Building Survivable Systems: An Integrated Approach based on Intrusion Detection and Damage Containment

T. Bowen D. Chee M. Segal
Telcordia Technologies
445 South Street, Morristown, NJ 07960
{bowen,dana,ms}@research.telcordia.com

R. Sekar T. Shanbhag P. Uppuluri
State University of New York
Stony Brook, NY 11794
{sekar,tushar,prem}@cs.sunysb.edu

Abstract

Reliance on networked information systems to support critical infrastructures prompts interest in making network information systems survivable, *so that they continue functioning even when under attack. To build survivable systems, attacks must be detected and reacted to* before *they impact performance or functionality. Previous survivable systems research focussed primarily on detecting intrusions, rather than on preventing or containing damage due to intrusions. We have therefore developed a new approach that combines early attack detection with automated reaction for damage prevention and containment, as well as tracing and isolation of attack origination point(s). Our approach is based on specifying security-relevant behaviors using patterns over sequences of observable events, such as a process's system calls and their arguments, and the contents of network packets. By intercepting actual events at runtime and comparing them to specifications, attacks can be detected and operations associated with the deviant events can be modified to thwart the attack. Being based on security-relevant behaviors rather than known attack signatures, our approach can protect against unknown attacks. At the same time, our approach produces few false positives – a property that is critical for automating reactions. Our host-based mechanisms for attack detection and isolation coordinate with network routers enhanced with active networking technology in order to trace the origin of the attack and isolate the attacker.*

1 Introduction

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

In our approach, we compile BMSL specifications into efficient *detection engines.* The efficiency of these engines opens up the possibility of augmenting real-time intrusion detection with real-time intrusion reaction. We are experimenting with BMSL in two contexts; considering incoming network packets as events, and considering system calls requested by executing processes as events. For network packets, BMSL specifications are based on packet contents, and for system calls, BMSL specifications are based on both system calls and the values of system call arguments. In both contexts, BMSL supports a rich set of language constructs that allow reasoning about not only singular events, but also temporally related event sequences. Since our approach enables real-time reaction, capabilities for specifying the reactions to be invoked when violations are detected are integrated into BMSL.

While we are currently experimenting with packet and system call interception, interception of other events is possible as well. For instance, we may develop a runtime environment that will generate an event corresponding to the logging of each entry in the system log file `syslog`. Similarly, the Management Information Bases (MIBs) accessible through the Simple Network Management Protocol (SNMP) provide important security-relevant data that can be monitored using BMSL specifications.

The overall architecture of our intrusion detection/response system is shown in Figure 1. For a given event stream such as packets or system calls, an interceptor component placed in the stream provides efficient interception of raw events. The interceptors deliver raw event streams to a runtime environment associated with each stream. The runtime environments further demultiplex the event streams into the event streams for individual detection engines, which implement the actual intrusion detection and reaction specifications. The runtime environments also provide common functions that facilitate execution of the detection engines, and isolate the detection engines from the details of specific interfaces and data formats. Typically, a single detection engine monitors each defended process, and another detection engine monitors all of the network traffic of one or more hosts.

Reaction to detected intrusions is enabled by the detection engine's *interposition capabilities*. For example, a system call detection engine can interpose a reaction program

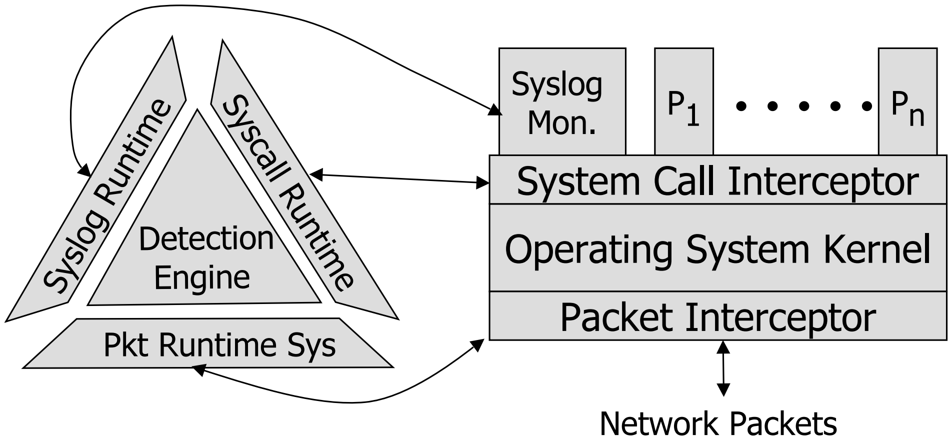

Figure 1. Runtime view of the system. $P_1, ..., P_n$ **are processes; Syslog Mon is a system log watcher.**

of arbitrary functionality either before, in place of, or after executing the intercepted system call's kernel functionality. Similarly, the network packet detection engine can alter, drop or spontaneously generate packets. In our prototype implementation, interposition capabilities have been implemented for system calls, while a passive interception capability has been implemented for network packets. Using interposition, a wide variety of reactions are possible. The simplest reactions merely terminate intruder access to the victim host, while the most complicated reactions seek to entrap the intruder and waste the intruder's resources by allowing intruder access, but placing compromised processes an isolated environment where they cannot cause damage to the victim.

The system call and packet interception defenses and reactions can help protect a host from damage, but ultimately computer security requires identification and isolation of the intruder. We are experimenting with techniques enabled by active networking technology to trace and isolate intruders.

Our main results are:

- BMSL, an *expressive, easy-to-use and and robust language* for capturing behaviors of processes and hosts as patterns over sequences of events such as system calls and network packets. Robustness is achieved by restricting the expressive power of BMSL, and by strong data typing that offers enhanced expressive power to capture network packet and system call arguments types. Since the detection engines may operate within the operating system kernel, robustness is an important goal for BMSL.
- *protection against known as well as unknown attacks.* The intruder's main goal is to gain unauthorized access to the resources on the victim host. Unauthorized access is difficult to gain when all the programs on the victim host are observing their security-relevant properties. By enforcing security-relevant properties, we protect against known and unknown intrusions.
- *efficient enforcement of normal behaviors, and isolation of misbehaving programs.* A key to damage prevention and/or isolation is the development of techniques for efficient interception and/or modification of system calls and network packets.
- *fast pattern-matching algorithms* that can detect deviations from normal behavior (or adherence to under-attack behavior) specified in BMSL. In general, the space required for efficient matching of BMSL patterns can be extremely large. To reduce space requirements, we develop a novel runtime model based on *quasi-deterministic extended finite state machines* that trade off space requirements against matching speed in a selective fashion, and achieve very good detection speeds and space utilization.
- *active-networking based techniques for tracing of attacker origin and isolation.*
- *effective and efficient implementation.* We present an analysis of the performance of our approach in terms of effectiveness and efficiency. Our experiments consisted of in-house simulations of attacks, as well as a standardized evaluation conducted by MIT Lincoln Labs [11] with the support of DARPA. The evaluation results indicate that our approach is effective, fast, and has low memory requirements.

This paper is intended to present a broad overview of the research results from our "Survivable Active Networks" project. A more detailed treatment of our language design and compilation techniques can be found in [38] and [37]. Our network monitoring system design, implementation and evaluation are presented in more detail in [36]. (Online versions of these papers can be accessed on the world-wide web at `http://seclab.cs.sunysb.edu/publicat.htm`.)

2 Specification Language 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, and the scope of damage that may result due to such errors
- *simplicity,* to control language and compiler complexity
- *amenability to efficient monitoring,* to keep overheads for runtime behavior monitoring low
- *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 achieved by associating each security property with the reaction to be taken when the property is violated. We describe the components of BSML below.

2.1 Types

BMSL types consist of primitive types such as integers, booleans and floats, event types that capture the structure of events, and aggregate types to capture the structure of system call arguments and network packets. BSML also supports various utility data structures such as lists, arrays and tuples. We confine the description below to the types that are unique to BMSL.

2.1.1 Events

Events may be *primitive* or *user-defined.* Primitive events are generated by a runtime system and constitute the input to our detection system. 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(`$client, server, username$`)`

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 2.3. All of the variables in $params$ must appear in pat.

2.1.2 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; rather, the representations are prespecified as part of protocol standards
- 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 may be 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 not powerful or flexible enough to satisfy our needs. In particular, the structure describing a lower layer protocol data unit (PDU) 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.

2.2 Class Types

It is not appropriate to describe and manipulate some of the data that is exchanged over the detection engines' 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 are shown below:

```
class CString {
        string getVal() const;
        void setVal(string s);
}
class StatBuf {
        int getDev()const;
        int getIno()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 as described later.

2.3 Patterns

Security-relevant properties of programs are captured as patterns over sequences of events such as system calls and network packets. In its simplest form, a BMSL pattern captures the occurrence of a single event. It is of the form $e(x_1, ..., x_n)|cond$, where $cond$ is a boolean-valued expression on the event arguments $x_1, ..., x_n$, as well as other variables that may appear earlier in the same pattern. The condition component can make use of standard arithmetic, comparison and logical operations and several support functions. The support functions allowed in a pattern correspond to "read" operations that do not modify the state of the monitored process. An example of such a function is $realpath()$ which translates a file name into a canonical form that does not contain "`.`", "`..`", or symbolic links.

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. The meaning of event patterns and the sequencing operators is best explained by the following definition of what it means for an *event history* H (a sequence of events observed at runtime) to match a pattern:

- *event occurrence:* $e(x_1, ..., x_n)|cond$ is satisfied by the event history $e(v_1, ..., v_n)$ if $cond$ evaluates to $true$ when variables $x_1, ..., x_n$ are replaced by the values $v_1, ..., v_n$.
- *event nonoccurrence:* $!e(x_1, ..., x_n)|cond$ is matched by H if it does not match $e(x_1, ..., x_n)|cond$.
- *sequencing:* $pat_1; pat_2$ is satisfied by an event history H of the form $H_1 H_2$ provided H_1 satisfies pat_1 and H_2 satisfies pat_2.
- *alternation:* $pat_1||pat_2$ is satisfied by an event history H if either pat_1 or pat_2 is satisfied by H.
- *repetition:* $pat*$ is satisfied by an event history $H_1 H_2 \cdots H_n$ iff H_i satisfies pat, $\forall\ 1 \leq i \leq n$.
- *realtime constraints:* pat **within** t is satisfied by an event history H_1 if H_1 satisfies pat and the time interval between the first and last events in H_1 is less than or equal to t.
- *atomicity:* **nonatomic** d **in** pat denotes that accesses to data d be atomic in pat, i.e., without any intervening operations by other processes that could modify this data.

When a variable occurs multiple times within a pattern, an event history satisfies the pattern only if the history instantiates all occurrences of the variable with the same value. For instance, the pattern $e_1(x); e_2(x)$ is not satisfied by the event history $e_1(a)e_2(b)$, but is satisfied by $e_1(a)e_2(a)$.

2.4 Response Actions

The reaction associated with a rule $p \rightarrow a$ is launched if *a suffix of the event history matches p.* The reaction component consists of a sequence of statements, each of which is either an assignment to a state variable or invocation of a support function provided by the runtime system[1]. The important classes of response actions are specified below.

2.4.1 Data Aggregation Operations

To identify network attacks it is often necessary to aggregate information across many network packets, and act on the basis of this information. Aggregation needs to be more sensitive to recently received packets than older packets. 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 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 are both time and space efficient. For a more detailed presentation of data aggregation operations, the reader is referred to [36].

2.4.2 Event Modification Operations

Event modification operations are realized as a set of support functions provided by the runtime system. Thus the event modification capabilities will differ for different runtime systems. 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 return a fake return value. This is accomplished using a support function $fake$, which takes an argument that corresponds to the value to which the variable $errno$ should be set to.

[1]Knowledge about these support functions are not integrated into BMSL, but us declared in header files that can be included in the specification.

2.4.3 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) by 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 [37] — 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 Illustrative Examples

3.1 Simple Examples

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 server program `fingerd` from executing any program, modifying file permissions, creating files or directories or initiating network connections. 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.

$execve||connect||chmod||chown||creat||truncate$
$||sendto||mkdir \rightarrow fake(EINVAL)$

We may also restrict the files that a process may access for reading or writing, for example, the the following rule prevents writes to all files, and reads from any files other than those mentioned in `admFiles` defined below.

$admFiles =$
`{"/etc/utmp","/etc/passwd", "datadir/*"}`
$open(f, mode)|(realpath(f) \notin admFiles$
$||\ (mode \neq \texttt{O\_RDONLY}) \rightarrow fake(EPERM);$

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.

$openExit(fd) ::= open\_exit(..., fd)||creat\_exit(..., fd)$
$rwOp(fd) ::= read(fd)||readdir(fd)||write(fd)$
$openExit(fd); (!rwOp(fd))*; close(fd) \rightarrow \cdots$

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
$(access(f); (!open(f))*; open(f)) \rightarrow fake(EACCES);$

3.2 Case Study: Specification for `ftpd`

Our starting point in developing a specification of `ftpd` is the documentation provided in ftpd's manual pages. 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 BMSL in Figure 2. Although we can convert the English descriptions directly into BMSL specifications we usually cross-check (or "debug") the specifications by monitoring ftpd under typical conditions. We use a hybrid approach, where we first manually inspect system call traces produced by ftpd, and used them to further narrow down the actions/behaviors that the ftpd server may exhibit. In most cases, we have not attempted a sophisticated reaction, instead opting for a simply terminating ftpd using a support function named $term()$.

Figure 2 shows a partial specification for `ftpd`. A complete specification can be found in [37], which consists of about 25 rules and event definitions. Explanations for (some of) these rules are as follows:

- ftpd attempts to authenticate the client host before proceeding to user authentication. 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 between installations. We treat $getpeername$ as a marker that indicates client host authentication related processing. Similarly, we treat opening of $/etc/passwd$ as a marker for user authentication related processing. Rules 4 and 5 capture these English descriptions by stipulating respectively that an open of the password file should never happen before invocation of $getpeername$, and that $setreuid$ system call should not be executed before opening of the password file.
- 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 2).
- prior to user authentication, only files beginning with names identified in the set `ftpAdmFilePrefixes` can be accessed (rule 6).
- certain system calls are never used before user authentication, and others are never used after user authentication (rules 7 and 8). Let `ftpInitBadCall` denote a pattern that matches system calls not used prior to user authentication. Similarly, let `ftpAccessBad-Call` match system calls that are not used after user authentication. (Definitions of these abstract events are omitted in order to conserve space.)
- for anonymous users, the userid `FTPUSERID` is used; moreover, the `chroot` system call is used to restrict access only to the subtree of the filesystem rooted at `~ftp` (rule 9).
- ftpd resets its effective userid to root in order to bind to socket 20 (ftp data port). The userid is restored to that of the logged in user immediately afterwards (rule 11).
- 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 10). In addition, we make sure that any file that is opened with superuser privilege is closed before execve (rule 12).

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 our language.

The specification enforces 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 14), race conditions in signal handling (rules 11) and site-exec (rule 13) [4].

```
    /* 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). */
1.  ftpPrivCalls ::= close||uidgidops||socket||setsockopt||(bind(s,sa)|port(sa)=20)
    /* Use a state variable to remember the uid of user logging in and client host name */
2.  begin();(!setreuid)*;setreuid(r,e) → loggedUser := e
3.  begin();(!getpeername)*;getpeername_exit(fd,sa,l) → clientIP := getIPAddress(sa)
    /* Host authentication phase must precede user authentication. */
4.  begin();(!getpeername)*;open(/etc/passwd) → term()
    /* User authentication must precede before userid changed to that of the user. */
5.  begin();(!open(/etc/passwd))*;setreuid() → term()
    /* Access limited to admin-related files before user login is completed. */
6.  begin();(!setreuid())*;open(f)|(!isExtension(ftpAdmFilePrefixes, f)) → term()
    /* Access limited to certain system calls before user login. */
7.  begin();(!setreuid())*;ftpInitBadCall() → term()
    /* Certain system calls are not permitted after user login is completed. */
8.  setreuid();any()*;ftpAccessBadCall() → term()
    /* Anonymous user login: must do chroot before setreuid. */
9.  begin();(!(setreuid||chroot(FTPHOME)))*;setreuid(r,FTPUSERID) → term()
    /* Userid must be set to that of the logged in user before exec. */
10. begin();(!setuid(loggedUser))*;execve → term()
    /* Resetting userid to 0 is permitted only for executing a small subset of system calls. */
11. setreuid(r,0);ftpPrivCalls*;
        !(setreuid(r1,loggedUser)||setuid(loggedUser)||ftpPrivCalls) → term()
    /* Any file opened with superuser privilege is either explicitly closed before an exec, or has close-on-exec flag set. */
12. (open_exit(f,fl,md,fd)|geteuid()=0);(!close(fd))*;(execve|!closeOnExec(fd)) → term()
    /* Site-specific: ensure ftp cannot execute arbitrary programs. */
13. execve(f)|(f ∉ ftpValidExecs) → term()
    /* Site-specific: ftp cannot connect to arbitrary hosts or services. */
14. connect(s, sa)|((getIPAddress(sa)!=clientIP)&&(getPort(sa)∉ftpAccessedSvcs)) → term()
```

Figure 2. A specification for `ftpd`.

3.3 Network-Based Attacks

3.3.1 Very Small IP Fragments

We begin with a simple example to identify unusual network packets that can often be used to launch attacks. For instance, very short IP fragments that are smaller than TCP headers can be used to bypass packet-filtering firewalls. We can detect such packets using:

```
MY_NET = 129.186.44.0
MY_NET_MASK = 255.255.255.0
my_net_addr(a) = ((a&MY_NET_MASK)=MY_NET)
is_frag(p) = (p.more_frags)||(p.frag_offset!=0)

MFUTable tcpFrag(
  unsigned int, /*key is IP address, no data*/
  100, 30, /*size 100, time window 30 sec*/
  1, 0,          /* hi, lo thresholds */
  tcpFragBegin, tcpFragEnd) /*threshold fns */
rx(p)|my_net_addr(p.daddr) &&
      is_frag(p) && p.protocol=IP_TCP &&
      p.tot_len < 48 -> tcpFrag.inc(p.saddr)
```

The functions `tcpFragBegin` and `tcpFragEnd` write records to a log file. They both take an argument that is the value of the key field corresponding to the table entry for which the action is being executed.

The threshold values in the example make attack detection to be very deterministic: an attack is recognized even if a single packet matching the criteria is received. The reasons for using a table in such a case (as opposed to directly invoking a function that generates an attack report) are as follows. First, we are able to distinguish among packets received with different source addresses and treat them as separate attacks. Second, the attacking host may generate a large number of fragmented packets that match this criteria. Rather than generating many attack messages, we may generate just two messages that indicate the beginning and end of the attack.

3.3.2 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 simply duplicated.[2]

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

[2] Not all overlaps correspond to teardrop attacks, but we used this pattern since it is simpler than the one that would permit legitimate fragment overlaps, and since overlapping IP fragments never appeared in the environments where our IDS was tested.

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

The pattern matches any sequence of packets that spans a period less than sixty seconds (one may choose a larger or smaller time frame), begins and ends with fragments of the same IP packet, and these fragments overlap partially.

3.4 Using Specification for Isolation

The isolation component is integrated seamlessly into our specification-based framework for detection of attacks on host and processes. When we detect an attack on a host that is delivered via network packets, we can quietly drop those packets. When we detect an attack on a process, we can use the switch action to switch to a new specification that contains BMSL rules to isolate the process. The new specification contains rules to:

- return faked return value, especially for system calls that can potentially damage the system
- log the activity for later analysis.
- reduce limits on resources that the rogue process can consume.
- restrict access to files.

For instance, the specification

```
exec -> chroot("/altroot"); setuid(-1);
        nice(20); switch genericIsolate;
```

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

```
connect || sendto || recvfrom
   -> sleep(60); fake(ETIMEDOUT)
bind || recv -> sleep(5); fake(EADDRINUSE)
open || read || write -> sleep(1)
```

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

4 Compilation of BMSL

4.1 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.

4.1.1 Packet Types

The semantics of the constraints in packet types is that they must hold before fields corresponding to a derived type are accessed. In particular, note that at compile time, we do not know the type of a packet received on a network interface, except for the lowest layer protocol type. 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 accessing the IP-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.

4.1.2 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 wanted to really store one or more components of some foreign object in BMSL, we need to use the appropriate accessor functions on this data to obtain the components of interest and store them as BMSL native types.

4.2 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

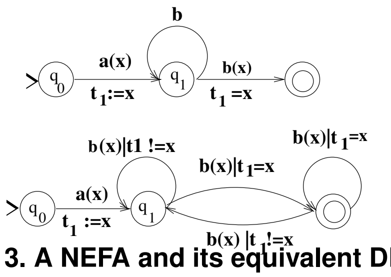

Figure 3. A NEFA and its equivalent DEFA

in a manner analogous to compiling regular expressions into finite-state automata. We call these automata *extended finite-state automata* (EFSA). EFSA are simply standard finite state automata (FSA) that are augmented with a fixed number of *state variables,* each capable of storing values of a bounded size. Every transition in the EFSA is associated with an event, an enabling condition involving the event arguments and state variables, and a set of assignments to state variables. The final states of the EFSA may be annotated with actions, which, in our system, correspond to the reactions given in our rules. For a transition to be taken, the associated event must occur and the enabling condition must hold. When the transition is taken, the assignments associated with the transition are performed.

An EFSA is normally nondeterministic. The notion of acceptance by a nondeterministic EFSA (NEFA) is similar to that of an NFA. A deterministic EFSA (DEFA) is an EFSA in which at most one of the transitions is enabled in any state of the EFSA. A NEFA for the pattern $a(x); b*; b(x)$ is shown in Figure 3. The equivalent DEFA is also shown in the same figure. We have shown that translating a NEFA to a DEFA can result in an unacceptable increase in the size of the automaton. Therefore we have developed a new approach that is based on translating NEFA into a *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 [37].

5 Runtime Infrastructure for System Calls

The BMSL specifications for system call based defenses must be translated into a detection engine and linked with a runtime infrastructure to produce an executable. The executable is a single Dynamically Loadable Kernel Module (DLKM) called `imod.o`. The runtime infrastructure provides the following capabilities:

- *mechanisms for associating running applications with BMSL specifications,*
- *interception of system calls, delivery of an application's intercepted system calls to its associated BMSL specification,*
- *delivery of a specification's response to intercepted system calls to the application,* and
- *support functions to simplify BMSL specification development.*

The runtime infrastructure imposes a few constraints on BMSL compilation. For each BMSL specification, the BMSL compiler must generate C++ source code defining a unique class. The generated classes must be derived from a default specification class called `baseProg`. For each system call that a BMSL specification needs to intercept, the class must implement interception methods having predefined signatures. The basic idea behind these constraints is that `baseProg` implements default methods for interception of all system calls at entry and exit. The default methods allow all intercepted system calls to execute normally. To achieve non-default treatment of selected intercepted system calls, the derived classes override the method implementations in `baseProg` corresponding to these system calls.

Given a set of BMSL specifications conforming to the runtime infrastructure's constraints, translation of the specifications and the runtime infrastructure into `imod.o` is basically a matter of C++ compilation and link editing, with one important exception. The exception is that the association between applications and their defensive specifications is encoded at compile time. The association requires the programmer to create a configuration file comprising multiple entries, where each entry gives the command line name of an application, the name of the C++ class associated with that application, and the name of the C++ source file defining that class. A single application can be associated with only one specification but many applications can be associated with a single specification. Using the configuration file, the translation selects the appropriate BMSL specifications to link edit into `imod.o`, and populates data structures within `imod.o` so that at runtime, the association between applications and specifications is known. The translation also examines the BMSL specifications to determine what subset of system calls must be intercepted to meet their combined needs, and uses this information to minimize runtime overhead by avoiding unnecessary system call interception.

The primary function of the infrastructure is real-time system call interposition. System call interposition is accomplished by system call table overwriting. The system call table is a kernel resident table of pointers to functions implementing system calls that is indexed by predefined system call numbers. When an application requests a system call, the system call's number is used to select the appropriate function pointer from the system call table. When `imod.o` is loaded, it first copies the system call table, and then overwrites the system call table so that the system call table entries point at interception functions implemented in `imod.o`, rather than at the real system call functions. (For efficiency, the table entries for system calls that no BMSL specification requires interception of are not overwritten.) All `imod.o` interception functions have the same structure. The interceptor functions start with pre-system call functionality, implemented by a function called `preTrap()`, then invoke the real system call, using the entry in the system call table copy, and conclude with post system call functionality, implemented by a function called `postTrap()`. The `preTrap()` and `postTrap()` functions are roughly equivalent, consisting of finding the object associated with the application requesting the system call, or instantiating such an object based on the application to specification association if this is the application's first system call, and then invoking an interception method on that object.

The interception method invoked is the method implemented for the particular system call that was intercepted, with `preTrap()` using the intercepted system call's entry method and postTrap using the intercepted system call's exit method. Note that if BMSL specification does not specify special treatment of an intercepted system call, the default treatment implemented by `baseProg` is invoked. The return values of both `preTrap()` and `postTrap()` are used to convey the BMSL specification's decision regarding treatment of the intercepted system call. The decision can be *normal*, meaning the request system call is allowed, or *faked*. For pre-system call interception, faked means that the system call is not to be executed, but instead an immediate return to the application is to be made using supplied results that mimic actual system call execution. For post-system call interception faked means that the results of the system call (which has already been executed) are to be overwritten. In both cases, a decision of faked requires the BMSL specification to supply appropriate results, such as a return code and population of output arguments, which vary between system calls.

A secondary function of the runtime infrastructure is to provide an environment that is conducive to the development of powerful specifications. This functionality is provided through support functions implemented within the infrastructure. The support functions provided are either in common to a large number of specifications or cannot reasonably be implemented within BMSL specifications. The following capabilities are currently provided in our implementation of the support functions:

- *a mechanism by which specifications can themselves use system calls.* The infrastructure provides support for BMSL specifications use of system calls in two contexts: backwards, or in the application's context, and forwards, or in `imod.o`'s context. Backward system calls support defenses that replace requested system calls with other system calls, while forward system calls enable `imod.o` to perform operations such as maintaining its own log files.
- *maintenance of a dynamic file descriptor- to-filename mapping.* The mapping between file descriptors in file names is fluid since many system calls can change the mapping, and because of inheritance of file descriptors across system calls such as execve, fork, and clone. Defenses that are based on file name could implement the functionality required to maintain file descriptor to file name mapping themselves, but to do so would unduly complicate them. Therefore, the functionality is consolidated and the mapping made available to BMSL specifications through a simple interface.
- *a unified method of handling system call arguments that are pointers.* Since BMSL specifications execute in kernel space, they have to be aware of the differences between kernel and user memory. Particularly troublesome are pointers passed through system calls, since they point at user memory which requires different access mechanisms than kernel memory. As mentioned earlier, this memory access problem is handled in BMSL by encapsulating the system call arguments using class types. The runtime infrastructure provides the implementation of these class types. The infrastructure converts pointer arguments into the appropriate class when the system call interception method is invoked.
- *implementation of subroutines commonly available at user level through standard system libraries.* In general, these subroutines are unavailable at kernel level. Our current approach is to port subroutines on an "as needed" basis, since their conversion into functions that can execute at kernel level is sometimes non-trivial.

6 Global Isolation Via Active Networking

Emerging active network technology presents appealing defensive capabilities to augment those of the host-based system call interception approach. There are a variety of attacks for which a host based approach can detect the attack, but cannot react in a useful manner. Many denial-of-service (DOS) attacks fall into this category: the host can detect that it is swamped by meaningless requests, and may even know the (spoofed) source IP address from which the requests originate, but cannot do anything to preserve itself under the attack. (Hosts can always respond to the attack by shutting down the attack services, but this reaction is not useful, since it accomplishes the aims of the attacker.) Given emergent active network capabilities, a more useful reaction would be for the attacked host to inform the nearest active network element of the attack and request that active network elements work together to find and isolate the source of the attack. With such a defensive strategy, not only can a host protect itself, but also, elimination of attack traffic near the source benefits all hosts by reducing unnecessary traffic.

We have a rudimentary implementation of an active network defense against flooding attacks using PLAN [16]. The implementation allows a host under attack to send a PLAN packet to its nearest PLAN enabled router. The packet contains a program which examines the router's routing table to see if the IP address responsible for the flood is immediately adjacent to the router. If so, that IP address is disabled using the router's administrative commands, if not, the PLAN packet is forwarded to the router's neighbors. Eventually, the PLAN packets is delivered to the router nearest the attacking IP address, and the flood is shut off at that point.

We believe that our current research into active network technology for defense is illustrative, but insufficient. Progress toward truly powerful active network based defenses requires further maturity of the technology, which is currently in prototype form. In particular, PLAN supports interception of only PLAN packets, not all other packets (such as TCP packets). These other packets can be monitored by PLAN programs as they pass through routers, but they cannot be deflected or have their contents altered, as envisioned defenses would require. Currently active network based defenses are limited to what can be achieved by using active packets to cause routers to re-configure themselves through their own administrative interface.

7 Performance Results

We have studied the effectiveness and performance of our approach experimentally. The effectiveness was also measured via our participation in the intrusion detection evaluation conducted by MIT Lincoln Laboratories.

7.1 ID Evaluation

Our network packet monitoring system (NMS) participated in the evaluation conducted by MIT Lincoln labs [11]. Due to the lack of availability of system call data for Linux in the evaluation, our system call monitoring system (SMS) did not participate in the evaluation. The focus of NMS and SMS are complementary. SMS is concerned with attacks that are targeted at specific processes, and thus manifest themselves at the system call level. The NMS is concerned with low-level network attacks that exploit errors in operating system kernels, and are not directed at any process.

Other participants in the evaluation included research groups from UC at Santa Barbara, Columbia University, RST Corporation, and two groups from SRI. A baseline system comparable to commercial intrusion detection systems was also included in the evaluation. It was determined that all of the systems participating in the evaluation provided

Attack Category	No. of Attacks	False positives	Score	Best score in evaluation
Probe	17	1	86%	86%
DOS	43	4	60%	65%

Figure 4. Scores by attack category.

significantly better detection rates over the baseline system, while reducing false positive rates by an order of magnitude or more.

The evaluation organizers set up a dedicated network to conduct a variety of attacks. Care was taken to ensure the accuracy of normal traffic as well. All of the network traffic was recorded in tcpdump format and provided to the participants of the evaluation. The data provided consisted of seven weeks of training data, plus two weeks of test data. The tcpdump files were 0.4 to 1.2GB in length per day.

The attacks were classified into four categories. Of these, only two categories related to low-level network attacks that are the focus of NMS. These two categories were probing and denial-of-service.

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.

Figure 4 shows the overall scores assigned to our system by Lincoln Labs [11]. The scoring scheme assigned fractional credit to each attack based on the percentage of the attack-containing packets (or sessions) identified by the IDS being evaluated. For instance, a port sweep may occur over hundreds or thousands of packets. Any IDS is able to identify only a subset of these packets as being part of a port sweep. This scoring procedure is not favorable for systems such as ours that emphasize low false positives. Such systems tend to err on the side of not identifying individual packets as attack-bearing, as long as a substantial number of packets within the attack can be tagged. Nevertheless, our system finished among the top two in both categories at low false-positive rates of 0.05 to 0.1 false alarms per attack. At false positive rates that are several tens of times higher (e.g., 2 to 3 false alarms per attack), some of the other systems perform better than us. However, it is quite likely that our system would sport higher detection rates if we increased the false positive rate to such high levels.

Figure 5 shows the scores obtained by our IDS for each kind of attack. Since it omits some of the higher-level probing and denial of service attacks that are not addressed by NMS, the aggregate score shown in this table is an improvement over that given in Figure 4. This table also shows the result under a different scoring scheme that attempts to

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

Figure 5. Scores on low-level network attacks.

Week	Day	Data file size (GB)	Time/GB of data (sec)	Memory (MB)
1	Mon	0.41	7.6	< 1
1	Tue	0.84	21.4	< 1
1	Wed	0.46	12.2	< 1
1	Thu	0.76	21.8	< 1
1	Fri	0.43	17.4	< 1
2	Mon	1.20	21.4	< 1
2	Tue	0.45	15.3	< 1
2	Wed	0.54	8.7	< 1
2	Thu	0.60	13.0	< 1
2	Fri	0.50	10.6	< 1

Figure 6. Performance of NMS.

identify whether each an attack is completely missed by a system. If a substantial fraction of the attack-bearing packets (say, 50%) are detected by the system, then we treat the attack as having been detected. Otherwise, we treat the attack as having been missed. Our system demonstrated excellent detection capability (96%) when using this criteria. The only attacks missed were due to the fact that the tcpdump contained only packets arriving into the network from outside, while we had assumed that it contained all of the internal traffic as well. As such, the explosion in the number of packets expected by our system as part of a UDP loop attack was not present in the tcpdump data, and hence the attack was missed by our system.

7.2 Performance of NMS

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 Figure 6. While running on a 450MHz Pentium II PC running RedHat Linux 5.2, our system can sustain intrusion

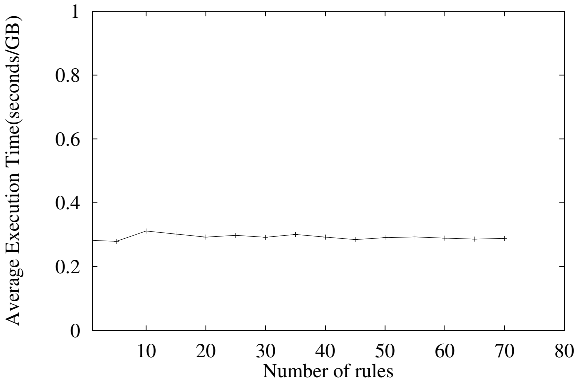

Figure 7. Pattern-matching time Vs number of rules.

detection at the rate of over 500Mb/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 low, largely the result of our choice of data aggregation operations. 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.
- *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.

We note that the time for detection does not monotonically increase 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. This factor can lead to the situation where the addition of rules *decreases* the execution time.

7.3 Performance of SMS

We studied the performance of our system with three server programs, namely, `ftpd`, `telnetd` and `httpd`. The specification for `ftpd` was as described earlier. The specification for `telnetd` and `httpd` are not shown, but are comparable in size and complexity to that of `ftpd`. All of the results in this section were taken on a 350MHz Pentium

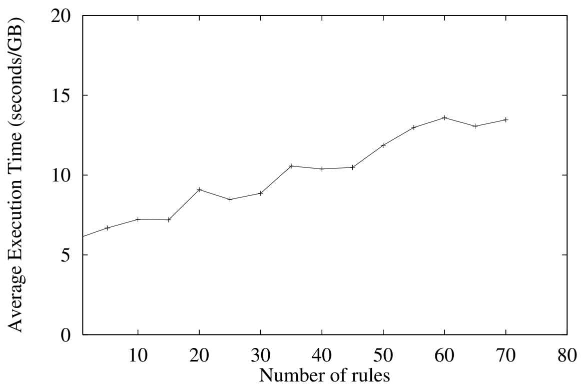

Figure 8. Intrusion detection time Vs number of rules.

TestCase	Time to run(s)	Time to match all sysCalls (s)	Overhead
ftpd	2.2s	0.03	1.5%
telnetd	3.1s	0.04	1.3%
httpd	5.8	0.09	1.5%

Figure 9. System call pattern matching overhead.

II Linux PC with 128MB memory and 8GB EIDE disk.

In measuring the runtime overhead we have separated measurements of the cost of system call interception, which are essentially constant for all system calls, from measurements of the cost of system call pattern matching, which varies based on the complexity of individual BMSL specifications. Our measurement of the cost of system call interception indicate that it adds only a minor overhead to most applications, for example, for `ftpd`, system call interception increases total time spent processing system calls by about 4.5%. In our test cases, system call processing time accounted for about 20% of total ftpd processing time, so 4.5% increase in system call processing has a virtually immeasurable impact on user perceived execution time.

7.3.1 Timing Results

Figure 9 summarizes our experimental results. They show that in fact, the overheads due to the monitoring are almost imperceptible for all three applications. The runtime data storage requirements for the FSM are too small to be measured.

Figure 10 shows the overhead for matching each system call, averaged across the three programs. Although the graph seems to indicate a slight increase in matching time per system call with increase in number of system calls, this increase is too small to be meaningful — the increase in time is about 5% when the number of system calls has increased by about 2000%. Thus, it is meaningful to talk about overhead as a percentage of the total runtime, as was done in Table 9.

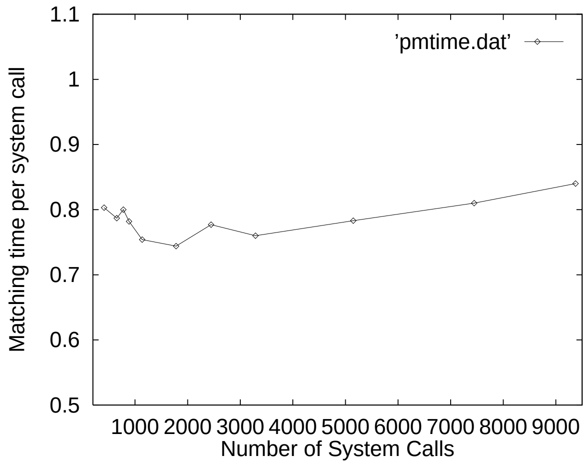

Figure 10. Matching time per system call.

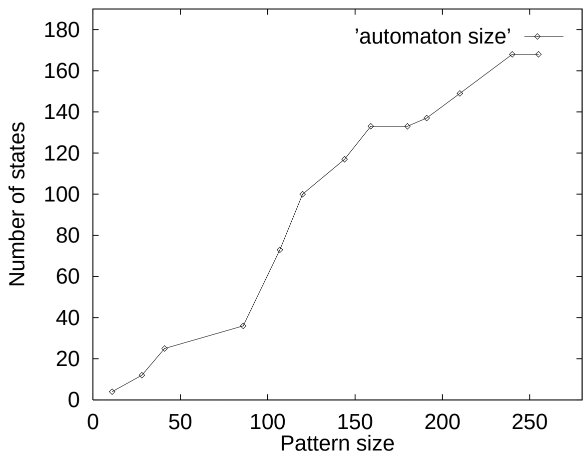

Figure 11. Increase in automaton size with specification.

7.3.2 Automaton Size

To evaluate the increase in size of the automaton when the number or complexity of patterns is increased, we have plotted the automaton size as a function of the size of the patterns. The size measure we use is given by the number of REE events and operators, taken over all of the patterns. Event arguments and conditions are not included, as the number of states is not very much affected by their presence or absence. The REE patterns of interest were those corresponding to our three benchmark programs, `ftpd`, `telnetd` and `httpd`. Randomly chosen subsets of these patterns were compiled and the corresponding size of the automaton was identified. These were then plotted as shown in Figure 11.

Although the worst-case automaton size is exponential in the size of REE, we find that in practice, the size increases more or less linearly with the total REE size.

8 Related Work

8.1 Host-Based Detection

Host-based techniques are aimed at protecting individual hosts and operate on the basis of information contained in audit logs or other similar sources of data. These techniques can be broadly divided into *misuse* detection [33, 20], *anomaly* detection [1, 8, 13], and *specification-based* detection [19, 38].

Among misuse-based approaches, a state-transition diagram based approach is used in [33] to capture signatures of intrusions. [20] uses colored petri nets to specify intrusive activity. This language is more expressive than ours in some ways (e.g., ability to capture occurrence of two concurrent sequences of actions), and less expressive in some other ways (e.g., ability to capture atomic sequences or the occurrence of one event immediately following another). Nevertheless, most intrusion signatures expressed in [20] can be easily captured in our language as well and hence our compilation techniques are applicable to their approach.

Among anomaly detection approaches, one of the first works based on program behaviors (as opposed to user behaviors) was that of [8]. Recently, these results have been improved by [13] using a neural network based approach that produces very accurate anomaly detectors. These approaches deal only with system call names, not with arguments. This simplifies the problem of *learning* normal behaviors of processes, which is the main focus of their work. However, for intrusion prevention or confinement, argument values are indeed important, e.g., we cannot otherwise distinguish an action to write a log file from one to modify the `/etc/passwd` file. Thus a language like REE is more appropriate in this context.

A specification-based approach achieves the accuracy of misuse detection, while addressing one of its deficiencies, namely, the inability to deal with unknown intrusions. It was first proposed in [19]. They use a pattern language based on context-free grammars extended with variables, and formulate the intrusion detection problem as one of parsing the audit logs with respect to these grammars. In contrast, our language is based on an extension of regular languages with variables. While context-free languages are more expressive than regular languages, this is not necessarily true when variables have been added to these languages. On the other hand, a regular language based formulation lends itself more readily to an automaton based pattern-matching approach that can be implemented efficiently.

8.2 Network Intrusion Detection

Most network intrusion detection systems [15, 32, 17, 21, 29, 31, 40, 34] operate by inspecting IP (or lower level) packets, most of them attempt to reconstruct the higher level interactions between end hosts and remote users, and identify anomalous or attack behaviors. Based on this, they attempt to identify a broad class of attacks, focusing particularly on malicious attacks on network servers and other processes running on the target system. We employ a different approach – in particular, attacks that are identifiable using higher level information are left to be detected by the SMS. Attacks that are invisible to SMS since they do not manifest themselves at the level of processes are left to be detected and handled by the NMS. Thus NMS is mainly concerned with detecting low-level attacks that exploit vulnerabilities in the design and implementation of host operating systems and network protocols. Most surveillance, probing and a large number of denial-of-service attacks in existence fall into this category of low-level network attacks. This two-part approach enables us to simplify the detection of diverse kinds of intrusions.

A completely different approach is taken for intrusion detection in [21], where techniques based on data mining are employed. Several previous works such as [15] also employed statistical and expert-system based techniques for detecting anomalous behaviors that could be indicative of attacks. These techniques largely complement pattern-matching based schemes such as ours. In particular, the benefits of our approach are speed, specificity and reduction of false positives. The downside is that unknown attacks, hitherto not captured, may go undetected. The anomaly detection systems are typically better at detecting unknown attacks, but their downside include high false-positive rates, nonspecific attack indicators, and need for extensive training. Combination approaches, such as those envisioned in EMERALD, can give us the benefits of both approaches while largely avoiding their drawbacks.

8.3 Related Work in Languages for Network Intrusion Detection and Packet Filtering

The use of special-purpose languages for network intrusion detection has been studied earlier. The choices range from scripting languages that make it easier to write intrusion detection code [34], C-like-but-strongly-typed languages such as that used in Bro [31], to a pattern-matching language in NetSTAT [40]. A common feature of these languages is that they are based on an imperative programming paradigm, whereas our language is declarative. Moreover, our language permits us to more easily capture patterns on sequences of packets, as opposed to other languages where patterns can characterize only individual events. This capability, together with the data aggregation features provided by our language, contributes to the conciseness of intrusion specifications. Another important distinction of our approach is that our language is designed to support efficient implementations of the pattern-matching and data aggregation operations.

Our type system for network packets is similar to *packet types* that have been developed independently in [5]. Their notion of type refinement is similar to our notion of inheritance for packets in that both approaches make use of constraints to augment the traditional notion of inheritance. This gives both approaches the ability to model layering of protocols. However, there are several significant differences as well. For instance, our notion of disjunctive inheritance is not captured in [5]. Moreover, we provide a general purpose algorithm that avoids repetition of constraint checking operations even if they are repeated along an inheritance chain or within rules. A more detailed treatment of the differences can be found in [36].

8.4 System Call Interposition

Interception of system calls, followed by interposition of arbitrary code at this point, has been proposed by many researchers as a way to confine applications. The Janus system [14] incorporates a user-level implementation of system call interception. It is aimed at confining helper applications (such as those launched by web-browsers) so that they are restricted in their use of system calls. Our approach improves on theirs by providing a more powerful language for capturing allowable behaviors. The kernel hypervisor [28] approach is similar to the Janus approach, but is implemented within an operating system kernel using loadable modules.

A more comprehensive set of system call interposition capabilities was developed in [10]. [9] focuses on the related problem of developing languages customized for writing interposition code (also known as *wrapper* code), and runtime infrastructure for their installation and management. Unlike the preceding approaches, their language can more easily capture sequencing relationships among system calls. But they do not focus on pattern-based techniques for intrusion detection. Moreover, computational issues in efficient matching of system call sequences, or robustness of interposition code, are not addressed.

9 Conclusions

In this paper we presented an approach for building survivable systems. Our approach is based on monitoring events such as system calls and network packets, comparing them against patterns characterizing normal (or abnormal) event sequences, and initiating appropriate responses when (potentially) damaging events are intercepted. These responses may preempt intrusion, or otherwise isolate and contain any damage. Since attacks can be prevented and/or contained, our approach can satisfactorily address intrusions arising due to software errors in otherwise trustworthy programs, as well as malicious programs (e.g., Trojan horses) from untrusted sources. Moreover, when new vulnerabilities (not protected by existing specifications) are identified, we can protect against them using appropriate specifications, instead of disabling vulnerable software until the vendor provides a patch.

One of the main challenges in making our approach practical is the ability to perform runtime prevention/detection that is fast enough to be included as part of processing every system call made by every process. We proposed a solution to this problem in this paper by developing appropriate compilation and runtime techniques. Our implementation results demonstrate that the overhead for intrusion detection is low. A key benefit of our approach (supported by theory as well as performance experiments) is that the detection time is insensitive to the complexity or number of patterns used in the specification. In most cases, our algorithm takes a constant time per system call intercepted, and uses a constant amount of storage. These advantages mean that specification developer can focus exclusively on conciseness, clarity and correctness of specifications, rather than be concerned about efficiency. Many of these advantages make our algorithm attractive for methods other than our own.

Future work will focus on tools and techniques to simplify the task of developing specifications.

References

[1] 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.

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

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

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

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

[6] 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.

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

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

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

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

[11] 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.

[12] B. Guha and B. Mukherjee, Network Security via Reverse Engineering of TCP Code: Vulnerability Analysis and Proposed Solutions, Proc. of the IEEE Infocom, March 1996.

[13] 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.

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

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

[16] 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.

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

[18] M. Jones, Interposition Agents: Transparently Interposing User Code at the System Interface, 14th ACM Symposium on Operating Systems Principles, December 1993.

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

[20] S.Kumar, Classification and Detection of Computer Intrusions, Ph.D Dissertation, Department of Computer Science, Purdue University, 1995.

[21] W. Lee, C. Park and S. Stolfo, Automated Intrusion Detection using NFR: Methods and Experiences, USENIX Intrusion Detection Workshop, 1999.

[22] R.W. Lo, K.N. Levitt, R.A. Olsson, MCF: a Malicious Code Filter, Computers and Security, Vol.14, No.6, 1995.

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

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

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

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

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

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

[29] B. Mukherjee, L. Heberlein and K. Levitt, Network Intrusion Detection, IEEE Network, May/June 1994.

[30] K. Olender and L. Osterweil, Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation, IEEE Transactions on Software Engineering, 16(3), 1990.

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

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

[33] P. Porras and R. Kemmerer, Penetration State Transition Analysis:A Rule based Intrusion Detection Approach, Eighth Annual Computer Security Applications Conference, 1992.

[34] M. Ranum et al, Implementing A Generalized Tool For Network Monitoring, LISA, 1997.

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

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

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

[38] R. Sekar, T. Bowen and M. Segal, On Preventing Intrusions by Process Behavior Monitoring, USENIX Intrusion Detection Workshop, 1999.

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

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