager, and
3) **Object manager** which instantiates the objects, passes system call events to objects as method invocations, deletes objects when they are no longer needed, and provides a set of convenience functions to simplify program development.

The implementation of each of these components is described in detail in separate subsections.

The utility methods are implemented as methods defined on baseProg to provide a library of common or difficult functions readily available to all defensive programs.

4.3.1 Interception Support Function

4.3.1.1 System call interceptor

4.3.1.1.1 `Libc` Version

As previously mentioned, the early version of the prototype intercepted system calls by modifying the standard system call interface library, named `libc`. Unlike the kernel version of SMS, the `libc` version is not considered appropriate for further development into a production quality system. Nevertheless, it is still useful for experimentation with the basic concepts of SMS in a safe environment, and may not be as difficult as the kernel version to migrate to other platforms. The `libc` modifications are described in this section. Most applications link `libc` either dynamically or statically. For applications that dynamically link `libc`, for the application to be protected, the dynamic library linkage conventions need to be changed so that the modified `libc` is used. For applications that statically link `libc`, for the application to be protected, the applications need to be re-built using the modified `libc`.

Before describing the modification to libc for interception, we first provide an overview of libc's design. Libc implements function Il( ) for each system call. Most Il( ) functions have the example structure. For a system call named X, libc implements an Il( ) function named X( ), and the kernel implements a Kl( ) function named __X( ). There is always a 1:1 mapping between Il( ) and X, and usually a 1:1 mapping between Kl( ) and Il( ). Usually Kl( ) and Il( ) have the same number of arguments, although Il( ) can have an arbitrary number of arguments of any type, while Kl( ) can have at most five arguments that must be unsigned integers. Il( )'s functionality consists of marshalling its arguments into the form required by Kl( ), calling Kl( ), and, after Kl( ) returns, marshalling Kl( )'s return value, the global variable errno, and the output arguments into the form defined for the system call, and returning to the application.

The interception modifications for system calls with this structure are trivial. The modifications for intercepting Kl( ) are applied to Il( ). The modifications are the same for all system calls and have the structure shown in Code Example 4-2.

Code Example 4-2 libc system call interception

```
 int Ii(arg1, ... arg5) {
 int fakedRc, realRC, fakedErrno;
 .//Unchanged instructions in Ii
 .
 .
 if (preTrap(SYS_Ki, arg1, ... arg5, &fakedRC,&fakedErrno) == faked){
       errno = fakedErrno;
       return(fakedRC);
 }
 realRC = Ki(arg1,...,arg5);
 if (postTrap(SYS_Ki, arg1,...,arg5, realRC, &fakedRC)==faked){
       errno = fakedErrno;
       return(fakedRC);
 }

return(realRC);}
```

Immediately prior to Il( )'s call to Kl( ), a call to a function called preTrap( ) is inserted. Immediately after the invocation of Kl( ), a call to a function called postTrap( ) is appended. PreTrap( ) and postTrap( ) are new functions that interface with the delivery component. PreTrap( ) causes delivery of a system call entry event, and postTrap( ) causes delivery of a system call exit event. Both functions have as return values the disposition code returned as a result of processing the reported event. If either function returns a disposition of faked, then the value contained in the variable called fakedRC is returned to the application. If neither function returns a disposition of faked, then the value returned by executing the requested system call is returned to the application. Note that if preTrap( ) returns faked, then the requested system call is not executed.

The parameters of the system call are conveyed as writeable variables so that they may be changed during processing of the reported event. Since preTrap( ) and postTrap( ) have access to system call arguments, the object processing the event can initiate defensive actions even though normal is returned. For example, consider the open( ) system call, which has file name as an argument. The object processing entry interception of open( ) may decide that the application should not open the requested file, and to fool the attacker, a safe, look-alike file should be opened. The object can implement this reaction by merely changing the value of the file name argument and returning normal, causing the succeeding steps to open the look-alike file. An alternative implementation requires the object to itself open the look-alike file, set the substitute return value to that obtained by the substituted open( ), and to return faked. The choice of implementations is at the defender's preference, but we have found that reactions involving merely changes to argument values are usually simpler using the first implementation. Note that the availability of alternative implementations should not be considered a drawback; rather it is a desirable characteristic of any general-purpose programming environment. Further details of the two functions are provided in the next subsections.

4.3.1.1.1.1 PreTrap

PreTrap()'s input arguments are the identity of the Kp() being intercepted and the at most five arguments to Kp(). The identity of Kp() is the number defined by the macro SYS_X, where X is the system call name. The arguments are all unsigned integers. No Kp*()* has more than five arguments, so always passing five arguments, with unused arguments set to NULL, is simpler that the alternative of writing multiple preTrap() functions differing only in the number of input arguments. PreTrap()'s output arguments are error number and return value. PreTrap()'s output arguments have meaning only when $O_{i,j}$•Kp_entry() returns faked. When preTrap() receives the faked return, preTrap() places the value of the output parameter error number into $A_{i,j}$'s global variable errno, which is used by convention in libc to return status to invokers of system calls. Likewise, preTrap() uses the value in the output argument return value as Kp()'s return value, and does not execute Kp().

4.3.1.1.1.2 PostTrap

PostTrap()'s input argument is the identity of the Kp() being intercepted. PostTrap()'s input/output arguments are the return value which Kp() produced, and errno as it was set by Kp(). (Recall that postTrap() is invoked after *K_p()* executes). The input/output arguments are used when $O_{i,j}$•Kp_exit() returns faked, in which case postTrap() sets Kp()'s return value and errno to the returned values.

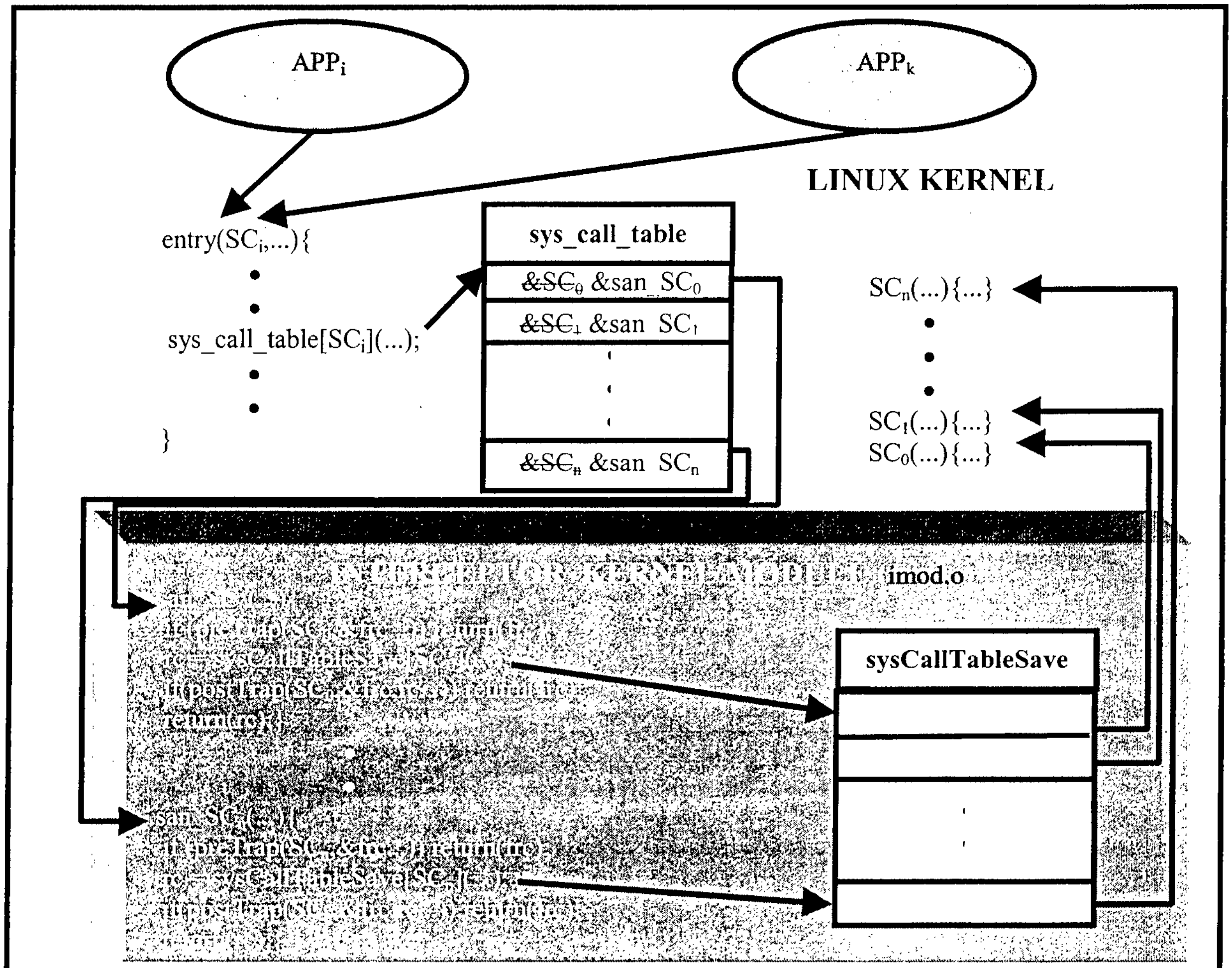
4.3.1.1.2 Kernel Version

In the latest version of the prototype, system call interception occurs in the OS kernel through a Dynamically Loadable Kernel Module (DLKM), as illustrated in Figure 3, and described in this section.

Before describing interception, we first review system call execution as it occurs in most UNIX variants. When an application makes a system call the application executes an instruction to move from user mode to kernel mode. Once in kernel mode, execution passes to the entry() function. Entry() uses the system call identifier of the requested system call to index sys_call_table, which contains pointers to system call functions. Entry() uses the pointer to invoke the function and upon its return, moves the application back to user mode, and returns.

Within the framework of the existing design, system call interception is easily accomplished by overwriting sys_call_table. The DLKM, called imod.o, copies the existing sys_call_table into sysCallTableSave and overwrites each entry in sys_call_table with a pointer to an imod.o function designed to intercept the system call pointed to by that entry in sys_call_table. For each system call, SC_i(), imod.o defines an interceptor function named san_SC_i(). The function signature of san_SC_i() is identical to SC_i(). The structure of all san_SC_i() interception functions is identical, as shown in Figure 3. First, the intercepted system call data is passed to the imod.o function called realDetectEntry(), which passes the data to the appropriate object, using the system call entry and exit methods previously described, for analysis. Through realDetectEntry() these methods can return one of two values: normal, meaning that entry interception decided that the system call should be executed, or faked, meaning that entry interception decided that the system call should not be executed. If realDetectEntry() returns faked, then it also supplies a substitute return value to be passed to the application, and san_SC_i() merely returns this value. If realDetectEntry() returns normal, then san_SC_i() invokes SC_i() using the pointer in sysCallTableSave. When SC_i() returns, the interceptor uses the imod.o function called realDetectExit() to perform exit interception. Like realDetectEntry(), realDetectExit() can return normal or faked. Normal means that exit interception decided that the application should be returned the actual output of $SC_i()$. Faked means that instead of the actual output of $SC_i()$, the return value supplied by `postTrap()` should be used.

Like their `libc` version counterparts `preTrap()` and `postTrap()`, `realDetectEntry()` and `realDetectExit()` have write access to system call arguments, so it is possible to implement powerful reactions event though the normal disposition is returned.



APP$_i$ - a user level application
P$_j$ - process running an instance of a user level application
SC$_i$ - the numeric identifier assigned to a specific system call
entry(SC$_i$, ...) - the kernel function invoked for all system calls.
sys_call_table - table of function pointers whose ith element points to the kernel function for the system call with identifier i.
SC$_i$(...){...} - the kernel system call function for the system call with numeric identifier i.
&SC$_i$ - a pointer to SC$_i$(...){...}
san SC$_0$(...){...} - the function providing interception of SC$_i$(...){...}

Figure 3

4.3.1.2 Object Manager

The object manager is the most complicated component in the prototype. The object manager provides a central interface point to which system call entry and exit events are reported. The object manager converts incoming system call events into invocation of methods on the appropriate objects. Underlying this general task are three subtasks:

- Object instantiation when a protected process requests its first system call,

- Selection of proper object when a protected process requests additional system calls, and

- Cleanup when a process terminates,

which will be explained by walking through incoming event processing. In addition to the object manager's main task of converting system call events into object method invocations, the object manager performs additional convenience tasks aimed at simplifying defensive program development. These tasks are remote originator identification and file descriptor mapping.

4.3.1.2.1 Converting events into method invocations

The event delivery component uses the object manager's `realDetect()`method to report incoming system call events. The previously defined system call event data is passed to the object manager through the arguments of `realDetect()`. When `realDetect()`is invoked, it must first determine the protection status of the process, $P_{i,j}$ that requested the system call. That status may be either:

1) Known and protected ($P_{i,j}$ is running an application that is listed in the config file, and this is not the process's first system call event),
2) Known and unprotected ($P_{i,j}$ is running an application that is not listed in the config file, and this is not the process's first system call event), or
3) Unknown (this is the first system call event for $P_{i,j}$).

If the status is known and protected, the object manager would have already instantiated $O_{i,j}$ to monitor $P_{i,j}$ and the object manager basically needs to invoke the appropriate method on $O_{i,j}$ returning that method's return value to the system call interceptor as the disposition. Invoking the appropriate method on $O_{i,j}$ presents a minor difficulty: the only identity of the system call available to `realDetectEntry()` and `realDetectExit()` are the integer identifier. The integer identifier must be converted into a method name. The conversion is accomplished via a `switch` statement derived from `asm/unistd.h`, which is the header file defining the system call identifiers. The `switch` statement includes a `case` for each system call identifier, and the `case` statement's action is to invoke the appropriate method. The `case` statement also organizes the method's arguments as required for the method being invoked, since the method's arguments, like the intercepted system call's arguments, are specific to the system call. The method `baseProg::evalSysCall()` provides the system call identifier to method conversion. (As will be seen later, `baseProg::evalSysCall()` can be overloaded by any derived class, and since it is invoked for every intercepted system call, and overloading it rather than overloading entry and exit methods is sometimes convenient.)

If the status is known and unprotected, then $P_{i,j}$ is not being protected, so the object manager merely needs to pass the normal disposition code back to the system call interceptor.

If the status is unknown, then the object manager needs to consult the config file (stored as an internal data structure for efficiency) to determine if $P_{i,j}$ is running an application for which a C_i has been assigned. If there is no C_i assigned to the application then the status of $P_{i,j}$ is set to known and unprotected, and the object manager passes back the normal disposition code to the system call interceptor. If there is a C_i assigned to the application then the status of $P_{i,j}$ is set to known and protected, and the object manager instantiates $O_{i,j}$, and invokes the appropriate method on $O_{i,j}$, returning that method's return value to the system call interceptor as the disposition. The object manager maintains the state of $P_{i,j}$ in a table indexed by j, that is, the process identifier. If the state is known and protected, the table entry for $P_{i,j}$ also contains a pointer to $O_{i,j}$. When $P_{i,j}$ requests its first system call the object manager's table will contain no entry for process identifier j, so the object manager will create the entry and update the state to known and protected or known and unprotected depending on the contents of the config file.

The object manager's table contains an entry for every process that has requested a system call, regardless of whether or not the processes are protected. Retaining knowledge for protected processes is of course necessary, since the processes must be associated with their objects for the duration of their lives. Retaining knowledge for unprotected processes is an optimization, it is not necessary, but avoids the overhead of re-examining the config file for all but the first system call requested by a process. For both protected and unprotected processes it is therefore necessary to remove the process information when the process terminates. Removal is simple, however knowing when to remove is complicated. There are two system calls: exit and execve, that serve as one indication of process termination, so the object manager always examines the type of the system call event that it is receiving to decide whether removal is required.

If the status of $P_{i,j}$ is known and unprotected, then immediately upon seeing the system call entry event for exit, or the system call exit event for execve, the object manager can remove the information it is storing for $P_{i,j}$. Since it may not be obvious why these two events are the events signifying termination, the following explanation is offered. If an exit entry event for $P_{i,j}$ is received, then it means that $P_{i,j}$ has called the exit system call. Since $P_{i,j}$ is not being protected, the only possible outcome of exit is process termination. If an execve exit event for $P_{i,j}$ is received, then it means that $P_{i,j}$ has called the execve system call, and furthermore, that execve has executed. If the execve was successful, as indicated by its return code, then $P_{i,j}$ has been transformed into Pk,j, which is a process having the same process identifier, but running a possibly difference application. Successful execution of execve, as evidenced by receipt of an execve exit event having a return code of success, is therefore an indication that information related to $P_{i,j}$ should be removed. Since an execve exit event with a return code of success is actually the first system call event associated with a newly started application, instead of the last system call event of a terminating application, in addition to signaling clean up of $P_{i,j}$, the event also initiates the object manager's previously described processing in response to the first system call event from Pi , k.

If the status of $P_{i,j}$ is known and protected, then the removal decision is similar to that described for the known and unprotected status with three exceptions. First, since the exit entry event may be arbitrarily modified by $O_{i,j}$, mere knowledge that $A_{i,j}$ has requested the exit system call is insufficient. Instead removal should only occur if $O_{i,j}$::exit_entry() returns the disposition code normal. (Admittedly it seems pathological for the disposition of an exit entry event to be anything other than normal, but we prefer a design that admits all possibilities rather than one that imposes limits based on preconceived assumptions.) Second, in addition to removing only the status information associated with $P_{i,j}$, the object manager must also delete $O_{i,j}$ by calling its destructor. Third, when $P_{i,j}$ is transformed into Pk,j through an execve, the status will change from known and protected to known and unprotected if new application is not listed in the config file. We believe that in most cases if the application that did the execve is protected, then the application that starts running as a result of the execve should also be protected, so by default, if the object manager finds that the new application has no entry in the config file, then the object manager retains the previous object to monitor the new application, updating the application-specific values (name, start time, etc) in the object. This default is specified by the member function baseProg::copyOnExec() and can be overwritten in the derived classes if desired.

While the exit entry and execve exit events indicate that $P_{i,j}$ is terminating, they are insufficient to cover error cases in which $P_{i,j}$ terminates due to a failure, which may produce no system call event evidence. To account for $P_{i,j}$ failures, the event based termination detection is augmented with a polling technique. On a periodic basis, the object manager examines the /proc directory to determine if any process whose status is known and protected or known and unprotected is no longer listed. If so, then the object manager assumes the process has crashed and initiates removal actions for that process.

The preceding discussion of how process termination is handled not only describes the mechanics of how the operation is done, but additionally reveals a design philosophy that applies to several other object manager functions. The philosophy is to simplify the methods that C_i must implement by providing a common implementation of methods that are either required by many objects, or which are so difficult that they might be a burden to the programmer. Handling process termination is in the first category, it is required by all objects, and therefore even though it would be relatively easy for each object to implement by placing the functionality in the entry_exit() and execve_exit() methods, we decided instead to have the object manager perform the function. The next object manager function that we describe—maintenance of a mapping between open file descriptor and file names, belongs to the second category, that is, we expect that only a few objects will need the functionality, but since it is hard to implement, we prefer a common implementation.

4.3.1.2.2 File Descriptor Mapping

The need for the open file descriptor mapping arises because some objects need the capability to reason about the name of a file during processing of system call events at which the file name is no longer easily available, because previous system calls have associated an open file descriptor with the name, and latter system calls use the file descriptor rather than the name. Because there are multiple system calls that can alter the mapping between open file descriptors and names, we decided to centralize this function and perform it for all protected applications. The file descriptor map is implemented by having the object manager check all incoming system call events to determine if the event is associated with one of the system calls that can allocate, deallocate, or modify file descriptors. These system calls are: execve, creat, open, clone, fork, dup, pipe, fcntl, dup2, close and socketcall. The impact all of these system calls except execve, fork, and clone can have on a process's file descriptors should be immediately obvious. The impacts of execve, fork, and clone on file descriptors arise because they impact the open file descriptors a child process inherits from its parent. The objects manager's check for possible changes to the file descriptor map occurs during the processing of system call exit event, rather than system call entry events, since changes only occur through successful execution of a system call, which is only evidenced by system call exit events which have a success return code. Furthermore, in addition to changes to file descriptors that occur due to successful execution of system calls requested by $P_{i,j}$, the possibility of changes to file descriptors caused by backward system calls made by $O_{i,j}$ must also be considered. (Recall that a backward system call is one that $O_{i,j}$ wishes to be executed as if $P_{i,j}$ requested it.) For this reason, backward system call methods also perform the file descriptor update function. The file descriptor map is maintained in an object that is encapsulated in baseProg, from which all C_i inherit, and is therefore available to all objects regardless of whether or not they use it. When a process forks, clones, or execves, the file descriptor map is copied to the object instantiated for the new process, subject to the file descriptor's inheritance policy. While it might have been possible to avoid a common implementation of the file descriptor map by requiring those objects which actually need it to intercept the require events themselves, however, we believe the design description illustrates that it might be excessively burdensome to require individual objects to implement so complex a function.

4.3.1.2.3 Remote Originator Identification

Another common function provided by object management is remote originator identification. Remote originator identification is used to identify the originator of a TCP connection when the server side of the connection is being protected by an SMS defensive program. The object manager performs remote originator identification to record the IP address and port for applications that are being accessed as servers by remote clients through establishment of a TCP connection. Since SMS monitors the TCP connection establishment system calls to obtain the originator's address, SMS is no more immune from IP source address spoofing than the underlying system calls themselves are. The typical architecture of UNIX servers, in which a parent process listens to a well-known port for connection requests, accepts connections, and then forks a child to actually provide service makes remote originator identification somewhat complicated. The remote originator address can easily be obtained from the system calls events that are reported to the parent, but the address is not easily known to the children, who typically are passed the socket as file descriptor zero. The object manager therefore always looks for those system call events which contain the originator's address, and when they occur automatically stores the originator's address in the object associated with the process requesting the system calls. The address is stored in a data member defined for baseProg, and is therefore inherited by all derived classes. Like the file descriptor map, the remote originator address is copied to the object created in response to forks, clones, and execves. Thus the object monitoring the child created to respond to a server request has available to it the originator's address. Availability of the address to the child object is of vital importance, since it is usually the child object that detects and reports intrusions, and in intrusion reports the originator should be identified.

4.3.1.2.4 Differences between libc and kernel versions

While the functionality of the object manager is identical for both versions, their architectures are slightly different and this difference simplifies how some of the functions are implemented in the libc version. The difference is that the libc version object manager is a user level multi-threaded server process, while the kernel version object manager is a monolithic kernel level sub-routine. The architecture differences manifest themselves as follows.

The libc object manager listens for connections on a well-known port and receive the first system call event reported by the system call interceptor for a process over this well-known port. If the object manager determines that the process should be protected, by consulting the config file, the object manager allocates a separate thread for the process, and within that thread sets up a socket to be used only for that process. Because each process has its own connect, the method for detecting abnormal process termination is simplified. If the process dies for any reason while it is connected to the object manager, the object manager will have evidence of the process's death through the connection. Thus the multi-connection nature of the libc version obviates the periodic polling of the /proc directory which the kernel version requires.

4.3.1.3 Event Delivery

4.3.1.3.1 Libc Version

Both preTrap() and postTrap() use askDetectionEngine() to deliver events to the object manager. AskDetectionEngine() maintains a socket connection to the object manager and delivers intercepted system call data to it in a simple ASCII format. Most of askDetectionEngine() is straightforward and therefore is not described in detail. However, one aspect—dealing with system call arguments that are pointers—involves an optimization that is described.

Some objects require access to the data pointed to system call pointer arguments in order to make system call disposition decisions. For example, an object attempting to prevent opening of /etc/passwd needs to read the path argument of open(), which is a pointer, in order to determine if /etc/passwd is being opened. AskDetectionEngine() could dereference all pointers and send the data they point to the object manager, but this would frequently result in large system call event messages. Since the objects can read the $A_{i,j}$'s memory, and because we believe that most intercepted system calls are not part of any vulnerable pattern, and therefore do not require argument dereferencing, we optimize for the typical case. The strategy which askDetectionEngine() implements is to never dereference pointers in the system call event message. In those rare cases in which the object requires dereferencing, that object must obtain dereferencing by reading $A_{i,j}$'s memory. The libc version object manager provides common functions for dereferencing all types of pointers that occur in system calls, as described in a later section.

4.3.1.3.2 Kernel Version

Event delivery is simpler in the kernel version than the libc version because there is no need for interprocess communication. In the kernel version, realDetectExit() and realDetectEntry() are invoked directly as functions, as described in Section 4.3.1.1.2.

4.4 Utility Methods

4.4.1 Forward System Calls

The forward system calls enable a defensive program to invoke a system call that is executed according to the context of the detection engine. A simple example of the need for forward system calls is a defensive program that maintains a log file of the protected application's actions. Obviously, this log file needs to be created and accessed with the permissions of the detection engine, not the application, since it is probably desirable to hide the file from the application.

4.4.1.1 `Libc` Version

Since the `libc` version of the detection engine runs as a separate user level process, no special handling for forward system calls is required. Any system call that the detection engine makes is a forward system call by default. However, for consistency with the kernel version, in which forward system calls do require special treatment, a set of baseProg methods, defined as `baseProg::forward_X(args...)`, is defined. One such system call is defined for each system call, with X taking the value of each system call, and args taking the value of definition of X's arguments. For the `libc` version, these methods merely invoke the system call and return its return value.

4.4.1.2 Kernel Version

When the kernel level interceptor reports a system call event to an object, all processing takes place in the context of the application that requested the intercepted system call. To provide forward system calls, the forward system call methods defined on baseProg, as described in the previous section, switch to the context of the detection engine before making the system call, and then switch back to the context of the application before returning. The switch is accomplished by swapping the effective and real user identifiers between those of the detection engine and those of the application.

4.4.2 Backward System Calls

The backward system calls enable a defensive program to invoke a system call that is executed according to the context of the application being defended. A simple example of the need for backward system calls is a defensive program that uses the `exit` system call to terminate the application after an intrusion is detected. Obviously, it is the application that is to be terminated, and not the detection engine, so it is vital that a backward `exit` be done instead of a forward `exit`.

4.4.2.1 `Libc` Version

In the `libc` version, backward system calls require communication between the detection engine and the protected application. This communication has two components: the poking of data from the detection engine into the modified `libc`, and a sub-protocol under the protocol for delivering intercepted system calls from the modified `libc` to the detection engine. The sub-protocol can only be used after the delivery of a system call event from the modified `libc` to the detection engine, and before the delivery of that event's disposition to the modified `libc`. The modified `libc` includes a special purpose in-memory structure dedicated to backward system calls. The detection engine pokes values into the structure describing the backward system call's identity and its arguments. The detection engine then sends a special code over its socket connection to the modified `libc`. The special code tells the modified `libc` to execute the instruction in the memory block. After the system call is executed, its return value is passed over the connection to the detection engine. The instructions comprising these actions are embedded in baseProg methods named: `baseProg::backward_X(args)`, where X is the name a system call and args are the arguments for that system call. Since backward system calls are actually executed in the protected application, special processing is required to prevent their interception and avoid recursion.

4.4.2.2 Kernel Version

In the kernel version, any system call made by a defensive program executes in the context of the application whose system call was intercepted, so no special processing to obtain backward system calls, is required. However, for consistency with the `libc` version, the kernel version provides an implementation of backward system calls that utilizes the `libc` naming convention.

4.4.3 Memory Dereferencing

Both the kernel and the `libc` versions require special processing associated with memory accesses. For the `libc` version, the need for memory access methods allowing the detection engine to access memory in the protected application should be obvious, since the detection engine and the application run as two separate processes. For the kernel version the need is less obvious, but is obviated by the use of virtual memory which causes the address space to change when the boundary between kernel and user space is crossed.

4.4.3.1 `Libc` Version

The detection engine accesses memory in the application using `peeks` and `pokes`, which are a mechanism that Linux provides for interprocess memory access. BaseProg implements methods for generic reading and writing of application memory based on address and size, and also implements several type specific methods for accessing commonly used built-in types or defined structures. Special care must be taken when writing into the application's memory to not exceed the memory's size. When it is sometimes necessary to replace a memory structure with a larger one, it is necessary to use the backward system call facility to allocate additional memory.

4.4.3.2 Kernel Version

While in kernel mode, accesses to user memory space must be dereferenced. Two generic, pre-existing kernel functions provide read and write dereferencing based on address and size. From these generic methods, special purpose dereference methods are implemented as baseProg methods to simplify dereferencing of commonly used built-in types and defined structures. When it is necessary to allocate more user memory to accommodate a write, the backward `brk` system call must be used.

A special and noteworthy case of dereferencing exists with respect to system call arguments that are pointers. All system calls that have pointers as arguments expect those arguments to point to memory in user space, hence the system calls are written with the appropriate dereferencing code. If an interceptor desires to change the value of a pointer argument, that interceptor must not allow the replaced pointer to point at kernel memory, otherwise the system call will produce an exception. To change the content of a pointer, an interceptor must allocate user memory, and point the pointer at the new user memory.

As a convenience in generalization, baseProg implements methods which test whether a specified pointer points at user or kernel memory,

4.4.4 Section 3 Sub-Routines

Most C++ programmers are accustomed to the sub-routines of section 3 of the UNIX user manual. These sub-routines perform many useful functions and in many case offer a simplified interface to the section 2 system calls. The section 3 sub-routines are available to BMSL programmers in the `libc` version, but not the kernel version.

While we desired to make the full section 3 sub-routine functionality available to BMSL programmers using the kernel version, we decided that doing so would be too much effort for a research prototype. The difficulty is that each sub-routine has to be ported so that it can run in kernel space, as opposed to user space. The difference in memory accesses between user and kernel space makes the porting non-trivial. Consequently we decided to port only those sub-routines which we actually need for our defenses. These sub-routines are implemented as baseProg methods.

4.4.5 Common Intrusion Detection Framework Interface

The CIDF interface allows the detection engine to issue intrusion reports in a standardized manner to other intrusion detection components. The CIDF interface was added late in the project and has not been integrated into the older `libc` version of the prototype.

When a defensive program encounters conditions which it considers to be an intrusion, in addition to reacting to the intrusion to prevent damage, the program should issue an intrusion report to alert other components in the security hierarchy. The method

```
     int baseProg::logIntrusion(unsigned int startTime, unsigned int endTime,
unsigned int attackCode, unsigned int severity, unsigned int certainty);
```

is used to issue intrusion reports. The arguments allow the BMSL to characterize the intrusion in terms of when it started, when it ended, the canonical name of the attack, how severe the BMSL program believes the attack to be, and how certain the BMSL program is that it has actually detected an attack. The assignment of values to the arguments is subject to the BMSL program's interpretation.

`LogIntrusion()` merely writes the intrusion report to a simple ASCII file. In addition to the arguments passed to it, `logIntrusion` also includes some of the common data members of baseProg in the intrusion report. The common data members included are process name, process identifier, originator, and effective and real user identifiers. Note that if the process being protected is not a server connected to a remote client, then the originator field is blank.

The file written by `LogIntrusion()` is the interface to the CIDF environment. We use the IDIP reference implementation of CIDF, and developed a CIDF client to read the file and a CIDF manager to receive the reports generated when the CIDF client reads the file. We realize that the coupling between CIDF and the detection engine is too loose for a production environment, but chose not to integrate IDIP more tightly into the detection engine. Such integration would have been extremely difficult because it would require porting IDIP into the Linux kernel.

Since the intrusion reports are ultimately consumed by IDIP, we limited parameter values to those acceptable by IDIP, in particular, the canonical representation of attack types was restricted to those defined by IDIP. We found the organization of attack types to be limiting. The attack types are chosen from a flat space, where the attack name encodes both the application being attacked and the type of attack. We would have preferred at least a two dimensional attack name space in which application name and attack name are decoupled. We believe the flat space to be far less extensible than a two dimensional space.

4.5 Developing BMSL Specifications for SMS

By far, the biggest impediment to writing BMSL specifications for applications is the sheer number of system calls in UNIX. Our early efforts in developing a specification for FTP server showed us that most often, we need not be concerned about so many different system calls; it suffices to consider classes of system calls that perform conceptually similar functions. We now describe our specification for FTP to illustrate the process of writing specification in detail. We then proceed to describe a classification scheme for grouping system calls. We then provide additional specification examples that make use of this classification scheme.

4.5.1 Specification for FTP Server

```
    Define useful constants.
 1. ftpAdmFilePrefixes ::= {“/etc/”, “/lib/”, “/usr/lib/”, “/dev/null/”,
    “/var/run/ftp”}
 2. ftpInvalidUsers ::= {0,BINUID,SYSUID,MAILUID}
 3. ftpInvalidPutDirs ::= {“/”, “/bin/”, “/sbin/”, “/usr/”, “/etc/”}
    Define useful abstract events.
```

We assume that certain abstract events such as privileged which denotes certain privileged system calls that are not used by most programs, and wrOpen which denotes any file open operation that can create or modify the file.

```
 4.  ftpInitBadCall ::= (wrOpen(f)|(f != /dev/null) &&
     !isExtension(/var/run/ftp,f))||permChange||rename||link||delete||mkdir|
     |rmdir||admin||execve||clone||bind||listen||connect||accept||recvfrom||
     recvmsg||sendto||sendmsg
 5.  ftpAccessBadCall ::=
     admin||accept||recvfrom||recvmsg||sendto||sendmsg||clone
 6.  ftpPrivCalls ::=
     close||uidgidops||socket||setsockopt||(bind(s,sa)|port(sa)=FTPDATAPORT)
 7.  ftpValidExecs ::= {/bin/ls,/bin/tar,/bin/gzip}
 8.  ftpAccessedSvcs ::= {NAMESERVER}
     Use a state variable to remember the uid of user logging in and client
     host name}
     int loggedUser := NOBODYUID
     int clientIP := 0
 9.  begin();(!setreuid)*;setreuid(r,e) --> loggedUser := e
10.  begin();(!getpeername)*;getpeername_exit(fd,sa,1) --> clientIP :=
     getIPAddress(sa)
     Host authentication phase must precede user authentication.
11.  begin();(!getpeername)*;open(/etc/passwd) --> term()
     User authentication must precede before userid changed to that of the
     user
12.  begin();(!open(/etc/passwd))*;setreuid() --> term()
     Access limited to admin-related files before user login is completed.
13.  begin();(!setreuid())*;open(f)|(!isExtension(ftpAdmFilePrefixes, f)) --
     > term()
     Access limited to certain system calls before user login.
14.  begin();(!setreuid())*;ftpInitBadCall() --> term()
     Certain system calls are not permitted after user login is completed.
15.  setreuid();any()*;ftpAccessBadCall() --> term()
     Anonymous user login: must do chroot before setreuid.}
16.  begin();(!(setreuid||chroot(FTPHOME)))*;setreuid(r,FTPUSERID) -->
     term()
     Userid must be set to that of the logged in user before exec.
17.  begin();(!setuid(loggedUser))*;execve --> term()
     Resetting userid to 0 is permitted only for executing a small subset of
     system calls.
18.  setreuid(r,0);ftpPrivCalls*;!(setreuid(r1,loggedUser)||setuid(loggedUse
     r)||ftpPrivCalls) --> term()
```

```
    Any file opened with superuser privilege is either explicitly closed
    before an exec, or has close-on-exec flag set.
19. (open_exit(f, fl, md,
    fd)|geteuid()=0);(!close(fd))*;(execve|!closeOnExec(fd)) --> term()
    Site-specific: ensure that ftp cannot be used to write files into
    certain directories.
20. wrOpen(f)|(f $in$ ftpInvalidPutDirs) --> term()
    Site-specific: ensure certain users cannot login using ftpd.
21. begin();(!setreuid)*;setreuid(r,e)|(e $in$ ftpInvalidUsers) --> term()
    Site-specific: ensure ftp cannot execute arbitrary programs.
22. execve(f)|(f $not in$ ftpValidExecs) --> term()
    Site-specific: ftp cannot connect to arbitrary hosts or services.
23. connect(s, sa)|((getIPAddress(sa) != clientIP)&&(getPort(sa) $not in$
    ftpAccessedSvcs)) --> term()
```

Our starting point in developing a specification of `ftpd` is the documentation provided in its manual pages. Specifically, we identified the following properties for `wu-ftpd` by examining its manual page and based on our knowledge of UNIX. These properties are captured in our specification language as shown above. Although it is possible to turn the English descriptions directly into specifications in our language, it is usually necessary to cross-check (or "debug") the specifications by monitoring `ftpd` under typical conditions. We have therefore used a hybrid approach, where we first manually inspected system call traces produced by `ftpd`, and used it to further narrow down the actions/behaviors that the `ftpd` server may exhibit. In most cases, we have not attempted a sophisticated response, instead opting for a simple action such as terminating the process using a support function named *term()*.

- `ftpd` attempts to authenticate a client host before proceeding to user authentication phase. Precisely identifying the sequence of system calls that correspond to client authentication is hard, as it involves a large number of steps that may vary from installation to installation. As such, we treat `getpeername` as a marker that indicates host authentication related processing. Similarly, we treat opening of */etc/passwd* as a marker for user authentication related processing. Rule 11 captures this English description by stipulating that an open of the password file should never happen before invocation of `getpeername`.

- Users need to be first logged in before most files can be accessed. Rule 13 uses *setreuid* as an indicator for completion of user login process.

- After user authentication is completed, `ftpd` sets the *userid* to that of the user that just logged in. We remember this *userid* for later use (rule 9).

- Prior to user authentication, only files beginning with names identified in the set *ftpAdmFilePrefixes* can be accessed (rule 13).

- Certain system calls are never used before user login, and certain others are never used after login process (rules 14, 15). Let *ftpInitBadCall* denote a pattern that matches system calls not used prior to user login. Similarly, let *ftpAccessBadCall* match system calls that are not used after login.

- for anonymous login, the *userid* FTPUSERID is used; moreover, the *chroot* system call is used to restrict access only to the sub tree of the file system rooted at `~ftp` (rule 18).

- `ftpd` resets its effective userid to root in order to bind certain sockets to ports numbered below 1024. The userid is reverted back to that of the logged in user immediately afterwards (rule 20).

- To eliminate possible security loopholes, `ftpd` must execute a *setuid* system call to change its real, effective and saved userid permanently to that of the logged in user before executing any other program; otherwise, the executed process may be able to revert its effective userid back to that of *superuser* (rule 19).

In addition, we make sure that any file that is opened with *superuser* privilege is closed before *exec* (rule 21).

- Finally, we model certain site-specific policies that override any access policies configured into ftpd. These policies are captured by rules 22 through 25.

4.5.1.1 Discussion

We make the following observations about the specification for ftpd.

1. In order to reduce clutter, we have deliberately abbreviated some of the lists (e.g., *ftpInvalidUsers*), while leaving out definitions of some abstract events (e.g., *wrOpen*) in the specification for ftpd.
2. Typical specifications need not be as comprehensive as for ftpd—we have made it comprehensive in order to better illustrate what sorts of properties can be captured in BMSL.
3. The specification was developed using the principle of least privilege, without really paying attention to known vulnerabilities. Nevertheless, it does address most known ftp vulnerabilities (many of which have since been fixed) such as FTP bounce (rule 25), race conditions in signal handling (rules 20, 19) and site-exec (rule 24) [CERT].
4. Availability of fast matching algorithms described in the subsequent sections enables us to focus on developing these rules independent of each other, as opposed to worrying about how they may be modified to enable more optimal checking, e.g., rules 11, 15, 16, and 23 have prefixes in their pattern component that are similar, and we can in fact combine them into a single rule. But this will lead to a specification that is much less clear, so we avoid this. Since the matching algorithms give us the benefit of such combination for free, there is no cost to pay for the clarity of specifications.
5. We can develop a more concise language that gives us the ability to specify dependencies among events e_1 and e_2 more directly, rather than using a pattern such as *begin();(!e*$_1$*)*;e*$_2$. However, the focus here is on a core language that has the necessary expressive power.
6. The examples illustrate that the expressive power of REE is far beyond that of regular expressions. For instance, rules 20 and 21 capture associations among events that are beyond what can be captured even by context-free grammars. (The associations are similar to checking that a variable is defined before use, and can be captured by attribute grammars.)

4.5.2 Classification of system calls

When writing specifications it sometimes becomes necessary to list out a large number of system calls made by a process, which may be a tedious process. Also it is cumbersome to write variants of specifications all dealing with system calls of similar functionality. In addition, to express certain security properties, we need to consider a group of system calls with similar functionality rather than just one system call. A specification writer could easily miss out certain system calls while trying to list out all of them.

The following example illustrates all the above concerns. Consider a property such as "no file can be opened for writing by this program." The BMSL specification:

```
        (any)*;open(file,O_WRITE||O_CREAT),
```

can be used to detect that a process is opening a file for writing. The term *any* here refers to a list of all the system calls. However this specification is not complete. If a process does not allow a file to be written, it is reasonable to expect that the process does not allow a file to be removed or a new directory to be created either. Hence we need to make sure that system calls with functionality similar to *open in write mode* are not allowed, i.e., a system call such as open (file, O_APPEND*)*, which opens a file to append data to it, or mkdir, which creates a new directory, should not be allowed. So we add properties such as: *no directory should be created*, *no directory should be removed* and *no file should be removed*. The specification now becomes:

```
(any)*;open(file,O_WRITE)|||mkdir(file)||rmdir(file)||...||unlink(file)) .......... (1)
```

where mkdir is the system call that creates a new directory, rmdir deletes a directory and unlink removes a file if it exists.

Now suppose, for a different program, we wish to express the properties: *no file can be opened for writing or reading*, we end up rewriting the specification (1) as follows:

```
(any)*;open(file,O_WRITE)|||mkdir(file)||rmdir(file)||...||unlink(file)
      || open(file, O_READ)|| read(f,,..) ||...||readv(fd,...)) ................. (2)
```

where `open(file, O_READ)` says that the file should be opened in read mode and `read(fd,buf,size)` reads from a file descriptor

The above example highlights the following difficulties for a specification writer:

- The sheer number of system calls that need to be considered could cause a specification writer to inadvertently omit some system calls.
- For writing specification **(2)** the entire process we went through for generating **(1)** had to be repeated and additional system calls had to be listed out.

4.5.2.1 Grouping System Calls

The examples above illustrate two problems: simple sounding rules (prohibit opening files for writing) require specifications of constraints on many system calls; and some rules have commonality that could force redundant specifications. Both problems are solved by a facility that allows the grouping of system calls into events. The key issue is that the criteria used for grouping should be useful from a security point of view. We find that grouping system calls based on the following two criteria is useful in writing specifications:

- Based on similar functionality. A group of system calls may do different things but the resources they manipulate and the effect they have on the resources may be similar. For example when we wanted to specify the property *no file can be opened for writing*, we had to list out the system calls *unlink*, *mkdir*, *rmdir* etc., as they all have similar effects on the file system. If a program is prohibited from opening a file for writing, it is reasonable to expect that it cannot remove any file or it cannot create any new directory. Hence we could group all the above system calls into a group called **WriteOps**. This example illustrates the use of grouping system calls that manipulate resources in a similar manner.
- Based on privilege: Certain system calls with different functionality can be execute only by a process being run as a root or only by the calling process or are made very infrequently. E.g.: system calls such as *mount* or *reboot* are made only be the root and are made very infrequently. In our classification we group the above calls along with some others as privileged calls.

4.5.2.2 Event Definition

Once we group the system calls, we can think of all the system calls in a group as an implementation of a higher-level abstract system call. For instance, we can define an event such as **WriteOps** as follows:

```
WriteOps(path) = {
      open(path, flags) | (flags & (O_WRONLY | O_APPEND | O_TRUNC)),
      creat(path, mode);
      mkdir(const char *pathname, mode_t mode);
      truncate(path, len);
      rmdir(const char *pathnme);
      unlink(const char *pathname);
  }
```

which states that any occurrence of one of the system calls inside the braces is to be treated as an instance of the abstract call `WriteOps`. `WriteOps` is an event that writes a file using its complete path name. The grouping also defines the mapping between the values of the arguments of the abstract call and the actual system call, for instance, in `WriteOps(path)`, the path refers to the file name to be opened and this is mapped to the first arguments for all the system calls within `WriteOps(path)`. While grouping, we may also need different levels of abstraction depending upon the context and this may in turn cause overlaps among different groups. For instance, we may have an abstract call that corresponds to ReadOps (which is an event that includes all the system calls that correspond to opening a file or directory for reading using a file name) and another event AllRWOps that captures all read and write operations on file names.

AllRWOps thus includes, all the system calls in ReadOps and WriteOps, so we define this event as

```
AllRWOps(path) {
        WriteOps(path) ;
        ReadOps(path);
 }
```

In this case, we have a hierarchy with individual system calls at the lowest level, **ReadOps** and **WriteOps** at the next level, and then AllRWOps at the next higher level.

- AllRWOps
    - ReadOps
        - E.g.: open(file, O_RDONLY|O_READ), readir(…)
    - WriteOps
        - E.g.: open(file, O_WRONLY|O_CREAT|…), mkdir(…)

In our specification language, we provide the grouped system calls and the event definitions as macros in a header file with the ability for the users to define their own groupings.

4.5.2.3 Classification chart.

With the criteria for classification listed in the above subsection, we classified the system calls into the following groups:

File accesses, *privileged* calls, *signal synchronization* calls, *process related* calls, *resource related* calls, *network* calls, *ipc* calls and *unimplemented* calls.

Each of the groups is further divided into sub groups.

Classification:

```
|---- File Access
|  |
|  |-Write(fd)                       These are the write operations using file descriptors
|  |-OtherWriteOps                   All the other system calls which have same
|  |                                 Functionality as open(file,O_WRITE)
|  |-WriteOps Write(fd) + OtherWriteOps
|  |-Read(fd)                        The read operations which use file descriptors
|  |-OtherReadOps
|  |-ReadOps Read(fd) + OtherReadOps
|  |-FileAttributeChecks             System call which return the inode
|  |                                 contents of a file like access permissions.
|  |                                 of files
|  |-FileAttributeChange             Calls which change the file attributes
|  |- MiscFDOps                      Calls like dup, lseek which take file descriptors.
|  |-AllFileOps Write+Read+OtherWriteOps+OtherReadOps
|
|---- Process Operations all calls which operate on process id's.
```

```
|   |
|   |-ProcOps                          Calls which create or execute a process
|   |-ProcessInterference              Calls which may interfere with the execution of
|   |                                  other processes.
|   |-GetProcessAttributes             Return various attributes of processes
|   |                                  like process group id, real user ID.
|   |-SetProcessAttributes             Set attributes like the effective user ID.
|   |-UnusualProcessOp                 The clone system call which is made very
|   |                                  infrequently.
|
|---- Resource Related Calls
|   |
|   |-SetResourceAttributes            Sets resource attributes like
|   |                                  scheduling parameters etc.
|   |-GetResourceAttributes
|
|---- Network Calls
|   |
|   |- Connect calls:                  Network system calls made by a client
|   |                                  trying to connect to a server.
|   |- Accept calls:                   Calls made by a server to accept a
|   |                                  connection.
|   |-OtherSocket calls:               Calls such as gethostname which are made
|                                      by both client and server.
|
|---- Signal Related Calls:            System calls which deal with sending
|                                      signals, setting timers.
|
|---- Privileged Calls                 Calls which require root access and are
|                                      executed very rarely. E.g.: mount, umount.
|
|---- MiscUnPrivilegedCalls:           Calls which are made by many processes
|                                      for memory management like mmap, munmap
|
|---- IPC system calls
|---- Unimplemented:                   Obsolete system calls, like oldstat etc.
```

4.5.3 Applications of classification of system calls:

Based on the classification we discussed earlier we developed a generic specification that can be used to monitor any program. In the next subsection we discuss the generic specification. We then show how we extended the generic specification to monitor the *GlobalView* program that is discussed in Section 7.3.1.

4.5.3.1 Development of a generic specification:

The goal for classifying system calls was to make writing specifications easier. To further this cause, we developed a specification which uses the classification scheme, and which can be used as a basis to start writing specifications to monitor most of the commonly used programs. The specification was developed using the principle of least privilege. In particular, we wrote the specifications based on the following observations:

1. Every program writes some files, reads some files and executes some files. The only difference between the programs as far as the above three properties are concerned is the actual files that they write, read or execute. Hence we don't have to change the specifications that correspond to the three properties but

instead we change the list of files on which these system calls operate. For example, the generic specification to express the property that "if the process is attempting to write a file which is not within the list of files which it can write, then terminate the process": *any*; WriteOps(fileWriteList) → term,* where the *fileWriteList* is the list of files which a process can write. This specification holds for all processes, only the list of files, *fileWriteList*, needs to be changed for each process. Note that if a program doesn't execute, write or read a file, then the specifications still hold, except that the corresponding file lists or process lists are empty.

2. Most programs do not make privileged calls hence we disallow any *privileged* calls. Similarly, *unimplemented system calls, clone system call, setting resource attributes* are disallowed.
3. Most programs do not make process interference calls. We disallow them in the generic specification. However this can be tuned during the training phase. Based on the above criteria we came up with the following generic specifications:

In the following rules *any* refers to the list of all system calls and *reaction* is a reaction some reaction, details of which are not important for this example.

Code Example 4-3

```
// Based on the classification of file access operations the following rules
are included.
 // If a write to a file other than the one in the list fileWriteList is made
 then terminate the process.

rule:  (any)*;otherWriteOps(a1,fileWriteList) --> term.

 // read operations should read only those files in the list fileReadList. A
 message is logged if some other file is read.

rule: (any)*;otherReadOps(a1,fileReadList) --> log.
 // Attribute change operations can only be on files in the list of files
 fileAttributeList.

rule: (any)*;fileAttributeChange(a1,fileAttributeList) --> term.
 // ************************************************************************
 // privileged calls are not allowed

rule:  (any)*;(privileged ) --> term.

 // ************************************************************************
 // **** The following rules relate to Process and resource operations ****

// ************************************************************************
 // Clone system call is not allowed

rule: (any)*;unusualProcOp --> term.
 // process interference calls e.g:

rule: (any)* ;procInterference --> term

 // If an execve doesn't execute a program in the fileExecList then
 // terminate the process. procOps is the macro with execve system call

rule:  (any)*; procOps(a1,fileExecList) --> term.
 // ************************************************************************
 // Set resource attribute calls not allowed.

// ***********************************************************************

rule: (any)*;setResourceAttributes --> term.
 // ***********************************************************************
 // UserId modifying calls.

// ***********************************************************************
 // the userid being set is different from the user id of the process.
 // @getuid is a call that tells us what the user id of the process is.

rule: (any)*;setuid(a1)| a1 != @getuid()--> term.
 // The effective user id is being set to some other user and the real user
 is
 // not the root or the user id of the process.

rule: (any)*;setreuid(a1,a2)| (a2 != @getuid() && a1 != 0) -->  term.
 // ***********************************************************************
 // IPC calls

// ***********************************************************************
 // log all ipc calls

rule: (any)*;ipcl(a1) --> log.
 // ***********************************************************************
 // Unimplemented calls

// ***********************************************************************
 // Obsolete system calls are not allowed
 rule: (any)*; unImplemented --> term.
```

4.5.3.2 Application in developing GlobalView defense

GlobalView was a program developed by the AFRL research labs for our online testing. Its functionality is covered in Section 7.3.1. To summarize, *GlobalView* goes through the following stages:

1. It authenticates a user.
2. It then gets the server name from a database.
3. It creates a temporary file.
4. It connects to the server from step (2) using port 5432 and reads some data.
5. The data is written into the temporary file created in step (3) after which it is closed.
6. The temporary file is then renamed into a file called *GlobalView.out*
7. The ownership of *GlobalView.out* is changed to root.

Our starting point for writing the *GlobalView* specification was the generic specification described earlier.

1. The description provided to us listed that the only file *GlobalView* writes is the temporary file created in step (3) in the state machine. We populated the *fileWriteList* (the list of files a process is allowed to write) used in the generic specification with this file.
2. The only attribute change operations *GlobalView* made were on the file *GlobalView.out*. So we populated the *fileAttributeList* (the list of files on which file attribute change operations are permissible) in the generic specification.
3. Since *GlobalView* does not fork or execute any process we used the generic specification to disallow *fork*, `execve` and *clone* system calls.

4. Since the description did not mention any *privileged calls*, *process interference calls*, *set process attribute calls*, *set resource attribute calls*, *unimplemented calls* and *ipc calls*, we just used the generic specifications which by default disallow these calls.

The next step to write rules based on the state machine of *GlobalView*. The following rules were found to be useful:

- *GlobalView* should not connect to the database to get server name before authentication.
- *GlobalView* should not rename any file or create a temporary file before authentication.
- *GlobalView* should not change ownership of the temporary file (using `chown`) before renaming the temporary file.

The next step was to deduce rules based on functionality of *GlobalView*:

- The rename call can only rename the temporary file to *GlobalView.out*.
- The rename call cannot rename a world write-able file or a symbolic link or a *setuid-to-root* file. This rule avoids a possible race condition where the temporary file could be overwritten if it was world writeable or if it was a symbolic link, and some other sensitive file, like */etc/passwd* might be renamed.
- The *chown* system call can only change the ownership to root for *GlobalView.out*. Also a rule made sure that *GlobalView.out* is not world write-able or a symbolic link or a *setuid-to-root* file.
- *connect* system call could only connect be to ports 53 or 5432.

Next, we obtained a system call trace of *GlobalView* (using the UNIX `strace` utility) which led us to write the following additional rule: The files which *GlobalView* could read were only `*.so` (shared library) files and 5 application files. We added this list to the generic specification.

Finally, since the source for *GlobalView* was given to us, we came up with rules to address the following vulnerabilities:

1. *GlobalView* uses `mktemp` call that has a race condition.
2. *GlobalView* has a potential buffer overflow when reading the password file.
3. Before exiting, *GlobalView* does not close any file.
4. If *GlobalView* process is stopped before it can complete, it may end up leaving a temporary file that may contain sensitive information

4.5.4 Anomaly Detector

While our specification-based approach to intrusion detection is much different from the anomaly-based approach, since SMS provides a general-purpose programming environment, we can implement an anomaly-based intrusion detector using SMS, as described in this section. The idea behind anomaly-based intrusion detection is that individual applications manifest certain observable behaviors during normal operation, and different characteristics while under attack. By subjecting an application to a training phase during which the characteristics of normal behavior are learned, intrusions can be detected by looking for deviation from learned characteristics during actual operation. One of the first anomaly detectors was six-system call sequence anomaly detector [PHA98]. The characteristic that this anomaly detector learns during training is all unique sequences of six system calls requested by an application. These sequences comprise a training data set for an application. When the application is used, the system calls it requests are grouped into sequences of size six, and if any requested sequence is not in the training data set, an intrusion is reported.

The SMS anomaly detector requires two programs to be written, one for the training phase, and one for the operational phase. The training phase decomposes into two separate component: an on-line component that records system calls on a per-process basis, and an off-line component that analyzes the recorded system calls to reduce them to application-specific training data set containing all unique sequences of size six. The training data set is represented as a C structure so that it can be compiled into the operational program. The operational program merely searches the C structure for each requested sequence of six system calls and if the requested sequence is not in the training sequence, reports an intrusion. The training data is stored in sorted order to minimize look-up time, which uses a binary search.

The six-sequence system call anomaly detector was an early anomaly detector, and more advanced algorithms are being developed for anomaly based intrusion detection. We were able to implement the six-sequence anomaly detector because of its simplicity, but we did not attempt to implement any more advanced anomaly detectors because our research efforts are focused on the specification-based approach. Nevertheless, our anomaly detector enabled some contrastive experiments between anomaly-based and specification-based approaches and those results are reported in sections 7.4.4 and 7.4.5.

When using anomaly-based intrusion detection, selection of the application inputs during the testing phase are vitally important. If legitimate paths are not executed during the training phase, then when those paths are executed during operation, false positives may be reported. Similarly, if some legitimate paths are avoided during the training phase, then the likelihood of a true positive may actually increase, although for the wrong reason. It is possible that a missed path contains a six-system call sequence that matches a sequence produced by an actual attack, therefore, inadequate training may lead to detection of such an attack. We believe that anomaly-based detector false and true positive rates are only valid when the training phase executes all legitimate application paths. The same belief applies to specification-based detectors although to a lesser extent because the specification-based detector has no training phase. Specification-based detectors should be evaluated by forcing all legitimate application paths to be executed, to properly ascertain false positive rates. To force, or at least measure, the adequacy of application inputs during the training and operational phases, we use a code-coverage tool [TEL99], which instruments an application's source code to collect statistics on which source statements are executed by each input. By aggregating the statistics over a set of test inputs the code coverage can be expressed in a variety of ways. For our purposes we chose block coverage, which measures the percentage of source code blocks executed.

4.6 Performance overhead

Since SMS is a real-time system, it is important to consider how its use impacts the performance of the applications that it protects. How much overhead it adds is in general application-specific, since, as this section shows, that overhead is largely a function of the complexity of a specific defense.

To measure application overhead added by SMS we ran experiments using interceptors of different complexity on a single application multiple times with identical inputs and the same execution environment. Four interceptor complexity levels were studied:

1. No Intercept: `imod.o` was not installed
2. Default Intercept: `imod.o` was installed and configured to intercept every system call using normal disposition.
3. Anomaly Defense Intercept: `imod.o` was installed and configured to perform the applications anomaly-based defense which requires interception of every system call and checking for deviations from the six-system call sequences learned for that application during training.

Specification Defense: `imod.o` was installed and configured to provide a defense based on the specification-based approach, which for the specific applications under observation required interception of two system calls and very minor processing to set or check a object data member.

The specific application used for the observation was `rshd`, and it was given the same input that was used to generate its training data, as described in Section 7.4.5. For each of the four interceptor complexity levels, 500 `rshd` invocations were observed. Each invocation resulted in `rshd` requesting about 20,000 system calls. The UNIX `time` utility was used to obtain `rshd`'s user, system, and elapsed time. The results are shown in Table 1.

Table 1 SMS Performance Overhead

	noIntercept	Specification Defense		Default Intercept		Anomaly Defense	
	Average Time	Average Time	Average Overhead	Average Time	Average Overhead	Average Time	Average Overhead
user	0.242064128	0.241983968	-0.033115324	0.381002004	57.39713552	0.362304609	49.67298617
system	0.384549098	0.401503006	4.408775861	2.200280561	472.1715566	1.9798998	414.8626817
elapsed	2.673527054	2.675591182	0.077206185	4.915230461	83.84816617	5.009478958	87.37341559

The results are exactly as expected. The default intercept complexity level shows there is a rather significant penalty just for intercepting all system calls. The anomaly defense incurs greater overhead because in addition to the cost of intercepting each system call, each system call requires that a new six-system call sequence be formed and searched for, using a binary search, in a table of about 500 such sequences. Finally, the specification-based approach gives the best performance of any of the interceptors (other than no intercept) because it both limits the number of system calls that need to be intercepted, and limits the amount of processing required per system call. We believe that `rshd`'s specification based defense represents the typical case, and this belief is evident in the example specification based defenses presented in this report. Consequently, we are confident that powerful specification based defenses that have low performance impact on the applications that they protect can be the rule rather than the exception. We consider the slight negative overhead reported for user time with the specification defense to be a consequence of normal variations and the accuracy of `time`, and not an indication that the specification defense actually improved performance.

4.7 Operating Instructions

SMS is easy to install and use. SMS is packaged in a single tar file named imod.tar that has three components:

- BMSL compiler source code,
- Detection engine source code,
- Source code for representative BMSL specifications
- The CIDF manager source code (including IDIP).

(Note, in earlier implementations of the prototype, the BMSL component was named ASL, and consequently the previous name persists in some portions of the current prototype.)

4.7.1 Installing the system.

Place the tar file in the directory in which SMS is to be installed, and unzip and untar the source code using :

```
tar -xf | gunzip imod.tar.gz
```

creating the following directory structure.

```
Installation directory
|
|- Parser Contains the source for the ASL compiler
|
|- DE      Contains the detection engine source
|   |- tools     contains the compiler executable
|   |- ASLsrc    The ASL specifications
|   |- ASLprogs  The compiled specifications
|   |- CCprogs   Specification written in C++
|
|-- CIDF
   |-- tfb  The CIDF working directory
   |--idiplib
     |-- Code: Source code for IDIP
     |-- Linux : object files
```

The tools which build `imod.o` require the above directory structure. After installing SMS, one minor configuration edit must be made. The file DE/interceptor/config.tcl must be edited and the assignment of the HOME variable must be changed to the name of the installation directory.

4.7.2 Quick Start

The distribution includes the defensive programs described in this report. This section describes how to build and install a detection engine containing one or more of these programs. Building and installation is controlled by the tcl script named config.tcl in the DE directory. Upon starting config.tcl, a GUI will pop up. The GUI allows configuration in two ways:

A **new configuration can be built** by inputting the names of programs to be protected and the names of defensive specifications to protect them. The GUI lists all the specifications available in the ASLsrc and CCprogs directory, and provides two data entry windows, one for the application name, and one for the specification name. Choose one of the available specifications by double-clicking on the specification or you enter the complete path name for the specification. Then enter the name of the application that specification is to protect. Finally, press the "ADD" button to enter that association. Iterate until all applications that are to be protected have been associated with a specification.

1. A **previous configuration can be selected** from among those listed on the GUI. Double click on any configuration name to view its contents. This can be repeated until the desired configuration is found. Configurations that are no longer needed can be deleted with the DELETE button.

After building or choosing a configuration, press the NEXT button. A new window displaying the configuration will appear. The build button will create an `imod.o` corresponding to the configuration. The install button will load that `imod.o`. While `imod.o` is loaded, the enable/disable button allows it to be quickly turned on and off, which is sometime useful during debugging. The uninstall button unloads `imod.o`.

For a real time display of intrusion reports issued by SMS, the CIDF manager can be run as follows:

1. cd to cidf/ directory,
2. run the script ./idipStart.

There is one known error in the operation procedure, and it affects the unloading of `imod.o`. Unloading `imod.o` may occasionally cause other processes to crash due to out-of-range memory access. The cause of the problem is that some processes may have long running system calls in-progress when `imod.o` is removed. While the unload procedure waits up to two minutes to give these system calls the opportunity to complete, and also sends signals to the processes to try to force system call interception, these measures are not always effective. Consequently on some occasions `imod.o` may be unloaded while it is still needed by an in-progress system call. When the system call eventually returns, the return address used by the system call will no longer be valid and the process will crash.

4.7.3 Detailed Procedures

The information in the quick start section should suffice for most users, but this section provides more detail about SMS configuration and installation as a trouble shooting aid.

4.7.3.1 Developing or Modifying Specifications

The BMSL specifications included with the distribution are not intended to be exhaustive or complete. As new vulnerabilities are discovered, development of new specifications or modification of existing specifications may be desired. Developing new specifications is the task of a competent programmer, and requires an understanding of BMSL, as described earlier. This section describes the mechanics of BMSL development.

BMSL specifications are stored in files whose suffix is .asl in the DE/ASLsrc directory. BMSL files can be created with any editor. BMSL files are compiled using the ASL compiler, which is stored in the DE/tools directory. BMSL compilation of file.asl produces two files: file.h, and file.C. File.h contains the definition of the C++ class, named file, that corresponds to the specification in file.asl. File.C contains the definition of a single function used to instantiate an object of type file. While placement of file.C and file.h are command line options of the ASL compiler, the current system configuration requires them to be placed in the DE/ASLprogs directory. The config.tcl script makes these conventions transparent to the programmer.

4.7.3.2 BMSL Compiler

The compiler source is in the Parser directory. The compiler can be built using the command make asl, which places the compiler in a file named asl in the tools directory. The BMSL compiler usage is as follows:

```
    asl [v [1234]] [o <C_file_name>][c <class_name>][h <header_file_name>]
specification_name
```

v verbosity level :
1 is the default and the compiler operates quietly,
2 causes the abstract syntax tree to be displayed,
3 causes the parsing actions to be displayed,
4 causes both the abstract syntax tree and the parsing actions to be displayed.

o specify an output file for the C++ class corresponding to the specification (default is specification name with a .C suffix).

c specify a name for the C++ class. Default is the specification name.

h specify an header file to be included by the specification (default is specification name with a HEADER.h suffix).

4.7.3.3 Building the Detection Engine

The detection engine, which is stored in the DE directory in a file named imod.o, is built in the DE directory using the command make imod.o. The Makefile in the DE directory controls the building of imod.o, which consists of a constant procedure for the infrastructure parts of imod.o, and a variable procedure for the defensive specification parts of imod.o. The constant procedure uses standard makefile techniques and requires no explicit description. The variable procedure does, however, warrant special treatment.

The variable procedure is responsible for building an imod.o that is customized based on the applications that are to be protected and the specifications associated with them. The desired associations are stored in the file DE/config, which consists of multiple lines, each of which has three tab-separated attributes. The first attribute is the command line name of the application to be monitored. The second attribute is the name of the class that is to be instantiated to monitor that application. The third attribute is the name of the file in which the class definition is stored. By convention, behavioral specifications are stored in the DE/ASLsrc directory. In the config file, the .asl prefix is omitted. The following is an example of a config file line:

```
in.ftpd ftp      ASLsrc/ftp.
```

The Makefile uses the asl compiler to compile the BMSL programs. The compilation is configured so that the generated .h and .C files are stored in the DE/ASLprogs directory.

The Makefile uses several custom scripts that are stored in the tools directory to perform an important optimization during the building of imod.o. The optimization ensures that only those system calls that are actually listed in the specifications are intercepted, thus minimizing the performance impact of system call interception. The scripts scan those .h files in DE/ASLprogs that are to be included in the imod.o being built, to determine what system calls are must be intercepted. The scripts build a file named interceptedSystemCalls.h that contains the symbolic names of the intercepted system calls in a C array. InterceptedSystemCalls.h is compiled into imod.o, and when imod.o overwrites the system call table to establish interception, only those system calls listed in the array have their system call table entries overwritten.

The Makefile also uses several custom scripts that are stored in the tools directory to automatically generate the file allocProg.C, which is used to instantiate objects when their associated applications make their first intercepted system call. Automatic generation of allocProg.C, minimizes its contents to include instantiation for only those objects required for the current configuration.

As previously mentioned, expert programmers may wish to generate C++ classes directly rather than having the ASL compiler produce them. Each C++ class requires creation of two files; a .h file and a .C file in the CCprogs directory. The .h file is the definition of the monitoring class. The .C file contains a single function that is called to instantiate the object. The distribution includes example .h and .C files in the CCprogs directory that can be consulted as models. When using C++ specifications, the config file specification is similar to that of BMSL specifications except that the directory containing the files must be CCprogs instead of ASLsrc. By convention, the file name part of the .h and .C files are assumed to be identical, therefore the .h or .C prefix is omitted.

4.7.3.4 Installation, Operation, and Removal

The detection engine produced by compilation is stored in DE/imod.o, which is a DLKM. DLKMs are installed using the Linux insmod command. Because of unresolved portability issues, some newer versions of insmod may not install imod.o, therefore an older version is included in the distribution in the DE directory. Installation is accomplished using the following command in the DE directory: ./insmod imod.o. When imod.o is no longer needed, it is removed using the Linux rmmod command, as follows: rmmod imod. (Note that the rmmod command does not allow the .o suffix.)

When imod.o is installed, it overwrites the system call table and begins to receive intercepted system calls, but it merely gives normal treatment to intercepted system calls until it is enabled. Imod.o is enabled by the installation of another DLKM, called santog.o. Santog.o can only be loaded after imod.o, and must be removed before imod.o. When both are loaded, the defenses contained in imod.o will be executed. Removing santog.o while imod.o is still installed causes execution of defenses to cease. Thus, loading and removal of santog.o allows imod.o to be disabled and enabled without removing and installing imod.o.

The Linux command lsmod is also of value to administrators dealing with DLKMs. Lsmod lists the currently installed DLKMs.

Occasionally, we have experienced difficulty in loading *large* DLKMs. The definition of large is site-specific and is related to the amount of memory a particular computer has. Insmod treatment of failure due to DLKM size is currently inadequate, it crashes with a segmentation violation. If insmod fails with a segmentation violation, the likely cause is that imod.o is too large. To correct the problem, imod.o must be reconfigured, either to contain fewer specifications, or to contain specifications that have been optimized with respect to their sizes.

5 Network Monitoring System

The SAN Network Monitoring System is concerned with the interception of network packets, detecting intrusions on the basis of this information, and possibly modifying the network packets. The NMS could be deployed at each host, or at an external gateway to the system being protected. The latter deployment is more common, as it enables the NMS functions to be consolidated at a single location (or a few locations), thus simplifying its administration. Moreover, several attacks (e.g., port sweeps across multiple hosts) can be identified only by collectively looking at the aggregate network traffic, rather than monitoring individual hosts.

Previous efforts in network intrusion detection [H90, H93, Pax98, VK98] typically attempted to use the network packet data as the prime source of information regarding intrusions. In SAN, we view NMS as complementing the function of SMS, rather than duplicating it. As such, NMS is geared towards identifying low-level attacks that do not manifest changes to behaviors of processes, since the latter class of attacks can be readily detected by the SMS. For this reason, NMS does not concern itself with reconstructing higher-level information from packets, such as the TCP/IP streams seen by a higher-level application. Of course, our BMSL-based framework permits us to make use of such higher level information, provided an external system performs the requisite TCP packet reassembly and provides the resultant information to NMS. This is certainly feasible if we used a runtime environment that can intercept the interaction between the session layer and higher layers of the network protocol stack. (In the case of UNIX implementation of the TCP/IP stack, this interface would expose similar information as available to the SMS, which can intercept data as it is read or written by the application that is participating in the TCP session.)

Since an attacker is likely to attempt to disable the intrusion detection system by any means possible, it is particularly important for the system to be robust under all traffic conditions, e.g., malformed network packets should not crash the system. We have developed a novel type system that enables compact declarations of network packet structure and the constraints on their contents, so that these conditions can be automatically checked at compile-time and/or runtime without programmer involvement. Moreover, we need to ensure that NMS is not easily disabled (or even worse, used to effect denial-of-service attacks) by overload. Our fast pattern matching algorithm has resulted in very low overhead for runtime monitoring, enabling NMS to sustain network traffic rate of 1-2 Gbps.

In the rest of this section, we illustrate NMS specifications with several examples. We describe our prototype BMSL compiler for NMS. We describe the installation, configuration and use of NMS. Performance and effectiveness of NMS are partially described in this section, with some of the detailed experimental efforts described in Section 7.

5.1 Examples of NMS Specifications

Development of NMS specifications was largely driven by the Lincoln Lab IDS evaluation effort. Since evaluation stressed the ability to accurately detect and identify attacks, the NMS specifications characterized misuse more often than normal use. The examples chosen for illustration in this section reflects this bias towards misuse specifications rather than normal use specifications in the NMS.

5.1.1 Very small IP Packets

We begin with a simple specification that essentially asserts that IP fragments are of reasonable size. In particular, the following specification will report an anomaly when the initial fragment of TCP packets is less than 48 bytes, which is too small to hold even the TCP header. There is no legitimate reason to use a very small packet fragment, unless it is the last fragment of an IP packet that contained multiple fragments. On the other hand, such a fragment can be used to inflict attacks, e.g., some packet filtering firewalls can be tricked into permitting TCP traffic to certain ports that are normally filtered out by the firewall.

```
   MY_NET = 129.186.44.0
   MY_NET_MASK = 255.255.255.0
   event my_net_addr(a) = ((a & MY_NET_MASK) == MY_NET)

event is_frag(p) = ((p.more_frags) || (p.frag_offset != 0))
   Table tcpFrag(unsigned int, /* keyed by ip address */
                 100, 30,      /* table size 100, time window 30 seconds */
                 1, 0,         /* hi, lo thresholds */
```

```
           tcpFragBegin, tcpFragEnd) /* invoked when hi or lo threshold
crossed */
rx(p) | my_net_addr(p.daddr) && is_frag(p) && (p.protocol == IP_TCP)

           && (p.tot_len < 48)  -> tcpFrag.inc(p.saddr)
```

The first two lines define constant values that provide the address of the network to be protected. Following this, two abstract events are defined that make it easier to specify fragmented IP packets. The next line declares an MFU table used to keep track of the hosts that have recently sent fragmented IP packets. The table is declared to have up to 100 entries. A time window of 30 seconds is associated with the table, which means that we remember events that happened roughly within the past half minute. We associate two functions, `tcpFragBegin` and `tcpFragEnd` with the MFU table that will be invoked when the number of entries in the table crosses high and low threshold. (These functions can each write out an intrusion report for consumption by a higher-level system.)

The thresholds are being set at 1 and 0 respectively, which means that `tcpFragBegin` will be invoked as soon as a single IP fragment of very small size (<48 bytes) is encountered, while `tcpFragEnd` will be invoked when the MFU table becomes empty. These threshold values make the detection of this anomaly deterministic. The reason for using a table in such a case (as opposed to directly invoking a function that generates an attack report) is as follows. First, we are able to distinguish among packets received from different sources and treat them as separate attacks. Second, the attacking host may generate a large number of fragmented packets that match this criterion. Rather than generating many attack messages, we may generate just two messages that indicate the beginning and end of the attack.

5.1.2 SYN-flood Attack

As a second example, we consider the SYN-flood attack, which involves sending a TCP connection initiation packet to a victim host with a spoofed (and typically nonexistent) source address. The victim host sends back a SYN-ACK packet, but since the source address of the first packet is spoofed, it does not receive the ACK packet to complete the connection. Consequently, the connection remains in a half-established state until a timeout occurs after a period of more than a minute. The number of such half-open connections is limited to a small number. Thus the ability of the victim host to accept further TCP connections on a socket can be effectively eliminated by an attacker that sends in a small number of such attack packets. We detect this attack using the following set of rules:

```
tcp_conn(p1, p2) = (p1.daddr == p2.saddr) && (p1.tcp_dport ==
p2.tcp_sport)
                 && (p1.saddr == p2.daddr) && (p1.tcp_sport ==
p2.tcp_dport)
tcp_syn(p) = my_net_addr(p.daddr) && (p.tcp_syn) && (!p.tcp_ack)
tcp_synack(p1, p2) = tcp_conn(p1, p2) && (p2.tcp_syn) && (p2.tcp_ack)
                  && (p1.tcp_seqnum+1 == p2.tcp_acknum)
tcp_ack(p2,p3) =

tcp_conn(p2,p3)&&(!p3.tcp_syn)&&(p3.tcp_ack)&&(p2.tcp_seqnum+1==p3.tcp_ack
num)
Table neptune((unsigned int, unsigned short) /* keyed by ip address and
port */
           1000, 120,     /* table size 1000, time window 120 seconds */
           4, 1,         /* hi, lo thresholds */
           neptuneBegin, neptuneEnd) /* invoked on crossing  hi or lo
thresholds */
(rx(p1)|tcp_syn(p1)..tx(p2)|tcp_synack(p1,p2));
```

```
((!rx(p3)|tcp_ack(p2,p3))*     over     60)     ->     neptune.inc((p1.daddr,
p1.tcp_dport))
```

After defining a few abstract events, the attack is specified in a simple and natural fashion in BMSL. A SYN-flood attack is characterized by reception of a TCP SYN packet, followed sometime later by transmission of a SYN-ACK packet, which is followed by a long time delay (60 seconds, in particular) over which no ACK packet is received. Isolated occurrences of this sequence of events do not indicate an attack, since the ACK packet may be lost under normal circumstances as well. To avoid false alarms due to such isolated occurrences, the response action is to simply increment the counter associated with the occurrence of a SYN-flood attack sequence for a given TCP destination endpoint. When the attack sequence occurs frequently over a short time (in particular, if it occurs more than 4 times within a 120 second interval), an attack report will be generated via execution of the function neptuneBegin. Note that this reporting function has the associated TCP endpoint information, which can be included in the attack report.

5.1.3 Teardrop Attack

The teardrop attack involves fragmented IP packets that overlap. The following pattern captures any such overlap, without flagging those cases where a fragment is duplicated. Note that not all overlaps correspond to teardrop attacks, but there is no reason to use overlapping IP fragments. As in the case of small TCP fragments, this example indicates the ability of BMSL to constrain system behaviors to correspond to "reasonable use" of protocols, rather than capturing a specific attack signature. This factor, in turn, aids discovery of unknown attacks that may vary in minor ways from the known attack (and hence be missed by a method looking for an exact signature), but nevertheless uses protocols in unexpected ways. (We also remark that it is in general a good idea to constrain unreasonable use of protocols, since the program paths handling unusual cases may not have been well tested.)

```
    frag_begin(p) = p.frag_offset*8
    frag_end(p) = frag_begin(p)+p.tot_len-20
    same_pkt(p1,p2) = (p1.daddr == p2.daddr) && (p1.saddr == p2.saddr) &&
    (p1.id == p2.id)
    overlapping_frag(p1,p2) = is_frag(p2) && same_pkt(p1,p2) &&
        ((frag_begin(p2) < frag_end(p1)) && (frag_begin(p1) < frag_end(p2)) &&

      !((frag_begin(p1)==frag_begin(p2) && frag_end(p1)==frag_end(p2))))

(rx(p1)|is_frag(p1);(rx|tx)*;rx(p2)|overlapping_frag(p1,p2)) within 60 -> ...
```

This example again illustrates the ability of BMSL to capture even complex behaviors (or attacks) in a simple fashion. We define what it means for IP fragments to overlap. Then, a teardrop attack is identified as an occurrence of a sequence of overlapping IP fragments, where arbitrary other packets may appear in the middle. This specification illustrates the ability of BMSL to capture *what* constitutes an attack, without having to burden the specification writer with *how* the attack is to be detected. The "how" part is nontrivial in the case of teardrop attack, since we have to compare an incoming IP fragment against all of the preceding fragments that are part of the same IP packet.

5.1.4 Port sweep attack

Finally, we present an example to detect port sweeps on TCP ports. The basic idea is to count the number of different ports on a host for which packets are delivered over some period of time. If we count all of the packets delivered to a host, then we will typically end up with many false positives. So we omit any TCP packet that is being received in response to a packet originally sent by the victim host. We keep track of TCP endpoints for which we have seen originating packets, and ignore these endpoints for detecting sweeps.

```
  Table orig((unsigned int, unsigned int, unsigned short, unsigned short),
              1000, 300); /* thresholds etc. omitted */
Table TCPsweep((unsigned int, unsigned int), /* source and destination IP
addr */
                1000, 1200,

              8, 3, ....) /* threshold functions not shown */
tx(p)|p.protocol==TCP -> orig.inc((p.saddr, p.daddr, p.tcp_sport,
p.tcp_dport))
rx(p)|(p.protocol==TCP) && (!orig.member((p.daddr, p.saddr, p.tcp_dport,
p.tcp_sport)))

-> TCPsweep.inc((p.saddr, p.daddr))
```

The first rule accumulates all of the TCP endpoints for which packets have originated in the protected network in a table `orig`. Any TCP packet that is received is first checked against this table. If it is found in the table, the packet is not considered as part of a sweep. Otherwise, it increments an entry in the table corresponding to the pair of source and destination IP addresses. When this number crosses a threshold of 8, an attack is reported. (Note that the choice of thresholds is highly dependent on the network environment being protected, and the choice of number 8 here is for illustrative purposes only.)

5.2 NMS Runtime Environment

In theory, NMS is capable of intercepting network packets, as well as modifying them. It is also capable of generating or discarding packets. However, our current implementation of NMS runtime environment supports only an interception capability. This restriction enabled us to reuse the popular packet capturing software BPF (Berkeley Packet Filter) to capture packets. BPF supports the ability to capture packets from either a network interface or a `tcpdump` file. The latter alternative was used in the Lincoln Labs evaluation. The former alternative was used in our internal experiments.

In addition to packet capturing, the runtime environment provides the support functions used by the detection engine. The most notable among these are for creation and manipulation of decay counters and MFU tables. The design of these data structures is interesting in its own right, as it forms the basis of high performance of the NMS. However, in the interest of space, we skip a full discussion here, instead referring the reader to our paper [SGSV99].

5.3 BMSL Compiler for NMS

Although BMSL is a single language that is applicable to both SMS and NMS, the compilers used in SMS and NMS are currently distinct. The reason for this is that by the time we were getting ready to participate in the 1998 Lincoln Labs evaluation, the work on the BMSL compiler had barely begun. Only the parser had been implemented at that time. In order to get a reasonable version of the compiler ready before the evaluation, we partitioned the compiler into two parts, one for SMS and another for NMS. From the point of implementing the needed features for SMS and NMS compiler quickly, this turned out to be a very good decision. However, from a longer-term maintenance and feature extension viewpoint, it would be desirable to integrate the two versions of the compiler.

The SMS compiler supports class types and matching of complex patterns, while the NMS compiler supports packet types and simple event patterns without any sequencing operations. In particular, the NMS event patterns are restricted to mention just a single event, possibly with conditions on the event. The more complex rules shown earlier were hand-translated (on an as-needed basis) into a collection of simple patterns and a collection of C++ functions that performed the operations on the right-hand sides of the rules. While this hand-translation approach was intended to be an interim measure, the continuing demands of intrusion detection evaluation in 1999 did not spare any time to upgrade the NMS compiler to support complex patterns. As such, the integration of the SMS and

NMS compiler is only partially complete at this point. In particular, the parsers used by both compilers have been integrated, but the type checking and code-generation portions remain independent.

5.4 Installing, Configuring and Running NMS

NMS analyzes, identifies and reports any observed intrusive activity in a network packet-stream. NMS is capable of directly accepting the network packet-stream from an Ethernet interface, as well as reading from compressed or regular `tcpdump` save files. NMS can be instructed to produce output to a front-end GUI that displays the intrusion reports. It can also be instructed to produce output to files, in the format desired by the DARPA 1999 evaluation.

NMS principally consists of four components, namely, the attack specifications, NMS compiler, NMS runtime environment, and a GUI front-end. The front-end GUI is written in Java and the specifications in BMSL, while both the BMSL compiler and the runtime modules are written in C++.

5.4.1 Installation

For installing NMS the following directories are needed:

parser: this directory contains the source, object and executable files for the parser module of the BMSL compiler. The executable is used to compile the attack specifications.

pfmCodegen: this directory contains the source and object files for the code-generation module of the BMSL compiler. These object files are linked with those of the parser module to generate the compiler executable in the parser directory.

asl: this directory contains the attack specifications used to generate the packet-filter. The specification files have the extension .asl.

runtime: this directory contains the driver program that accepts the command line arguments, accepts input, processes the input packet stream, and produces output accordingly. Other files in this directory include ones for implementation of data structures like event aggregators and threshold decay counters and MFU tables, and the runtime functions referenced by the attack specifications.

gui: this directory contains the source, class and executable files for the front-end GUI.

In order to generate the runtime executable, the `pcap` library (the BPF packet capturing library) is needed. These libraries need to be installed in a separate directory.

5.4.2 Configuration and tuning

NMS configuration changes are mainly done by changing parameter values in various BMSL files and C++ header files in the runtime directory. The specific files that may need to be changed are as follows:

1. `asl/finalBackup.asl` : change the constants `MY_NET1` and `MY_NET2` to indicate the first two bytes of the IP address of the monitored network.
2. `runtime/Makefile`: assign to the variable `ASL_FILE` the name and relative path of the ASL file that contains the attack specifications. Assign to the variable `PCAP_DIR` the directory in which the libpcap libraries have been installed. Assign to the variable `PARSERDEMOPATH` the name and path of the ASL compiler.
3. `runtime/functions.h`: change the constants `MY_NET1` and `MY_NET2` to indicate the first two bytes of the IP address of the monitored network. Also modify the thresholds for detecting various attacks. There are usually two thresholds, a low threshold and a high threshold. Below the low threshold, no attack will be reported. When the low threshold is reached, an attack with a probability of 0.2 is reported. This increases steadily to about 0.8 by the time the high threshold is reached.
4. `runtime/functions.C`: Include the IP address of every host in the network to be protected, by appropriately initializing the array `my_ip_addr`. Also, modify the size and time window parameters of the MFU tables (MFU tables are implemented using a C++ class called `Aggregator`) if necessary.

5.4.3 Running NMS

The NMS is invoked using the following command line syntax:

```
runtime [-o99] | [-m] [-np | -nd] [-r <tcpdumpFileName>] | [-i
<interface>] |
     [-z compressedTcpdumpFileName]
```

The meanings of the options are as follows.

-o99: produce output for 1999 DARPA ID evaluation. The output is written to two files, namely, identification.list and detection.list. The detection.list file contains a brief listing of the attacks describing the date, time, name, category and target of the attack. In addition it also gives the probability that the detected activity is actually an intrusion. The identification.list file contains a more detailed report of the attacks listed in detection.list, containing additional information like the duration of the attack, address of the attacker and the ports involved.

-m: produce output on stdout that in the format required by the GUI front-end. Normally, this option is not used directly; instead, the GUI application starts up the NMS using this option for monitoring purposes.

-np: read packets, but do no processing on the packets. Used mainly for debugging and for performance measurements..

-nd: read packets, process them but do not detect attacks. Used for debugging and performance measurements.

-i <interface>: read packets from the Ethernet interface identified by the UNIX device name <interface>. The NMS needs to be run with superuser privileges in order to use this option.

-r <tcpdumpFileName>: read packets from tcpdump file <tcpdumpFileName>.

-z compressedTcpdumpFileName : read packets from gzip-compressed tcpdump file.
Without -r, -i or -z options, NMS reads tcpdump data from standard input. Not having any of these options is incompatible with the –m option.

6 Active Networking Response System

SAN uses active networking technology to implement defensive reactions on behalf of a host that believes that it is under attack. Owing to the immaturity of active network prototypes, fairly limited defensive reactions have been explored thus far. We have implemented one active network based reaction capability, called trace and isolate. The trace and isolate capability allows a host that is under attack to request the active network to find the physical source of the attack and isolate that source from the network at the active network router closest to the source. Thus, in addition to detecting attacks and protecting itself using the SMS and NMS, an attacked host can help protect the network from excessive and illegitimate usage by causing the attacker to be isolated.

The active network based response is interfaced with SMS and NMS through the CIDF manager. SMS and NMS themselves do not decide to invoke active network reactions, but instead merely issue intrusion reports to the CIDF manager. If the CIDF manager detects conditions that warrant the active network reaction, the CIDF manager requests the active network to initiate the reaction. We are not researching correlation techniques, so our current CIDF manager is simply hardcoded to request active network reaction for one attack only. When the CIDF manager receives more than a threshold number of pop3d password cracking attempts, the CIDF manager responds by asking the active network to trace and isolate the source of the attack. For the pop3d password-cracking exploit, the source IP address is not spoofed, and so using it as the basis for the trace and isolate capability is correct. For other attacks, such as the r-utilities trust exploit, the source IP address is spoofed, and so using it for tracing and isolating would be incorrect, and might even deny service to an innocent bystander. While we believe that an active network trace and isolate capability for spoofed IP addresses is feasible, we have not yet implemented it, so we are currently limited to tracing and isolating non-spoofed sources.

The ANRS uses Programming Language for Active Networks (PLAN) [PLA99], which is a resource-bounded functional programming language that uses a form of remote procedure call to realize active network packet programming. PLAN Demons execute on the host computer being protected and are assumed to run on some of the routers in the network.

When the CIDF manager decides to invoke an active network reaction, that CIDF manager informs the local PLAN demon of the reaction to initiate and its parameters. In our prototype, we implemented two reactions: trace and isolate (a.k.a. `stopHim`), and trace and restore (a.k.a. `startHim`), and they both have a single argument: an IP address. The local PLAN Demon launches PLAN packets containing the algorithm for the requested reaction into the network, where routers running the PLAN Demon receive them and execute their contents. The trace and isolate reactions is shown in Code Example 6-1. Trace and restore is similar.

Code Example 6-1

```
  (*
 PseudoCode:
 Is this address homed on one of my interfaces?
      Yes -> shut down packets from this address
            Exit
      No -> send this packet over all of my interfaces except
            The one on which this packet arrived.

*)
 (*
 This routine takes the address of an unwanted host and
 blocks his packets at the source.
  *)
 fun stopHim (addr: string, aport: string, aproto: string) : unit =
     let val fldret:string * bool = foldr(checkAddr, thisHost(), (addr,
 false))
         val nlist: (host * dev) list = getNeighbors()
         val islocal:bool = snd fldret in
         if islocal
         then blockHim(addr, aport, aproto)
         else if nlist = []
              then print("I have no neighbors\n")
              else foldr(sendPacketS, nlist, (addr, aport, aproto))

   end
```

Trace and isolate uses flooding to determine the location of the attacker. The feasibility of flooding algorithms in large networks is an open issue. The algorithm uses information about the statically configured routing tables to decide if a particular router is connected to the attacker, and thus the algorithm is appropriate only for non-blind attacks. Also, the issues of authentication and security for communication between PLAN demons have not been addressed.

7 Experimental Results

7.1 1998 Lincoln Labs Evaluation of NMS

7.1.1 NMS Detection Accuracy

Our NMS participated in a comprehensive evaluation of intrusion detection systems conducted by MIT Lincoln labs [Graf et al 98]. As mentioned earlier, NMS was designed to complement (but not duplicate) the detection capabilities of the SMS. As such, NMS is tuned mainly for identifying attacks that do not manifest clearly in system call events. Such attacks include most probing attacks, since the probes are typically seen at the OS level, but not at the application level. It also includes several denial-of-service attacks that exploit OS errors to bring the system down. However, denial-of-service attacks that disable a specific higher-level service are not detected, and are left to be dealt with by the SMS. The other two categories of attacks in the Lincoln Labs data, namely, user-to-root and remote-to-local, invariably involve application processes, and as such are not within the scope of NMS.

At the time of 1998 evaluation, the following attacks were within the NMS repertoire. With a few exceptions (such as `fraggle` and `nestea`), all of these attacks were present in the Lincoln Labs data.

Attack Name	Description
`ipsweep`	Surveillance sweep performing port sweep or ping on multiple hosts
`land`	Denial of service using TCP packet with the same source and destination address
`neptune`	SYN-flood denial of service on one or more ports
`pod`	Ping-of-death denial of service attack
`teardrop`, `nestea`, `boink`	Denial-of-service attacks involving overlapping IP fragments
`portsweep`	Indiscriminate sweep through a large number of ports to identify available services.
`smurf`	ICMP echo reply flood, caused by sending an ICMP echo request to an intermediate network broadcast address using a (spoofed) source address of the victim host.
`nmap`	probing attack that attempts to map a victim network using a combination of stealthy and non-stealthy `portsweeps` and `ipsweeps`
`satan`	Vulnerability probing tool which checks for a variety of vulnerabilities. Typically recognized by the `portsweep` that is an integral component of `satan` probing.
ping flood	Denial of service involving a flood of ICMP packets, but no sign of `smurf`
`UDP` flood	Similar to ping flood, but involves UDP rather than ICMP
`smurf` intermediate site	Use of protected network as the intermediary network for launching `smurf`
`fraggle`	Similar to `smurf`, but involves simple UDP services rather than ICMP
`fraggle` intermediate site	Similar to `smurf` intermediate site, but applies to UDP rather than ICMP

The attack data was provided in the form of `tcpdump` files that ranged in size between 300MB and 1.2GB for each day of the two-week evaluation period. Participants were required to tag each packet in the `tcpdump` file as representing an attack or not. Optionally, a probability could be assigned to indicate the likelihood of an attack. For TCP, entire sessions were tagged as opposed to individual packets. These "raw results" were then processed by MIT Lincoln labs to produce standardized scores for all the participants.

The following table summarizes the results of the 1998 evaluation. The attacks are identified using rules that are generally similar to the examples discussed earlier. However, in the process of training and debugging the system, we have found that the rules tend to get a bit more complicated than the examples. At times, we have also had to change the rules due to certain idiosyncrasies or artifacts in the test data. The table lists only those attacks in the

1998 evaluation data that are within the scope of NMS. The table shows that NMS is very effective, missing only two of the attacks in the evaluation data. Even these misses were due to the fact that the UDP storm attack had not been properly coded or tested previously in the NMS.

Attack	Number	Detections	Score
Smurf	8	**100%**	100%
Teardrop	4	**100%**	100%
Land	2	**100%**	100%
Ping of Death	5	**100%**	99%
IP sweep	3	**100%**	96%
satan	2	**100%**	94%
portsweep	5	**100%**	90%
saint	2	**100%**	89%
nmap	4	**100%**	78%
Neptune	7	**100%**	70%
mscan	1	**100%**	55%
UDP Storm	2	**0%**	0%
Total	45	**96%**	85%

The third column in the table shows the score assigned to each attack by Lincoln Labs. They assigned fractional scores for attack detection, based on the fraction of sessions (or packets, in the case of non-TCP attacks) that had been tagged as attacks. Thus, if a neptune attack consists of 20 SYN packets, and an IDS tags the last 15 SYN packets as attack bearing, it would score 75% rather than 100%. Clearly, this is not a good way to score, since no IDS will have a basis to tag the first 5 SYN packets as part of an attack, unless it can predict that in the future, it is going to receive many more SYN packets. For this reason, it appears better to count the number of detections, as we show in the second column, rather than assigning fractional scores. We define an attack to have been detected if a majority of the attack-bearing packets (or sessions) have been identified. It is worth noting that in 1999 evaluation, Lincoln Labs abandoned the idea of fractional scores, and went with a different method that counted number of detections.

7.1.1.1 False Positives

One of the strengths of NMS lies in its low false positive rate. We report the false positives as a fraction of true positives, instead of the "false positives per day" approach used by Lincoln Labs. In our opinion, counting the number of false positives in a day is not meaningful: this number will increase in proportion to the amount of traffic encountered in a day, and also the number of attacks detected by a system. Consequently, one cannot compare the false positives across different IDS or different protected systems. In contrast, when we express false positives as a fraction of true positives, we can compensate for these factors, and it is hence meaningful to compare the measures across different systems and installations.

In 1998, NMS achieved a false positive rate of 5% to 10% when it is expressed as a fraction of true positives. Most other systems, especially those based on anomaly detection, reported false positives that were one to three orders of magnitude higher.

7.1.2 NMS Performance

Our emphasis on efficiency of implementation paid off in terms of performance, as shown by the CPU and memory usage of our IDS for the ten days of test data as shown in the table below. While running on a 450MHz Pentium II

PC running RedHat Linux 5.2, our system can sustain intrusion detection at the rate of 15s/GB, or equivalently, about 600MB/second. (In measuring the CPU time, we considered only the time spent within the intrusion detection system, and ignored the time for reading packets from the `tcpdump` file.) Its memory consumption is also very low, largely the result of our design of the decay counter and MFU table data structures.

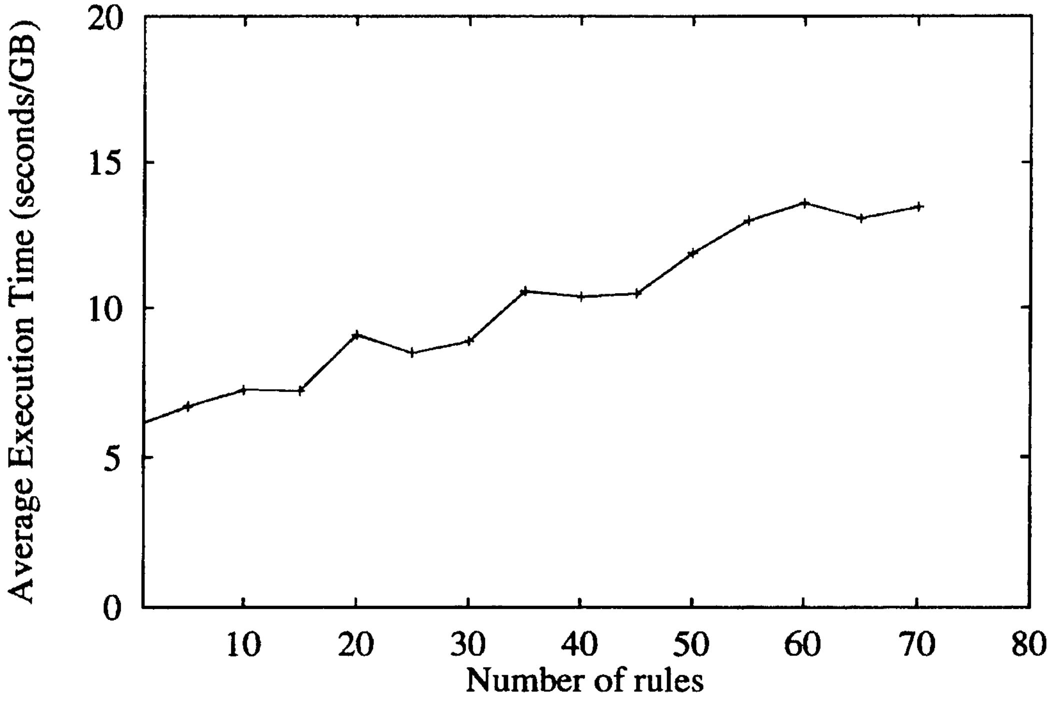
Week	Day	`tcpdump` file size (GB)	Time taken/GB (data)	Memory usage
1	Monday	0.41	7.6	<1MB
1	Tuesday	0.84	21.4	<1MB
1	Wednesday	0.46	12.2	<1MB
1	Thursday	0.76	21.8	<1MB
1	Friday	0.43	17.4	<1MB
2	Monday	1.2	21.4	<1MB
2	Tuesday	0.45	15.3	<1MB
2	Wednesday	0.54	8.7	<1MB
2	Thursday	0.6	13.0	<1MB
2	Friday	0.5	10.6	<1MB
	Average	0.62	13.8	<1MB

The high performance is the result of our emphasis on the following aspects:

insensitivity of the pattern-matcher to the number of rules. Our IDS currently contains about 75 rules, so any pattern-matching approach that involves checking each of this patterns individually will be slow. By compiling the patterns into an automaton, we are able to identify all pattern-matches, while spending essentially constant time per packet that is independent of the number of patterns. Thus the pattern-matching time remains independent of the number of rules. This factor is further illustrated in the graph shown below.

fast implementation of data aggregation operations. As described earlier, we have implemented the weighted counter and table data structures so that operations on them have an amortized O(1) cost per operation. As a result, detection time increases only linearly (and slowly) with the number of attacks.

The following picture shows how the performance of NMS changes with increase in the number of rules in the specification. The intrusion detection time increases slowly with the number of rules. If we split the time into the time spent for pattern matching and for intrusion detection, we note the pattern matching time remains constant even when the number of rules is increased. However, the time for intrusion detection increases slowly. This is due to the fact that when more rules are added to the specification, the likelihood of any event matching a pattern is increased, so the number of (right-hand side) code sections that executed on reception of an event increases. Naturally, this increases the runtime. Note, however, that the runtime is not increasing monotonically with the number of rules. This is because of the fact that the addition of a new rule can reduce the frequency with which an earlier rule was matching, e.g., in the port sweep example, if we had included the second rule but not the first one, the second rule would be applicable much more frequently. This can lead to the situation where the addition of rules *decreases* the execution time.



7.2 1999 Lincoln Labs Evaluation of NMS

Much of the SAN development efforts were spent in the SMS system during the 9 months between the 1998 and 1999 evaluations. As such, the changes to NMS were not extensive. The specific changes were:

Addition of new attack specifications: Several new attacks had been included for the 1999 evaluation. These include:

UDP Storm: This attack sets up a loop between two simple UDP services (e.g., `chargen` and `echo` services), which results in a flood of packets back and forth between the two services, saturating the network.

Very small IP fragments: Use of IP fragments that are too small to hold TCP or UDP headers, typically aimed at spoofing some implementations of packet-filtering firewalls.

IP spoofing: Attempt by an attacking machine to establish a TCP connection with a victim machine, while using the source address corresponding to a different machine.

Use of unusual IP parameters: Some attacks use unusual IP options as a way to confuse or circumvent firewalls or other security mechanism built into servers.

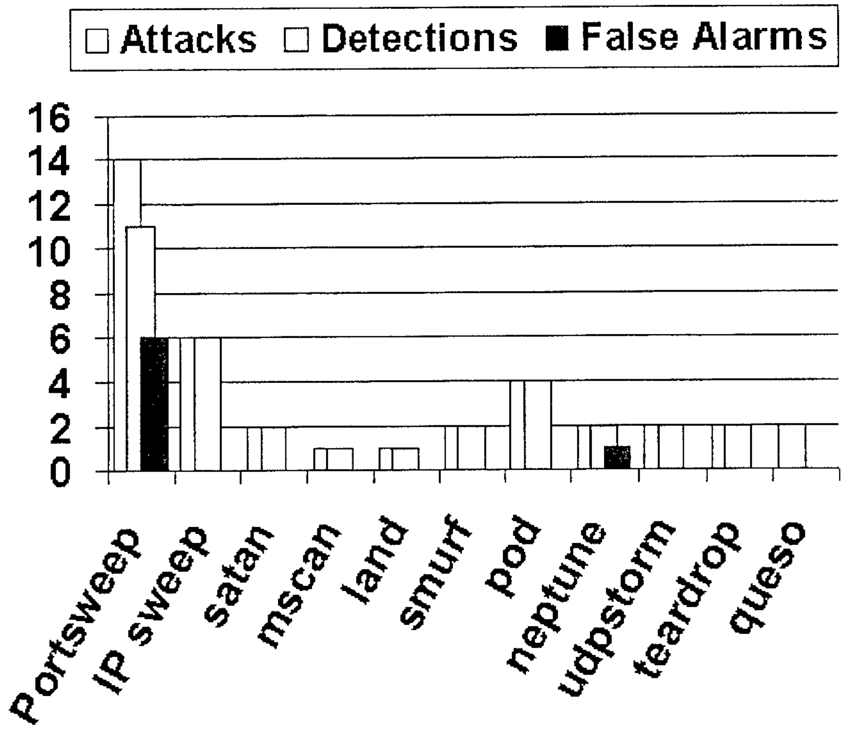
Improving accuracy of detection of known attacks: Techniques for detecting many attacks such as IP and portsweeps and `neptune` were improved for the 1999 evaluation.

Improved reporting of attacks: The attack-reporting format was significantly improved. In particular, the detection reports identify the affected host and service, the attacking source and service (if not spoofed), and begin and end times for the attack. The format of a report looked like the table below at the beginning of the 1999 evaluation. For the evaluation, we modified this format further so that it conforms to the format prescribed by Lincoln labs.

Time	Attack Name	Begin or End	Source and/or Destination Info
09:51:21	Portsweep	attack began	207.136.086.223→ 172.016.114.050
09:51:21	Portsweep	attack ended	207.136.086.223→ 172.016.114.050
13:49:52	Ping of death	attack began	172.016.114.050
13:49:52	Ping of death	attack ended	172.016.114.050
15:08:17	Teardrop	attack began	172.016.113.050
15:08:18	Teardrop	attack ended	172.016.113.050

7.2.1 Effectiveness of NMS

The following picture summarizes the performance of NMS. It considers only those attacks in the 1999 Lincoln Labs evaluation data that were within the scope of NMS.



At the time the results were submitted to Lincoln Labs, NMS had missed some of the IP sweeps and port sweeps in the attack data. This was because of the fact that the definition of these attacks had not been specified clearly. In the 1998 data, port and IP sweeps always touched many hosts and/or ports. For 1999, Lincoln Labs changed this assumption, so attempts to access the three different hosts or ports over a short time were to be considered as sweeps. Since this change of assumption was not communicated to the participants, most participants did not recognize these attacks. After this assumption was made known, we changed our thresholds for portsweep and IP sweep to very small values. The picture above shows the results obtained after this change.

NMS failed to detect three of the port sweeps and all of the queso attacks. All of the missed port sweeps consisted of two or more hosts that probed exactly three ports (i.e., no host probed more than 2 ports). Such port sweeps are virtually impossible to detect in any system, so it is not clear to us why these were considered to be "attacks" by Lincoln Labs. The queso attack was not detected because we had not modeled the relevant protocols or the attack itself.

Our detection rate in 1999 was very similar to that obtained in 1998. Every attack that had been modeled in the NMS was detected, except for the questionable port sweep attack discussed above. This reflects a significant degree of improvement in detection accuracy over 1998, since the 1999 attacks were much more stealthy and sophisticated.

7.2.2 NMS Performance

No performance improvements were made to the NMS in 1999. As such, our performance in 1999 was essentially unchanged from what was described in 1998.

7.3 1999 AFRL evaluation experience

The SMS participated in the 1999 DARPA sponsored on-line evaluation conducted by Air Force Research Laboratory (AFRL). The SMS performance was completely successful. SMS detected all of the attacks presented to it and reported no false positives. Furthermore, the test was conducted at AFRL–Rome Site facilities, which necessitated remote SMS installation and operation. No significant installation or operational difficulties were encountered, in contrast to other systems being evaluated, some of which where so difficult to install and configure that they delayed the evaluation by over one month.

The 1999 AFRL evaluation comprised defending the standard telnet and ftp applications, and defending one customized application, called GlobalView. We were instructed to disable our reaction capabilities for the evaluation so that we would only detect attacks, not prevent them from succeeding. While we accommodated this requirement it did create two problems. First, it meant that some of the power of the SMS could not be shown through the evaluation, and second, for some attacks, such as the race condition exploit, detection and reaction are inseparable. That is, our defense remediates the underlying error, but cannot tell if stimulation of the code containing the error is for the purpose of legitimate or attack usage. To accommodate we increased the complexity of the race condition defense to enable some degree of separation between detection and reaction, however, this ultimately proved unnecessary since AFRL did not actually exploit the race condition.

7.3.1 GlobalView

GlobalView was written by AFRL for the purpose of the evaluation and it contained numerous vulnerabilities, all of which, it can be argued, mimic vulnerabilities in real applications. We were provided with the GlobalView source code, and a usage document similar in content to UNIX manual pages, and given approximately one week to prepare a defense.

The details contained in the user documentation enabled us to quickly develop the operational view of GlobalView that is shown in Figure 4. From this understanding we deduced the following security related issues.

- Permissible files for GlobalView to access and their access modes,
- Lack of forks, clones, or execves
- Race condition on rename of temporary to final output files, and
- Buffer overflow potential on command line, password file, and bases file reads.

When we inspected GlobalView's source code we discovered additional security vulnerabilities:

- Race condition in mktemp,
- Failure to handle syntax errors or large fields in password file,
- Failure to handle reads > 4096 from server,
- Failure to check return codes from mktemp, calloc, fclose, and fopen and
- Numerous failures to check strings for null terminator

Finally by executing GlobalView (in a limited functionality mode since we did not have access to the server) we observed the GlobalView fails to delete the temporary file, which may contain secure information if exit is premature, for example due to a crash.

In general, the race conditions enable an attacker to corrupt information that legitimate users obtain, the stack overflows enable execution of arbitrary programs as root, and the formatting and failure to check return codes create the possibility of a DOS attack by stimulating unchecked failure conditions that, because they are unchecked, cause `GlobalView` to crash. In addition to a crash causing DOS, failure to delete the temporary file may enable an attacker to obtain sensitive information that is in the file at the time of the crash.

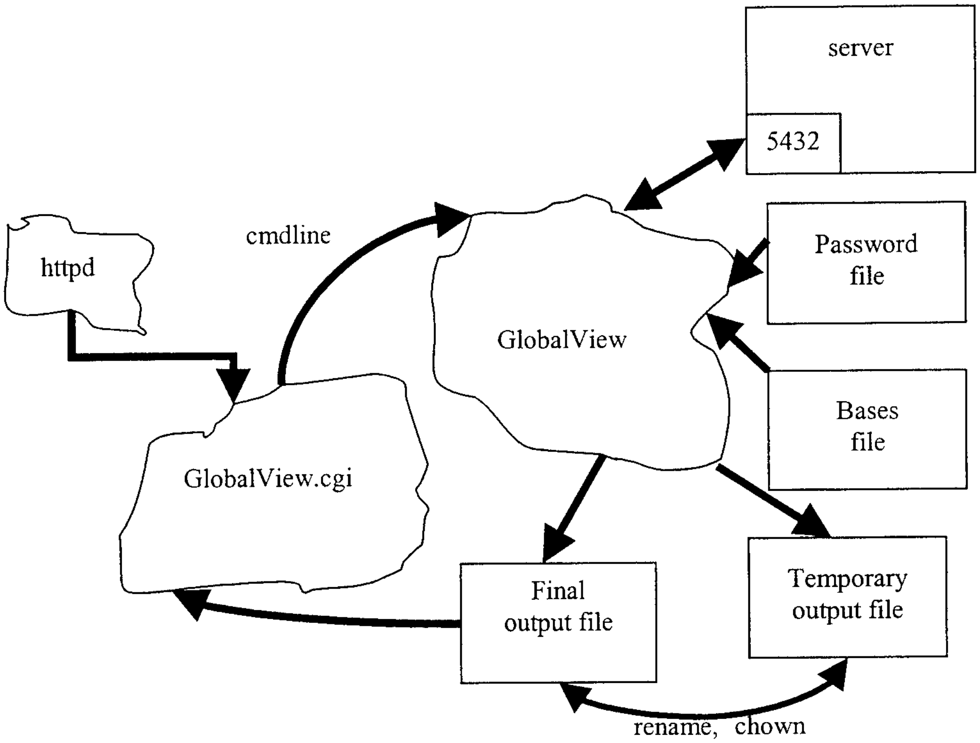


Figure 4--GlobalView Specification

Our experience with other programs like `GlobalView` (that is, server programs) previously led to the development of a set of generic defensive rules that seem to be appropriate to many of these types of programs, and we found these generic rules were appropriate to `GlobalView`. The rules prohibit:

- 32 privileged system calls (`fchown, mknod, mount, umount ...`),
- 9 inter-process system calls (`kill, ptrace,...`),
- 4 userid modifying system calls (`setuid, setpgid,...`),
- 4 setuid-to-root system calls (setuid(a1)|a1=0, setpgid(a1,a2)|a2=0,...),
- 5 resource limit modifying system calls (`sched_setparam, nice,...`), and
- 15 obsolete system calls (`vma86_old,...`).

Since GlobalView does not use any of these system calls, by ensuring at runtime that GlobalView is not requesting any of them we are providing some verification that GlobalView has not been altered, either during or before execution.

Our examination of GlobalView's operation led us to add the following GlobalView specific rules to the generic rules:

- prohibited file access:
- writes to world writeable files,
- writes to all but temporary file,
- read from all but *.so and 5 application files,
- rename to all but GlobalView.out, and
- chown for all but GlobalView.out,
- ensure integrity of mktemp -> open race vulnerability,
- ensure integrity of close -> rename -> chown race vulnerability,
- prohibit unused system calls: execve, fork, clone, ipc, and socketcall with 13 argument values,
- ensure syntactical integrity of password file,
- prohibit connects to ports other than 53 or 5432, and
- prohibit connect before authentication.

Because we wanted to achieve the best possible evaluation result, we spent the entire week allotted to us studying GlobalView and improving our defense. This amount of effort was far more than actually necessary, as a post-evaluation analysis revealed that the rule which detected the GlobalView intrusion was a simple rule which had become apparent to us with only a few hours of GlobalView analysis. The resulting BMSL program for defending GlobalView contained about 30 rules and compiled into a C++ program for about 10,000 lines. The imod.o which was ultimately produced has a size of about 1 Mbytes, and about 750 Kbytes of that imod.o is required for the infrastructure functions, so about 250 Kbytes of object code was needed for the GlobalView defense.

The AFRL's attack against GlobalView consisted of only stack overflow attacks, and those attacks followed the popular approach of using the stack overflow vulnerability to obtain a root shell, by tricking GlobalView into requesting the execve("/bin/sh") system call. Our defenses detected the attack perfectly. Furthermore, AFRL applied normal traffic to GlobalView with our defense enabled, and the defense produced no false positive reports. It is also of interest to note that for their convenience in launching the attack, AFRL used a modified version of GlobalView, so the source code from which our defense was derived did not exactly match the source code of the object our defense actually protected.

We believe that our excellent performance with the GlobalView defense during the AFRL evaluation does not begin to illustrate the full potential of the SMS approach. As the previous description of our defense for GlobalView shows, we could have defended against many more attacks than the one that AFRL actually used. Moreover, AFRL did not attempt to make the stack overflow attack stealthy by exploiting it for purposes other than obtaining a root shell. (With the SMS approach, stealth refers to how similar the behavior stimulated by an attack is to the behavior stimulated by legitimate usage.) Had the AFRL attacks attempted to exploit the stack overflow to perform other damaging actions, such as destroying important files, then out defense might have been able to demonstrate detection of stealthy attacks. As an experimental method, we believe that the attacks in the evaluation should progress from non-stealthy to extremely stealthy for each vulnerability, and should cover a breadth of different vulnerabilities in order to determine the true limits of the defense.

7.3.2 Telnet

Telnet is a standard client/server application that allows general-purpose remote access. The client side program is called telnet, and the server side program is called telnetd. We built a defense for telnetd based on the same generic defensive rules as those described for the GlobalView defense, plus additional rules to defend against publicly described telnetd vulnerabilities. AFRL did not actually attack telnetd, but instead used telnet to obtain legitimate remote access to the victim computer, and then within that telnet session AFRL conducted an insider attack using another application. However, since our telnetd defense was configured in such away that it would apply the defense to child processes, it was able to detect the attack.

The insider attack that AFRL used within the telnet session was to run the vulnerable set-uid-to-root program called libterm. Libterm is vulnerable to stack overflow, and the vulnerability was exploited to coerce it into creating an xterm window as root, essentially giving the attacker a root shell. Our telnet defense detected the inappropriate execve system call because it occurred while the effective user was root. The telnet defense was also subjected to normal usage and it reported no false positives.

As with GlobalView, we do not believe that the AFRL attack on telnet in any way stressed our detection capabilities, so the fact that we had perfect performance is not a strong characterization of our approach. Rather, we believe that if AFRL had performed more complex attacks, the telnet defense would have detected some of them, and might have missed some others. Furthermore, the degree to which the normal usage pushed our defense with respect to false positives is unknown.

7.3.3 Ftp

Ftp is a standard client/server application that allows special purpose remote access for the purpose of file transfer. The client side program is called ftp and the server side program is called ftpd. Our defense for ftpd was similar to our defense for telnetd, although the ftpd defense is possibly stronger because we had actually inspected the source code in order to produce it.

AFRL attacked ftpd by exploiting a stack overflow vulnerability that is triggered by creating a directory with an exceptionally long name to obtain a root shell. Our defense detected the attack, and when subjected to normal usage, did not report any false positives.

As with GlobalView and telnet, we believe that a more thorough set of attacks on ftpd would have both demonstrated more of our detection capabilities, while at the same time identifying the limits of our current detection capability.

7.4 Internal evaluation experience

7.4.1 Classification of CERT Advisories (1993-1998)

One of our first internal efforts in assessing the SMS/NMS combination was based on the CERT database of UNIX-related advisories over a five-year period (1993-1998). We classified these advisories into several categories based on the underlying vulnerability. Based on this classification, we proceeded to examine if each of these attacks could be detected by SMS or NMS. The following table summarizes the results of this (theoretical) assessment. Clearly, it would have been more useful had we actually established these results experimentally, but this is not feasible, since it requires that (a) NMS and SMS be ported to many different UNIX versions, and (b) that the exact configuration of target machine be created, including the specific version of OS and applications.

Category	Count	Detectability
Buffer Overflow	29	Most
Insecure filename handling	14	Most
Inadequate argument checks	7	Most
Insecure program features	7	Most
Trojan Horse	3	Most
Kernel-level errors	3	Most
Weak authentication/encryption	7	Some
Configuration errors	10	Some
Unknown	16	Unknown
Total	96	

The classification itself is described in more detail below. The classification is structured so that there is a good correlation between the category and the technique we can use to prevent the attack (if we can). Due to the fact that the alerts do not provide detailed information about attacks, we have to perform the classification as well as assessment of detectability (using our approach) with limited information. To that extent, there may be some errors and unknowns in the information provided below.

7.4.1.1 Buffer Overflow

Attacks in this category involve exploiting buffer overflow errors to have programs (typically privileged programs) to execute arbitrary commands, such as execve'ing an interactive shell to the attacker. Typically, the commands executed will cause the attacked program to take actions that it would not take under normal circumstance. As such, these attacks can be detected by our specification-based approach.

1. CA-98.05 Topic 1:Inverse Query Buffer Overrun in BIND 4.9 and BIND 8
2. CA-97.26 Buffer Overrun Vulnerability in statd(1M) Program
3. CA-97.24 Buffer Overrun Vulnerability in Count.cgi cgi-bin Program
4. CA-97.23 Buffer Overflow Problem in rdist
5. CA-97.21 SGI Buffer Overflow Vulnerabilities
6. CA-97.19 lpr Buffer Overrun Vulnerability
7. CA-97.18 Vulnerability in the at(1) program
8. CA-97.17 Vulnerability in suidperl (sperl)
9. CA-97.13 Vulnerability in xlock
10. CA-97.12 Vulnerability in webdist.cgi
11. CA-97.11 Vulnerability in libXt
12. CA-97.10 Vulnerability in Natural Language Service
13. CA-97.09 Vulnerability in IMAP and POP
14. CA-97.08 Topic 2: Second vulnerability related to INN - ucbmail, Topic 1: Vulnerability in innd
15. CA-97.07 Vulnerability in the httpd nph-test-cgi script
16. CA-97.06 Vulnerability in rlogin/term
17. CA-97.05 MIME Conversion Buffer Overflow in Sendmail
18. CA-97.04 talkd Vulnerability
19. CA-97.02 HP-UX newgrp Buffer Overrun Vulnerability
20. CA-96.24 Sendmail Daemon Mode Vulnerability
21. CA-96.20 Sendmail Vulnerabilities
22. CA-96.18 Vulnerability in fm_fls
23. CA-96.14 Vulnerability in rdist

24. CA-96.13 Vulnerability in the dip program
25. CA-96.04 Corrupt Information from Network Servers
26. CA-95:17 rpc.ypupdated Vulnerability
27. CA-95:13 Syslog Vulnerability - A Workaround for Sendmail
28. CA-95:04 NCSA HTTP Daemon for UNIX Vulnerability
29. CA-94:02 Revised Patch for SunOS /usr/etc/rpc.mountd Vulnerability

7.4.1.2 Insecure Handling of Filenames

This is another class of attacks that involve programs that fail to perform adequate permission checking on files. This may be because the program fails to perform any permission checking, fails to verify presence of a file before creating (and erasing previous content), fails to check if the file is a symbolic link, etc. Even if permissions are checked, there may be race conditions between the file access check and the actual open operation. Intrusions involving these problems are aimed at fooling a program into modifying critical files such as the password file, .login or .profile files, etc. In particular, they typically involve accessing files that the program would not normally access, or involve unexpected overwriting of files or use of symbolic links. Consequently, we expect to be able to capture most of these attacks using our specification-based approach.

1. CA-98.03 SSH agent (Inadequate checking of arguments to setuid program)
2. CA-98.02 CDE vulnerabilities (Inadequate checking of arguments to setuid program)
3. CA-97.03 Vulnerability in IRIX csetup (It is possible to configure csetup to run in DEBUG mode, creating a logfile in a publicly writeable directory)
4. CA-96.27 Vulnerability in HP Software Installation Programs
5. CA-96.25 Sendmail Group Permissions Vulnerability
6. CA-96.23 Vulnerability in WorkMan When a file is specified with "-p", WorkMan simply attempts to create and/or truncate the file, and if this succeeds, WorkMan changes the permissions on the file so that it is world-readable and world-writeable.
7. CA-96.19 Vulnerability in expreserve
8. CA-96.17 Vulnerability in Solaris vold The handling of these files is not performed in a secure manner. As vold is configured to access these temporary files with root privileges, it may be possible to manipulate vold into creating or over-writing arbitrary files on the system.
9. CA-96.16 Vulnerability in Solaris admintool
10. CA-96.15 Vulnerability in Solaris 2.5 KCMS programs
11. CA-96.08 Vulnerabilities in PCNFSD
12. CA-95:09 Solaris ps Vulnerability
13. CA-95:02 Vulnerabilities in /bin/mail
14. CA-93:17 xterm.login

7.4.1.3 Inadequate Argument Checking

This category covers several instances where a program (typically a privileged one) is made to perform actions chosen by an attacker via carefully crafted input or arguments to the program. Buffer overflows and insecure filename handling are in fact special cases of this category, but we have listed them separately since they are so prevalent, and also because more specific defenses (e.g., attempt to execute a system call from code in the data segment or attempt to overwrite unrelated files) can be designed to capture them. The attacks in this category have the typical result of making a process perform actions that it normally does not do, and hence these attacks are generally detectable using our specification-based approach.

1. CA-97.25.CGI_metachar Sanitizing User-Supplied Data in CGI Scripts (other) the author of the script has not sufficiently sanitized user-supplied input.
2. CA-97.15 Vulnerability in SGI login LOCKOUT When LOCKOUT is enabled users may be able to create arbitrary or corrupt certain files on the system, due to an inadequate check in the login verification process
3. CA-97.14 Vulnerability in metamail (insufficient variable checking in some support scripts)
4. CA-96.22 Vulnerabilities in bash (Inadequate checking of arguments to setuid program)

5. CA-96.06 Vulnerability in NCSA/Apache CGI example code (a library function escape_shell_cmd(), which attempts to prevent exploitation of shell-based library calls, such as system() and popen(), contains a vulnerability.)
6. CA-95:14 Telnetd Environment Vulnerability (bug). The extension to telnet provides the ability to transfer environment variables from one system to another. By influencing that targeted system, a user may be able to bypass the normal login and authentication scheme and may become root on that system.
7. CA-95:12 Sun 4.1.X Loadmodule Vulnerability Because of the way the loadmodule program sanitizes its environment, unauthorized users can gain root access on the local machine.

7.4.1.4 Insecure Program Features

Problems in this category involve using the features provided by a program in a manner that would compromise security. Whereas the previous categories typically involved exploitation of a "bug" in a program, this category involves exploitation of a (questionable?) "feature." Some of these attacks can be detected as they introduce unusual behavior in the exploited programs, while many others may be undetectable. In the list below, we use the flag D to indicate that an attack is likely detectable using our technique, and E to indicate that some non-specific effect can be detected. Absence of any flags indicates that the attack is unlikely to be detected by the SMS or NMS.

1. CA-97.27 (D) FTP Bounce by using the PORT command in active FTP mode, an attacker may be able to establish connections to arbitrary ports on machines other than the originating client (bad protocol)
2. CA-98.05 (E) Topic 3: Denial-of-Service Vulnerability in BIND 8 (Invalid DNS record causing a loop in server)
3. CA-97.22 BIND - the Berkeley Internet Name Daemon (cache poison, bad protocol) Cache poisoning occurs when malicious or misleading data received from a remote name server is saved (cached) by another name server. This "bad" data is then made available to programs that request the cached data through the client interface.
4. CA-96.09 (D) Vulnerability in rpc.statd. The vulnerability in rpc.statd is its lack of validation of the information it receives from what is presumed to be the remote rpc.lockd. Because rpc.statd normally runs as root and because it does not validate this information, rpc.statd can be made to remove or create any file that the root user can remove or create on the NFS server.(bad protocol)
5. CA-96.01 (D) UDP Port Denial-of-Service Attack D. When a connection is established between two UDP services, each of which produces output, these two services can produce a very high number of packets that can lead to a denial of service on the machine(s) where the services are offered
6. CA-95:10 (D) GhostScript Vulnerability Older versions of GhostScript do not completely disable the pipe operator that can be used execute commands that can modify files
7. CA-95:08 (D) Sendmail v.5 Vulnerability

7.4.1.5 Trojan Horse

Attacks in this category involve a maliciously altered program that may perform destructive actions when executed. For the most part, this class of attacks is detectable if we developed a reasonable profile of the program. In many cases, simple sandboxing that restricts the file accesses to those that are known to be needed will itself provide a significant degree of protection against this class of attacks. In the specific cases below, the first two are detectable, but the third one is not.

1. CA-94:14 Trojan Horse in IRC Client for UNIX
2. CA-94:07 wuarchive ftpd Trojan Horse
3. CA-94:05 MD5 Checksums (Trojan Horse)

7.4.1.6 Weakness in Encryption or Authentication

The vulnerabilities in this category involve the use of poor authentication or encryption protocol. Some of them are detectable using SMS or NMS, and are flagged as (D). Others are not detectable.

1. CA-96.21 (D) TCP SYN Flooding and IP Spoofing Attacks

2. CA-96.03 Vulnerability in Kerberos The Kerberos Version 4 server is using a weak random number generator to produce session keys. On a computer of average speed, the session key for a ticket can be broken in a maximum of 2-4 minutes.
3. CA-95:01 (D) IP Spoofing Attacks and Hijacked Terminal Connections
4. CA-94:15 (D) NFS Vulnerabilities
5. CA-94:9 /bin/login Vulnerability
6. CA-94:01 ongoing network monitoring attacks. The intruders first penetrate a system and gain root access
7. through an unpatched vulnerability. The intruders then run a network monitoring tool that captures up to the first 128 keystrokes of all newly opened FTP, telnet, and rlogin sessions visible within the compromised system's domain.
8. CA-93:19 Solaris System Startup Vulnerability. If fsck(8) fails during system boot, a privileged shell is run on the system console.

7.4.1.7 Configuration Errors

Configuration errors involve improper file permission settings or improper configuration files to applications. For most part, these errors are either meaningless to detect (e.g., item 2 below) or cannot be detected (e.g., item 5) by our technique. Those that can be detected (D), partially detected (PD), or detected with a nonspecific indication (E) are indicated below.

1. CA-97.01 Multi-platform Unix FLEXlm Vulnerabilities. Insecure configuration of vendor product installation
2. CA-96.11 Interpreters in CGI bin Directories
3. CA-96.10 NIS+ Configuration Vulnerability
4. CA-95:16 (D) wu-ftpd Misconfiguration Vulnerability. The problem is that the variable _PATH_EXECPATH was set to "/bin" in the configuration file src/pathnames.h when the distribution binary was built. _PATH_EXECPATH should be set to "/bin/ftp-exec" or a similar directory that does not contain a shell or command interpreter, for example.
5. CA-95:15 SGI lp Vulnerability The SGI IRIX system as distributed has some accounts without passwords. Among the accounts that are password-less is the lp account
6. CA-94:06 (PD) Writeable /etc/utmp Vulnerability
7. CA-93:6 (D) wuarchive ftpd Vulnerability. A vulnerability exists in the access control mechanism in this version of ftpd.
8. CA-93:3 (PD) SunOS File/Directory Permissions File permissions on numerous files were set incorrectly in the distribution tape of 4.1.x. A typical example is that a file which should have been owned by "root" was set to be owned by "bin".
9. CA-93:13 (PD) SCO Home Directory Vulnerability. Some users have /tmp as home directory
10. CA-93:11 (D) UMN UNIX gopher and gopher+ Vulnerabilities. Ability to read password file.

7.4.1.8 Kernel-Level Problems

This category consists of attacks that are aimed at bugs within the kernel itself. Exploitation of kernel bugs can cause the kernel to crash. Since these problems do not happen at the process level, they are all detected by the NMS.

1. CA-98.01 smurf (ping packet to broadcast address, with spoofed source = attacked host)
2. CA-97.28 IP Denial-of-Service Attacks (KC) (Teardrop: overlapping IP fragments, Land: SYN packet with source = destination)
3. CA-96.26 Denial-of-Service Attack via ping some systems will react in an unpredictable fashion when receiving oversized IP packets. Many ping implementations by default send ICMP datagrams consisting only of the 8 octets of ICMP header information but allow the user to specify a larger packet size if desired(KC)

7.4.1.9 Other Implementation Errors

For a majority of attacks in this category, we do not have sufficient information about the attack to be able to predict whether the problem can be detected using our specification-based approach.

1. CA-98.05 Topic 2: Denial-of-Service Vulnerabilities in BIND9 and 8 (Read invalid regions of memory and crash)
2. CA-97.20 JavaScript Vulnerability Security flaws exist in certain Web browsers that permit JavaScript programs to monitor a user's browser activities beyond the security context of the page with which the program was downloaded
3. CA-97.16 ftpd Signal Handling Vulnerability D (?) This vulnerability is caused by a signal handling routine increasing process privileges to root, while still continuing to catch other signals. This introduces a race condition.
4. CA-96.12 Vulnerability in suidperl. On systems that support saved set-user-ID and set-group-ID, suidperl does not properly relinquish its root privileges when changing its effective user and group IDs.
5. CA-96.07 Weaknesses in Java Bytecode Verifier. A maliciously written applet can perform any action that the legitimate user can perform.
6. CA-96.05 Java Implementations Can Allow Connections to an Arbitrary Host The Applet Security Manager allows an applet to connect to any of the IP addresses associated with the name of the computer from which it came.
7. CA-95:07 SATAN Vulnerability: Password Disclosure depending on the configuration at your site, the supporting HTML browser, and how you use SATAN, your session key may be disclosed through the network.
8. CA-95:03 Telnet Encryption Vulnerability
9. CA-94:13 SGI IRIX Help Vulnerability
10. CA-94:11 Majordomo Vulnerabilities
11. CA-94:10 IBM AIX bsh Vulnerability
12. CA-94:08 ftpd Vulnerabilities (RC)
13. CA-94:03 IBM AIX Performance Tools Vulnerabilities
14. CA-93:7 Cisco Router Packet Handling Vulnerability a router which is configured to suppress source routed packets may allow traffic which should be suppressed.
15. CA-93:18 SunOS/Solbourne loadmodule and modload Vulnerability
16. CA-93:8 SCO /bin/passwd Vulnerability This potential will not allow unauthorized access to a system, but it may deny legitimate users the ability to log onto the system.

7.4.2 Stack Overflow Exploit

The stack overflow exploit remains the most popular of all exploits due to its power; in its most popular form it enables an intruder to obtain a command line shell as root, the highest privilege user. While many programs can have the stack overflow vulnerability, it is only when those programs are also run as set-uid-to-root processes that the vulnerability can be exploited to gain an advantage. (To be precise, an advantage can be gained whenever user Y can exploit a stack overflow vulnerability in a process run set-uid-to-X, where X is a user with more privilege than Y.)

7.4.2.1 Description of Vulnerability

Code Example 0-1 gives an example of a program containing the stack overflow vulnerability. In the example program the character buffer b is intended to store the value of the program's first command line argument, for which the programmer allocated 32 bytes of storage. The instruction which populates the buffer from the command line argument—`strcpy`—knows nothing of the 32 byte limit, and will continue to copy the argument until a null character occurs. Thus if the argument is greater than 32 bytes, b will overflow, and with C programs, the overflow will corrupt the memory space occupied by machine instructions. Thus the overflow actually changes the program's object code. By carefully constructing the argument, an intruder's machine language program can be used to overwrite the original program, and with further care, the address jumped to by the return instruction can be overwritten so that it is replaced with the address of the intruder's program in the overflowed b. Consequently, an arbitrary intrusion program can be installed and executed.

Code Example 0-1

```
main(int argc, char **argv){
   char b[32]
   strcpy(b,argv[1]);
   return;
}
```

7.4.2.2 Description of Exploit

Code Example 0-2 shows an example of a C program that provides a generalized mechanism for exploiting stack overflow vulnerabilities. The buffer shellcode is initialized to a hexadecimal string that encodes a machine language program for invoking the `execve("/bin/sh")` system call. The content of shellcode is ultimately placed in the environmental variable named `EGG`, where it is available for use as input to the program being attacked. The rest of the program merely manages placement of shellcode within `EGG`, which is padded with no-op instructions, in order for the overflow to be properly aligned in the stack of the attacked program. Two command line variables, stack size and offset, give the attacker control over placement. Acceptable values for stack size and offset can either be determined by trial and error, or by study of the program being attacked. Once the proper values are determined, using `EGG` as a command line argument causes the vulnerable program to become a command line shell program with the privilege of the effective user of the program. Since we assume the program is set-uid-to-root, then the result is a command line root shell.

Code Example 0-2 C Program for Exploiting Stack Overflow Vulnerability

```
#include <stdlib.h>
#define DEFAULT_OFFSET 0
#define DEFAULT_BUFFER_SIZE  512
#define NOP0x90
char shellcode[] =
     “\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b”
     “\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd”
     “\x80\xe8\xdc\xff\xff\xff/bin/sh”;
unsigned long get_sp(void) {__asm__(“movl %esp, %eax”);}
void main(int argc, char *argv[]) {
     char *buff, *ptr;
     long *addr_ptr, addr;
     int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
     int i;
     if (argc > 1) bsize = atoi(argv[1]);
     if (argc > 2) offset = atoi(argv[2]);
     if (!(buff = malloc(bsize))) exit(0);
     addr = get_sp() - offset;
     ptr = buff;
     addr_ptr = (long *) ptr;
     for(i=0;i<bsize;i+=4) *(addr_ptr++) = addr;
     for(i=0;i<bsize/2; i++) buff[i] = NOP;
```

```
        ptr = buff + ((bsize/2) - (strlen(shellcode)/2));
        for(i=0;i<strlen(shellcode);i++) *(ptr++) = shellcode[i];
        buff[bsize - 1] = '\0';
        memcpy(buff,"EGG=",4);
        putenv(buff);
        system("/bin/bash");
}
```

7.4.2.3 Avoiding the Vulnerability

The stack overflow vulnerability is best avoided during implementation phase by adopting the coding practice of not copying data from a source buffer to a destination buffer without providing for termination when the size of the destination buffer is exceeded. While this seems a simple enough coding practice it has not been followed in the past, so many existing programs contain the vulnerability, nor is it being followed currently, as hackers find many new programs having the vulnerability. The vulnerability is particularly insidious when it is contained in a library that is used by many applications. Even if the application developers use good coding practices, they cannot eradicate the error in the library, and moreover, once the library vulnerability is found, then intruders can easily identify good candidates for future attacks, since all applications using the vulnerable library are themselves vulnerable.

7.4.2.4 Defending Against the Exploit

In addition to the SMS method of defending against stack overflow vulnerabilities, described later in this section, another technique, embodied in the StackGuard [COWAN ET AL] product, also bears consideration. StackGuard has a different approach than the SMS method because it detects the actual mechanism of the attack whereas the SMS defense detects the damaging behavior about to be caused by the attack. Unlike SMS, StackGuard requires recompilation and is applicable to only one type of overflow, that associated with the stack, whereas the SAN defense does not require recompilation and can detect intrusions which overflow any buffer. The StackGuard defense works by modifying the C compiler to place special values on the stack. Then, at runtime, the stack is checked to determine that the values are indeed on the stack. Since the exploit overwrites stack data, there is a high probability that the exploit will destroy the values. Thus if the values are not found when and where expected, a stack overflow exploit has been detected.

In contrast to the StackGuard technique, which detects actual corruption of the stack, the SMS defense, shown Code Example 0-3, actually allows the overflow to occur, but prevents damage by detecting that a prohibited system call is being requested. Since the example vulnerable program does not contain any `execve` system calls, then it can be assumed that any `execve` request is the result of an intrusion. For other programs, defining the appropriate behavior to be prevented is more difficult. Many real-world vulnerable programs must be allowed to do some `execve` system calls, since their functionality requires it. However, it is usually possible to disallow intrusive `execves` based on either their argument or they time at which they occur during application execution.

To date, nearly all stack overflow exploits try to execute `execve("/bin/sh")` and since this is a very rare system call for any program to execute legitimately, seeing it nearly always means that an intrusion is occurring. In the future, more stealthy exploits may be seen, in which the intrusive program attempts to perform some other damaging system call. For certain applications it may be possible to chose a damaging system call that so closely mimics legitimate behavior that a BMSL program detecting it would be extremely difficult to devise.

The specification in Code Example 4-1 detects the stack overflow exploit against the example program. Furthermore, a similar defense was used for protecting applications in the AFRL evaluation and it successfully defended them.

Code Example 0-3 BMSL Defense against Stack Overflow Exploit

```
rule:    execve ->
      {

      logIntrusion(time(0),time(0),IDIP_ATTACK_CODE_BUFFER_OVERFLOW,100,100)
              exit(-1)
      }
```

7.4.3 *POP3D Password Cracking Exploit*

Attackers use the Post Office Protocol 3 Demon (POP3D) password cracking vulnerability to conduct remote dictionary style password cracking attacks on victim computers. POP3D is a server program that allows remote access to mail services. POP3D requires user/password style authentication, like many other server programs, and it uses /etc/passwd as its password file. Unlike most other server programs, POP3D is fairly silent about failed authentication requests. Whereas most server programs terminate the connection after a few failed user/password combinations, POP3D has no such threshold, and will permit an unlimited number of failures with virtually no report. Thus POP3D is an excellent target for a dictionary style password cracking approach in which numerous user/password guesses most be tried [PHA98].

Unlike vulnerabilities such as stack overflow, it is not correct to call the lack of a threshold an error. It is more of a poor policy decision on the part of the implementers than an error. Nevertheless, SMS allows the vulnerability to be overcome. As such, POP3D demonstrates an important capability of SMS that we only discovered the need for as we began to investigate real-world attacks. That capability is that through system call interception, a programmer can customize the behavior of programs for which source code is not available, for example, to disable unwanted features. This capability seems especially relevant given that recent popularity of the email-borne viruses that use the arguably excessively powerful features of mailer to spread themselves. Although we have not investigated these attacks in detail, we believe that system call interception could be used to limit powerful features that have dubious value to some users.

To disable the unlimited authentication feature of POP3D, we first had to decide what system call sequence uniquely marked the beginning of an authentication attempt. Several markers were possible, but we chose what appeared to be the most intuitive. We defined an authentication attempt as beginning when the sequence of two reads, the first returning a string starting with "user" and the second returning a string starting with "pass" are seen. This marker is consistent with the POP3D protocol definition and is less likely to change as POP3D evolves than other markers we could have chosen. Every time the marker is seen, our defense increments a counter, and if the counter exceeds a threshold, the defense reports an intrusion. Thus the defense is possible through the interception of the read system call and requires a small string comparison for each intercepted read.

We developed an exploit, named pop3Crack for the pop3d password cracking vulnerability, and our defense was able to detect the attack. Given that we have good detection, we decided to add reaction capabilities. Our first reaction capability was to try to deceive the attacker by lying about correct guesses after the threshold had been reached. There are several ways in which the deceptive reaction could be implemented and we chose an implementation that meshed well with our detection implementation. The reaction works by having the read interceptor replace the actual string the attacker supplied for the user name with a name that is known not to exist in the password file. Thus, regardless of what user name the attacker supplies, the read interceptor will overwrite that value with an unacceptable name. POP3D then responds to this user name as it would any other invalid user name. We also observed that the attack can have a secondary effect on the victim in terms of increasing resource utilization even though the defense prevents disclosure of secure information. To mitigate excessive resource utilization we introduced delays by adding the sleep system call to all intercepted reads after the threshold. The delay is observable to the attacker, but most attackers would probably attribute it to network or server congestion rather than detection.

One final aspect of the pop3d defense is that through the intrusion reports that it sends to the CIDF manager, the CIDF manager is able to decide to escalate the defenses by initiating an active network based defense to trace the attack to its source and isolate the attacker's computer from the network by causing the attacker's packets to be discarded as the first router they encounter upon entry into the network.

7.4.4 Race Vulnerability Exploit

As this section explains, the race condition exploit shows a clear distinction between the capabilities of the system call anomaly detection based approach and the system call specification based approach. The exploit also shows that detection and reaction are sometimes inseparable, in the sense that specification based remediation of the underlying vulnerability merely eliminates the vulnerability, it does not detect attempts to exploit the vulnerability.

7.4.4.1 Description of Vulnerability

The race condition vulnerability exists in many commonly used set-uid-to-root programs. The vulnerability stems from an underlying Unix operating system deficiency. The deficiency is lack of operating system support for atomicity within a sequence of system calls. The Unix system call decomposition is such that for some commonly needed operations, no single system call that performs the operation exists. To compensate for this deficiency the programmer must resort to a sequence of system calls to achieve the necessary operation. However, since Unix provides no way for programmers to specify that a sequence of system calls is to execute atomically, it is possible for external factors to change shared resources in a way that invalidates important decisions made during the system call sequences. By way of analogy, the problem is similar to that of concurrency control in database systems: Unix operating system calls are like database read and write operations since they can read and write shared data, and the system calls of one process are interleaved with the system calls of all other concurrently running processes. It is well-known in database systems that when concurrent applications read and write shared data, the interleaving of the concurrent reads and writes must be carefully coordinated so that one application's writes do not corrupt another applications in-progress reads. This coordination is supplied by programmer defined transaction delimiters. The absence of an abstraction such as transactions for system calls means that programmers have no way to ensure that decisions made based on early system calls are not invalidated by other applications before those decisions can be acted upon by later system calls.

The preceding abstract description of the race condition vulnerability is rendered concrete by the C program in Code Example 0-4. Race.C is a contrived program that encapsulates the race vulnerability as it exists in an older version of the popular xterm application, as well as other applications.

Code Example 0-4 Race.C --program with race vulnerability

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
main(int argc, char **argv)
{
     int euid,uid,i=0,n,fd;
     char logFileName[512];
    logFileName[0]=0;
    char buf[512];
```

```
if (argc > 1){
      for(i=1;i<argc;++i){
           if (strcmp(argv[i],”-l”)==0)
                strcpy(logFileName,argv[i+1]);
      }
}
if (logFileName[0] != 0){
     if (access(logFileName,W_OK) != 0) {
          if (errno != ENOENT){
               printf(“no write access to %s\n”,logFileName);
               exit(-1);
          }
     }
     printf(“The race starts now\n”);
 }
 i=0;
 while ((buf[i++]=getchar())!=EOF){
      if (i == 512){
           if (logFileName[0] != 0){
                if ((fd=open(logFileName,O_RDWR|O_CREAT,0666))==-1){
                     perror(“open”);
                     exit(-2);
                }
                if (lseek(fd,0,SEEK_END)==-1){
                     perror(“lseek”);
                     exit(-2);
                }
                if ((n=write(fd,buf,i)) != i){
                     perror(“write”);
                     exit(-3);
                }
                close(fd);
           }
           i = 0;
      }
 }
 if (logFileName[0] != 0){
      if ((fd=open(logFileName,O_RDWR|O_CREAT,0666))==-1){
           printf(“%s\n”,logFileName);
           perror(“open”);
           exit(-2);
      }
      if (lseek(fd,0,SEEK_END)==-1){
           perror(“lseek”);
           exit(-2);
      }
      if ((n=write(fd,buf,--i)) != i){
           perror(“write”);
           exit(-3);
```

```
                }
                close(fd);
        }
}
```

The race condition vulnerability of race.C mirrors that of common race condition vulnerabilities in standard applications set-uid-to-root applications, such as xterm. The vulnerability concerns the access() at line 20 and the open()s at lines 32 and 50. The intent of the access() at line 20 is to guard the open()s at lines 32 and 50, that is, to prevent them from opening any file for which real user does not have write permission. Set-uid-to-root programs frequently use access() to guard open() whenever it is necessary for the set-uid-to-root program to open a file specified by the real user, to ensure that this file is opened in accordance with real user's privilege, and not effective user's (that is root's) privilege. The technique is widespread because it is the technique recommended by Unix manual pages.

7.4.4.2 Description of Exploit

The race condition vulnerability is exploited by changing the file referenced by the variable logFileName and used as the first parameter to access() and open() after the access() has been executed, but before the open() is executed. Assuming that logFileName initially names a file which real user can write, but that after executing access()the attacker can change the file to one that real user cannot write, but that root can write, then the attack gains the ability to write into supposedly protected files. If logFileName has the value "myLogFile", then the two Unix commands:

```
        rm myLogFile
        ln -s /etc/passwd myLogFile
```

if executed in-between access() and open() allow unauthorized writes to the password file.

The attacker's main difficulty in exploiting the race vulnerability is in getting the file changing commands to execute at the right time, that is, before open(), but after access(). Race.C simplifies this difficulty by printing a message—"The race starts now"—after access(), and then waiting for input before executing open(). Race.C also repeatedly closes and opens the file, giving multiple opportunities for the exploit to succeed. These expedients merely reduce the difficulty in exploiting the race vulnerability, they do not elevate the vulnerability from being impossible to exploit to being possible. Even without the expedients, an attacker would merely repeat the exploit until success was achieved. Furthermore, it is not unreasonable for a programmer to write a program that unintentionally contained the expedients. A programmer may decide to do the access() check early in the execution sequence, immediately after the command line arguments are processed, but may defer opening the file until later in the execution sequence, when the program actually has data to write to the file. Also, the technique of multiple closes and opens is often used to conserve file descriptors since a single process usually can have only a small number of open file descriptors at any one time. In our experiments, we used a simple script which scanned the output of the Unix ps command looking for an invocation of race with a -l option, and when such an invocation was seen executed the two file changing commands on that file (not using /etc/passwd for obvious reasons). The script is nearly always successful in exploiting the race condition.

7.4.4.3 Avoiding the Vulnerability

Unfortunately, for many Unix operating systems, there is no absolute way to write correct set-uid-to-root applications that need the functionality implied by the access/open sequence. The underlying problem is insufficient system call functionality, and this deficiency cannot be corrected by any application level method, in general. What is needed is either a single system call that atomically provides the functionality of the access/open sequence, or a transaction-like mechanism by which applications can enforce atomicity over a sequence of system calls.

While there is no general solution to the problem without operating system modification, for specific applications there are some specific solutions that avoid the problem. Primary among these solutions is to change the application so that it does not need to run as a set-uid-to-root program. For many applications, running as set-uid-to-root is merely an expedient; it simplifies application development and installation, so with more effort the application can be re-written so that it does not need to run as root. For other applications, the functionality that requires root permission can be grouped such that it is placed early in application execution, and after performing this functionality, the application can demote itself to real user privilege. There is still a class of application that these solutions do not apply to, and this is the class for which root privilege is required at arbitrary times during execution. Ideally, such programs would be written following the principle of least privilege, promoting and demoting their own privilege level to meet their immediate needs. However, while all Unix operating systems allow demotion, few allow promotion, rendering it impossible to write portable applications based upon this principle.

7.4.4.4 Defending against the Exploit

The race vulnerability appears to be a vulnerability that is best defended against using the specifications-based approach as opposed to the anomaly detection approach, since exploit of the vulnerability does not seem to produce any abnormal process behavior. To test this hypothesis, we produced two defenses; an anomaly based defense using the six system call sequence algorithm, and a specification based defense that recognizes and remediates the vulnerability at the operating system level.

7.4.4.4.1 Anomaly-Based Defense

For the training phase of the algorithm, we achieved 66.7% coverage (measured as described in Section 4.5.4) of race.C using 24 test cases. The only instructions in race.C not covered were the failure cases for the opens, seeks, writes, and closes. The 24 test cases produced 108 unique sequences of six system calls each. After training, the 108 sequences were used to monitor actual execution of race.C. Re-running the 24 test cases produced no anomaly reports, thereby verifying that the training phase was correct. Furthermore, manual execution of race.C, which may have produced variations not seen in the training data, for example, due to timing differences, or different input, also produced no anomaly reports, indicating that training data was sufficient to prevent false positives under typical usage. Convinced that our training phase was adequate, we ran race.C under attack conditions. We discovered that although the attack succeeded, no anomaly reports were produced. This experimental result confirms our analysis that exploitation of the race vulnerability does not cause the process containing the vulnerability to produce system call sequences that differ from those produced during normal execution.

7.4.4.4.2 Specification-Based Defense

The specification-based defense against the race condition is shown in Code Example 7-5. The race condition defense does not fit the usual decomposition of a defense into separate detection and reaction phases. Rather, the defense merely remediates the underlying vulnerability, and the remediation takes place regardless of whether or not the vulnerability is being exploited. The essence of the defense is to recognize the intent of the programmer, that is, to ensure that logFileName is opened with respect to the privileges of real user, not effective user. While the programmer's expression of this intent, access followed by open, where both have the same file name, is not sufficient to actually achieve the intent, the expression is enough to allow recognition of the intent. Once the intent is recognized, a sufficient implementation of the intent can be interposed. The sufficient implementation is to "wrap" the open system call with instructions to interchange effective and real user before the open, and then to restore them after the open. Thus the file will only be opened for writing if real user has write permission for it. Because the remediation is interposed in all executions, whether or not the vulnerability is being exploited, the exploit is not actually detected; it is instead prevented, just as if the vulnerability had never existed. If precise detection, for example for logging purposes were desired, the defense could do some additional checks to see if the file had been

Code Example 0-5 BMSL Defense Against Race Vulnerability

```
interface rProg1_if {
    class CString {
      String get();
      void set(String s);

  };
    event open(CString pathname,int flags, int mode );
    event access(CString pathname,int mode);

  event $open(CString pathname,int flags, int mode );
    int @geteuid();
    int @getuid();
    int @setreuid(int i, int ruid);

  String realpath(String c) const;

};
  module main() {
  int savedEuid;

int changedEuid;
  rule:  access(name, mode)|((ruid==@getuid())&& (rn == name.get())) ..
               open(name1, flags, mode1)|(rn == name1.get()) -->
         {
               changedEuid = 1;
                savedEuid  = @geteuid();
                @setreuid(-1,ruid);
         }
  rule: $open(f, flag, mode)|(changedEuid ==1) -->
         {
               changedEuid = 0;
               @setreuid(-1,savedEuid);
```

changed after the access, however for simplicity our defense does not include this refinement. For our own monitoring purposes, the defense does create an intrusion report every time the remediation takes place, however in a production environment, the defense would either be silent, or assign a very low severity to the report to avoid high false positive rates. With the defense installed, none of the 24 test cases used for anomaly detector training produced any false positives, nor did any of the manual executions produce any false positives. However, all race condition exploit attempts were prevented. The prevention of the race condition exploit caused `open()` to fail, causing the user to see an unexpected, but nevertheless legitimate result.

7.4.4.4.3 Related Issue: Difficulties in preparing the anomaly-based defense

In preparing the anomaly-based defense for race.C we encountered an unexpected degree of difficulty, and while we overcame the difficulty to produce an acceptable training data set, the difficulty is nevertheless worth discussing here. We discovered that even for a trivial program like race.C, production of a sufficient set of test cases to generate a good training data set is not trivial. Our first attempt was driven by consideration of code coverage, and we discovered that 12 test cases produced the desired 66.7% coverage, and that these 12 test cases produced 63 unique system call sequences of length six. However, when we attempted to verify the adequacy of this training data set we discovered that it produced unexpected, and erroneous anomaly reports when race was run manually. The problem with the training data set was that it was produced by running race.C with standard input redirected, but when run manually, race.C did not have standard input redirected. This minor variation caused one additional system call in the training data which did not appear when race.C was executed manually. Also, the way the user typed the input when race.C was run manually affected the number of reads that were actually issued to consume that input. We corrected these problems, adding more test cases to cover them. The additional 12 test cases did not increase code coverage, so while code coverage appears to be of some value in judging adequacy of the test cases for producing the training data set, it is not an absolute metric. Given the trivial nature of race.C we assume that the difficulty we had in producing adequate training data will be compounded for more complicated programs, and that execution environment parameters (for example input buffer size) need to be consider along with internal structure of the program being defended.

7.4.5 R-utilities Trust Exploit

The r-utilities exploit takes advantage of a convenience feature that allows users to define a trust relationship for a set of computers. The relationship lets users with accounts on multiple computers in the set access one computer from another without a password. The exploit described here enables an attacker who has knowledge of the trust relationship, but who does not have access to any of the computers in the set for which the trust relationship is defined, to masquerade as a trusted user and thereby gain access to one of the computers without a password. The exploit is based on two intrusion techniques: denial-of-service, (for example SYN flooding) and IP spoofing. The denial-of-service attack disables one of the trusted computers during the attack, so that it cannot respond while the attacker is using spoofing to assume the identity of the disabled computer. The attack must be done blindly, since spoofing in general means that the attacker cannot see response from the victim computer. Consequently, the attack is unsuitable for interactive sessions, and is typically used to execute a single command that will create a backdoor through which a later, interactive session may be established, for example, by adding the attacker's computer to the set of trusted computers.

7.4.5.1 Description of Vulnerability

The r-utilities are a collection of client/server applications allowing remote computer access. The trust relationship utilized by the r-utilities is established by the presence of a file called `.rhosts` in home directories on each computer that trusts remote connections.

The `.rhosts` file contains multiple records of two attributes each. The attributes are a computer name and a user name. So, for example, if user1 tries to access computer1 from computer2 without a password, access is granted if the following line is in the `.rhosts` file of user1 in computer1.

```
computer2 user1
```

If this record is not present in the `.rhost` file in user1's home directory on computer1, then a valid password is required before access is granted. In the .rhost file, the "+" symbol is a wildcard. If a computer has an `.rhosts` file with a "+ +" line then the r-utilities on that computer will trust all computers and all users.

Given that an attacker has determined (for example, through social engineering techniques) that the example `.rhost` file exists in the home directory of user1 on computer1, the attacker can exploit the trust relationship to gain access to computer1 as user1 from any computer. The exploit requires that the attacker simulate an r-utilities access to computer1 while masquerading as user1 and computer2. Simulating r-utilities access is trivial, since the r-utilities protocol is well-known. Within the r-utilities protocol, the user name is merely passed from client to server with no authentication, so masquerading as user1 is trivial, it merely requires delivery of a packet containing the string "user1" at the appropriate point in the protocol. Masquerading as computer2 is more complicated, and is explained below.

To use r-utilities, a client at computer2 sets up a TCP connection to a well-known port on computer2. All TCP connections are uniquely identified by their source and destination IP addresses, so after the TCP connection is established, the r-utilities use the source IP address as the identity of the computer requesting the r-utilities access.

The r-utilities use this source address when consulting the `.rhost` file to determine if non-password protected access is authorized. Therefore, to exploit the trust relationship, the attacker must spoof computer2 during establishment of the TCP connection to the well-known port. To spoof computer2 during TCP connection establishment, the attacker must construct an initial SYN packet that contains computer2's IP address in the source field of the packet and deliver that packet to computer1, initiating the three-way TCP connection establishment protocol, disable computer2 so that it cannot respond to the SYN+ACK packet of three-way handshake that computer1 sends it after receiving the spoofed SYN packet, and blindly construct and send the third packet of the three-way handshake to computer1.

The first step is trivial, and the second step is easily accomplished using well-known denial-of-service attacks. The third step is difficult, however, because it requires the attacker to construct the third packet of the three-way handshake without ever having seen the second packet. Timing is one difficulty, since the attacker must make sure that the third packet is not sent before computer1 has actually sent the second packet. Given the long timeouts used in typical TCP connection establishment implementation, the timing difficulty is generally not hard to overcome by merely waiting a few seconds. The major difficulty in constructing the third packet is in deciding what sequence number to use, since computer1 will reject the third packet if the proper sequence number is not used. The only sequence number that computer1 will accept is the sequence number in the second packet, but the attacker cannot, in general, see the second packet. The attacker must therefore guess the sequence number.

Guessing sequence numbers requires knowledge of the algorithm that computer1 is using to generate sequence numbers. In theory, computer1 picks sequence numbers at random, but randomness is difficult to achieve. Moreover, randomness implies that a sequence number might be reused shortly after a previous connection using that sequence number was disconnected, and immediate re-use of a previous sequence number can confuse many current TCP implementations. Consequently, most computers today do not generate random sequence numbers. A trivial, but popular sequence number generator merely uses a sequence number that is incremented by a large number every five seconds, and by one for every incoming connection request. If computer1 uses this algorithm, given that the attacker knows the current sequence number at time t, then the attacker can guess the sequence number at t+i with reasonable accuracy. A less trivial, and less popular sequence number generator uses the trivial algorithm, but adds some randomness, so that accurate prediction is impossible.

To determine what sequence number generator is being used by computer1, the attacker must probe computer1, without spoofing, to obtain samples of sequence numbers over time. Imap (a host probing tool that samples sequence numbers and performs other probes) reports that our lab computers have the best sequence number generator, that is purely random. We tried to determine if it was still possible to attack these computers using an r-utilities trust exploit. Our conclusion is that such an attack is possible, but that random sequence number generators require attackers to use sequence number guessing techniques that have signatures that our SMS and NMS can observe.

Sequence number guessing can still be used to attack computers using purely random sequence number generators because of the property shown in Figure 5, which shows the difference between the sequence numbers generated by computer1 for successive connection requests. The graph shows that sequence numbers are not purely random at all, but instead given that the attacker knows the current sequence number, the sequence number for the next connection is likely to be no more than 10,000 away from that number. While 1 in 10,000 odds at picking the right sequence number might seem practically unbeatable, in actuality, the odds can be easily be increased to n in 10,000, by a bursty guess. For a bursty guess, the attacker merely sends multiple copies of the third packet of the three-way handshake to computer1. The copies differ only in their sequence numbers, and all sequence numbers are chosen to fall within the predicted range. Increasing burst size increases likelihood that a correct guess is made; fortunately, increasing burst size also increases the likelihood of detection.

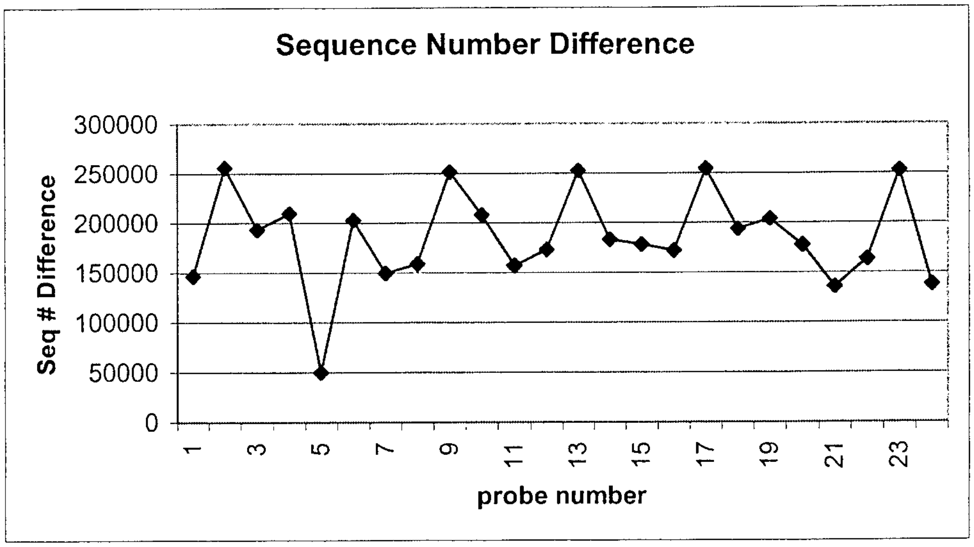


Figure 5—Sequence Number Difference

While the previous discussion shows that general-purpose TCP spoofing, denial-of-service, and sequence number guessing attacks are necessary to mount an r-utilities trust exploit, it has not addressed the specifics of the r-utilities application that the attacker must also overcome. Fortunately for the attacker, there is only one r-utility obstacle to overcome. To explain this obstacle we move from a general discussion of r-utilities to a discussion of the specific r-utilities application likely to be attacked. That application is RSH. RSH allows remote execution of a single command, and since it is inherently not interactive, it is the most likely target of r-utilities trust exploits. The RSH application consists of a client program named rsh and a server program named rshd. Figure 6 shows the protocol used between rsh and rshd. When a remote user invokes rsh, rsh initiates the standard three-way handshake protocol to establish a TCP connection between rsh and rshd. Next rsh sends a data packet containing a single ASCII encoded integer, called the error port number. Following the style of many server programs rshd establishes a second TCP connection to rsh using the error port number, and then forks and execves a child process, passing it one of the connections through inheritance. Rsh then sends a packet with the local user name and a packet with the remote user name and the command to be executed to the rshd child process, which performs the rhost check, executes the command, returns the results over the connection, and terminates the connection. In the described RSH protocol, there is only one obstacle to the r-utilities trust exploit, and that obstacle is the establishment of the second connection. The obstacle imposed by the second connection is exactly the same as that imposed by the first connection, that is, sequence number guessing. In an absolute sense, sequence number guessing was shown not to be an obstacle, and the same technique used to overcome it for the first connection can also be used for the second connection. However, as a practical matter, the effort required for sequence number guessing gets increasingly difficult as time from the last sample increases, so if a second round of sequence number guessing is required, attackers may decide to take their attack elsewhere. Unfortunately, the RSH protocol itself comes to the attacker's aid, since the protocol allows the second connection, as well as the fork and execve, to, be bypassed using a feature we have named the port number zero feature. The feature is activated by setting the error port number, which is contained in the first data packet sent to rshd, to zero. In response to an error port number whose value is zero, rshd skips the second connection establishment, and the fork and execve. Consequently, the parent rshd process receives the remaining data packets itself over the single connection. While the r-utilities trust exploit does not require the port number zero feature, we believe that most attackers would use the port number zero feature to simplify their attack.

In summary, the r-utilities trust exploit is a direct consequence of the r-utilities convenience feature for establishing trust relationships that depends upon unauthenticated information obtained during TCP connection establishment.

The TCP information, that is, the source IP address, is subject to forgery through the well-known attack techniques of denial-of-service, spoofing, and sequence number guessing. The target r-utilities application, RSH, presents only one minor application-specific impediment to the attack, and even this impediment can be disabled through legitimate use of the RSH protocol.

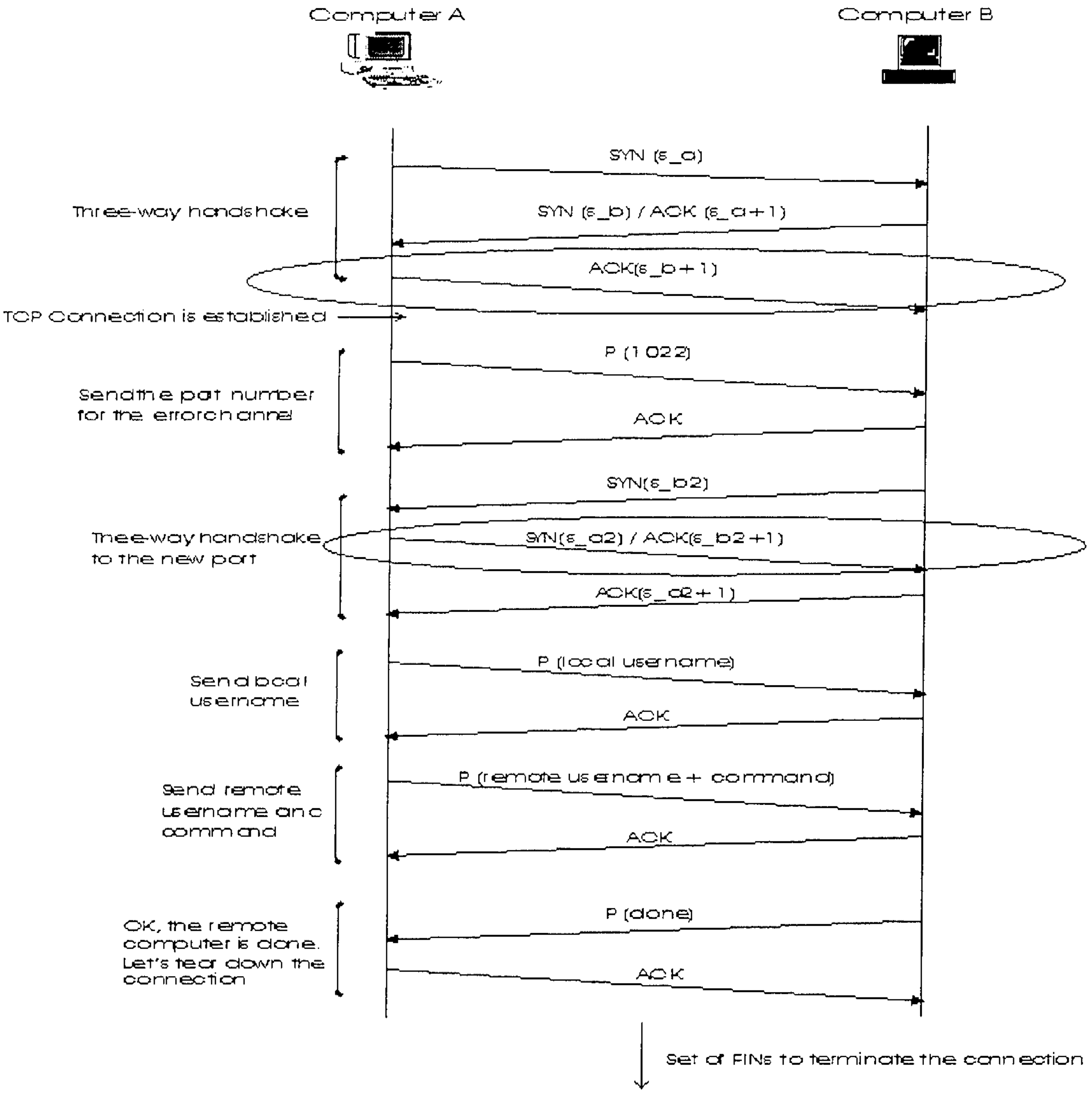

Computer A
Computer B
SYN (s_a)
Three-way handshake
SYN (s_b) / ACK (s_a+1)
ACK(s_b+1)
TCP Connection is established
P (1022)
Sendthe pat number for the errorchannel
ACK
SYN(s_b2)
Thee-way handshake to the new part
SYN(s_a2) / ACK(s_b2+1)
ACK(s_a2+1)
P (local username)
Send bcal username
ACK
P (remote usename + command)
Send remote username anc command
ACK
P (done)
OK, the remote computer is done. Let's tear down the connection
ACK
Set of FINs to terminate the connection

Figure 6—RSH Initiation Protocol

7.4.5.2 Description of Exploit

To experiment with the r-utilities trust exploit, we built a tool called rshx to perform the exploit. The tool produces outgoing packets that correspond to those of rsh, but adds functionality to support sequence number guessing and use of the port number zero feature.

Rshx can operate at three levels of stealth as Table 2 shows.

Table 2

Stealth Level	*Sequence Number Guessing*	*Port Number Zero Feature*
1	one guess	not used
2	one guess	used
3	n guesses	used

Stealth levels 1 and 2 are practical when the attacker believes that there is a high probability of guessing the correct sequence number. At stealth level 1, two correct guesses are needed, while at level 2, only one correct guess is needed. An attacker would use stealth level three if the attacker believes that there is low probability of correctly guessing a sequence number. At level three, the attacker may try to choose an n that achieves an optimal tradeoff between probability of detection and likelihood of success.

In developing `rshx`, we allowed for a special network configuration that greatly simplifies the attacker's task. In this special configuration, the attacker and one of the trusted computers are both on the same shared Ethernet segment, so the attacker can actually sniff the segment to see packets destined for the trusted computer. This special case allows perfect sequence number guessing, since the sequence numbers are not actually being guessed, but are instead being observed. While the special network configuration is in no way required by `rshx`, we took advantage of it to simulate connection to a computer having a trivial sequence number generator for purposes of experimentation. As previously mentioned, our lab computer uses what `imap` calls purely random sequence number generation, (although it is at best only pseudo-random) so using the special configuration allowed us to run two sets of experiments: one in which the victim computer uses trivial sequence number generation, and one in which the victim computer uses purely random sequence number generation, without requiring any changes to our actual experimental network configuration.

We discovered that `rshx` performed exactly as intended. `Rshx` is always able to exploit an r-utilities trust relationship if perfect sequence number guessing is possible. If perfect sequence number guessing is not possible, the probability that `rshx` can exploit the trust relationship is a function of the size of the sequence guessing burst.

7.4.5.3 Avoiding the Vulnerability

The primary means for avoiding the r-utilities trust exploit is to disable the trust feature at a policy level. Disablement can be accomplished in two ways:

`rshd` can always be started with options that preclude checking for user `.rhost` files, and
users can be prohibited from creating `.rhost` files.

Enforcement of these policies is difficult and depends upon an organization's ability to exert strong controls over the actions that computer users can perform. Any user who can become root can start `rshd`, and that user might either not know the security policy and subvert it by accident, or may just make a mistake in specifying `rshd` parameters. Even worse, since all users can write in their home directories, any user can subvert the policy by creating a `.rhost` file. Given users' preference for convenience over security, it is likely that a user would deliberately subvert the policy, albeit for purposes that user considers legitimate. We know of no tool, other than SMS, that would enforce, in real time, either the start up of `rshd` with parameters allowing use of the trust feature, or the creation of `.rhost` files in user directories. There are tools for file system and process scanning that can be configured to detect `.rhost` files in user directories, and an improperly configured `rshd`, but these tools allow the vulnerability to exist between the time of its creation and the next scanning interval.

7.4.5.4 Detecting the Attack

Attack detection depends upon the ability to define observable signatures of the attack. In the SAN project, observable signatures are limited to network packets that the victim receives, and system calls which applications—in this case rshd—running on the victim request.

Rshx at stealth level 1 has no observable packet or system call signature. Fortunately, we expect stealth level 1 to be rare, since it requires either a rare network configuration, or a weak sequence number generator.

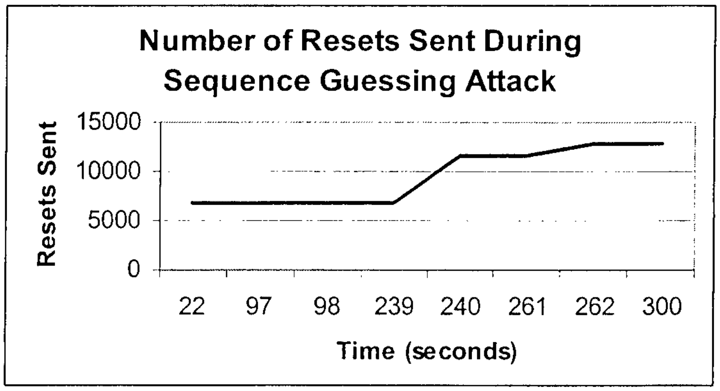


Figure 7—Resets sent in response to bursty sequence number guessing

Rshx at stealth level 2 has one observable signature. The signature is related to the use of the port number zero feature. We believe that while the port number zero feature is a legitimate feature, since legitimate rshs never use port number zero, its use is always coincident with an attack. The port number zero feature signature can actually be seen in both the intercepted incoming packets destined for rshd and the intercepted system calls requested by rshd. In the packets, the port number zero feature is observed by modeling the RSH protocol and detecting that the content of the first data packet in the protocol contains port number 0 in the data field. In the system calls, port number zero is observed by noticing the omission of the system calls associated with establishing the second connection and doing the fork and execve. While either signature is valid, we prefer the system call signature, since building a packet monitor to understand the RSH protocol may require more effort.

Rshx at stealth level 3 has two observable signatures. One signature is the same as that for rshx at stealth level 2, that is, use of the port number zero feature. The other signature is the burst of packets with incorrect sequence numbers that is produced by the bursty sequence number guessing attack. The burst of sequence numbers with incorrect sequence number is observable through the packets visible to the victim in two ways. First, receipt of packets with incorrect sequence numbers prompts the victim to respond with reset packets. Since the number of reset packets sent is maintained in an SNMP MIB variable, then periodic probing of the variable can provide evidence that bursty sequence number guessing is occurring. Figure 7 shows the how the bursty sequence number guessing attack affects the MIB variable. Note the rapid increase of resets at minute four, which coincides with an attack. Resets may be generated for a variety of reasons, so probing of the MIB variable is not conclusive, and requires definition of a threshold, making it subject to evasion by stealth. Second, the NMS can be programmed to detect the actual burst of packets with bad sequence number guesses. Since the NMS can look directly at the attacking packets, rather than looking at merely an aggregation of an artifact caused by their receipt, it can provide more accurate detection than MIB probing. (There is another practical advantage of NMS over SNMP: the operating system used in our lab, Linux, does not support the required MIB variable.)

Rshx at stealth level three represents the most likely form of the exploit, and by correlating its two different signatures, rsh using port number zero, and sequence number guessing, which are on individually weak, high quality detection can be achieved.

To experiment with the system call based detection of the r-utilities exploit we used our previously described anomaly detector. We collected training data by monitoring rshd's system calls when it was used in a legitimate fashion by rsh. A set of about 150 test cases for rsh produced about 65% code coverage of rshd. From the monitored system calls, a training data set comprising about 520 sequences of six system calls was produced.

An anomaly detector utilizing the training data monitored rshd during legitimate and attack usage. We discovered that the anomaly detector reported no anomalies under normal usage. The anomaly detector reported some anomalies under attack usage. Investigation of the reported anomalies revealed that only if the attack caused stimulation of the port number zero feature would the anomaly detector report an anomaly. This result corresponds to our previous conclusion that the only evidence of the attack that is observable at a system call level is use of the port number zero feature. It is interesting to note that while one may naively expect the use of the port number zero feature to produce radically different system calls than normal use, in reality, the reported anomaly was the omission of a single system call. The omitted system call was alarm, which in normal usage provides a timeout mechanism to guard establishment of the second connection. While the anomaly detector was able to find an anomaly associated with the port number zero feature, we believe a competent programmer would have easily produced a better system call signature that was more closely tailored to use of the port number zero feature. (The weakness of using omission of alarm is that it is an indirect consequence of the port number zero feature, while other system calls, for example fork and execve are direct consequences and are more likely to remain valid across different versions of rshd.)

In contrast to the anomaly-based detector, a specification-based detector is shown in Code Example 0-6. The specification notices the omission of a fork system call before an execve. This specification enforces what can be considered a general constraint for many server programs; that is, that the programs use the fork/execve mechanism for servicing requests. (Not all server programs use this mechanism, but many do.) Thus the specification-based defense may be usable by many server programs, while the anomaly-based defense is probably limited to rshd.

Code Example 0-6 Specification Based r-utilities trust exploit detector

```
 #include “common.h”
 #define EINVAL 22
 module main() {
 int okToExecve_;

int globalTemp;

 // privileged calls
 rule:  (any)*;fork()-->
         {
                  okToExecve_ = 1;

        }
 // execve
 rule:  (any)*; execve(a1,a2,a3)|okToExecve_ != 1 -->
         {
         logIntrusion(10);
            fake(-1,EINVAL);

        }
}
```

A concern with both the anomaly and the specification based SMS detector is that since they detect use of a legitimate, but seldom used feature, there is some possibility of a false positive. To reduce the probability of a false positive, we used NMS to monitor for the other intrusion that accompanies the exploit in its most popular use. We programmed NMS to monitor for bursts of packets with incorrect sequence numbers which occur due to a bursty sequence number guessing attack. The NMS specification, shown in Code Example 0-7, recognizes bursts with more than 5 incorrect sequence number packets.

Code Example 0-7 NMS Sequence Number Guessing Detector

```
packet(i, p) | addr_my_net(p.d_addr) && (p.tcp_syn == 0) && (p.tcp_ack ==
16)

       -> `decNeptune(p);`
void decNeptune(const u_char *p)
{

 Neptune *n = neptuneAgg.lookup(Neptune(source_ip(p), dest_ip(p),
  source_pt(p), dest_pt(p)));
  if (n == NULL)
        return;
  if (n->mySeqNo_+1 != tcp_ack_num(p)) {
        EpTriple ept(dest_ip(p), dest_pt(p), source_ip(p));
        int m = ipSpoofAgg.insert(ept, 0, pktTime);
        if (m > IP_SPOOF_BAD_ACK_THRESHOLD_LO)

     printAttacks(“ipspoofing”, p, mkProb(m, IP_SPOOF_BAD_ACK_THRESHOLD_LO,
    IP_SPOOF_BAD_ACK_THRESHOLD));
  }
  else {
        TcpEndPoint    *tep     =     tcpEPAgg.lookup(TcpEndPoint(dest_ip(p),
dest_pt(p)));
        if (tep != NULL) {
          if (tep->attack_)
            /*printAttacks(“override_prevattack”, p, 0.9)*/;
          tcpEPAgg.dec(*tep, pktTime,10);
        }
        neptuneAgg.remove(Neptune(source_ip(p), dest_ip(p),
                                                              source_pt(p),
dest_pt(p)));
        neptuneHostAgg.remove(source_ip(p));
  }

}
```

Both NMS specification and the SMS specification detect the two different signatures of the `rshx` at stealth level three, and send IDIP attack detection code to the CIDF manager. As soon as the CIDF manager gets one of the signatures, it starts a timer. If it gets the other signature before the timeout, it reports the intrusion with a high level of certainty. Thus, through correlation of two low certainty intrusion reports produced by SMS and NMS, the CIDF manager can deduce that a single underlying exploit is responsible, and can report that intrusion with high certainty.

7.4.5.5 Comments on the Difficulty of Producing Training Data

In producing the training data set which resulted in the described behaviors, we encountered many difficulties which illustrate how difficult the creation of adequate training data can be. First, as we were developing the training data we noticed some correlation between coverage, size of training data, and accuracy of anomaly detector. In general, as we added more tests to increase coverage, the training data grew in size, and incidence of false positives decreased. This supports the hypothesis that increased coverage during the training phase improves detector performance. However, when we reached the 65% coverage we noticed that the anomaly detector still reported false positives for test inputs which produced identical coverage to that for the training data. Investigation revealed that these false positives stemmed from minor runtime differences, and they were eliminated by repeating the training phase several time. We repeated the training phase about five times, at which point we observed that additional repeats did not result in training dataset growth. Believing that we had an adequate training data set we began conducting the experiment and observed an unexpected result; the anomaly detector actually detected an attack which we expected it to miss, that attack being the stealth level 1 attack, which neither guesses sequence numbers nor uses the port number zero feature. Also, the anomaly detector reported a false positive that was unexpected when subjected to legitimate use. Furthermore, the anomaly reported for both the true positive and the false positive was identical. The anomaly was an extra `lseek` system call on the `/etc/passwd` file that occurred immediately prior to `rshd` termination, which was actually long after the attack had succeeded. Extensive investigation revealed that the anomaly being reported had absolutely nothing to do with the attack. The anomaly occurred because of an artifact of the experimental setup. This artifact was the position of the entry for the user accessing `rshd` within the `/etc/passwd` file. If the entry occupied the last line of the file, then the extra `lseek` occurred, otherwise it did not occur. To obtain the results reported above, we adjusted the user's position within the `/etc/passwd` file and repeated the training phase, which added the new signatures to the training data. We believe that this experience illustrates the difficulty of producing a training data set for an anomaly detector. While code coverage is a good start, it is insufficient, as execution environment differences can cause different system calls for inputs that give identical coverage. Furthermore, there appear to be what we consider situations that even a skilled programmer would never expect to influence the training data—such as position of an entry within a file—which need to be discovered in order to produce a high quality training data set. On the contrary, we believe that a competent programmer could write a specification for detecting port number zero feature usage which is not only better than that learned by the anomaly detector, but with less effort required to produce an adequate training data set.

8 Summary

In this section, we summarize the accomplishments of the SAN project. These accomplishments are organized into the following categories:

- new capabilities that have been developed and validated
- new techniques that form the basis of these capabilities
- new results from recent research efforts that have built on and extended the techniques developed in SAN
- technology transferred
- publications resulting from SAN
- software prototypes developed
- other technology transfer efforts

8.1 New Capabilities Developed in SAN

Detection of attacks before they cause damage. SAN project has established the feasibility of detecting attacks early enough that responses can be taken to prevent and/or contain damages that may result from the attack.

Low-overhead, high-speed intrusion detection. Our SMS imposes runtime overheads in the range of 5% or less in practice—an overhead that is almost imperceptible. Our NMS supports detection at very high speeds, approaching gigabit rates even when run on a typical PC.

Accurate detection of known as well as novel attacks. The SAN approach combines the principal advantages of misuse and anomaly detection, while eliminating their main drawbacks. Like misuse detection techniques, the SAN approach can accurately identify known attacks with a low degree of false positives. Like anomaly detection, it can identify novel attacks.

Automatic initiation of intrusion response: SAN provides the capability for a user (i.e., a system administrator) to specify customizable responses to attacks or other unintended uses of the protected system. These responses are initiated automatically by SAN when an attack takes place.

Active-Networking based response for tracing and isolating an attacker. The SAN prototype incorporates capabilities for global isolation of an offending host by making use of routers that have been enhanced to support active networking. The AN-based technique enjoys the advantage of being able to isolate and seal off the attacker even if the attack uses a spoofed source address.

These capabilities have been demonstrated through a number of experiments, including the Lincoln Labs IDS Evaluation, AFRL Real-time IDS, and our own internal experimentation efforts.

8.2 New Techniques and Technologies Developed in SAN

In order to realize the capabilities described above, we had to develop several new techniques and technologies as described below.

An expressive and easy-to-use language for capturing system behaviors. Most previous approaches for characterizing system behaviors in terms of events (such as system calls or network packets) ignored either the event parameters, or the temporal relationships between the occurrences of different events. We have developed a new pattern language in BMSL that can express event parameters as well as temporal relationships. This enables BMSL users to write concise specifications of normal behavior. Conciseness leads to clarity, which in turn increases our confidence in the correctness of these specifications and reduces development and debugging efforts.

Novel type system that affords type-safe access to complex data types. We have developed a novel type system for accessing complex data such as network packets. This type system ensures that event arguments and other data will be accessed correctly at runtime, eliminating such problems as invalid memory accesses and race conditions. This factor, together with the high detection speed, enables our IDS to protect itself from subversion by attacks that saturate the network, or contain malformed packets or data. Moreover, the safety and robustness offered by the type system makes BMSL a much more suitable language for developing OS kernel-resident real-time defenses than languages such as C or C++.

Fast pattern-matching algorithms that can detect deviations of observed system behavior from specifications. We have shown how BMSL pattern matching can be done using a class of automata called *extended finite state automata* (EFSA). We have investigated and shown that the space requirements for a deterministic EFSA corresponding to a BMSL pattern are unacceptably large. To overcome this problem, we developed a novel runtime model based on a new class of automata called quasi-deterministic EFSA. QEFA trades off space usage in return for potentially slower matching for certain classes of patterns. In practice, however, such patterns occur rarely, and thus we realize the benefits of low space utilization and high detection performance. In particular, the time for matching an event at runtime remains a constant that is independent of the number of BMSL patterns.

Availability of such fast pattern-matching algorithms simplifies specification development significantly. A BMSL user can focus exclusively on expressing acceptable behaviors clearly and concisely, rather than being concerned about efficient enforcement of these behaviors.

Modular design that offers portability and extensibility. We have developed an elegant design for the interface between the detection engines and the runtime environments. This design makes it possible to interface the detection engine to a wide variety of runtime environments *without making any changes to the BMSL compiler that generates the detection engines.* Already, we have integrated the detection engine to the `libc` and kernel version of the system call interceptor, the network packet interceptor, and a user-level system call interceptor described in [JS00]. Integration with a runtime system that feeds off Solaris audit logs is currently underway.

Efficient and effective system call interposition for Linux. A key to damage prevention and/or isolation is the development of techniques for efficient interception and/or modification of system calls.

The applicability of our techniques extends well beyond the domain of survivable systems. Many of the concepts developed in BMSL, such as event patterns and fast pattern matching algorithms are applicable in network monitoring, software monitoring, debugging and quality assurance. Packet types as well as the pattern-matching algorithms can be applied to packet filtering systems to make them extensible to support new network protocols, and also make them faster. The system call interposition capability can be used to implement other sorts of application extensions, such as application-transparent file encryption and decryption.

8.3 Principles Established and Lessons Learnt

The SAN project has met almost all of the objectives we started out with, and validated our main hypotheses. We have shown that a specification-based approach yields high detection accuracy, protection against unknown attacks and a low degree of false positives. The approach also enables development of defenses tailored for different kinds of attacks. Our event interception techniques enabled these defenses to be launched before attacks can compromise the system. Finally, our compilation and event interception techniques provided adequate performance to enable online detection and responses to be supported without noticeable performance degradation. In addition to establishing the basic hypotheses we started out with, our research and experimental efforts in SAN lead us to make the following observations and conclusions:

- *The use of BMSL, a high-level specification language, for building the interceptors and interposers worked well for several reasons:*
  - BMSL frees the programmer from one of the most burdensome problems in developing event pattern matching programs, that problem being the details associated with overlapping partial patterns.
  - The BMSL compiler performs optimizations that would be extremely difficult for a programmer to apply manually. These optimizations are especially important in light of the need to consider both system calls and their arguments, since the arguments can lead to an explosion of space or time complexity.
  - BMSL provides a degree of type safety over lower level languages like C++, and this safety is of the utmost importance because the code ultimately executes within the OS kernel where errors can be disastrous.
- *From our research we learned that our programming environment allows the construction and operation of defenses that resolve many real world computer security problems.*
  - For application vulnerabilities and/or poorly designed features, programs which either correct the vulnerability/feature and/or detect its malicious stimulation can be created in the absence of source code by a competent programmer with modest effort.
  - For OS vulnerabilities that are directly associated with system calls, the SAN system call interposition capability allows remediation.
  - For OS vulnerabilities that are not directly associated with system call processing, the network packet interception appears to offer some protection against stimulation of the vulnerability by detecting and deleting the input that stimulates it.
  - For vulnerabilities that are purely the result of application misconfiguration, the SAN approach offers some protection if the misconfiguration causes an observable improper behavior to occur, however misconfigurations may obscure the behavior to the degree that it cannot be observed.
- *We found that the object-oriented approach of associating an object with a process to be monitored and using method invocation to report events to that object provided a convenient programming mechanism.* Related to this mechanism, the use of inheritance to provide default treatment for uninteresting events also simplified programming.

- *We also learned that demand-driven active network defenses can provide a useful adjust to host-based defenses.* Active network defenses can perform functions, such as trace and isolate, that are impossible for hosts to perform. Making these defenses demand-driven so that they are enabled only when actual attack probability is high helps focus the defenses in a way that conserves router processing capacity. Limited active network prototype capabilities prevented us from realizing our initial concept of using active networking to build defenses based on real-time packet modification.

The prototype shows that the overhead involved with system call and packet interception is low. Although the overhead associated with processing complex interceptors and interposers can become significant, such interceptors were not required in any of our experiments involving real world vulnerabilities.

Some of the unanticipated challenges that arose in the SAN research and how we dealt with them are described below. In some cases, it was simply an issue of understanding the issues involved and developing solutions to these problems. In other cases, the challenges led us to explore alternative approaches for intrusion detection and response. (Many of these new approaches are described in more detail in the next section.)

- The most challenging aspect of our experience is associated with the programming of system call interceptors and interposers. Although the high level language provides type-safety, kernel resident programs are inherently difficult to write, restricted in terms of programming environment and library facilities, and provide poor support for debugging. We are developing a new approach that tackles this problem by splitting the interposition code into two pieces: a kernel-resident interceptor and a user-level analyzer, with the goal of minimizing the size, complexity, and the need to modify the kernel-resident interceptor. This hybrid approach mimics the successful model that has been used in packet filtering approaches *ala* Berkeley Packet Filter [MJ92].

- There is a trade-off between the specification development effort and how closely this specification matches the actual application behavior. As we get closer and closer to a specification that exactly matches an application behavior, our returns (in terms of increased security) diminish rapidly in comparison with the increase in development effort. These factors led us to pursue two approaches for simplifying specification development. First, we developed a classification of Linux system calls, based on which we could write a generic specification for most applications that required only changes to protect against a majority of attacks. Second, we are pursuing a resource-centric approach for protecting against damage due to attacks. Whereas the number of applications that may need defenses is large and growing, the number of different kinds of resources that need protection seems to be limited and somewhat more static.

- Specification-based approaches such as ours, which rely on manually developed characterizations of intended behavior, nicely complement anomaly detection approaches that learn normal behaviors by observing a system under operation. Specification-based approaches can protect against vulnerabilities that arise due to errors of omission in an application, while anomaly detection approaches would simply learn these erroneous behaviors as normal. On the other hand, subtle changes in behavior of an application may be detected by a learning-based approach, but missed by a specification based approach, since a manually developed specification is necessarily less detailed than a model that is learnt via machine learning. For additional discussion of this topic, please see Section 4.5.4.

8.4 Results of Research Efforts Related to SAN

The techniques and technologies developed in SAN inspired follow-on research on many topics that were closely related to SAN efforts. These results are an indirect outcome of SAN effort, and thus enhance the return for the efforts invested in SAN.

- *A runtime environment for system call interposition at the user-level.* Although a kernel level implementation of system call interposition offers many advantages, it has some drawbacks as well, in the form of increased development and debugging efforts. A user-level interposition avoids these drawbacks. However, the conventional wisdom suggested that user-level interposition would be slow, and could be easily subverted. In our recent research, we have shown that these issues can be addressed satisfactorily [JS00]. Although the performance of user-level interposition will be necessarily lower than kernel-level interposition, we showed that the decrease in performance is not significant for most applications. Moreover, the user-level interception system is portable across different UNIX variants, such Linux, Solaris and IRIX. Our follow-on work in the PARR project will combine the benefits of kernel level and user level interposition.

- *An automaton-based approach for learning program behaviors in terms of system calls.* In SAN, we developed highly efficient runtime matching techniques based on compilation of patterns into finite-state automata. Partially motivated by this success, we investigated the use of the FSA representation for learning normal program behaviors. Unlike previous approaches such as that of [FHS97], the automaton approach does not need to break system call sequences into short subsequences. Instead, it can program as a single sequence. This increases the ability of the approach to capture long-term correlations, which are necessarily lost by the process of breaking the execution sequence into short strings. Prior experience seemed to indicate that the problem of learning FSA is computationally hard, its storage requirements may be high, and it may require a long training period. However, we developed a clever approach that leveraged additional information about processes (such as the contents of program stack) to develop an efficient algorithm that overcame these drawbacks. Our early experiments suggest that the automaton-based learning approach improves over [Forrest et al 96] in terms of the false alarm rate as well as the training time by one to three orders of magnitude.
- *Automated vulnerability analysis of computer configurations.* Intrusion detection is concerned with the problem of detecting exploitation of vulnerabilities on a system. Vulnerability analysis, on the other hand, is aimed at detecting the underlying vulnerabilities themselves. Previous approaches for vulnerability analysis, such as those embodied in COPS and SATAN, were based on encoding known causes of vulnerabilities in computer system configurations. In contrast, we have developed a novel approach [RS00] that can identify previously known as well as unknown vulnerabilities. Our approach is based on high-level models of behavior of the system and its components, and formal specification of desired security properties. By analyzing the potential interactions and conflicts among the system components, our approach can identify as-yet-unexploited vulnerabilities. Even more important, our technique can be used to generate misuse signatures that correspond to exploitations of these vulnerabilities. This capability enables us to guard against these vulnerabilities using intrusion detection, even if the underlying vulnerability cannot be eliminated.

8.5 Publications

8.5.1 Publications in Refereed Conferences and Journals

The following papers were published, based on the research conducted with whole or partial support from the SAN project.

1. **A Specification-Based Approach for Building Survivable Systems**, R. Sekar, Y. Cai and M. Segal, *National Information Systems Security Conference,* 1998.
2. **Model-based vulnerability analysis of computer systems,** C.R. Ramakrishnan and R. Sekar, *2nd Int'l Workshop on Verification, Model Checking and Abstract Interpretation,* 1998.
3. **On Preventing Intrusions by Process Behavior Monitoring**, R. Sekar, T. Bowen and M. Segal, *USENIX Intrusion Detection Workshop,* 1999.
4. **Synthesizing Fast Intrusion Detection Systems from High-Level Specifications,** R. Sekar and P. Uppuluri, *USENIX Security Symposium*, 1999.
5. **A High-Performance Network Intrusion Detection System**, R. Sekar, Y. Guang, S. Verma and T. Shanbhag, *ACM Symposium on Computer and Communication Security,* 1999.
6. **Abstracting Security Specification for Building Survivable System**, J. J. Li and M. E. Segal, *22nd National Information System Security Conference*, 1999.
7. **A User-Level Infrastructure for System-call Interposition**, K. Jain and R. Sekar, *ISOC Network and Distributed Systems Symposium,* 2000.
8. **Building Survivable Systems: An Integrated Approach based on Intrusion Detection and Damage Containment**, T. Bowen, D. Chee, M. Segal, R. Sekar, T. Shanbhag, P. Uppuluri, *DARPA Information Survivability Conference (DISCEX),* 2000.

9. **A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors**, R. Sekar, M. Bendre and P. Bollineni, Under review for *ACM Symposium on Computer and Communication Security,* 2000.
10. **Model-Based Analysis of Configuration Vulnerabilities**, C. Ramakrishnan and R. Sekar, Under review for *ACM Symposium on Computer and Communication Security,* 2000.
11. **Remediating Application-Specific Vulnerabilities at Run Time**, T. F. Bowen, *Accepted for publication in IEEE Software special issue on Malicious Code Detection,* September/October, 2000.

8.5.2 Graduate theses

The following students completed their Master's projects and theses on SAN-related topics.

1. *A Specification-Based Approach for Intrusion Detection,* Y. Cai, Iowa State University, Dec 1998.
2. *A Real-time Packet Filtering Module for Network Intrusion Detection System,* Y. Guang, Iowa State, Jul 1998.
3. *ASL: A specification language for intrusion detection and network monitoring,* R. Vankamamidi, Iowa State University, Nov 1998.
4. *A Software Infrastructure for Building Intrusion Detection and Confinement Systems,* K. Jain, Iowa State University, July 1999.
5. *A New State-based approach for Learning Program Behavior for Intrusion Detection,* P. Bollineni, Iowa State University, Nov 1999.
6. *Synthesizing fast pattern-matching algorithms for Intrusion Detection and Prevention,* P. Uppuluri, Iowa State University, April 2000.
7. *A state-machine approach for network intrusion detection,* T. Shanbhag, Iowa State University, Jul 2000.
8. *Model-Based Intrusion Detection,* S. Zhou, SUNY at Stony Brook.

8.6 Software Developed

We developed the following software components in the SAN project:

- compiler for BMSL that generates code for system call interposition
- compiler for BMSL that generates code for network packet monitoring
- runtime environment for system call interposition, implemented by modifying libc.
- runtime environment for system call interposition, implemented within the OS kernel
- runtime environment for network packet interception
- BMSL specifications and defenses for several server applications
- BMSL specifications for detecting low-level network probing and denial-of-service attacks.

These components have been packaged into two modules, namely, the SMS and NMS.

References

[ALE96] Aleph One, Smashing the Stack for Fun and Profit, Phrack Online, Volume 7, Issue 49, File 14 of 16, www.fc.net/phrack, November 9, 1996

[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, National Computer Security Conference, 1996.

[BCG87] G. Berry, P. Couronne and G. Gonthier, Synchronous Programming of Reactive Systems: An Introduction to Esterel, Technical Report 647, INRIA, Paris, 1987.

[Berry86] G. Berry and R. Sethi, From Regular Expressions to Deterministic Automata, Theoretical Computer Science 48 pp. 117-126, 1986.

[Bishop96] M. Bishop and M. Dilger, Checking for Race Conditions in File Access. Computing Systems 9(2), pp. 131-152, 1996.

[Brzozowski64] J. A. Brzozowski, Derivatives of Regular Expressions, Journal of ACM Vol. 11, No. 4, pp. 481-494, 1964.

[Cai98] Y. Cai. A Specification-Based Approach for Intrusion Detection. M.S. Thesis, Department of Computer Science, Iowa State University, Dec 1998.

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

[CM 99] S. Chandra and P. McCann, Packet Types, Workshop on Compilers Support for Systems Software, 1999.

[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.

[Cowan et al 98] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle and Q. Zhang, StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, 7th USENIX Security Symposium, 1998.

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

[FBF99] T. Fraser, L. Badger, M. Feldman, Hardening COTS software with Generic Software Wrappers, IEEE Symposium on Security and Privacy, 1999.

[FHS97] Forrest, S.; Hofmeyr, S.; Somayaji, A.; Computer Immunology, *Communications of the ACM*, Volume 40, No. 10, 1997.

[FHS98] S. Forrest, S. A. Hofmeyr, A. Somayaji, Intrusion Detection using Sequences of System Calls, Journal of Computer Security Vol. 6 (1998) pg 151-180.

[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.

[GPRA98] D. Ghormley, D. Petrou, S. Rodrigues, and T. Anderson, SLIC: An Extensibility System for Commodity Operating Systems, USENIX Annual Technical Conference, 1998.

[Graf et al 98] I. Graf, R. Lippmann, R. Cunningham, D. Fried, K. Kendall, S. Webster and M. Zissman, Results of DARPA 1998 Offline Intrusion Detection Evaluation, http://ideval.ll.mit.edu/results-html-dir/index.htm , 1998.

[GSS99] 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.

[Guang98] Y. Guang. A Real-time Packet Filtering Module for Network Intrusion Detection System, M.S. Thesis, Department of Computer Science, Iowa State University, Jul 1998.

[GWTB96] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer, A Secure Environment for Untrusted Helper Applications, USENIX Security Symposium, 1996.

[H90] L. Heberlein et al, A Network Security Monitor, Symposium on Research Security and Privacy, 1990.

[H93] J. Hochberg et al, NADIR: An Automated System for Detecting Network Intrusion and Misuse, Computers and Security 12(3), May 1993.

[HKMGN98] M. Hicks, P. Kakkar, J. Moore, C. Gunter, and S. Nettles, PLAN: A Packet Language for Active Networks, Proceedings of the Third International Conference on Functional Programming Languages, pages 86-93, ACM, 1998.

[JS00] K. Jain and R. Sekar, User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement, ISOC Network and Distributed Security Symposium, 2000.

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

[Ko96] C. Ko, Execution Monitoring of Security-Critical Programs in a Distributed System: A Specification-Based Approach, Ph.D. Thesis, Computer Science, University of California at Davis, 1996.

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

[KS94] Kim, G. H., Spafford, E. H.; Experiences with Tripwire: Using integrity checkers for intrusion detection, *Systems Administration, Networking, and Security Conference III*, Usenix, April 1994

[Kumar94] S. Kumar and E. Spafford, A Pattern-Matching Model for Intrusion Detection, National 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.

[LIN76] Linden, T. A. Operating System Structures to Support Security and Reliable Software, NBS Technical Note 919, Institute for Computer Sciences and Technology, National Bureau of Standards, US Department of Commerce, Washington DC 20234, August 1976.

[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.

[LV95] D. Luckham and J. Vera, An Event-Based Architecture Definition Language, IEEE Transactions on Software Engineering, 21(9), 1995.

[MF98] McGraw, G.; Felten, E.; *Securing Java*, 1998, John Wiley & Sons, Inc.

[MJ92] S. McCanne and V. Jacobson, The BSD Packet Filter: A New Architecture for User-level Packet Capture, Lawrence Berkeley Laboratory, Berkeley, CA, 1992.

[MLO97] T. Mitchem, R. Lu, R. O'Brien, Using Kernel Hypervisors to Secure Applications, Annual Computer Security Application Conference, December 1997.

[MY60] R. McNaughton and H. Yamada, Regular expressions and state graphs for automata, IRE Trans. on Electronic Computing, EC-9(1), 1960.

[Mukherjee94] B. Mukherjee, L. Todd Heberlein, Karl N. Levitt. Network Intrusion Detection, IEEE Network, pp. 26-41, May/June 1994.

[NIT98] Nitzberg, S., Conflict and the Computer: Information Warfare and Related Ethical Issues, In *Proceedings of the 21st National Information System Security Conference*, Arlington, VA, 126-135.

[Pax98] V. Paxson, Bro: A System for Detecting Network Intruders in Real-Time, USENIX Security Symposium, 1998.

[PHA98] Phrack Magazine, Volume 8, Issue 54, 12/25/98 article 3 of 12, http://phrack.infonexus.com/search.phtml?view&article=p54-3.

[PLA99] http://www.cis.upenn.edu/~switchware/PLAN.

[PN97] P. Porras and P. Neumann, EMERALD: Event Monitoring Enabled Responses to Anomalous Live Disturbances, National Information Systems Security Conference, 1997.

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

[RS95] Russinovich, M. and Segall, Z. Fault-Tolerance for Off-The-Shelf Applications and Hardware, *Proceedings of the 1995 25th International Symposium on Fault-Tolerant Computing*, 1995.

[RS00] C. Ramakrishnan and R. Sekar, Model-Based Analysis of Configuration Vulnerabilities, ACM Intrusion Detection and Prevention Workshop, 2000.

[S99] Common Intrusion Detection Framework, S. Staniford-Chen et al, http://seclab.cs.ucdavis.edu/cidf

[SBS98] Sekar, R.; Bowen, T.; Segal, M.; On Preventing Intrusions by Process Behavior Monitoring, *Workshop on Intrusion Detection and Network Monitoring Proceedings,* Santa Clara, CA, April 1999, The USENIX Association, 1999.

[Sch98] F. Schneider, Enforceable Security Policies, TR 98-1664, Department of Computer Science, Cornell University, Ithaca, NY, 1998.

[Sekar95] R. Sekar, I.V. Ramakrishnan and R. Ramesh, Adaptive Pattern Matching, SIAM Journal of Computing, 1995.

[Sekar98] R. Sekar, Y. Cai and M. Segal, A Specification-Based approach for Building Survivable Systems, 21st National Information Systems Security Conference.

[SPA89] Spafford, E. The Internet Worm Program: Analysis, *Computer Communication Review*, January 1989.

[Spafford91] E. H. Spafford. The Internet Worm Incident, Technical Report CSD-TR-993, Purdue University, West Lafayette, IN, September 19, 1991.

[SGSV99] R. Sekar, Y. Guang, T. Shanbhag and S. Verma, A High-Performance Network Intrusion Detection System, ACM Computer and Communication Security Conference, 1999.

[SU99] R. Sekar and P. Uppuluri, Synthesizing Fast Intrusion Prevention/Detection Systems from High-Level Specifications, USENIX Security Symposium, 1999.

[TEL99] Telcordia Software Visualization and Analysis Toolsuite, http://xsuds.argreenhouse.com/what.htm

[Uppuluri00] P. Uppuluri, Synthesizing Fast Intrusion Prevention/Detection Systems from High-Level Specifications, Master's Thesis, Department of Computer Science, Iowa State University, Ames, IA 50014, 2000.

[Van98] R. Vankamamidi. ASL: A specification language for intrusion detection and network monitoring. M.S. Thesis, Department of Computer Science, Iowa State University, Dec 1998.

[VK98] G. Vigna and R. Kemmerer, NetSTAT: A Network-based Intrusion Detection Approach, Computer Security Applications Conference, 1998.

Survivable Active Network (SAN) Project
(DARPA ITO Contract Number F30602-97-C-0244)
Final Report Addendum
for the no-cost contract extension period from 8/1/00 - 1/31/01

Unclassified

Extension of SAN Prototype
To Process Basic Security Module (BSM) Data
and
Evaluation of SAN Specification-based Approach
for Intrusion Detection
Using 1999 Lincoln Labs BSM Data

Prepared by: R. C. Sekar, SUNY at Stony Brook

Principle Investigator: Mark Segal, Telcordia Technologies, Inc.

Submitted 1/31/01

Final Report Addendum

A1 Background

The SMS component of SAN uses a specification-based approach for intrusion detection. In this approach, legitimate program behaviors are first specified using our BMSL specification language. These specifications are then compared with behaviors that are observed when a program is run. Any deviation between specified and observed behaviors is deemed to indicate an attack. One of the main benefits of this approach is its ability to detect novel attacks that have not been encountered previously, as the approach is based on legitimate program behaviors, rather than attack signatures. A potential disadvantage is that for maximum protection against attacks, specifications may have to be developed individually for every program. In this experiment, we evaluate the benefits as well as costs of the specification-based approach used in SAN.

The experiment concerns the evaluation of the SMS component of SAN. The NMS component previously participated in the 1998 and 1999 evaluations, and the results of these evaluations were described in the final report.

A2 Objective

The main objective of the experiment was to evaluate the effectiveness of the SAN specification-based approach for detecting intrusions. We used the intrusion detection evaluation data provided by Lincoln Labs as part of the 1999 evaluation. Since the experiment is concerned only with the SMS, we used only the subset of Lincoln Labs data that recorded program behaviors. Lincoln Labs recorded program behavior data using the Basic Security Module (BSM) of Solaris, which produces a log file containing BSM data.

A secondary objective of the experiment was to evaluate the level of effort that needs to be spent for developing specifications of program behavior for intrusion detection.

A tertiary objective was to analyze the attack data itself, and to identify how well it taxes the intrusion detection systems.

A3 Approach

The following steps are involved in this experimental evaluation:

- Development of a runtime environment that parses BSM data and feeds system call events contained there into the SMS detection engine
- Development of specifications for programs that may have been attacked in the Lincoln Labs evaluation.

We describe these steps in more detail below.

A3.1 Development of BSM Runtime Environment

The SMS component of SAN consists of two main components:

- a detection engine (DE) that is generated from BMSL specifications, and
- a runtime environment (RTE) that feeds system call data into the DE.

SMS has a modular design, with a well-defined and documented interface between the DE and RTE. Earlier, we had integrated the DE with the in-kernel system call interception system. In order to process the BSM data, we had to develop a new RTE that would conform to the interface specifications, but will get its event data from the BSM audit files provided in the Lincoln Labs 1999 evaluation data.

Development of the BSM RTE consisted of the following tasks:

- analysis and selection of BSM record types that are relevant for SMS. In order to ensure that this experimentation effort can be completed within our prescribed time frame, we omitted (a) all audit records that do not correspond to system calls, and (b) those system calls that were unlikely to cause significant damage even if abused. Through a manual review process, we identified that approximately 40% of the BSM system call records are useful for intrusion detection.
- implementation of a parser for the selected audit records. The format of the BSM file is clearly specified in its documentation. Thus, implementation of a parser is straightforward, but tedious due to the large number of record types that need to be processed.
- development of classes that provide a way for the DE to examine system call arguments. Parsing of BSM argument tokens is performed within these classes.
- integration of the DE with the BSM RTE and testing.

A3.2 Development of Specifications

As mentioned earlier, one of the main efforts involved the SAN approach is that of specification development. In order to evaluate the level of effort needed, we used the following approach for specification development:

1. Begin with the set of specifications we had developed in our earlier experimentation efforts. In particular, we started with our specifications for ftp, telnet, and http servers, and a generic specification that is applicable to most programs.
2. Refine the generic specifications for all setuid programs. The refinement consists of specifying additional system calls that may or may not be executed by each specific program, and additional files that may or may not be read or written by a program.
3. Add site-specific security policies to these specifications. One such policy was stated explicitly by Lincoln Labs, pertaining to the accessing of files in a directory named "secret." There were also some implicit policies, e.g., anonymous ftp users should not write files into the directory `~ftp2`, nor should they read or write files in hidden directories.
4. Use training data to "debug" and/or "augment" the specifications. This step assumes particular significance in view of the fact that our specifications were originally developed for Linux, whereas the BSM data pertains to process behaviors under Solaris. The main changes made in this step pertain to the list of files and directories accessed by various programs.
5. Experiment with the test data, and revise the specifications as needed. We originally anticipated that we might have to develop additional specifications, as well as tune existing specifications as part of this step. It turned out, however, that the specifications obtained at the end of step (4) were adequate for the 1999 BSM evaluation data.
6. Explore refinement of specifications that would enable the specifications to detect behaviors that are actually unsuitable for detection using the method of specifying legitimate behaviors. While Lincoln Labs classifies the behaviors as attacks, we are not in complete agreement that they are attacks, nevertheless, we wanted to ensure that SMS could be programmed to conform to Lincoln Labs classification, even though we disagreed with it. Section 4 gives more details.

A3.2.1 Notes on Porting Specifications from Linux/Kernel RTE to Solaris/BSM RTE

Not all of the functionality provided by the kernel system call interception RTE can be made available by the BSM RTE. In particular, the BSM RTE is limited to providing only the data that is recorded in the audit file. In contrast,

[2] Since a number of attacks on anonymous ftp involve writing to the home directory of ftp, this policy could (arguably) be considered a general policy rather than a site-specific one.

the kernel RTE allows the DE to access every system call argument, and also query the system state at any time. Moreover, the kernel RTE provides an API for the DE to request corrective actions to be taken when intrusions are detected. These differences imply that the BSM RTE cannot provide the same semantics for the DE/RTE interface as the kernel RTE. These changes imply that the specifications developed for SAN with the kernel RTE may have to be changed when using the BSM RTE. In practical terms the following kind of changes were necessary:

- Specifications that made use of system call arguments that were not recorded in the BSM file had to be changed. There was only one instance where this aspect posed a problem in our experiments: the rules we had developed for the "guest" attack had to be changed, as they required access to the data buffer argument of a read system call, which was not available in the BSM audit file.
- Specifications that needed current system state had to be changed. For instance, we had to delete rules that would check the target of a symbolic link before a file open operation. Similarly, rules designed to guard against race conditions also needed instantaneous system state, and these rules had to be deleted as well.

All response actions had to be disabled. but this posed no problem from the perspective of this experiment.

In summary, we note that although the differences between the two RTEs are significant, the modular design of SAN insulated the DE from most of these differences. The BMSL approach to interface specification, where the interface between the DE and the RTE is characterized through declarations in the BMSL specification file, provided further decoupling between the two modules – in particular, the changes in the interface required us to make changes to the interface declarations that were specified in a common include file, but did not require to the specifications themselves. These factors greatly eased the integration of the BSM RTE with the SAN DE.

A4 Experimental Results

A4.1 Attack Detection

Lincoln Labs provided a list of attacks present in the BSM data, along with the particular BSM files in which the attacks were present, the time intervals during which the attacks took place, the signatures of the attack and the simulation details. According to this list, 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 HTTP tunnel attack) was not visible in the BSM data. Offsetting these, there were five additional instances of the attack phase of the HTTP tunnel attack. Thus, the total number of attacks present in the data was 30.

The following table summarizes the 11 attack types and 30 instances of these attacks that were present in the data. The table specifies whether an attack was detected, and if so, what step of the specification development procedure outlined in the previous section revealed the attack. 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 6 of the specification development effort). At this point, all of the attacks could be detected.

Attack Name	**Total # of Instances**	**Percentage of instances detected after Step 4**	**Percentage of instances detected after Step 6**
Fdformat	3	100%	100%
Ffbconfig	1	100%	100%
eject	1	100%	100%
Secret	4	100%	100%
ps	4	100%	100%
Ftp-write	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%**

Most of the attacks were relatively simple, and were detected without any problem by SAN. This was in spite of the fact that little effort had been put into developing program-specific behavioral specifications. Below, we discuss the attacks that were moderately complex.

A4.1.1 Discussion

The `ps` attack was a buffer overflow attack, but was significantly more complex than other buffer overflow attacks. For one thing, it used a buffer overflow in the static area, rather than the more common stack buffer overflow. 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 in SAN.

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 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 using BMSL rules, 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: It involves an attacker using a dictionary attack, which requires a large number of login attempts. Such a high number of failed logins is highly unusual, and thus our normal behavior specification excludes such behaviors.

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 and sends confidential information to the site. The periodic connections are initiated by the UNIX `cron` daemon. The difficulty is that none of the steps involved in the attack can be considered as outside the scope of what a legitimate user can do. 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. A legitimate user may have used these facilities to periodically download news or stock quotes or other legitimate information, and there is no easy way to rule out these as the reasons for installation of the HTTP tunnel software. Since legitimate user behaviors cannot be distinguished from attacks in the case of HTTP tunnel, we developed a misuse specification that captures the following steps of the attack:

- periodic invocation of a program by `cron` such that this invocation results in a connection to a remote server

With this signature, we were able to detect the HTTP tunnel attack. The set up phase of the attack, however, was not visible, since some of the data that is crucial for the set up phase was not recorded in the BSM logs. (See Section A6.5 for details.)

A4.2 False Positives

The specifications had been tested previously against the training data. This meant that most of the false positives had already been eliminated. Consequently, our experiments did not result even in a single false positive.

A5 Conclusions

1. *SAN specification-based approach is effective.* It was able to detect 80% of the attacks with only specification of legitimate behavior. Since these specifications did not incorporate any knowledge about attacks or attacker

behavior, the experiment demonstrates the effectiveness of the SAN approach against unknown attacks. Of particular significance was the ps attack, which was considered very stealthy in the 1999 evaluation.

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 we expended. Thus, we believe that only a modest amount of specification development effort is allows defense against most attacks.
3. *SAN programmers can trade increased specification development efforts against decreased probability of successful attacks.* By investing some additional effort, we were able to detect all of the attacks included in the BSM data.
4. *100% detection rate could be achieved with 0% false positive rates.* 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.
5. *SAN 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. This is another benefit over anomaly detection techniques, where it is difficult to pinpoint the source of the anomaly.

Conclusions (1), (4) and (5) support our hypothesis that the ***SAN specification-based approach combines the benefits of misuse detection (low false positives and availability of attack-related information) with that of anomaly detection (ability to detect novel attacks)***. Conclusions (2) and (3) support our claim that modest specification development effort is sufficient to detect most attacks.

Our evaluation would have benefited significantly if the test data contained more stealthy and/or sophisticated attacks. We believe that the full capabilities of the SAN approach were not put to test by the Lincoln Labs data. (In comparison, the 1999 online evaluation used more stealthy attacks, and was able to stress SAN more.) This conclusion resonates with our argument that the evaluation efforts should be less focused on "coverage" and more on "sophisticated attacks" that would likely be launched by a skilled adversary.

A6 Addendum: Detailed Description of Attacks in BSM Data

In this section, we present the details of individual attacks and comment on their detection. The attack descriptions are based on the documentation provided by Lincoln Labs.

A6.1 Buffer overflows

A6.1.1 ps exploit

Attack description: This is a buffer overflow attack on the ps program on Solaris 2.5.1. This attack allows a user to execute invalid system calls with root privileges.

Comments:
When a *setuid to root* program is executed, the generic specification checks if any of the arguments being passed to the program in the execve system call are oversized. (The notion of "over-sized" may differ for different programs, and may have to be specified for each program. Alternatively, it can be defined based on the type of an argument, e.g., file names.) This resulted in detection of ps execution with an oversized argument.

For any *setuid to root* program, the generic specification maintains a list of files the program can write and perform attribute changes. We detected that ps makes a chmod system calls on a file that was not in the list.

A6.1.2 Exploits in `eject`, `Fdformat`, `ffbconfig` programs

Attack description: These attacks exploit a buffer overflow condition in the *setuid to root* programs `eject`, `Fdformat` and `ffbconfig`. The exploit code for the vulnerabilities was obtained and used unmodified from the *rootshell.com* website.

Comments: We detected this attack in two ways:
Whenever a *setuid to root* program is executed, the generic specification checks for oversized arguments in the `execve` system call. The specification detected an oversized argument in the executions of the three programs.
The generic specification has a list of invalid executable programs, and for the three programs, we included all "shell" programs in the list. We detected one of these shells--`ksh`--being `execve`'ed as part of the attack.

A6.2 FTP attacks

For anonymous `ftp`, we specified the following behavior constraints:

1. files in the `~ftp` directory may not be written
2. no file or directory with a hidden name (i.e., a name that begins with a ".") may be created
3. even if files were created that violated these policies, no one should try to download such files.

The following attacks were detected when these policies were violated.

A6.2.1 `ftp-write`

Attack description: 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.

In the simulation of the attack, the attacker logged in anonymously and created a `.rhosts` file with the string '++' in it within the `~ftp` directory.

Comments: We detected a file named `tricky` and a file named `pp`, both in `~ftp`, being written, thus violating behavior (1) above. We also detected a `rename` system call, which renamed `tricky` and `pp` to `.rhosts`, thus violating (2).

A6.2.2 Warezmaster

Attack description: Warez attacks are a combination of *warezmaster* and *warezclient.* During a *warezmaster* attack, the attacker logs into an anonymous `FTP` site and creates a file or a hidden directory. In *warezclient* attack, software, which was previously posted via anonymous `FTP` by the warezmaster, is downloaded.

Comments: The specification detected a file named `1rootk.tgz` being written into the `~ftp` directory, thus violating (1) above.

A6.2.3 Warezclient

Attack description: Users download software that was previously posted via anonymous `FTP` by the warezmaster.

Specification: Warezclient attack violates (3).

A6.3 Secret policy attack

Attack description: In a *secret* attack an attacker maliciously or mistakenly transfers data that is accessible to the attacker to a place where it doesn't belong. For example, transferring data from a classified computer/network to a non-classified computer/network would constitute a *secret* attack.

A policy indicating that all files in the directory `/export/home/secret/` are secret and cannot be moved out of the directory (whether by programs such as `cp`, `cat`, *etc.*) was published by Lincoln labs. During an attack, the attacker executes a command such as `cp`, `cat`, etc., on a file in `/export/home/secret/`

Comments: The generic specification contains a list of files that cannot be read by the processes it is monitoring. We added the secret directory to this list. We could detect all the instances of `cp` and `cat` reading files in the specified directory.

A6.4 Guest

Attack description: The *guest* attack is a variant of the *dictionary* attack. On badly configured systems, guest accounts are often left with no password or with an easy to guess password. In the simulation of the attack, an attacker tries to login using username, password combinations such as "guest/", "guest/guest", "anonymous/" and "unsigned/anonymous".

Comments: There are two approaches to detect this attack:

- The first approach is to monitor for the above-mentioned username/password combinations during a login session. This requires us to monitor all the `read` and `write` system calls. However, since BSM data does not record the data read or written by these system calls, this approach does not work with the BSM data.
- The second approach attempts to do the same thing as the first, but attempts to fill in the holes in the BSM data using other information. For instance, each login attempt would result in an open of the `passwd` and `shadow` files. Moreover, a successful login results in a system call to change directory to `~guest`. Based on this signature, the guest attack can be detected by SAN.

A6.5 Httptunnel

Attack description: During an *httptunnel* attack, the attacker gains local access to the machine that is to be attacked. The attacker sets up and configures an *http* client to periodically query a web server that the attacker has setup at some remote host. When the client connects, the server is able to send cookies that could request information be sent by the client, such as the password file on the victim machine. In effect, the attacker is able to *tunnel* requests for information through the *http* protocol. This attack is carried out in two phases. The first phase involves setting up the client . The second phase involves using the tunnel. During setup, the attacker logs in to the victim. The attacker might also add periodic web client starting instructions to the legitimate user's `crontab`. During the second phase the web client uses the tunnel to connect to the attacker's server, the attacker's server sends cookies requesting information to the victim, and the victim fulfills the requests. This web server runs on any port that the attacker chooses, however most often the attacker chooses a non-well known port above *1024.*

Comments: The critical part of the setup phase of the attack is that the user `rexn` sets up `cron` to periodically execute the program `~rexn/foo`, which was previously downloaded form outside. The setup phase involves the use of an editor to modify a configuration file. The name of the program to be executed, namely, `~rexn/foo`, is written into this file. Unfortunately, the BSM data does not record the data buffer argument of read and write system calls, and as such, the audit log contain no data to suggest that the set up of the attack took place. Thus, if we confine ourselves to the BSM audit data, the set up phase of the attack is not visible.

As for the "tunneling" phase, our detection efforts were rendered difficult by the fact that the operations of the `cron` daemon were not included in the BSM data. This meant that one could not tell whether a particular program was started by cron or some other process. To work around this problem, we developed a signature that captures the initial sequence of system calls made by a process started up by cron. This signature was used to detect execution of programs by cron.

To detect the HTTPtunnel attack, we developed a specification that disallows cron-spawned processes from connecting to external networks. With this specification, all instances of HTTPtunnel attack (there were 7 of them over the ten days of the attack) could be detected.

A6.6 Guess Telnet

Attack description: The attacker makes a number of telnet attempts and tries to guess the password.

Comments: This attack can be detected by a counting the number of login attempts made. A failed login attempt can be detected, when no seteuid is made after the passwd and shadow files are read. The specification monitored for such sequences of system calls. Whenever the number exceeded a certain constant over a period of *3* minutes, the specification flagged an attack.

A7 Addendum 2: Output of the SMS.

In this section, we show the identification list of attacks provided by Lincoln Labs and the output of the SMS after running it on the BSM files. Note that corresponding to an instance of an attack in the Lincoln Labs list, there could be multiple instances of the attack in the list generated by the SMS, e.g., for the fourth week data, according to the Lincoln Lab's list the Mondays' data contained a ps attack from 8:18:35 to 9:04:40. We detected that during this time frame 6 instances of ps attack occurred within very short intervals of time (< 1.5 seconds).

A7.1 Lincoln Labs Attack Identification List

The following is the list of attacks that Lincoln Labs identified as being present in the BSM data.

4th week, Monday (03/29/99)

1. ps 08:18:35 to 09:04:40
2. ftp-write 13:58:16 to 14:04:01

4th week, Tuesday (03/30/99)

1. httptunnel 09:09:17 to 09:11:56
2. ps 11:29:17 to 12:05:09

4th week, Wednesday (03/31/99)

1. Fdformat 10:38:00 to 10:41:00
2. warezmaster 11:25:39 to 11:26:26
3. secret 12:28:15 to 12:36:27
4. guest 15:53:15 to 15:55:58
5. guess telnet 17:58:17 18:02:05

4th week, Thursday (04/01/99)

1. guest 12:56:47 to 12:59:29
2. guess telnet

4th week, Friday (04/02/99)

1. secret 20:34:00 to 20:36:00

5th week, Monday (04/05/99)

1. warezclient 08:59:16 to 08:59:46

2. secret attack

ffbconfig 12:11:18 to 12:23:46

5th week, Tuesday (04/06/99)

1. ftp-write 10:19:16 to 10:33:20
2. ps 11:20:09 to 01:01:00
3. httptunnel 12:06:32 to 12:10:02
4. eject 12:55:14 to 01:11:46
5. Fdformat 16:24:15 to 17:20:39
6. secret 17:22:15 to 17:24:15

5th week, Wednesday (04/07/99)

no attacks on file ...

5th week, Thursday (04/08/99)

1. ps 08:33:00 to 08:36:00
2. httptunnel 12:06:30 to 12:10:01
3. Fdformat 12:57:17 to 13:38:27
4. warezclient
5. guess telnet

A7.2 Detection List by the SMS

Each line in the output has the following fields:

Process name: This is the name of the process being monitored. If SMS cannot identify the process name a ".." appears in its place. Process Id: This is the *pid* of the process being monitored. *Time :* Time of attack in Day Month Date Time Year format.

Path: Some attacks result in an invalid file operation. In such cases, the path refers to the path of the file. Some attacks involve executing an invalid program. The path is then the program being executed. In all other cases, path refers to the complete path of the program being monitored.

Attack description: Describes the invalid operation performed. The following bullets briefly explain the various attacks:

1. File Attribute Change: When an attack involves a system call that changes the attributes of a file. The most common system calls which change attributes on files are: chmod, *chown, chgrp, rename* etc.
2. Buffer Overflow: This message is displayed when an oversized argument is detected in an execve system call.
3. Invalid FTP write/ Invalid FTP read: These are messages when an invalid write or read is performed by *in.ftpd* program.

Invalid Write/ Invalid Read: These are attack messages, displayed when an invalid read or write is detected by the generic specification.

For instance, in the first line of the attack data:

in.ftpd 2581 Mon Mar 29 13:58:20 1999 /export/home/ftp/tricky Invalid FTP Write

in.ftpd is the program being monitored, *2581* is the process id, *Mon Mar 29 13:58:20 1999* is the date of the attack, /export/home/ftp/tricky is the file on which some invalid operation is being performed, and *Invalid* FTP *Write* implies that the file is being written into illegally.

Attack detection output from the SMS:

4th Week, Monday 03/29/99

```
.. 697 Mon Mar 29 08:50:18 1999 /usr/bin/ps Buffer Overflow
ps 697 Mon Mar 29 08:50:19 1999 /export/home/secret File Attrib Change
.. 706 Mon Mar 29 08:50:19 1999 /usr/bin/ps Buffer Overflow
ps 706 Mon Mar 29 08:50:19 1999 /export/home/secret/budget File Attrib Change
.. 715 Mon Mar 29 08:50:19 1999 /usr/bin/ps Buffer Overflow
ps 715 Mon Mar 29 08:50:20 1999 /export/home/secret/budget/spending File Attrib Change
.. 741 Mon Mar 29 08:51:20 1999 /usr/bin/ps Buffer Overflow
ps 741 Mon Mar 29 08:51:21 1999 /export/home/secret/budget/spending File Attrib Change
.. 750 Mon Mar 29 08:51:22 1999 /usr/bin/ps Buffer Overflow
ps 750 Mon Mar 29 08:51:22 1999 /export/home/secret/budget File Attrib Change
.. 759 Mon Mar 29 08:51:22 1999 /usr/bin/ps Buffer Overflow
ps 759 Mon Mar 29 08:51:22 1999 /export/home/secret File Attrib Change

in.ftpd 2581 Mon Mar 29 13:58:20 1999 /export/home/ftp/tricky Invalid FTP Write
in.ftpd 2581 Mon Mar 29 13:58:22 1999 /export/home/ftp/.rhosts Invalid FTP rename
```

4th Week, Tuesday (03/30/99)

```
in.telnetd 1032 Tue Mar 30 9:11:10 1999 /var/spool/cron/crontabs/rexn crontab entry
in.telnetd 1032 Tue Mar 30 9:11:10 1999 /tmp/crontaba000GM httptunnel
.. 1923 Tue Mar 30 12:01:05 1999 /usr/bin/ps Buffer Overflow
ps  1924  Tue  Mar  30  12:01:05  1999  /export/home/secret/crisis_plan/PANIC  File  Attrib
Change
```

4th Week, Wednesday (03/31/99)

```
.. 1205 Wed Mar 31 10:39:31 1999 ./usr/bin/Fdformat Buffer Overflow
Fdformat 1205 Wed Mar 31 10:39:31 1999 /usr/bin/ksh Invalid execve

in.ftpd 1356 Wed Mar 31 11:25:48 1999 /export/home/ftp/lrootk.tgz Invalid FTP Write
in.ftpd 1356 Wed Mar 31 11:25:48 1999 /export/home/ftp/lrootk.tgz Invalid FTP Read

.. 1530 Wed Mar 31 12:06:31 1999 connect httptunnel

cp 1786 Wed Mar 31 12:30:57 1999 /export/home/secret/personnel/byeltsin Invalid Read

in.telnetd 3118 Wed Mar 31 15:54:04 1999 /export/home/guest Guest Attack
in.telnetd 3119 Wed Mar 31 15:54:04 1999 /export/home/guest Guest Attack

in.telnetd 4231 Wed Mar 31 18:00:11 1999 /usr/sbin/in.telnetd Guess Telnet
in.telnetd 4265 Wed Mar 31 18:27:50 1999 /usr/sbin/in.telnetd Guess Telnet
```

4th Week, Thursday (04/01/99)

```
.. 1733 Thu Apr 1 12:06:32 1999 connect httptunnel

in.telnetd 2271 Thu Apr 1 12:57:36 1999 /export/home/guest Guest Attack
in.telnetd 2270 Thu Apr 1 12:57:36 1999 /export/home/guest Guest Attack

in.telnetd 2378 Thu Apr 1 13:23:33 1999 /usr/sbin/in.telnetd Guess Telnet
```

4th Week, Friday (04/02/99)

```
.. 1680 Fri Apr 2 12:06:31 1999 connect httptunnel

cat 1786 Fri Apr 02 20:35:15 1999 /export/home/secret/personnel Invalid Read
cat 1786 Fri Apr 02 20:35:15 1999 /export/home/secret/budget Invalid Read
cat 1786 Fri Apr 02 20:35:15 1999 /export/home/secret/crisis_plan Invalid Read
```

5th Week, Monday (04/05/99)

```
in.ftpd 566 Mon Apr 5 08:59:24 1999 /export/home/ftp/lrootk.tgz Invalid FTP Read

.. 1463 Mon Apr 5 12:08:31 1999 connect httptunnel
.. 1522 Mon Apr 5 12:22:31 1999 /usr/sbin/ffbconfig Buffer Overflow
ffbconfig 1522 Mon Apr 5 12:22:32 1999 /usr/bin/ksh Invalid execve

cp 1350 Mon Apr 5 11:44:35 1999 /export/home/secret/personnel/shussein Invalid Read
```

5th Week, Tuesday (04/06/99)

```
in.ftpd 483 Tue Apr 6 10:19:17 1999 /export/home/ftp/pp Invalid FTP Write
in.ftpd 483 Tue Apr 6 10:19:17 1999 /export/home/ftp/.rhosts Invalid FTP rename

.. 1133 Tue Apr 6 12:06:32 1999 connect httptunnel

.. 1494 Tue Apr 6 13:00:04 1999 /usr/bin/ps Buffer Overflow
ps 1494 Tue Apr 6 13:00:04 1999 /export/home/secret File Attrib Change
.. 1511 Tue Apr 6 13:00:05 1999 /usr/bin/ps Buffer Overflow
ps 1511 Tue Apr 6 13:00:05 1999 /export/home/secret/budget File Attrib Change
.. 1520 Tue Apr 6 13:00:05 1999 /usr/bin/ps Buffer Overflow
ps 1520 Tue Apr 6 13:00:06 1999 /export/home/secret/budget/spending File Attrib Change
.. 1556 Tue Apr 6 13:01:06 1999 /usr/bin/ps Buffer Overflow
ps 1556 Tue Apr 6 13:01:06 1999 /export/home/secret/budget/spending File Attrib Change
.. 1565 Tue Apr 6 13:01:07 1999 /usr/bin/ps Buffer Overflow
ps 1565 Tue Apr 6 13:01:07 1999 /export/home/secret/budget File Attrib Change
.. 1574 Tue Apr 6 13:01:07 1999 /usr/bin/ps Buffer Overflow
ps 1574 Tue Apr 6 13:01:08 1999 /export/home/secret File Attrib Change

in.telnetd 1595 Tue Apr 6 13:03:11 1999 ./usr/bin/eject Buffer Overflow

.. 3075 Tue Apr 6 17:01:14 1999 ./usr/bin/Fdformat Buffer Overflow

cat 3182 Tue Apr 6 17:22:45 1999 /export/home/secret/personnel/ghwbush Invalid Read
```

5th Week Thursday (04/08/99)

```
.. 470 Tue Apr 8 8:34:45 1999 /usr/bin/ps Buffer Overflow

.. 1979 Thu Apr 8 12:06:32 1999 connect httptunnel

.. 2361 Thu Apr 8 13:24:02 1999 ./usr/bin/Fdformat Buffer Overflow
Fdformat 2361 Thu Apr 8 13:24:03 1999 /usr/bin/ksh Invalid execve

in.ftpd 3936 Thu Apr 8 19:41:23 1999 /export/home/ftp/lrootk.tgz Invalid Read

in.telnetd 4018 Thu Apr 8 19:59:47 1999 /usr/sbin/in.telnetd Guess Telnet
in.telnetd 4057 Thu Apr 8 20:01:27 1999 /usr/sbin/in.telnetd Guess Telnet
in.telnetd 4092 Thu Apr 8 20:05:28 1999 /usr/sbin/in.telnetd Guess Telnet
```

☆U.S. GOVERNMENT PRINTING OFFICE: 2001-610-055-10143

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.