A Domain-specific Language for Intrusion Detection*†

Diomidis Spinellis‡ and Dimitris Gritzalis§

May 30, 2000

Abstract

We describe the use of a domain-specific language (DSL) for expressing critical design values and constraints in an intrusion detection application. Through the use of this specialised language information that is critical to the correct operation of the software can be expressed in a form that can be easily drafted, verified, and maintained by domain experts (security officers) thus minimising the risk inherent from the mediation of software engineers. Our application, panoptis is a DSL-based low-cost, easy-to-use intrusion detection system using the process accounting records kept by most Unix systems. A set of databases contain resource usage profiles for processes, terminals, users, and time intervals. Panoptis monitors new process data against the recorded profiles and reports on entities diverging from the established resource usage envelopes implying possible data security threats.

Keywords

Domain-specific languages, security monitoring, intrusion detection, Unix process accounting.

1 Introduction

Panoptis[1], is an intrusion detection system based on the process accounting records produced by all widely-used versions of Unix. These records, originally intended for producing billing information, can be used to detect anomalous situations and alert the security administrators. The voluminous nature of the process accounting records prohibits manual inspection; Panoptis keeps detailed databases keyed by users, terminals, processes, and time intervals containing typical usage profiles. A novel aspect of Panoptis is the use of a domain-specific language (DSL) for the specification of the items that will be checked.

Panoptis detects and reports all entities that execute outside the defined profile envelopes and automatically updates the databases to reduce the administrative burden and reporting volume. On a system that has an established pattern of use entities outside the normal usage envelopes are likely to be associated with information security breaches. Data threats that can be detected in this way include wiretapping, browsing, leakage, tampering, and masquerading [Den83]. An example of Panoptis's output can be seen in Figure 1.

The heuristic and quantitative nature of our approach extends the range of data security threats that can be detected beyond the closed computer system environment into the organisational environment that hosts Panoptis. As an example Panoptis could detect an employee transferring inordinately large amounts of data to a computer outside the organisation even if that employee had proper system authorisations to perform such transfers. Although Panoptis was implemented under the Unix operating system the approach and techniques we used are applicable to other operating systems keeping process accounting records. As an example, the Windows NT audit event log can be used in a similar way.

* *Proceedinds of the 1st ACM Workshop on Intrusion Detection Systems* ACM, November 2000.

‡ University of the Aegean, Department of Information and Communication Systems, Karlovasi, Greece

§ Athens University of Economics and Business, Department of Informatics, Athens, Greece

[1] Argos-Panoptis — the one who can see everything — is a Greek mythology canine creature whose body is covered with eyes. Even when Panoptis is sleeping half of its eyes remain open. For this it was given the task of guarding Io one of Zeus' lovers.

```
Database Users*Commands, key [root/grep]:
        New maximum user time (2.52 / 2.08)
        New maximum disk block input/output (10.94 / 8)
        New maximum clock time (14.55 / 13.63)
        New maximum character input/output (18427 / 13097)
Command: grep  Terminal: tty01  User: root
Executed from: 13/06/95 12:30:55 to: 13/06/95 12:31:09 (14.55 seconds)
spending 1.73 seconds in kernel space and 2.52 seconds in user space
(4.25 total) and using the CPU for 29% of the time.
Character I/O: 18427 characters (average I/O: 4335.74 characters/CPU second)
Disk I/O: 10.94 K (average I/O: 2.57 K/CPU second)
Memory accounted: 0.88 K (average size: 0.10 K)
```

Figure 1: An example of an Panoptis report.

1.1 Domain-specific languages

A domain-specific language [Ram97] is a programming language tailored specifically for an application domain: rather than being general purpose it captures precisely the domain's semantics. Examples of DSLs include *lex* and *yacc* [JL87] used for program lexical analysis and parsing, HTML [BLC95] used for document mark-up, and VHDL used for electronic hardware descriptions. Domain-specific languages allow the concise description of an application's logic reducing the semantic distance between the problem and the program [BBH+94, SG97].

DSLs are, by definition, special purpose languages. Any system architecture encompassing one or more DSLs is typically structured as a confederation of modules; some implemented in one of the DSLs and the rest implemented using a general purpose programming language. As a design choice for implementing security software DSLs present two distinct advantages over a "hard-coded" program logic:

Concrete Expression of Security Policies Security policies are not coded into the system or stored in an arcane file format; they are captured in a concrete human-readable form. Policies expressed in the DSL can be scrutinised, split, combined, shared, published, put under release control, printed, commented, and even be automatically generated by other applications.

Direct Involvement of the Security Officer The DSL expression style can often be designed so as to match the format typically used by the security officer. This results in keeping the experts in a very tight software lifecycle loop where they can directly specify, implement, verify, and validate, without the need of coding intermediaries. Even if the DSL is not high-level enough to be used as a specification language by the security officer, it may still be possible to involve the security officer in code walkthroughts far more productive than those over code expressed in a general purpose language.

1.2 Unix Process Accounting Records

Most modern versions of Unix provide the capability of *process accounting*[LMKQ88, pp. 62–63]. The operating system kernel creates a file containing an accounting record for every process that terminates. Each record contains for a given process the following vector:

- U_i, G_i: its user and group identification,
- C_t: its controlling terminal,
- T_b: the time the process began,
- T_r, T_s, T_u: the real, system and user times used by the process,
- M_t: its total memory usage,
- C_t, D_t: its total character and disk input/output,
- N: the name of the command that started the process, and
- E, F: its exit status and associated flags.

Based on the above data the following quantities can be derived for every terminated process:

- T_l: the local time of the day the process started found by converting the time the process began to local time,

- T_t: the total CPU time consumed by the process as the sum of the system and user times ($T_s + T_u$),
- M_a: average memory usage as the memory accounted divided by the CPU time (M_t/T_t),
- C_a, D_a: average character, and disk input/output as the respective quantity divided by the CPU time (C_t/T_t, D_t/T_t),
- H: CPU "hog" factor as the process's CPU time divided by the actual time it executed (T_t/T_r), and
- the number of times the process run in a specific time interval.

A number of programs are typically provided for processing the accounting records, but these are geared towards providing billing and system performance tuning information. In the following sections we will describe how a domain-specific language can be used to specify the way parts of the process accounting data space can be grouped and checked for intrusion detection purposes.

2 Intrusion Detection Data Space

Panoptis monitors the system processes in three independent dimensions:

1. **The accounting data** This data corresponds to a specific process, terminal, and user and consists of the values described in the previous section. It can be monitored for being above or below specific limits which are based on the system's historical data collected by Panoptis.

2. **The monitored entity** A monitored entity can be one of the following:

   - U: a user,
   - T: a terminal,
   - P: a process,
   - (U, P): a process executed by a specific user, and
   - (U, T): a user working on a specific terminal.

   An abnormal behaviour which could signify a security breach can be associated with any of the above entities. For example

   - a user may run programs at an unusual time ($K_0 \leq T_l(U) \leq K_1$),
   - a process may consume an inordinate amount of CPU time ($T_t(P) \geq K$),
   - a terminal may be exhibit abnormal input/output behaviour (e.g. $C_t(T) \geq K$),
   - a user may execute an uncommon command, or
   - a user may work from an unusual terminal.

3. **The monitoring time interval** Time intervals are defined by the system administrator. Typical intervals that provide useful data are:

   A fixed period As described in section 4, we found that storing data for twenty minute intervals, a day, and a week captures enough information about the system behaviour to cover a large number of possible security breach attempts. The twenty minute interval is useful for quickly detecting a large number of invocations of an important program such the password changing command, while the day and week databases can be run with a larger set of checks to detect finer changes in the system's behaviour indicating attempted security breaches.

   A specific period Panoptis can store separate data for every day and every hour (e.g. Mon, Tue, ... and 1200h, 1300h, ...) to capture behaviour that is occurring in non-standard days or times. An example of a security breach that can be detected using this method is the execution of an application used by personnel working nine to five late at night or over the weekend. We found it more convenient to group the specific period time interval databases into groups of larger granularity such as workdays/weekend.

   Continuous monitoring Finally, Panoptis can be run in a mode whereby the accounting log is continuously monitored and all records that are appended to it are checked against the specified databases. This execution mode provides immediate notification of possible security problems. A system administrator can run Panoptis in this mode with its output redirected to a hardcopy terminal to create a log that can not be erased even when the security of the system is compromised.

The three dimensions described above can be tailored via a configuration file to a setup that is suitable for the system being monitored. In addition, terminal and user names can be grouped in logical sets to avoid the generation of redundant messages. As an example all users of the same application or toolset can be defined as one group, because we expect them to have similar usage profiles. One profile will be defined and used for all of them, but any leap outside the profile will be directly attributable to a specific user. Similarly, a pool of terminals that are interchangeably used in a room should be grouped together, because they too will have statistically similar usage profiles.

3 The Panoptis Domain-specific Language

Panoptis consists of a single program that reads accounting records and updates profile databases optionally reporting cases that fall outside the existing profiles. Its arguments are a DSL-based configuration file that directs the program operation, the database to update, the interval to operate upon, and an optional list of process accounting files (the system accounting file `/var/adm/pacct` is the default record source).

Panoptis is configured by a domain-specific language. The language supports bindings over the following distinct databases:

tty Terminals.

uid Users.

uidtty Users logged in on a specific terminal.

comm Commands.

uidcomm Users executing a specific command.

The basename used for storing each one of the above databases is specified as a parameter in the *panoptis* invocation. As a result, different databases can be used to store process accounting history for different hosts, time intervals, or monitoring configurations.

For every process accounting record the following attributes can be checked:

maxaxsig Signal exit status.

maxhog Maximum CPU hog factor (CPU time over elapsed time).

maxmem Maximum memory usage.

maxavrw Maximum average disk block input/output.

maxstime Maximum system time.

minbmin Minimum daily start time (start time whithin the 24 hour interval).

maxutime Maximum user time.

maxbmin Maximum daily start time.

maxasu Superuser status.

maxcount Maximum number of times a given record has appeared in the database.

maxrw Maximum disk block input/output.

maxacore Core dump flag.

maxavio Maximum average character input/output.

maxafork Fork status.

maxetime Maximum clock time.

maxavmem Maximum average memory usage.

maxio Maximum character input/output.

Panoptis will report process accounting records whose attributes fall above (below) the values already recorded in a given database.

The *panoptis* monitoring options are also set in the DSL configuration file. The file contains the following elements:

Assignments Specific variables can be assigned values to control the *panoptis* behaviour.

Monitoring specifications These are given using the relation dbcheck(*database*, *attribute* ...) and specify that the given attributes should be monitored in a given database. The special attribute *ALL* can be used to specify that all attributes shall be monitored.

User maps These are given using the relation usermap(*abstract user*, *username* ...) and specify that all concrete users specified will be mapped to the given abstract user. This relation can be used to group users into specific monitoring groups (e.g. power users, administrators, typists).

Terminal maps These are given using the relation termmap(*abstract terminal*, *terminal name* ...) and specify that all concrete terminals specified will be mapped to the given abstract terminal. This relation can be used to group terminals into specific monitoring groups (e.g. network terminals, printers, data entry, etc.).

In addition, the following variables can be specified in a configuration file:

```
#
# Configuration file for host pooh
#
# $Id: paper.tex 1.6 2000/05/30 12:26:58 dds Exp $
#

HZ = 100                # "Floating point" value divisor
bigend = FALSE          # Set to TRUE for big endian (e.g. Sun), FALSE for
                        # little endian (e.g. VAX, Intel x86)
map = TRUE              # Set to TRUE to map uid/tty numbers to names
EPSILON = 0.001         # New maxima difference threshold
report = TRUE           # Set to TRUE to report new/updated entries
unlink = FALSE          # Set to TRUE to start fresh

# Reporting procedure
output = '| /usr/bin/tee /dev/console | /bin/mail root'

# Databases and parameters to check
dbcheck(tty, minbmin, maxbmin, maxio, maxcount)    # Terminals
dbcheck(comm, ALL)                                 # Commands
dbcheck(uid, ALL)                                  # Users
dbcheck(uidtty, maxcount)                          # Users on a terminal
dbcheck(uidcomm, minbmin, maxbmin, maxutime,       # Users of a command
        maxstime, maxmem, maxrw, maxcount, maxasu)

# Map users and terminals into groups
usermap(caduser, john, marry, jill)
usermap(admin, root, bin, uucp, mail, news)

termmap(room202, tty31, tty32, tty33, tty34, tty35)
termmap(ptys, ttyp01, ttyp02, ttyp03, ttyp04, ttyp05, ttyp06)
```

Figure 2: Sample configuration file.

report Boolean variable. Set to TRUE to report new/updated entries.

countreport Boolean variable. Set to TRUE to report time the command was started.

unlink Boolean variable. Set to TRUE to clear existing database entries.

map Boolean variable. Set to TRUE to map uid/tty numbers to names based on the mapping of the system where *panoptis* is run.

HZ Numeric variable. The divisor used by the system to store "floating point" values.

EPSILON Numeric variable. Maximum difference threshold. When this threshold is exceeded *panoptis* will report the specific command.

acct String variable. Set to specify the system source of the accounting records. The following values are currently supported:

- **'SVR3'** SunOS 4.X and XENIX,
- **'Linux'** e.g. Linux 2.2,
- **'SVR4'** POSIX, XOPEN, e.g. SunOS 5.6,
- **'fBSD'** Free BSD e.g. Free BSD 3.4.

bigend Boolean variable. Set to TRUE for big endian (e.g. Sun), FALSE for little endian (e.g. VAX, x86) accounting records.

output String variable. Set to specify how *panoptis* results will be output. The *Perl* syntax used for opening files can be used.

A sample configuration file is reproduced in Figure 2. Two variables (`HZ` and `bigend`) define the machine's hardware characteristics. These — in conjunction with the option `map` which specifies whether the local system user and terminal names should be used for reporting — made it possible for us to run Panoptis on our system cross-checking the accounting files of other systems. A possible setup based on this capability could be a centralised security server monitoring a large number of remote systems. The `report` and `unlink` settings are used for creating initial profiles. Setting `unlink` will create a fresh set of profile data. In that case `report` could be disabled while historical data is collected and stored in the database. The `output` parameter specifies the filename or process to receive Panoptis's output. In this example all reports are printed on the system console and a copy is mailed to the system administrator account.

The next section of the configuration file specifies for each of the databases outlined in section 2 the parameters — as described in section 1.2 — to be checked. These specifications are used to customise the profile databases for storing only relevant profile data. In the example we provide terminals (`tty`) are monitored for use outside the normal hours in order to detect physical or network security breaches and the number of characters transfered in order to detect attempts to transfer data outside the system. Commands (`comm`) and users (`uid`) have all their parameters monitored as these should quickly settle to an established pattern minimising false alarms. A subsequent divergence of any of the parameters is likely to be interesting. The database containing the users of a specific terminal is only monitored for the number of commands run from that terminal in order to catch intruders. Finally, the database containing data for every command a user executes (`uidcomm`) is monitored for the time that process is run, its use of CPU time, memory, and disk I/O, the number of times it was executed, and whether it was executed with superuser privileges. Divergence of these parameters can pinpoint Trojan horses, viruses, encryption crackers, and data browsers.

The last section of the configuration file contains the grouping tuples used to specify logical sets of terminals and users. In our example the users of the CAD application form one group (`caduser`) and the administrative accounts another (`admin`). All other system users are stored and checked as individuals. Records in the databases that are keyed by a user (`uid, uidtty, uidcomm`) will be reflect the behaviour of the whole group instead of a specific user. Similarly, some terminals that are shared in one room are checked as one group. Pseudo-terminals (*ptys*) which are often used for network connections are also grouped together as they are assigned to incoming connections in a random way.

Panoptis is typically installed as a program to be executed by the system's command scheduler *crontab*. Additionally, Panoptis can be run at system startup as a background task to continuously monitor the accounting files. A sample scheduling file for Panoptis that we used on our system is reproduced in Figure 3. In this example a few quick checks are run every twenty minutes (on the fifth, 25th, and 45th minute of the hour) against the profiles stored in the `pooh.20min` database. Every hour a more complete check is run. Its profiles are split into two databases, one stores the working hour (8am to 6pm) profiles (`pooh.workhour`) and one the night-hour (7pm to 7am) profiles (`pooh.late`). Daily checks are run every night at 4:50am. Again, the profile databases are split between workdays and weekends. Finally, the complete set of accounting files is checked using a full configuration every Sunday at 2:20am.

4 Evaluation

A monitoring system can fail in two different ways:

Type I error Failing to report an important event (since, false negative).

Type II error Reporting a large number of unimportant events letting important ones passing unnoticed (false positive, noise).

In addition, a security monitoring system can fail either because an intruder uses an attack mode not anticipated or covered by its design (a system *limitation*), or because the intruder intentionally tries to get around it (a system *weakness*).

Panoptis's heuristic nature will result in both silence and noise. Noise is gradually eliminated as more and more cases are added to the profile data. Silence can result either from security breaches that are outside the system's domain, or from an intruder's deliberate exploitation of the system's weaknesses. As the system is based on process accounting records, a number of other important information that could lead to the detection of security problems is not examined. Examples of other entities that could be monitored and included in the profile data include system calls made by a process, network connections, and patterns of file access. Monitoring these entities would require operating system kernel modifications [BK88]; we decided against them in order to keep the system portable and easy to install.

An intruder knowing Panoptis's architecture and configuration could also foil the system by:

- generating legitimate "noise" in order to hide a culprit process,
- an attack based on a non-terminating process (such as system daemons) which are not normally logged,

```
#
# Panoptis crontab file for host pooh
#
# The format of this file is:
# Hour Minute Day-of-month Month Day-of-week Command
*    5,25,45 * * *   panoptis pooh-quick.cfg pooh.20min    20m
8-18 05      * * *   panoptis pooh-hour.cfg  pooh.workhour 1h
19-7 05      * * *   panoptis pooh-hour.cfg  pooh.late     1h
4    50      * * 1-5 panoptis pooh-day.cfg   pooh.workday  24h
4    50      * * 6,0 panoptis pooh-day.cfg   pooh.weekend  24h
2    20      * * 0   panoptis pooh-full.cfg  pooh.weekly   7d \
                           /usr/adm/pacct? /usr/adm/pacct
```

Figure 3: Sample scheduling file.

- using an interpreter such as Perl [WS90] to access system resources without invoking external processes,
- changing the name of the offending command to a benign name,
- gradually and legitimately changing the usage profile of an entity avoiding the suspicion caused by a sudden change,
- filling the disk where administrative data is kept in order to disable process accounting, or
- exploiting Panoptis's relatively large time window between the occurrence of a suspicious event and its detection.

On the plus side, Panoptis's open ended nature can result in the detection of security problems unanticipated during its design and deployment. Some of the attacks described can be defended by careful installation and configuration. Countermeasures include keeping the accounting records in a filesystem that has no public writable directories (by default process accounting records and the temporary file directory reside on the same filesystem), and the protection of the configuration file and the reports from unauthorised reading to make the planning of an undetected attack difficult.

We have run Panoptis on the accounting records of our site, an academic site X-terminal server, a dialup/WWW server and a C/database development machine. After some time of tuning and profile collection Panoptis's reports are reduced to a steady trickle reflecting the users change of interests or mode of work and the introduction of new programs on the system. Although Panoptis has not yet caught any security violations the results we have so far obtained are encouraging. In some cases Panoptis has helped us identify sources of system performance degradation or potential security problems. Furthermore, in a Gedankenexperiment we performed based on five security breaches described in [BKS90] we found that four of them could have been caught by Panoptis.

5 Related Work

In an early study on real-time intrusion detection [And80], it was suggested that an intruder could be detectedby monitoring certain system-wide parameters (i.e. CPU use, memory use, disk activity, keystroke dynamics, etc.), and compare them with what had been historically established as normal or expected for that facility. It was, also, suggested to profile the normal behavior of programs, files, and other objects. This is often called a statistical anomaly detection approach. Until this study, the relevant work focused on developing procedures and algorithms for automating the offline security analysis of audit trails.

On the basis of the above, SRI scientists developed IDES (Intrusion Detection Expert System) [L+92] and Next-generation IDES [A+95]. IDES is a system that continuously monitors user behavior and detects suspicious behavior as it occurs. IDES takes the approach that intrusions can be detected by flagging departures from historically established norms of behavior for individual users. To support the idea, various intrusion detection measures are profiled for each user and statistical processing of them is carried out by the monitoring facility.

Intruders often use known paths to attack a system (e.g. programmed password attacks, access to privileged files, exploitation of known vulnerabilities, etc.). With a model-based reasoning, specific models of defending to prescribed attacks can be developed [GL91]. Other approaches are either defining acceptable, as opposed to intrusive, behavior [Kar87], or — on earlier stages of technology — are based on the introduction of trap doors for intruders (i.e. “bogus” user accounts with “magic” passwords, etc.) [Lin75]. None of them is sufficient alone, since it addresses a specific type of threats.

Several studies have demonstrated that the use of specialized (security-focused) audit trails is needed for security purposes. In addition to the raw audit data itself, additional data could prove to be useful or necessary for intrusion detection: external facts (e.g. changes in user job description), supporting facts (e.g. file attributes), and profiles of expected behavior (e.g. time schedules). It seems to be a fact, that effective intrusion detection will not come into widespread use until good security auditing mechanisms are in place [Lun93].

The appropriate level of auditing is really important. It has been suggested [Kuh86, Pic87] that the audit should be performed at the lowest possible level (e.g. monitoring system service calls), because in this case to circumvent auditing is harder.

The more recent studies on intrusion detection focus more on the topology of the modern information systems environment. As a result, network intrusion detection systems have been developed [S+99, VK98]. The cornerstone of these systems is also a domain-specific language that enables concise specification of network packet contents under normal/expected and/or attack conditions. These approaches claim to have easy-to-develop intrusion specifications, to carry out high-speed and large-volume monitoring, to be robust and extensible, and to use a comprehensive evaluation framework.

6 Conclusions and Further Work

The use of a domain-specific language can make process accounting data ammenable to intrusion detection. Panoptis first expands the accounting data space by deriving new quantities from the existing records and scattering the results into the three dimensions of value, monitored entity, and time interval. It then analyses the data by comparing it against the profiles of the past it has stored on a database and reports any significant changes. The numerous parameters that affect Panoptis's performance can be easily tuned to match the characteristics of the system being supervised forming heuristic rules. This approach is flexible and provides useful results while limiting extraneous noise.

After using Panoptis for some time we found out that the data evaluated can be expanded in a number of ways by increasing the number of derived properties (e.g. adding running averages). In addition, report triggering can be made more selective by introducing thresholds, counters, and combined conditions. This additional complexity will require the provision of a more sophisticated configuration system, probably a rule-based language. We are currently investigating the requirement specifications for such a language. We are also looking for ways to automate the administration of Panoptis's configuration based on templates suitable for different types of systems and checks.

References

[A+95] D. Anderson et al. Next-generation intrusion detection expert system (NIDES): A summary. Technical Report SRI-CSL-95-07, SRI Int'l., 1995.

[And80] J. Anderson. Computer security threat monitoring and surveillance. Technical report, J. Anderson Co., Pennsylvania, April 1980.

[BBH+94] J. Bell, F. Bellegarde, J. Hook, R. B. Kieburtz, A. Kotov, J. Lewis, L. McKinney, D. P. Oliva, T. Sheard, L. Tong, L. Walton, and T. Zhou. Software design for reliability and reuse: a proof-of-concept demonstration. In *Conference on TRI-Ada '94*, pages 396–404. ACM, ACM Press, 1994.

[BK88] David S. Bauser and Michael E. Koblentz. NIDX — a real-time intrusion detection expert system. In *USENIX Conference Proceedings*, pages 261–273, San Francisco, CA, USA, Summer 1988. Usenix Association.

[BKS90] Fuat Baran, Howard Kaye, and Margaritta Suarez. Security breaches: Five recent incidents at Columbia university. In *UNIX Security Workshop II*, pages 151–171, Portland, OR, USA, August 1990. Usenix Association.

[BLC95] T. Berners-Lee and D. Connolly. RFC 1866: Hypertext Markup Language — 2.0, November 1995. Status: PROPOSED STANDARD.

[Den83] Dorothy Elizabeth Robling Denning. *Cryptography and Data Security*. Addison-Wesley, 1983.

[GL91] T. Garvey and T. Lunt. Model-based intrusion detection. In *14th National Computer Security Conference*, 1991.

[JL87] Stephen C. Johnson and Michael E. Lesk. Language development tools. *Bell System Technical Journal*, 56(6):2155–2176, July-August 1987.

[Kar87] P. Karger. Limiting the damage potential of discretionary Trojan horses. In *IEEE Symposium on Security and Privacy*, pages 32–37. IEEE Press, 1987.

[Kuh86] J. Kuhn. Research towards intrusion detection through the automated abstraction of audit data. In *9th National Computer Security Conference*, 1986.

[L+92] T. Lunt et al. A real-time intrusion-detection expert system. Technical Report SRI-CSL-92-05, SRI Int'l., 1992.

[Lin75] R. Linde. Operating system penetration. In *National Computer Conference*, 1975.

[LMKQ88] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System*. Addison-Wesley, 1988.

[Lun93] T. Lunt. A survey of intrusion detection techniques. *Computers and Security*, 12(4):405–418, June 1993.

[Pic87] J. Picciotto. The design of an effective auditing subsystem. In *IEEE Symposium on Research in Security and Privacy*, pages 13–22. IEEE Press, 1987.

[Ram97] J. Christopher Ramming, editor. *USENIX Conference on Domain-Specific Languages*, Santa Monica, CA, USA, October 1997. Usenix Association.

[S+99] R. Sekar et al. A high-performance network intrusion detection system. In *6th ACM Conference on Computer and Communication Security*, pages 8–17. ACM Press, 1999.

[SG97] Diomidis Spinellis and V. Guruprasad. Lightweight languages as software engineering tools. In Ramming [Ram97], pages 67–76.

[VK98] G. Vigna and R. Kemmerer. NetSTAT: A network-based intrusion detection approach. In *Computer Security Applications Conference*, 1998.

[WS90] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly and Associates, Sebastopol, CA, USA, 1990.