The following paper was originally published in the

Proceedings of the 8th USENIX Security Symposium

Washington, D.C., USA, August 23–26, 1999

SYNTHESIZING FAST INTRUSION PREVENTION/DETECTION SYSTEMS FROM HIGH-LEVEL SPECIFICATIONS

R. Sekar and P. Uppuluri

© 1999 by The USENIX Association
All Rights Reserved
For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738
Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer. Permission is granted for noncommercial reproduction of the work for educational or research purposes. This copyright notice must be included in the reproduced paper.
USENIX acknowledges all trademarks herein.

Synthesizing Fast Intrusion Prevention/Detection Systems from High-Level Specifications[*]

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

Abstract

To build *survivable information systems* (i.e., systems that continue to provide their services in spite of coordinated attacks), it is necessary to detect and isolate intrusions *before* they impact system performance or functionality. Previous research in this area has focussed primarily on detecting intrusions after the fact, rather than preventing them in the first place. We have developed a new approach based on specifying intended program behaviors using patterns over sequences of system calls. The patterns can also capture conditions on the values of system-call arguments. At runtime, we intercept the system calls made by processes, compare them against specifications, and disallow (or otherwise modify) those calls that deviate from specifications. Since our approach is capable of modifying a system call before it is delivered to the operating system kernel, it is capable of reacting before any damage-causing system call is executed by a process under attack. We present our specification language and illustrate its use by developing a specification for the ftp server. Observe that in our approach, every system call is intercepted and subject to potentially expensive operations for matching against many patterns that specify normal/abnormal behavior. Thus, minimizing the overheads incurred for pattern-matching is critical for the viability of our approach. We solve this problem by developing a new, low-overhead algorithm for matching runtime behaviors against specifications. A salient feature of our algorithm is that its runtime is almost independent of the number of patterns. In most cases, it uses a constant amount of time per system call intercepted, and uses a constant amount of storage, both independent of either the size or number of patterns. These benefits make our algorithm useful for many other intrusion detection methods that employ pattern-matching. We describe our algorithm, and evaluate its performance through experiments.

1 Introduction

Our increasing reliance on networked information systems to support critical infrastructures (e.g, telecommunication, commerce and banking, power distribution, and transportation) has prompted interest in making the information systems *survivable,* so that they continue to perform their primary functions even in the face of coordinated attacks. In order to build survivable systems, it is necessary to detect and isolate attacks *before* they impact system performance or functionality. Thus a number of recent research efforts have focussed on the problem of *intrusion prevention* [GWTB96, SBS99, CPMWBBGWZ98, MLO97, FBF99].

Many known approaches for intrusion prevention (including the one described in this paper) are based on the following observation about attacks: regardless of the nature of an attack, damage can ultimately be caused only via system calls made by processes running on the attacked host. It is hence possible to prevent damage due to attacks if we can monitor every system call made by every process, and prevent damage-causing calls from being executed. Actions to contain or respond to the attack could be launched at this point, e.g., terminating the process. Some of the central problems in the context of these approaches are:

- efficient interception of system calls
- pattern languages to characterize normal/abnormal system call sequences for any given program/process
- efficient matching of actual system calls made by a process against such patterns

Several techniques for system call interception have already been proposed in [GWTB96, MLO97, GPRA98, FBF99]. This paper addresses the remaining two issues, focusing particularly on an expressive and easy-to-use specification language, and pattern-matching algorithms that are fast enough to be invoked on every system call.

[*] This research is supported in part by Defense Advanced Research Agency's Information Technology Office (DARPA-ITO) under the Information System Survivability program, under contract number F30602-97-C-0244.

1.1 Summary of Results

- In Section 2 we present a new and expressive language for capturing patterns of normal or abnormal behaviors of processes in terms of sequences of system calls and their arguments. As compared to the language described in [SCS98, SBS99], this paper focuses on a core language that we call regular expressions for events (REE). REEs extend regular expressions to model system calls that are characterized by a name as well as argument values. Response actions can be associated with patterns, and these will be launched automatically when our system observes a match for the pattern.
- In Section 3 we illustrate our language with several simple examples. We then detail the process of developing a complete specification for the `ftpd` server. This example, together with similar specifications for `httpd` and `telnetd` forms the basis of our experimental results in later sections.
- REE have much higher expressive power than regular expressions. The presence of variables makes them comparable to attribute grammars in terms of expressive power. The examples in Section 3 illustrate that in practice, REE language provides the capabilities needed to describe program behavior as needed for intrusion detection or prevention.
- In Section 4 we present our runtime model for fast pattern-matching, based on extended finite-state machines (EFSA). Just as REE's extend the power of regular expressions to permit variables, EFSA extend traditional finite-state automata to be able to assign or examine values of a finite set of variables. As in the case of regular expressions, every REE can be matched by a non-deterministic EFSA (NEFA) whose size is linear in the size of REE. But simulation of a NEFA at runtime can be very inefficient. We then examine the properties of deterministic EFSA (DEFA) which can be simulated efficiently. However, we show that the size of deterministic EFSA can be far too large (super-exponential in the size of NEFA) for the approach of using DEFA to be viable.
- Ideally, we would like an approach that achieves a balance between the space explosion implied by DEFA with the runtime inefficiency imposed by NEFA. We propose an algorithm to accomplish this in Section 5. Given an REE of size N, our algorithm produces an EFSA with a worst-case size that is $O(2^N)$. We show that this EFSA is deterministic for a subclass of REE's called REE(0) patterns.
- We present an implementation of our techniques in Section 6 and report its performance. In practice, our algorithm uses much less than exponential amount of space. This means that we can perform intrusion prevention/detection in essentially constant time per system call for most patterns. Our current implementation introduces about 1 to 2% overhead due to the pattern-matching operations.

The applicability of the techniques developed in this paper extends well beyond our system. For instance, many intrusion detection techniques (e.g., [Kumar95, PK92, Ko96]) are formulated using signature patterns that characterize attacks. In contrast with our approach, the performance of matching algorithms developed in these approaches worsens linearly with the number of patterns. By using the techniques developed in this paper, the performance of these approaches can be improved significantly.

2 Approach Overview

We model security-related behaviors in terms of sequences of (security-related) events. In general, events may be internal to a single process, e.g., a function call made by the process; or they may be externally observable, e.g., a message delivered to a machine, or a system call made by a process. We are particularly interested in the system call events, since it is possible to *enforce* secure behaviors if we can intercept and modify system calls. Our specifications characterize normal and/or abnormal behavior of programs as patterns over system call sequences. These specifications are compiled into optimized programs for efficient detection of deviations from the specified behavior. When discrepancies are detected at runtime, defensive actions are initiated to contain or isolate the damage.

Our intrusion detection/prevention system consists of an offline and a runtime component. The offline system generates detection engines from specifications, while the runtime system provides the execution environment for these engines. The input to the offline component is a specification S_P for each program P to be defended[1]. The offline component

[1] This specification may be developed based on known information about P, obtainable from such sources as manual pages, security advisories etc.

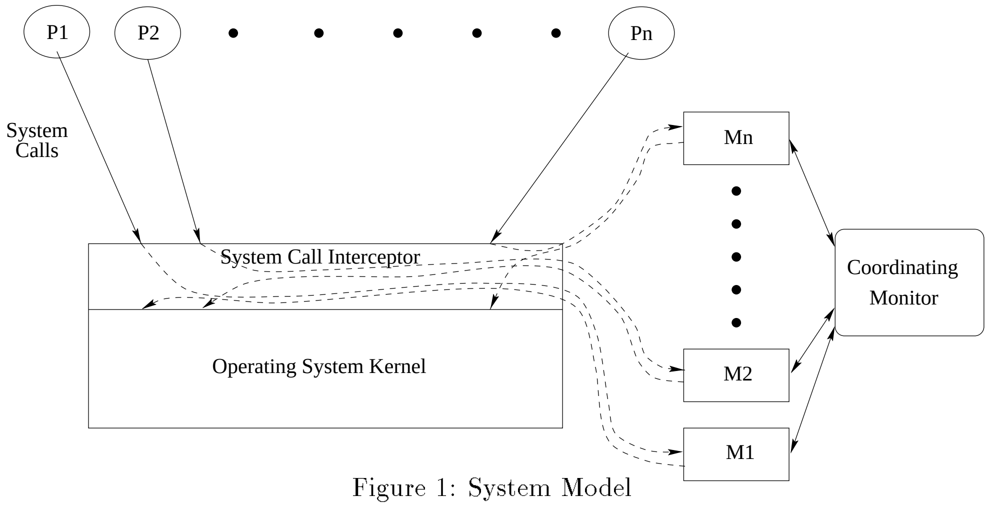

Figure 1: System Model

translates S_P into a C++ class definition C_P. This C++ class simulates an extended finite state machine for matching the patterns in S_P and initiating appropriate responses to intrusion attempts. This code is compiled and then linked with a runtime infrastructure to produce a detection engine.

Figure 1 shows the runtime operation of our system. When P executes as process P_j, it is monitored using the detection engine M_j, which incorporates an instance of C_P. System calls made by P_j are intercepted by the system call interceptor just before, and just after the system call's kernel level functionality is executed, and the system call information is passed to M_j. If M_j matches a pattern, it invokes the action associated in S_P with that pattern. This action would typically utilize the support functions provided by the runtime system to modify the just-initiated system call execution so as to prevent it from causing damage.

The $fork$ system call is given special treatment. When process P_j executes a fork system call, it results in two processes P_j and P_j'. At this point, the detection engine monitoring P_j also clones itself, with one instance monitoring P_j and another monitoring P_j'. If one of these processes, say P_j', executes another program using the *execve* system call, we may switch to monitoring the new program with respect to a new specification as described later.

Frequently, security specifications for a process are completely independent of the actions of other processes. However, some specifications concern interactions among multiple processes. To monitor the behavior of concurrent processes, we use a coordinating monitor that communicates with the monitors for different processes to identify deviant behaviors. We express specification of process interaction using a language construct called an atomic sequence, described later. The coordinating monitor is used to detect violations of atomic sequences. While we describe the atomic sequence construct and discuss a possible implementation to support it, we have not yet implemented it. As such, we do not provide experimental results regarding this construct.

2.1 Specification Language

We provide an overview of the principal components of our specification language here. The main components of a specification include variable declarations and rules. Variables declared in this manner are global, as opposed to *local variables* whose scope is limited to a single rule. Such global variables are referred to as *state variables*, while local variables are called *temporary variables.* State variables are restricted to be of primitive types, i.e., long, integer, boolean, char, and unsigned versions of these. A rule is of the form $pat \rightarrow actions$. Here, *pat* denotes a pattern on sequences of system calls. When a process is monitored using this specification and the process makes a sequence of system calls that matches *pat*, the responsive steps contained in *actions* is initiated.

2.1.1 Event Patterns

Event patterns are built from events using sequencing operators. Events are of the form $EventName(Arg_1, ..., Arg_n)$. For each system call, we identify two events: the entry event which corresponds to the invocation of the system call, and the exit event which corresponds to the return from the system call. The entry event uses the same name as that of the system call, while the exit event is obtained by appending `_exit` to the system call name. $Arg_1, ..., Arg_n$ denote the system call arguments. An *event history* is a sequence of events.

We define a special event *begin* which precedes any system call made by any process, and the event pattern *any* that stands for any event.

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. *Elementary patterns* in our language are of the form $e(x_1, ..., x_n)|cond$, where $cond$ is a boolean-valued expression on the event arguments $x_1, ..., x_n$, any temporary variables that may appear earlier in a pattern, and state variables. 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.

The meaning of event patterns and the sequencing operators is best explained by the following definition of what it means for an event history 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_1H_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 $H_1H_2 \cdots H_n$ 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 will satisfy the pattern only if the history instantiates all occurrences of the variable with the same value. For instance, the pattern $e_1(x); e_2(x)$ will not be satisfied by the event history $e_1(a)e_2(b)$, but will be satisfied by $e_1(a)e_2(a)$.

We relax the definition of the sequencing operators to minimize the need to include both entry and exit events in a pattern. For instance, when two system call entry events occur in sequence, we implicitly insert an exit event in the middle, e.g., $(open|C_1); (close|C_2)$ is treated as equivalent to $(open|C_1); open\_exit; (close|C_2)$. If the first event is an entry event while the second is an exit event for a different system call, then an exit event for the first call and the entry event for the second call are implicitly added, e.g., $(open|C_1); (close\_exit|C_2)$ is treated as equivalent to $(open|C_1); open\_exit; close; (close\_exit|C_2)$.

The value of a temporary variable should be defined before its first use via a *binding* that takes the form $tvar = expr$. Note that binding differs from assignment in that there can be at most one binding to any temporary variable, as subsequent conditions of the form $tvar = expr$ are treated as comparisons.

2.1.2 Response Actions

The response action associated with a rule $p \rightarrow a$ is launched if *a suffix of the event history matches* p. The action component consists of a sequence of statements, each of which can either be an assignment to a state variable or invocation of a support function provided by the runtime system[2].

A typical response is to prevent a system call from executing and/or return a fake return value. This is accomplished using a support function $fail$, which takes an argument that corresponds to the value to which the variable $errno$ should be set to. Sometimes, it is necessary (or just convenient) to switch to a new specification using the support function $switch(f)$, where f is the name of the new specification.

If multiple patterns match at the same time, all of the associated actions are launched. This leads to a problem when some of these actions conflict with each other. It is possible to address such interactions by (a) defining a notion of *conflict* among operations contained in the reaction components of

[2] Knowledge about these support functions are not integrated into the specification language, but are declared in header files that can be included in the specification. Due to space constraints, we do not treat these declarations in detail.

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 this paper — 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 approach yet, instead relying on the specification writer to deal with such conflicts.

3 Example Specifications

We begin this section with a few simple examples and then proceed to give a complete specification for the FTP daemon.

3.1 Simple Examples

To restrict a program from making a set of system calls, we can create a simple pattern that matches any of these disallowed system calls and then invoke an action that causes these calls to fail. For instance, we may wish to prevent a server program such as `fingerd` from executing any program, modify file permissions, create files or directories or initiate network connections. We use the shorthand notation of omitting some of the trailing (and sometimes, all of the) variables of a system call when we are not interested in their values.

$$\begin{aligned}&execve||connect||chmod||chown||\\ &\quad creat||truncate||sendto||mkdir\\ &\rightarrow fail(EINVAL)\end{aligned}$$

We may also wish to restrict the files accessed for reading or writing. For a program such as `fingerd`, we may use the following rule to prevent the program from writing any file, and reading any files other than those mentioned in `admFiles` defined below.

$$\begin{aligned}&admFiles = \{\texttt{"/etc/utmp",}\\ &\quad \texttt{"/etc/passwd", datadir/*\}}\\ &open(f, mode)|(realpath(f) \notin admFiles\\ &\quad || \ (mode \neq \texttt{O\_RDONLY})\\ &\rightarrow fail(EPERM);\end{aligned}$$

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

$$\begin{aligned}&openExit(fd) ::=\\ &\quad open\_exit(f, fl, m, fd)||creat\_exit(f, m, fd)\\ &rwOp(fd) ::= read(fd)||readdir(fd)||write(fd)\\ &openExit(fd); (!rwOp(fd))*; close(fd) \rightarrow \cdots\end{aligned}$$

Although regular expressions are not expressive enough to capture balanced parenthesis, the presence of variables in REE enables us capture the close system call matching an open. This issue of expressive power is discussed again later.

The example below illustrates the use of atomic sequence patterns. A popular attack uses race conditions in setuid programs as follows. Since the setuid process runs with effective user *root*, any open operation by the process will always succeed. If the process is running on behalf of a user, and wishes to open a file with the permissions of this user (i.e., with privileges corresponding to the real userid), it may do so by first using the *access* system call which determines if the real user has access to a file, and if so, it goes ahead and opens the file. The attacker exploits the time window between the access and open system calls as follows. The attacker uses a symbolic link as the name of the file in question, and changes the target of the link between the access and open system calls. To prevent this attack, we ensure that the object referred by the *access* and *open* system calls is accessed atomically:

$$\begin{aligned}&nonatomic(f.target)\ \text{in}\\ &\quad (access(f); (!open(f))*; open(f))\\ &\rightarrow fail(EACCES);\end{aligned}$$

3.2 Case Study: Specification for `ftpd`

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 in Figure 2. 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

```
    /* 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 */
9.  int loggedUser := NOBODYUID
10. int clientIP := 0
11. begin();(!setreuid)*;setreuid(r,e) → loggedUser := e
12. begin();(!getpeername)*;getpeername_exit(fd,sa,l) → clientIP := getIPAddress(sa)
    /* Host authentication phase must precede user authentication. */
13. begin();(!getpeername)*;open(/etc/passwd) → term()
    /* User authentication must precede before userid changed to that of the user. */
14. begin();(!open(/etc/passwd))*;setreuid() → term()
    /* Access limited to admin-related files before user login is completed. */
15. begin();(!setreuid())*;open(f)|(!isExtension(ftpAdmFilePrefixes, f)) → term()
    /* Access limited to certain system calls before user login. */
16. begin();(!setreuid())*;ftpInitBadCall() → term()
    /* Certain system calls are not permitted after user login is completed. */
17. setreuid();any()*;ftpAccessBadCall() → term()
    /* Anonymous user login: must do chroot before setreuid. */
18. begin();(!(setreuid||chroot(FTPHOME)))*;setreuid(r,FTPUSERID) → term()
    /* Userid must be set to that of the logged in user before exec. */
19. begin();(!setuid(loggedUser))*;execve → term()
    /* Resetting userid to 0 is permitted only for executing a small subset of system calls. */
20. 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. */
21. (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. */
22. wrOpen(f)|(f ∈ ftpInvalidPutDirs) → term()
    /* Site-specific: ensure certain users cannot login using ftpd. */
23. begin();(!setreuid)*;setreuid(r,e)|(e ∈ ftpInvalidUsers) → term()
    /* Site-specific: ensure ftp cannot execute arbitrary programs. */
24. execve(f)|(f ∉ ftpValidExecs) → term()
    /* Site-specific: ftp cannot connect to arbitrary hosts or services. */
25. connect(s, sa)|((getIPAddress(sa) != clientIP)&&(getPort(sa) ∉ ftpAccessedSvcs)) → term()
```

Figure 2: A specification for **ftpd**.

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 13 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 14 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 11).
- prior to user authentication, only files beginning with names identified in the set `ftpAdmFilePrefixes` can be accessed (rule 15).
- certain system calls are never used before user login, and certain others are never used after login process (rules 16, 17). 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[3].
- for anonymous login, 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 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.

3.3 Discussion

We make the following observations about the specification for `ftpd`.

- 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`.
- 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 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].
- 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.
- 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.
- 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 Runtime Model

Our runtime model for the detection engine is based on extended finite-state automata (EFSA). EFSA are simply standard finite state automaton that are augmented with the ability to store values in a fixed number of *state variables.* 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, will correspond to the response actions given in our specifications. For a transition to be taken, the associated event must

[3] We note that it may be hard to obtain a complete list of the system calls in either of these cases.

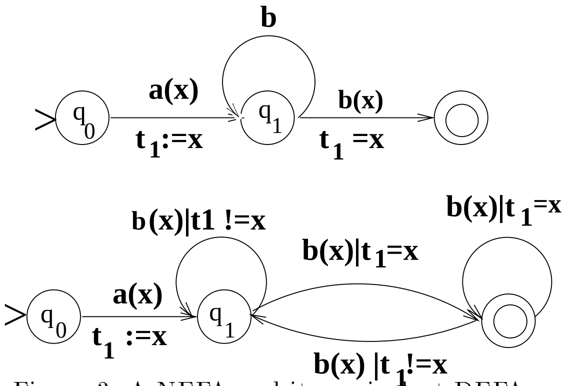

Figure 3: A NEFA and its equivalent DEFA

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 (abbreviated as NEFA) is similar to that of an NFA: a NEFA accepts an event history $e_1 e_2 \cdots e_n$ if there is a sequence of states $s_0, s_1, \ldots s_n$ such that s_0 is the start state, s_n is a final state, and $\forall\ 1 \leq i \leq n$, there exists a transition from s_{i-1} to s_i that can be taken on e_i. A deterministic EFSA (DEFA for short) 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 summarize some of the key properties of REE and EFSA, stated without proof. Let the REE be of size N, and use k variables $X_1, ..., X_k$ where X_i can assume n_i distinct values, for $1 \leq i \leq k$.

- there exists a NEFA of size N corresponding to the REE and it uses state variables $X_1, ..., X_k$
- this NEFA can be simulated with at most $O(N * n_1 * n_2 * \cdots * n_k)$ cost per event at runtime
- every NEFA can be transformed into an equivalent DEFA with the same set of state variables
- in the worst-case, the smallest DEFA for the above REE will have $2^{N * n_1 * n_2 * \cdots * n_k}$ states

The first and third properties are among those that carry over from regular expressions to REE. The second property shows that the overhead for simulating NEFA at runtime can be significantly higher than that for simulating NFA, the latter being just $O(N)$. Similarly, the fourth property shows that the size explosion due to NEFA to DEFA conversion can be significantly worse than NFA to DFA conversion, the latter explosion being limited to $O(2^N)$ in the worst case, and much smaller in practice. In fact, the explosion for NEFA to DEFA construction is unacceptably large, making such conversion impractical. Before we proceed to tackle this problem in the next section, we first develop an algorithm for using NEFA for matching event histories at runtime.

```
procedure $NEFAtrans$(HashTable $curStats$, Event $e$)
    foreach state $S \equiv (c, v_1, ..., v_n) \in curStates$ do
        delete $S$ from $curStates$
        foreach transition $T$ from $c$ to a state $c'$ enabled
                on $e$ with value $v_i$ for state variable $x_i$ do
            Let $v'_1, ..., v'_n$ be the new values of state
                variables after making the assignments in $T$
            foreach $S' \in epsilon((c', v'_1, ..., v'_n))$ that is
                    not already in $curStates$ do
                add $S'$ to $curStates$
        end
    end
end

procedure $NEFAsim$(EventHistory $H$)
    Let $S^i \equiv (c^i, v^i_0, ..., v^i_n)$ denote the initial state
    Let $H$ be of the form $e_1, e_2, ..., e_m$
    Initialize a HashTable $curStates$ to contain $S^i$
    for $1 \leq j \leq m$ do
        $NEFAtrans(curStates, e_j)$
    end
end
```

The algorithm uses a hash table to store all of the possible states in which the NEFA could be in. On receipt of an event, we compute the new states of the NEFA by making a transition from each of the current states. A state of the NEFA is represented by a tuple $(c, v_1, ..., v_n)$ where c denotes the control state of the NEFA and $v_1, ..., v_n$ denote the values of the state variables. The algorithm uses a function *epsilon* to compute the set of states reachable from a state S while only following ϵ-transitions.

We remark that our NEFA model has some similarities with the colored Petrinet model used in [Kumar95]. In particular, the notion of a token in their Petrinet corresponds to our notion of a NEFA state. Their semantics of matching differs from ours in some ways, and as a result, their algorithm for simulating a nondeterministic Petrinet results in cloning of tokens at each step of the simulation algorithm. This can result in unbounded increase in the number of tokens, whereas the number of distinct states of the NEFA is bounded as discussed above.

5 Translation of REE to NEFA

We propose an algorithm that achieves a balance between the space explosion implied by DEFA and the runtime inefficiency imposed by NEFA in this section. The key point is that the explosion due to state variables ($2^{n_1 * n_2 * \cdots * n_k}$) is unacceptable, whereas the explosion due to the size of REE (2^N factor) does not usually pose a problem in practice. Our algorithm is thus geared towards avoiding size explosion due to state variables. This is achieved by permitting nondeterminism in the automata on a subset of transitions where state variables are being assigned new values. This approach results in an EFSA that is deterministic for a subclass of REEs called REE(0) patterns. For non-REE(0) patterns, the EFSA produced may not be deterministic, but our experimental results show that the performance of the algorithm is very good even for such patterns. Regardless of the nature of the patterns, the EFSA size is $O(2^N)$. We present an overview of our algorithm here, leaving out the details due to space constraints. (See [SU99] for a more complete description.)

5.1 Translation Algorithm

Our algorithm has an initial preprocessing phase that includes the following steps:

- strip the leading *begin* from patterns; if a pattern does not start with a *begin*, prefix $any*$, so that the augmented pattern will match an event history iff the original pattern matched a suffix of the history.
- introduce an end-marker # to every pattern, and then combine them into one pattern using $||$.
- express realtime or atomicity using $*$, ; , $||$ and constraints on argument values
- rename event arguments so that ith argument to any event is named \$i. For arguments that need to be remembered for later, we assign their values to state variables using notation $/t_j = \$i$. A pattern that does not use a state variable after this step is known as an REE(0) pattern.
- number the positions in the REE from left to right, indicating the positions as superscripts

Given the patterns $begin(); (a(x,x)||b); (a||b)*; b(x)$ and $begin(); (a(y,3)||c(y)); (a||c)*; c(y)$, the preprocessing steps result in the following pattern:

$$(a^1|\$1 = \$2)/t_1 := \$1; (a^2|b^3)*; b^4/\$1 = t_1; \#^5 \;\; || \\ (a^6|\$2 = 3||c^7)/t_2 := \$1; (a^8||c^9)*; c^{10}|\$1 = t_2; \#^{11}$$

The purpose of numbering the REE positions is similar to that of earlier algorithms for constructing DFA from regular expressions [Aho90, MY60]. Each state S in the NEFA is associated with a subset P_S of positions in the REE as follows: a position $p \in P_S$ iff there is a path from the start state of the NEFA to S that matches the prefix of the REE up to position p. Intuitively, P_S denotes that set of positions in the REE up to which we may have matched an input event history that took the NEFA from its start state to the state S.

The construction of the NEFA for an REE R begins with the start state S_0. The positions P_{S_0} associated with the start state is defined as $first(R)$, where $first$ is the function defined below.

$$\begin{array}{rcll}
first(e|C) & = & \{pos(e|C)\} & \\
first(A;B) & = & first(A) \cup first(B) & \text{if } empty(A) \\
first(A;B) & = & first(A) & \text{otherwise} \\
first(A|B) & = & first(A) \cup first(B) & \\
first(A*) & = & first(A) &
\end{array}$$

Here, the notation $pos(e)$ denotes the position number associated with the event e. $empty(A)$ is true if A matches the empty string. The $first$ of the example REE would be the positions $\{1, 6, 7\}$.

The next step in construction is to identify the distinct events $e_1, ..., e_m$ that occur at one of the positions in P_S. We create m transitions from S to *dummy* states as shown with dotted lines in Figure 4. The set of positions P_{S_i} associated with each of these dummy states is given by:

$$P_{S_i} = \{j \in P_S | e_i \text{ occurs at } j \text{ in } R\}$$

Let C_j be the condition associated with a position $j \in P_{S_i}$. As the next step, we select a $k \in P_{S_i}$ and create two transitions from S_i to dummy states S_{i1} and S_{i2}, the first one to be taken when C_k holds and the second to be taken otherwise. $P_{S_{i1}}$ and $P_{S_{i2}}$ are given by:

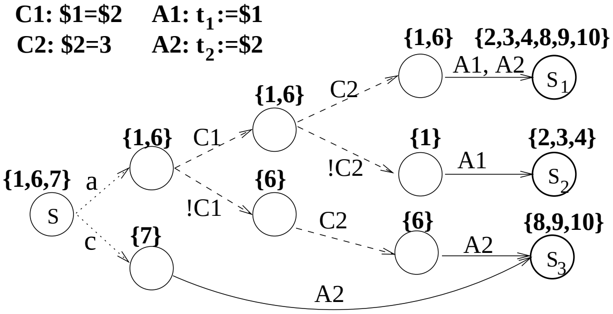

Figure 4: Partially constructed NEFA

$$P_{S_{i1}} = \{j \in P_{S_i} | C_j \wedge C_k \neq false\} \\ P_{S_{i2}} = \{j \in P_{S_i} | C_j \wedge \neg C_k \neq false\}$$

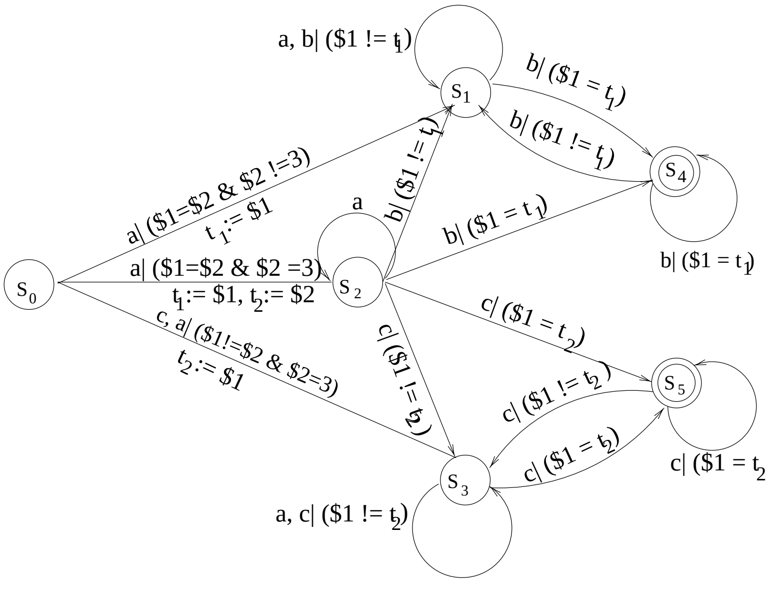

Figure 5: NEFA (also a DEFA) for the example patterns

We note that the correctness of the algorithm is not affected even if we included a j in $P_{S_{i1}}$ such that $C_j \wedge C_k = false$, but the size of NEFA may increase. A similar comment applies to $P_{S_{i2}}$ as well. (The size increase will likely be reversed by an optimizing compiler that may be used to compile the code generated by our NEFA construction algorithm.) This process is repeated at S_{i1} and S_{i2} recursively, while ensuring that each condition is tested at most once. The dashed lines in Figure 4 show these transitions.

When we reach a dummy state S_d where all conditions have been tested, we are ready to create new (non-dummy) states. Let $A_1, ..., A_r$ be the distinct assignment actions among the positions P_{S_d}. Then we create a total of r states $S'_1, ..., S'_r$, with ϵ-transitions from S_d to S'_k labeled with assignment actions A_n, for $1 \leq n \leq r$. $P_{S'_k}$ is determined as follows. Let P_k denote the subset of P_{S_d} where assignment actions A_k appear. Then $P_{S'_k} = \cup_{p \in P_k} follow(p)$, where $follow$ is defined by:

$follow(p) \supseteq first(Q)$, if p is rightmost for
a subexpression $Q*$ in R
$follow(p) \supseteq first(Q)$, if p is rightmost
within Q' where $Q'; Q$ is in R
$follow(p) \supseteq follow(q)$, if p and q are rightmost
in Q and Q', where $Q||Q'$ is in R,
or $Q; Q'$ is in R and $empty(Q')$

A position p' is included in $follow(p)$ only if required by the conditions above. We can reduce the number of ϵ transitions by merging states corresponding to assignments from different patterns. (The set of positions of the merged state is given by the union of these sets for the original states.) We can also avoid creating a new (non-dummy) state S'_n if $P_{S'_n}$ is identical to $P_{S''}$ for another state S'' in the NEFA. Figure 4 shows the partially constructed NEFA after these optimizations.

After the above algorithm, a post-processing phase merges sequences of dummy transitions into a single one, eliminating the dummy states in the process. It also marks any state that includes an end-marker position as a final state for the corresponding pattern. The NEFA constructed by the algorithm for our example is shown in Figure 5.

5.2 Properties of NEFA

Matching REE(0) Patterns As mentioned earlier, $REE(0)$ patterns are those that cannot refer to values of event arguments except immediately after the occurrence of the event. With our variable renaming scheme in place, this means that there would be no need for state variables, and hence no assignments. Note that the only source of nondeterminism in the above algorithm arises because of ϵ-transitions associated with each group of assignments. Therefore, the NEFA generated is a DEFA for $REE(0)$ languages.

Size It is easy to see from our algorithm that after the elimination of dummy states, there can be at most 2^n states, where n is the size of the REE, because each state is associated with a subset of positions in the REE, and there are at most 2^n such subsets. It is interesting to note that although we are dealing with a more complex language than RE, our bound for number of states is the same as the corresponding bound for RE.

To get an idea of the size of the NEFA, one has to consider the states as well as the space required for storing the transitions. If *maxp* denotes the maximum number of positions where the same event occurs in an REE, then we can show that the total space requirements for the states and transitions is bounded by $O(2^{n+maxp})$.

We note that as in the case of the DFA construction algorithm from [Aho90], the worst case space requirements do not reflect the space usage in practice. Often, as in the case of the above example and in the examples studied Section 6.3, we end up with an automaton whose size is much smaller than the exponential upper bound.

6 Implementation and Performance

Our system implementation consists of a compiler and a runtime system.

6.1 Compiler

The front-end of the compiler is responsible for parsing a specification. Its implementation is routine, based on standard compiler construction tools Flex and Bison. Type-checking is performed subsequently, and then we translate the pattern components of our specifications into a NEFA using the algorithm described in the previous section. The reaction components of the rules are attached to those states that accept the pattern corresponding to the respective rules.

The NEFA is then translated into a C++ class *NEFA* as follows. At runtime, the state of the NEFA is maintained in a structure called *FSM*, which stores the control state as well as the state variables. To support nondeterminism, multiple instances of an FSM can be created using a *clone* function that makes identical copies of an FSM. At any point during runtime, zero, one or more copies of the FSM may be active.

The transitions of the NEFA are captured in the C++ code as follows. A system call *sc* is delivered to the NEFA class by invocation of a member function on the NEFA class with the name *sc*. The code for this class sequences through the list of FSM's that are currently active. For each FSM, its state is updated to reflect a transition that it would have made on the event *sc* and its arguments. If multiple transitions are possible, suitable number of copies of the FSM are made and each copy follows one of the transitions.

6.2 Runtime System

The output code produced by the compiler is linked with a runtime support system that provides the infrastructure for intercepting and delivering system calls to the detection engine. Events are delivered by invoking the corresponding member functions on the NEFA class described above. The runtime system also provides the functions needed by the detection engine to alter the behavior of system calls. With regard to data access, it provides support functions to access system call argument values.

The runtime system has been implemented by modifying the system call interface within the operating system. We do not provide a detailed explanation of the runtime system here, as our focus in this paper is on efficient runtime matching operations. Our results indicate that the overhead for interception is more or less constant per system call, and adds about 30% overhead to the cost of a system call [Bow99]. However, since system calls account for only a small fraction of the CPU time used by an application, the increase in CPU time (counted as the sum of user and system time on UNIX) is in the range of 2% to 30%. Most applications, especially those that are CPU-intensive, impose overheads at the bottom of this range, while I/O intensive applications such as `tar` and `ftp` lead to higher overheads. [FBF99], which employs an interception mechanism similar to ours, reports overheads in the range of 2% to 4% for a different set of applications, namely, `gcc` and `httpd`.

We remark that the overheads for system call interception are nontrivial. In fact, our fast matching algorithm reduces the checking time to the point where system call interception accounts for the the dominant portion of the total overhead imposed by our approach. In particular, this implies that there is no penalty to be paid due to the use of sophisticated pattern-matching based approach such as ours over a method that uses simpler rules that are triggered on individual system calls.

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 6: Overhead due to system call pattern matching. Results taken on 350MHz Pentium II Linux PC with 128MB memory and 8GB EIDE disk.

6.3 Performance Results

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**. Our experiments were designed to evaluate the runtime overhead for monitoring (both in terms of time and space) and the size of the pattern-matching automata. In measuring the runtime overhead, we omitted the cost of intercepting system calls since our primary interest is in evaluating the performance of the pattern-matching algorithms. Moreover, the overhead for interception remains essentially a constant independent of the size of the patterns or length of the execution trace.

6.3.1 Timing Results

The best indicator of runtime overheads is the fact that only two of all the patterns in these three specifications were non-REE(0). This implies that the runtime matching effort consists of performing a transition on a deterministic automaton, which is about as expensive as the cost of the tests involved in the applicable transitions. As compared to the cost of a typical system call, this is indeed very low. The non-REE(0) patterns did not contribute much to the overhead either, as they led to very few cases where cloning was required. Specifically, the number of clones active at any point at runtime was less than five, with each FSM requiring several bytes of storage. Thus the runtime storage requirements were very low. Simulating such a small number of FSMs at runtime is also very fast.

Figure 6 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 7 shows the overhead for matching each system call, averaged across the three programs. The graph indicates a slight increase with increase in number of system calls, which is to be expected due to the presence of non-REE(0) rules. However, the rate of increase is extremely small, at 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 6. To obtain the results in the table and the figure, we exercised the three programs with a random combinations of valid commands of different lengths.

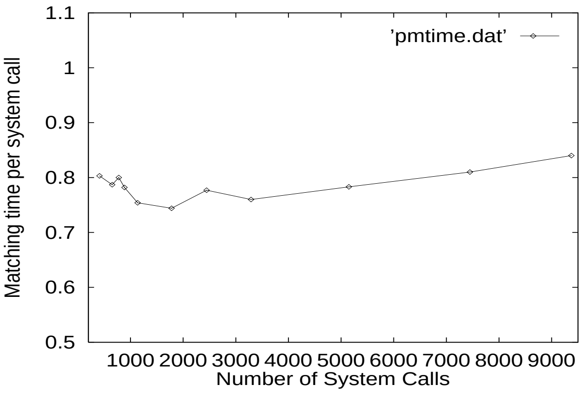

Figure 7: Matching time per system call (in microseconds).

6.3.2 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

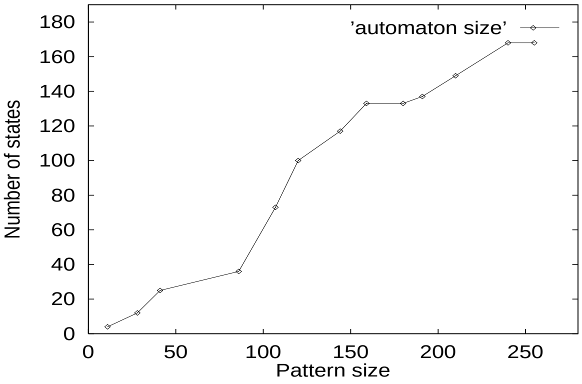

Figure 8: Increase in automaton size with specification size.

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 8.

Although the worst-case 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. Although it is not readily apparent from the graph, the rate of increase in number of states tended to decrease after a while in our examples. The reason for this is that existing patterns had already created many states that could already represent some of the stages of matching the new pattern, and thus only fewer additional states were required. However, we must exercise caution in generalizing from these three programs, as the relationship between pattern size and automaton size can vary greatly from one set of patterns to another.

7 Related Work

7.1 Intrusion Detection

Techniques for prevention of intrusions draw on previous research on (post-attack) intrusion detection. Intrusion detection techniques can be broadly divided into *misuse* detection [PK92, Kumar95], *anomaly* detection [ALJTV95, FHS97, GSS99], and *specification-based* detection [Ko96, SBS99].

Among misuse-based approaches, a state-transition diagram based approach is used in [PK92] to capture signatures of intrusions. [Kumar95] 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 [Kumar95] 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 [FHS97]. Recently, these results have been improved by [GSS99] using a neural network based approach that produces very accurate anomaly detectors. All 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 [Ko96]. 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.

7.2 Preventive Approaches

Some of the preventive approaches are based on protecting systems against underlying software errors that are exploited by attackers. For instance, malicious code detection techniques [BD96, GM96, LLO95] employ program analysis techniques to detect security-related errors in the source code. Similarly, [CPMWBBGWZ98] developed compilation techniques that add additional checks into the code generated to identify (and prevent) attacks that exploit buffer overflows to overwrite the return addresses stored on a process stack.

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 [GWTB96] 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. Their language is tailored to restrict individual system calls without any regard for the context in which they appear. This approach is well-suited for fine-grained access control and sandboxing. The kernel hypervisor [MLO97] 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 [GPRA98]. Their approach is geared for a broad range of applications aimed at augmenting software functionality in areas such as security and fault-tolerance. [FBF99] 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 are not addressed.

7.3 Other Work

There has been a significant amount of earlier work in regular languages and pattern-matching. In particular, our algorithm for NEFA construction is based in part on some of the ideas developed in [Aho90, MY60] for direct construction of DFA from regular expressions. The language design has been influenced by previous pattern-matching based languages such as Lex and Awk [AWK88].

Regular languages and ω-regular languages, together with their automata-equivalents, have been used widely in formal specification and verification of concurrent systems [Kurshan94]. In the related area of program analysis, [OO90] develops an approach for expressing sequencing constraints as regular expressions, and compile-time checking of these constraints using dataflow analysis. [Schneider98] proposes to use Buchi automata for monitoring security properties of programs. The main difference between these works and ours is that they operate on regular or ω-regular languages, while our language is REE. Another difference is that in the context of verification, we are often interested in properties of infinite sequences, while our interest is in behaviors that manifest themselves in finite sequences.

Describing properties of event sequences is an important problem in building, prototyping, debugging and monitoring distributed systems. Languages such as CSP [Hoare78], LOTOS [BB89] and Esterel [BCG87] support very simple patterns, permitting no sequencing or closure operators. The task sequencing language developed in [LHMBH87] and its successor Rapide [LV95] support an expressive pattern language that is significantly more expressive than ours. Moreover, they support a partially ordered set model of event histories, whereas our current model is a linear sequence. However, it is not clear whether this language is amenable to efficient pattern matching at runtime, as they do not address the problem of automata-based techniques for efficient matching of these patterns.

8 Conclusions

In this paper we presented an approach for building survivable systems. Our approach is based on monitoring system calls made by processes (running on the system to be protected), comparing them against patterns characterizing normal (or abnormal) system call sequences, and initiating appropriate responses when (potentially) damaging system calls are made. 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 an algorithm that compiles patterns in our specification language into an extended finite state automaton (EFSA). Our implementation results demonstrate that detection can be performed very fast (1 to 2% overhead to program execution time) using our approach. 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 make our algorithm attractive for any other approach that performs intrusion detection by some form of pattern-matching.

We showed that for a class of patterns (specifically, REE(0) patterns) the automata constructed by our algorithm are deterministic. A more precise characterization is desirable, since in practice, our algorithm builds deterministic automata for a much larger class of patterns. Such a characterization may also enable us to restrict the specification language so that patterns that can lead to very large overheads (associated with simulating a large number of NEFA instances at runtime) can be avoided. Another area of future research is the application of techniques presented in this paper to the larger problems of distributed system monitoring and software debugging.

References

[Aho90] A.V. Aho, Algorithms for Finding Patterns in Strings, Handbook of Theoretical Computer Science Vol A, Elsevier Science Publishers B.V., 1990.

[AWK88] A.V. Aho, B.W. Kernighan, and P.J. Weinberger, The AWK Programming Language, Addison-Wesley, Reading, MA, 1988.

[ALJTV95] 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.

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

[BD96] M. Bishop, M. Dilger, Checking for Race Conditions in File Access. Computing Systems 9(2), 1996, pp. 131-152.

[BB89] T. Bolognesi and E. Brinksma, Introduction to the ISO Specification Language LOTOS, The Formal Description Technique LOTOS. Amsterdam: North-Holland, 1989.

[Bow99] T. Bowen et al, Operating System Support for Application-Specific Security, under review for Symposium on Operating Systems Principles, 1999.

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

[CPMWBBGWZ98] 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.

[FHS97] S. Forrest, S. Hofmeyr and A. Somayaji, Computer Immunology, Comm. of ACM 40(10), 1997.

[FHRS90] K. Fox, R. Henning, J. Reed and R. Simonian, A Neural Network Approach Towards Intrusion Detection, National Computer Security Conference, 1990.

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

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

[GM96] 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.

[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.

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

[Hoare78] C. Hoare, Communicating Sequential Processes, Comm. of the ACM, 21(8), 1978.

[Jones93] M. Jones, Interposition Agents: Transparently Interposing User Code at the System Interface, 14th ACM Symposium on Operating Systems Principles, December 1993

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

[Kumar95] S.Kumar, Classification and Detection of Computer Intrusions, Ph.D Dissertation, Department of Computer Science, Purdue University, 1995.

[Kurshan94] R. Kurshan, Computer Aided Verification of Coordinating Processes: The Automata-Theoretic Approach, Princeton University Press, Princeton, NJ, 1994.

[LLO95] R.W. Lo, K.N. Levitt, R.A. Olsson, MCF: a Malicious Code Filter, Computers and Security, Vol.14, No.6, 1995.

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

[LHMBH87] D. Luckham, D. Helmbold, S. Meldal, D. Bryan, and M. Haberler, Task Sequencing Language for Specifying Distributed Ada Systems: TSL-1, PARLE: Conf. on Parallel Architectures and Languages, LNCS 259-2, 1987.

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

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

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

[OO90] K. Olender and L. Osterweil, Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation, IEEE Transactions on Software Engineering, 16(3), 1990.

[PK92] P. Porras and R. Kemmerer, Penetration State Transition Analysis:A Rule based Intrusion Detection Approach, Eighth Annual Computer Security Applications Conference, 1992.

[Schneider98] F. Schneider, Enforceable Security Policies, TR 98-1664, Department of Computer Science, Cornell University, Ithaca, NY, 1998.

[SCS98] R. Sekar, Y. Cai and M. Segal, A Specification-Based Approach for Building Survivable Systems, NISSC, October 1998.

[SBS99] R. Sekar, T. Bowen and M. Segal, On Preventing Intrusions by Process Behavior Monitoring, USENIX Intrusion Detection Workshop, 1999.

[SU99] R. Sekar and P. Uppuluri, Synthesizing Fast Intrusion Prevention/Detection Systems from High-Level Specifications, Technical Report 99-03, Department of Computer Science, Iowa State University, Ames, IA 50014.