PERFORMANCE BENCHMARKING OF UNIX SYSTEM AUDITING

A Thesis Submitted by

Sam Nitzberg

in partial fulfillment of the requirements

for the degree of

Master of Science in Software Engineering

at Monmouth College

August 18, 1994

Advisor: Edward Amoroso, Ph. D.

</HR>

Performance Benchmarking of UNIX System Auditing

Sam Nitzberg

Department of Software Engineering, Monmouth College, W. Long Branch, N.J. 07712

Abstract

A performance benchmark suite and associated methodology is introduced and demonstrated for the analysis of security-enhanced UNIX System® audit trails. To demonstrate and validate the proposed suite and methodology, a case study is presented using an existing off-the-shelf UNIX system environment (HP-UX). The results of this benchmarking case study demonstrate that the proposed suite and methodology provide a feasible and useful means for performance analysis of security-enhanced UNIX auditing in practical settings. In particular, a series of tests was executed with system auditing turned both on and off (for a full set of selectable audit channels) and performance degradation values ranged from negligible impact for arithmetic component benchmark tests to as high as 85% performance degradation for system call overhead tests. These results may have significant implications for the source selection and marketing of UNIX-based systems. Appendices detail the benchmarks and case study execution results.
<HR>

1 Introduction

Organizations deciding whether or not to use the security features provided by a given security-enhanced UNIX system® generally have no quantitative methods for assisting in this determination. In particular, benchmarking tools and methodologies for measuring the impact of on-line auditing on UNIX system performance and resource usage are not generally available. The implications of this are especially evident in the manner in which the U.S. Government, for example, selects and procures UNIX-based systems. Even on large efforts such as the U.S. Army Sustaining Base Information Services (SBIS) Project where considerable expense was allocated by the government to install UNIX-based systems that include auditing within the Army's Sustaining Base Automation infrastructure, no generally-accepted or off-the-shelf benchmarking suite or methodology was available and hence required for use during the proposal life cycle as a source selection criterion. The work reported in this paper will help to establish factors that provide concrete

insights into a means for comparison of existing system implementations.

The specific goal of the work reported here is to initiate the preliminary identification of a generally-accepted set of measures with which to determine if the security features for any given UNIX implementation are suitable for use in a given system. An additional goal is to report how these security features generally impact system resources. Particular emphasis is placed on the user-visible impact of auditing by employing off-the-shelf benchmarks that simulate typical system behavior. Problems that must be solved in this effort include the determination of criteria that characterize security feature performance, the identification of security critical events, and the implementation of the security feature benchmark tests. To validate and demonstrate these results, a case study experiment using an off-the-shelf UNIX system (i.e., HP-UX running on an HP 9000 workstation) was completed and is reported here. Since any benchmark is capable of producing unexpected results, a degree of interpretation is included in this paper as well.

Section 2 of this paper provides an overview of on-line auditing on UNIX-based systems. Section 3 addresses performance analysis concerns and benchmarking methods in general with emphasis on audit system performance. It also explains considerations regarding the choice of the benchmarking system utilized, together with a general discussion of benchmarking systems available. Section 4 explains the specific test environment, the analysis of results from the benchmarking runs performed, and an explanation of the analysis performed. Section 5 summarizes the findings of this effort, with suggested direction for further work. The Appendices provide descriptions of the benchmark tests for the suite used, with the reports generated from the benchmarking runs.

2 Security-Enhanced UNIX Auditing

A primary component of recent security-enhanced UNIX systems involves on-line auditing of identified security-critical events. This inclusion is a direct result of the types of attacks that have been reported in computing environments. In Peter Neumann's Risks to the General Public [1], for example, a considerable number of malicious attacks have been observed in which on-line auditing could have been a useful deterrent (i.e., because attackers, knowing they could be caught, might not attempt the attack).

The technology of on-line auditing has evolved gradually from simple operating system schemes that were designed for maintaining user accounting data (e.g., *acctcom (1)* on UNIX) to real-time data collection and processing features in fully-networked environments (i.e., LANs and WANs) that include routines for determining if potential attacks have been initiated and for notifying the affected personnel, sometimes immediately. The AT&T Audit Trail Analysis Tool [2] (previously referred to as ComputerWatch) is an example of such a system.

The diagram in Figure 1 represents an operational abstraction of a typical on-line security auditing architecture on a UNIX system. Security critical actions occurring on the given computer system are identified by carefully-placed kernel (and sometimes user-level) probes. These probes are essentially procedure calls that pass audit reports to a specific auditing routine, usually a dedicated audit daemon. The audit daemon accepts the audit records and commits them to a protected log for perusal by any existing audit trail analysis tools or for eventual transport to a possibly-remote system for storage or analysis. Obviously, this entire operation is intended to be as transparent to user-level processes as possible. In the optimal case, a user-level process initiates a kernel level sequence that passes the records smoothly and offers no performance degradation or visible effect as a result of the auditing calls, storage, or processing.

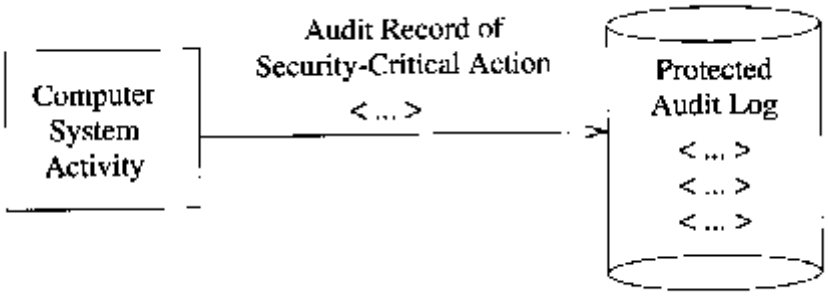

Figure 1: Typical Auditing Architecture.

Most existing and previous UNIX-based systems, such as those consistent with the AT&T System V Interface Definition (SVID) [3] have traditionally addressed security by a collection of familiar security routines such as the *chown* command to change ownership of files and the *chgrp* command to change a file's group and security attributes for files. Auditing as a security extension to UNIX, is generally less well-known and hence requires additional explanation in order that the benchmarking results reported here be properly interpreted and understood. Specifically, it is important to recognize that the two major classes of auditing systems can be referred to as *stateful* audit systems and *stateless* audit systems. A stateful audit system initiates its operation in some well-defined initial state, and as the system operation progresses, will record those events which can change its initial and all subsequent states. The stateful system thus must only record enough information to establish a traceable thread of investigation through the audit trail for all processes. A stateless system, on the other hand, will not maintain changes from an initial state. Instead, such systems will produce discrete, stand-alone, unrelated records. The major advantage of such a system is that the logical relationships between records can potentially be reconstructed even if certain records are missing or otherwise corrupted.

It is a straightforward enough matter to extract basic conclusions about system operation from audit trails. In the simplest case, operating system-supplied commands and utilities such as *grep* and *awk* can be used to pattern-search for specified information. All security-critical activity relating to a given user or files will thus have audit records associated with those entities recorded and may be viewed either off-line or in real-time. Such tools provided with the operating system usually are fairly limited in what they can perform with the audit trail. More sophisticated tools such as SRI NIDES (Network Intrusion Detection System) [4] from SRI use user profiling data to perform sophisticated analysis of audit trails.

An additional issue that is often considered with respect to auditing is the so-called rating that a given system might have with respect to the National Computer Security Center (NCSC) Orange Book [5] or a similar security criteria. Roughly, C2 certified systems require discretionary access control (DAC) mechanisms, audit trails, and appropriate destruction of residues within the system. B1 systems further require mandatory access control (MAC) with a security level assigned to each subject and object in addition to the DAC mechanisms which function at a lower level. These allow mapping of subjects to objects to be defined (see [6] for more details on DAC and MAC). If a user, as a subject does not have a defined access to a given object (generally a file), that access is denied. HP-UX, for example, has a collection of classes for auditable events which may be turned on or off. For example, the "delete" class is associated with the calls *rmdir(2)*, *semctl(2)*, and *msgctl(2)*. Auditing on these processes may be turned on or off at will by the administrator [7]. Self-auditing capabilities are provided so that programs may limit the audit data they produce to relevant items. In addition, the format in which audit data is presented to an administrator for perusal is an important factor in that system's usability. System V/MLS, for example, offers several views including formatted output.

Table 1 summarizes several factors with respect to auditing.

	HP-UX	SUN OS	System V/MLS
NCSC Rating	Unrated	C2	B1
State Model	Stateless	Stateless	Stateful
Audit Format	Fixed	Fixed	Verbose, Sensitive, Raw

Table 1: Audit Comparison of Three UNIX Systems.

Note that this table offers additional technical and source-selection insights into existing UNIX auditing functionality and features. Note that the products selected for inclusion in the table were selected as a representative set of existing products. System V/MLS is assumed in the B1 configuration as an add-on package to the NCR RAS UNIX operating system. The Sun OS version considered is the C2-rated system [8].

Since the primary goal of the work reported here involves the measurement of performance degradation due to the additional processing afforded by auditing, it is instructive to consider the type of processing that one might expect to find in a UNIX auditing system. The diagram in Figure 2 below offers a schematic view of the likely course of action that typically exists in a UNIX auditing scheme. The reader is advised, of course, that this schematic greatly simplifies the auditing design in most practical systems. For example, user-level auditing is often included and requires specialized design and integration. In addition, routines are not depicted for optimizing performance including caching schemes and complex file system maps that ease the amount of information that must be captured by a specific trail.

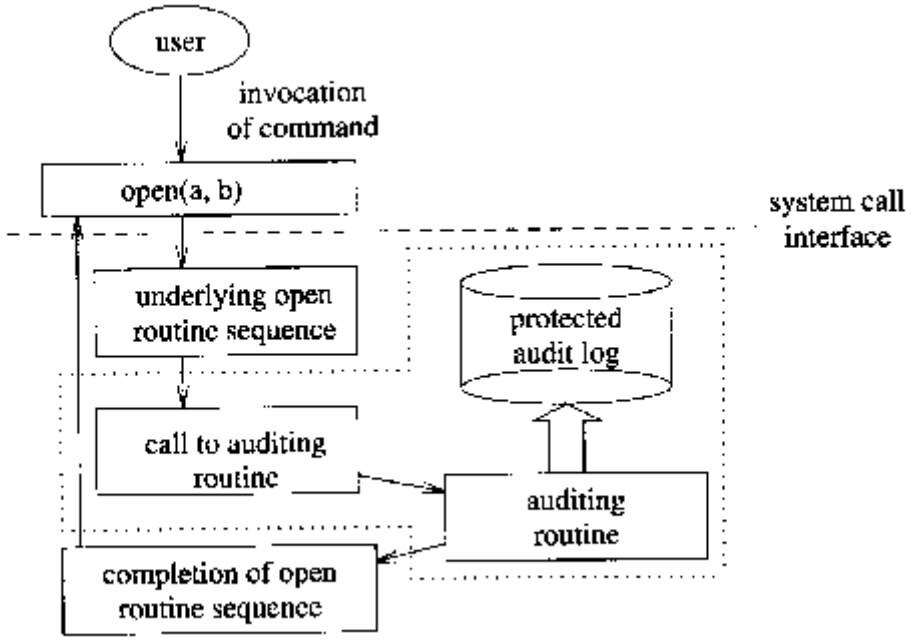

Figure 2: Auditing Operational View.

Figure 2 depicts the basics of a typical Unix auditing system's method for handling an auditable item, in this case, an *open()* call. The operational sequence is as follows: the user invokes the *open()* call, which contains instructions (at the kernel level) that

are to be audited. Control is then branched to the auditing facility which generates a record in a protected audit log. The auditing thread of execution is then completed for this function, and control returns to the open sequence which completes its function.

The diagram in Figure 3 below instantiates this general notion for a specific system, namely System V/MLS.

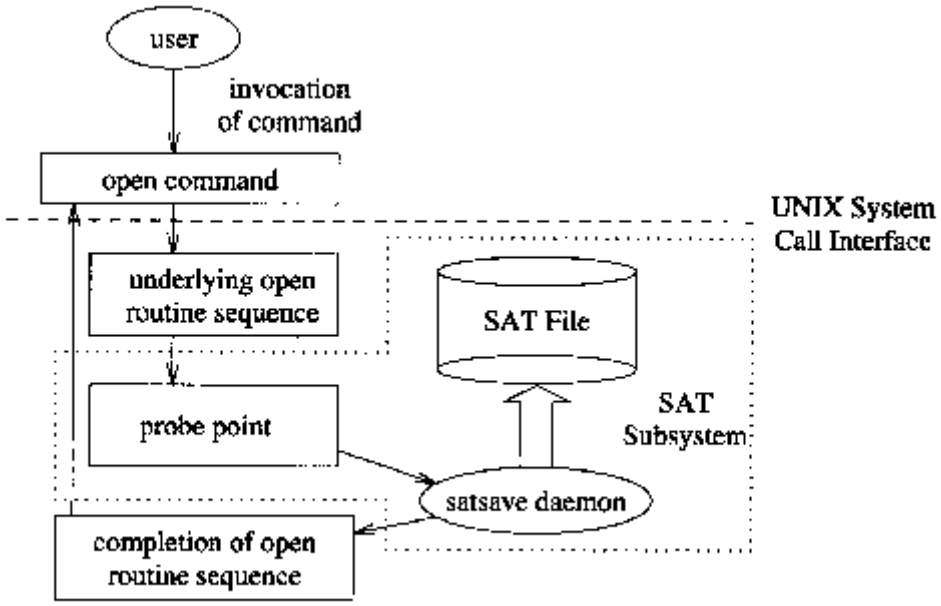

Figure 3: UNIX System V/MLS Auditing.

Note in Figure 3 that audit calls are referred to as probe points. The probe points (42 in all in System V/MLS) are embedded in both the kernel and user-level UNIX routines. The probe points transmit their information to the SAT security module. This module invokes a satsave daemon call to store the audit data. More information can be obtained from the NCSC final report on System V/MLS [9].

3 Benchmarking Considerations

In developing our methodology, we also examined existing approaches to workload and performance characterization. These investigations offered some insight into the optimal approach to benchmarking auditing systems. For example, the Performance Analysis Methodology [10]) recommended that one rule out any system limitations first. It then recommended examining whether the data indicates a particular bottleneck. Finally, it suggested using multiple metrics to validate any discovered bottlenecks. While this methodology was not used specifically, it did affect our thinking.

For example, the audit benchmarks were examined from a bottleneck perspective. That is, one significant use of benchmarks is to aid in identifying and correcting bottlenecks in systems. With a methodology such as the one outlined above, benchmarks can assist in resolving critical systems problems based on bottlenecks. Secure systems must be capable of being audited without the auditing system itself becoming a bottleneck, or otherwise impairing functionality.

Taking the considerations mentioned above into account, we selected four factors that should affect the benchmarking methodology. These were adapted from recommendations from the Transaction Processing Council (TPC) [11]. The TPC's benchmark selection criteria follow:

* Applicability (Is it relevant to large number of users?)

* Comparability (Will it lead to objective comparisons?)

* Understandability (Can users / press understand its significance?)

* Executability (Can it be run in a reasonable time period and for a reasonable cost?)"

Another consideration in selecting our benchmarking approach was the work reported by SPEC (Software Performance Evaluation Cooperative). SPEC makes this statement regarding how its benchmarks are to be selected [12]:

. . .the benchmarks must be large, long-running (5-10 minutes), realistic programs that test a wide range of applications and system resources (integer, floating-point, cache, memory system, I/O, etc.). Whenever possible, SPEC will seek benchmark candidates from public domain sources.

from which we may adopt these additional factors in selecting the benchmarking suite for a case study:

* The benchmark must exercise many aspects of the target system;

* The benchmark should be in the public domain.

For the benchmark used in the case study, the Byte UNIX benchmark suite was chosen. The Byte Unix Suite is applicable towards general user concerns [13]. This is a public domain software item, available by anonymous ftp from ftp.uu.net in the published/byte /benchmarking/sources directory Appendix A briefly describes the nature of each type of benchmark found in the benchmark suite. For the purposes at hand, this suite meets our criteria: The tests it performs are, generally of interest to a wide audience of users; It provides objective comparisons, is highly understandable, executes in a reasonable period of time, and exercises many properties of the operating system while residing in the public domain.

4 Case Study: Benchmarking HP-UX

In order to assess the adequacy and suitability of the proposed benchmark suite for measuring UNIX system audit trails, we utilized the suite in a specific case study environment. In particular, the case study environment consisted of an HP-9000 model 385, ruggedized system. This particular system is also known as a TCU, or Tactical Computer Unit. This system was installed with HP-UX version 8.0, a hybrid between AT&T System V and BSD Unix. The TCU was equipped with 64 megabytes of RAM, a MC68040 CPU, and a floating point accelerator. The storage medium was a magneto-optical disk, mounted to the TCU via an MSEU (Mass Storage Expansion Unit).

To prepare to invoke the benchmarks, non-essential system processes were killed. *Sockregd* (the socket registration daemon), *syncdaemon* (the disk buffer syncing daemon), and *rlbdaemon* (a networking daemon) were automatically respawned on the test system, and were not killed. The analysis which follows demonstrates that the test case was not detrimentally affected by these processes. The system was converted to a trusted unix system. The HP-UX Sam (Systems Administration Manager) program was used to control the audit environment. After this, SAM was used to disable and enable system auditing for each set of runs. When system auditing was to be enabled for a given trial run, all auditable events / items / system calls were set to be audited.

The Byte UNIX Benchmark suite was installed and invoked without modification. This suite automatically installs properly on either BSD or AT&T System V type environments. No special or custom software tools were used in the production of any reported statistics.

Two pairs of benchmark runs were performed. The first set of runs consists of RUN1 and RUN2. The second set of runs consists of RUN3 and RUN4. RUN1 and RUN3 were performed with the full auditing on, while RUN2 and RUN4 were performed without auditing. From the results of each benchmark invoked under the suite, performance degradation under the auditing system was calculated independently for each set of runs.

Each benchmarking component which had a calculated degradation of more than 1 percent for either of the two sets of runs has its larger value represented in Figure 4.

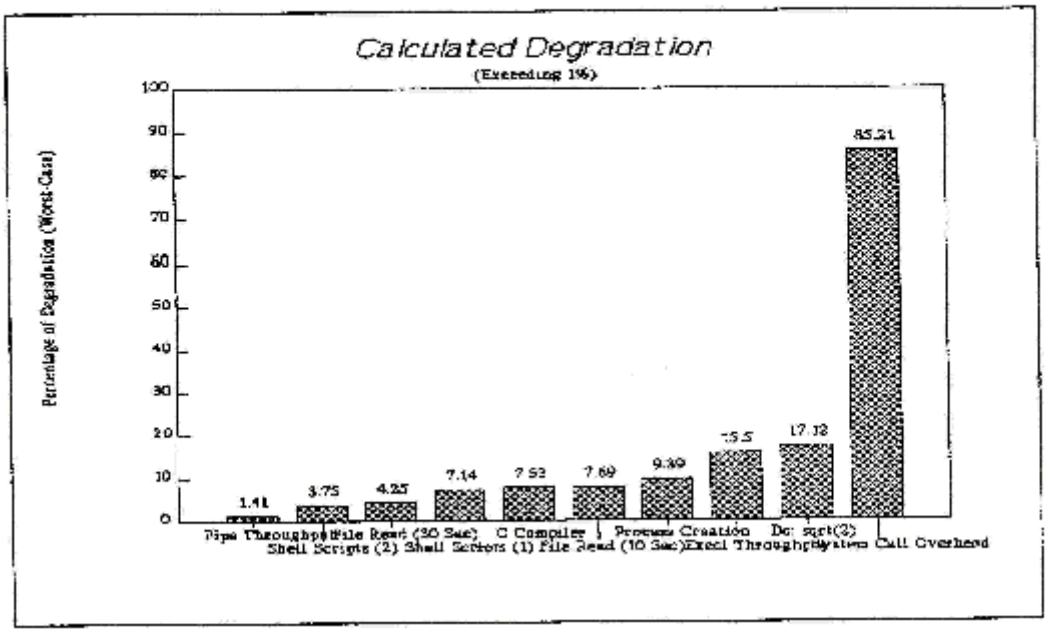

Figure 4.

Statistics have been drawn from the benchmark results (see Table 2). Delta-1 represents the percentage of deviation in recorded benchmark performance for RUN1 and RUN2. Delta-2 is similarly computed based on the output of the benchmark for RUN3 and RUN4. Negative values represent recorded performance degradation. Positive results represent cases where the test performed executed at a faster rate with auditing on; these results are generally of a very low magnitude. Due to the nature of Unix systems, operations which execute in the same time, allowing a very small deviation may be considered to be statistically identical. Delta-3 represents the absolute, simple difference between the Delta-1 and Delta-2 values for an exercise. For any result with a very small value for Delta-3, the confidence is high in the result. Any specific benchmarks for which Delta-3 had a significant value would have to be examined to determine the cause of the anomalous reading.

Benchmark	Delta-1	Delta-2	Delta-3
Dhrystone 2 w/o register variables	0.030809	0.01441	0.045227
Dhrystone 2 using register variables	0.018354	0.053415	0.035061
Arithmetic Test (type = aritoh)	0.04264	0.05603	0.013382
Arithmetic Test (type = register)	0.13912	0.077270	0.216395
Arithmetic Test (type = short)	0.054131	0.074080	0.019949
Arithmetic Test (type= int)	0.089644	0.071099	0.018544
Arithmetic Test (type= long)	0.080385	0.046354	0.034030
Arithmetic Test (type = float)	0.054738	0.054728	0.000009
Arithmetic Test (type = double)	0.088180	0.061033	0.027146
System Call Overhead Test	85.2194	84.7967	0.422698
Pipe Throughput Test	1.41787	1.41443	0.003433
Pipebased Context Switching Test	0.84433	0.78582	0.058510
Process Creation Test	8.74587	9.39044	0.644570
Execl Throughput Test	15.0561	15.5056	0.449438

File Read (10 Seconds)	4	7.69230	*11.69230*
File Write (10 seconds)	0	0	0
File Copy (10 Seconds)	0	0	0
File Read (30 Seconds)	3.67709	4.25205	0.574962
File Write (30 Seconds)	0	0.97087	0.970873
File Copy (30 seconds)	0	0	0
C Compiler Test	-7.53424	0	7.534246
Shell scripts (1 concurrent)	7.14285	7.14285	0
Shell scripts (2 concurrent)	-3.75	-3.75	0
Shell scripts (4 concurrent)	0	0	0
Shell scripts (8 concurrent)	0	0	0
Dc: sqrt(2) to 99 decimal places	17.5279	17.1205	0.407401
Recursion TestTower of Hanoi	0.078740	0.019685	0.059055

Table 2: Benchmark Statistics.

The suite is composed of 27 tests. Singly and grouped by commonality, a view of the results follows:

* The Dhrystone 2 and Arithmetic tests experienced no significant degradation, as shown by the reported Delta-1 and Delta-2 values. These values all hover about 0 and also indicate that there is no significant, general overhead imposed by the audit-trail mechanisms on the operating system.

* The System Call overhead test revealed high degradation. Auditing was invoked on all system calls, with a calculated 85% degradation. While a call-by-call analysis would provide the specific degradation for each call, this shows an almost doubling of the time required to perform system calls with auditing invoked.

* The Pipe Tests revealed minimal degradation for pipe-based communications.

* The Process Creation Test and the Execl Throughput Test experienced approximately 9 and 15 percent degradation, respectively.

* The File Read and Write Tests experienced degradations of up to 7.69% under auditing. Tests which revealed zero degradation appear to be due to caching. The system used for the test case had 64 megabytes of RAM.

* The C Compiler Test shows a worst-case result of 7.5% degradation under auditing.

* The Shell Scripts Tests performed a variety of functions on a file. For the cases of 1 and 2 concurrent executing scripts, the degradations were calculated as 7.1% and 3.75% respectively. All other cases indicate no degradation. This appears to be due to caching.

* The Dc test revealed 17% degradation for the dc Unix high-precision calculator utility.

* The Recursion Test--Tower of Hanoi reveals no significant degradation for recursive operations.

There are two Delta-3 values which are of interest. The most significant values listed are for the 10 seconds file read test, and for the C Compiler Test. The actual values returned from the File Read (10 Seconds) test has low granularity, and the values are closely grouped. The average value for these runs was 50.5 KBps with little actual deviation (values are 50KBps +/- 2 Kbps). This explains similar results with a significant calculated deviation between runs. The C Compiler Test encountered a degradation of 7.53% for the first run set, with no reported degradation for the second set of runs. Such deviations may be due to disk caching. Since the majority of the Delta-3 values are so small, there is high confidence in these results and methods. With more suite invocations, average deltas should stabilize, and figures which appear to be anomalous and due to disk caching should become clear.

5 Concluding Remarks and Future Work

The work described in this report represents an initial attempt at providing quantitative information to assist in the determination regarding whether to use or not use a security-enhanced UNIX system. It is worth noting that there are both pros and cons towards using the approaches described. On the positive side, the methods described do allow comparisons to be made in performance both between differing systems in secure modes, and for a single unit in both secure and non-secure modes (or with varying degrees of security). One shortcoming to be considered are the potentially large deltas which may be generated in reports due directly to caching. Due to the effects of caching, worst-case analyses are required.

Earlier in this paper, it was stated that problems to be solved in this effort include the determination of criteria that characterize security feature performance, the identification of security critical events, and the design and implementation of the security feature benchmark. In order, this is how these issues have been addressed. The chosen criteria to characterize security feature performance is the percentage of degradation experienced under benchmarking for specific tests. The security critical events for the purposes of this paper have been generally determined to be the complete set of auditable events for the system under test in order to provide a worst-case analysis; this may be different for specific, special cases. The implementation of the security feature benchmark was based on a COTS approach. Unless a demonstrably superior benchmark targeted at the UNIX security enhancement systems could be based on a view of user's interests, be easily understood, and readily and widely accepted, another approach would be preferable. Other COTS and commercial packages are available for UNIX systems benchmarking, and based on the results derived from the tests performed, would be expected to return worthwhile results.

Network benchmarking suites are commercially available. Through this effort, it appears that COTS and PDS software can be used towards measuring the cost of using secure extensions in general. By using such use-specific benchmarking systems, the costs of using these secure extensions while networking may be determined in terms of throughput and other measures (including storage cost for various degrees of auditing) for various platforms and configurations.

Benchmarking systems have not traditionally been invoked against auditing systems, with the results publicized to provide consumers of secure computing environments with information which would be highly useful in evaluating computing products towards their various missions prior to purchase. As the results of benchmark runs are freely available in magazines and from some anonymous ftp sites for various platforms, the results of benchmarking systems in various levels of their security should likewise be made available.

Current benchmarks produce discrete factors quantifying system performance, provide factors relating to various system throughputs reached, and describe the rate of performing certain mathematical functions. What these benchmarks presently lack from the view of those interested in security extensions is the ability to describe the cost of producing the audit trail on available storage for individual tests and cumulatively. This is a basis for comparison worth noting on these systems.

One general approach towards solution to the problem at hand seems plausible for future work. That approach is to exercise low-level (auditable) events of interest through high-level, user-typical code. That is, typical end-to-end user behavior (by some definition) is simulated. Statistical measures relating to performance (time of suite execution, and perhaps memory or other system utilization measurements) are taken with the audit trail facilities both enabled and disabled. The few resulting measures with some basic statistical analyses are used to indicate the performance of the audit trail system. Scaling benchmarks which operate in this fashion (simulating end-to-end user behvior) exist in the form of COTS (Commercial, Off-The-Shelf) and Public Domain Software. Investigation of existing packages for suitability to the problems at hand has been investigated towards the benchmarking environment to be incorporated under this effort.

Systems of interest would either provide data regarding simulated end-to-end user view

of performance, by simulating multiple users on the system under test and noting degradation, or would provide a collection of simple, understandable collection of statistics regarding various aspects of systems performance. This suite falls into the latter category.

Acknowledgments

The assistance of Chuck Flink of Bell Laboratories, Ram Chelluri of AT&T, and Ken Lee of Ft. Monmouth was greatly appreciated.
</HR>

References

[1] P. Neumann, RISKS: Cumulative Index of Software Engineering Notes -- Illustrative Risks to the Public in the Use of Computer Systems and Related Technology, *ACM Software Engineering Notes*, Vol. 14, No. 1, 1989.

[2] C. Dowell and P. Ramstedt, The ComputerWatch Data Reduction Tool, *Proceedings of the 13th National Computer Security Conference*, Washington, D.C. 1990.

[3] AT&T, UNIX System V Interface Definition , Volumes I and II, 1986.

[4] T. Lunt, Network Intrusion Detection Systems (NIDES), SRI Technical Report, 1994.

[5] Department of Defense, *Trusted Computing System Evaluation Criteria*, publication number CSC-STD-001-85

[6] E. Amoroso*, Fundamentals of Computer Security Technology*, PTR Prentice Hall, Englewood Cliffs, N.J., 1994.

[7] Hewlett Packard, *HP-UX System Security*, part number B1862-90009, 1991.

[8] Sun Microsystems, *Security Features Guide*, part Number:800-1735-10 Revision A, 9 May 1988

[9] National Computer Security Center, *Final Evaluation Report of American Telephone and Telegraph System V/MLS Release 1.1.2 Running on Unix System V Release 3.1.1*, publication number CSC-EPL-89/003

[10] R. Chelluri (Chair), *Uniforum Performance Measurement/Management and Capacity Planning of UNIX Systems*, 1993.

[11] Ibid.

[12] Ibid.

[13] Ibid.

[14] *Byte Magazine*, Unix Benchmark Suite, documentation file bench.doc.
</HR>

Appendix A: Benchmark Code [14]

This appendix describes the general nature of the benchmarks included in the benchmark suite used for the test cases:

Dhrystone 2 consists of conditionals, flow controls, arrays, character strings, and common programming constructs; no floating point operations are used.

Arithmetic Tests listed contain assorted basic arithmetic operations, with each test performing the set of operations on the indicated data type.

System Call Overhead Test involves the use of dup(), close(), getpid(), getuid(), and umask() calls.

Pipe Throughput Test evaluates time required to pass 1 megabyte of data through a pipe to and from a single process.

Pipe-based Context Switching Test evaluates the time required to conduct bi-directional communications between a process and its child through a pipe.

Process Creation Test evaluates time required to fork()

Execl Throughput Test - evaluates time for a process to repeatedly execl() a process. A process creates a child process, which dies and is repeatedly regenerated.

File operations tests indicates throughput for the indicated operations for the time period indicated.

C Compiler Test returns a rating for a compile and link in terms of lines per minute.

Shell Script Test is a script run by concurrent processes. The script performs a number of operations on a file, including sort, grep, and wc (word count) functions.

Dc: sqrt(2) to 99 decimal places performs the indicated function, using the dc high-precision calculator utility.

Tower of Hanoi (Recursion Test) performs the classical Tower of Hanoi as an exercise in recursion.

</HR>

Appendix B: Detailed Benchmark Execution Results

This section contains the reports generated by the Byte Benchmark Suite which are referenced in this paper. The next four pages of reports detail RUN1, RUN2, RUN3, and RUN4 in order.

RUN: AUDITING STATE:

RUN1 ON

RUN2 OFF

RUN3 ON

RUN4 OFF

</HR>

BYTE UNIX Benchmarks (Version 3.11)

System HPUX mcsexp B.08.00 B 9000/385 08000917228b

Start Benchmark Run: Fri Jul 8 09:37:08 EDT 1994

1 interactive users.

Dhrystone 2 without register variables 30519.5 lps (10 secs, 6 samples)

Dhrystone 2 using register variables 30516.5 lps (10 secs, 6 samples)

Arithmetic Test (type = arithoh) 1451221.0 lps (10 secs, 6 samples)

Arithmetic Test (type = register) 3230.0 lps (10 secs, 6 samples)

Arithmetic Test (type = short) 3511.9 lps (10 secs, 6 samples)

Arithmetic Test (type = int) 3237.9 lps (10 secs, 6 samples)

Arithmetic Test (type = long) 3237.0 lps (10 secs, 6 samples)

Arithmetic Test (type = float) 2741.8 lps (10 secs, 6 samples)

Arithmetic Test (type = double) 2951.1 lps (10 secs, 6 samples)

System Call Overhead Test 752.7 lps (10 secs, 6 samples)

Pipe Throughput Test 1474.0 lps (10 secs, 6 samples)

Pipebased Context Switching Test 645.9 lps (10 secs, 6 samples)

Process Creation Test 55.3 lps (10 secs, 6 samples)

Execl Throughput Test 37.8 lps (9 secs, 6 samples)

File Read (10 seconds) 52.0 KBps (10 secs, 6 samples)

File Write (10 seconds) 2200.0 KBps (10 secs, 6 samples)

File Copy (10 seconds) 214.0 KBps (10 secs, 6 samples)

File Read (30 seconds) 10190.0 KBps (30 secs, 6 samples)

File Write (30 seconds) 2266.0 KBps (30 secs, 6 samples)

File Copy (30 seconds) 139.0 KBps (30 secs, 6 samples)

C Compiler Test 13.5 lpm (60 secs, 3 samples)

Shell scripts (1 concurrent) 13.0 lpm (60 secs, 3 samples)

Shell scripts (2 concurrent) 7.7 lpm (60 secs, 3 samples)

Shell scripts (4 concurrent) 4.0 lpm (60 secs, 3 samples)

Shell scripts (8 concurrent) 1.6 lpm (60 secs, 3 samples)

Dc: sqrt(2) to 99 decimal places 920.8 lpm (60 secs, 6 samples)

Recursion TestTower of Hanoi 508.4 lps (10 secs, 6 samples)

INDEX VALUES

TEST BASELINE RESULT INDEX

Arithmetic Test (type = double) 2541.7 2951.1 1.2

Dhrystone 2 without register variables 22366.3 30519.5 1.4

Execl Throughput Test 16.5 37.8 2.3

File Copy (30 seconds) 179.0 139.0 0.8

Pipebased Context Switching Test 1318.5 645.9 0.5

Shell scripts (8 concurrent) 4.0 1.6 0.4

=========

SUM of 6 items 6.5

AVERAGE 1.1

</HR>

BYTE UNIX Benchmarks (Version 3.11)

System HPUX mcsexp B.08.00 B 9000/385 08000917228b

Start Benchmark Run: Fri Jul 8 11:16:30 EDT 1994

1 interactive users.

Dhrystone 2 without register variables 30510.1 lps (10 secs, 6 samples)

Dhrystone 2 using register variables 30510.9 lps (10 secs, 6 samples)

Arithmetic Test (type = arithoh) 1451840.2 lps (10 secs, 6 samples)

Arithmetic Test (type = register) 3234.5 lps (10 secs, 6 samples)

Arithmetic Test (type = short) 3510.0 lps (10 secs, 6 samples)

Arithmetic Test (type = int) 3235.0 lps (10 secs, 6 samples)

Arithmetic Test (type = long) 3234.4 lps (10 secs, 6 samples)

Arithmetic Test (type = float) 2740.3 lps (10 secs, 6 samples)

Arithmetic Test (type = double) 2948.5 lps (10 secs, 6 samples)

System Call Overhead Test 5092.5 lps (10 secs, 6 samples)

Pipe Throughput Test 1495.2 lps (10 secs, 6 samples)

Pipebased Context Switching Test 651.4 lps (10 secs, 6 samples)

Process Creation Test 60.6 lps (10 secs, 6 samples)

Execl Throughput Test 44.5 lps (9 secs, 6 samples)

File Read (10 seconds) 50.0 KBps (10 secs, 6 samples)

File Write (10 seconds) 2200.0 KBps (10 secs, 6 samples)

File Copy (10 seconds) 214.0 KBps (10 secs, 6 samples)

File Read (30 seconds) 10579.0 KBps (30 secs, 6 samples)

File Write (30 seconds) 2266.0 KBps (30 secs, 6 samples)

File Copy (30 seconds) 139.0 KBps (30 secs, 6 samples)

C Compiler Test 14.6 lpm (60 secs, 3 samples)

Shell scripts (1 concurrent) 14.0 lpm (60 secs, 3 samples)

Shell scripts (2 concurrent) 8.0 lpm (60 secs, 3 samples)

Shell scripts (4 concurrent) 4.0 lpm (60 secs, 3 samples)

Shell scripts (8 concurrent) 1.6 lpm (60 secs, 3 samples)

Dc: sqrt(2) to 99 decimal places 1116.5 lpm (60 secs, 6 samples)

Recursion TestTower of Hanoi 508.0 lps (10 secs, 6 samples)

INDEX VALUES

TEST BASELINE RESULT INDEX

Arithmetic Test (type = double) 2541.7 2948.5 1.2

Dhrystone 2 without register variables 22366.3 30510.1 1.4

Execl Throughput Test 16.5 44.5 2.7

File Copy (30 seconds) 179.0 139.0 0.8

Pipebased Context Switching Test 1318.5 651.4 0.5

Shell scripts (8 concurrent) 4.0 1.6 0.4

=========

SUM of 6 items 6.9

AVERAGE 1.1

</HR>

BYTE UNIX Benchmarks (Version 3.11)

System HPUX mcsexp B.08.00 B 9000/385 08000917228b

Start Benchmark Run: Fri Jul 8 12:48:24 EDT 1994

1 interactive users.

Dhrystone 2 without register variables 30512.0 lps (10 secs, 6 samples)

Dhrystone 2 using register variables 30531.7 lps (10 secs, 6 samples)

Arithmetic Test (type = arithoh) 1451755.7 lps (10 secs, 6 samples)

Arithmetic Test (type = register) 3237.9 lps (10 secs, 6 samples)

Arithmetic Test (type = short) 3512.3 lps (10 secs, 6 samples)

Arithmetic Test (type = int) 3237.2 lps (10 secs, 6 samples)

Arithmetic Test (type = long) 3237.4 lps (10 secs, 6 samples)

Arithmetic Test (type = float) 2742.3 lps (10 secs, 6 samples)

Arithmetic Test (type = double) 2951.0 lps (10 secs, 6 samples)

System Call Overhead Test 776.4 lps (10 secs, 6 samples)

Pipe Throughput Test 1484.6 lps (10 secs, 6 samples)

Pipebased Context Switching Test 643.9 lps (10 secs, 6 samples)

Process Creation Test 55.0 lps (10 secs, 6 samples)

Execl Throughput Test 37.6 lps (9 secs, 6 samples)

File Read (10 seconds) 48.0 KBps (10 secs, 6 samples)

File Write (10 seconds) 2200.0 KBps (10 secs, 6 samples)

File Copy (10 seconds) 214.0 KBps (10 secs, 6 samples)

File Read (30 seconds) 9998.0 KBps (30 secs, 6 samples)

File Write (30 seconds) 2244.0 KBps (30 secs, 6 samples)

File Copy (30 seconds) 139.0 KBps (30 secs, 6 samples)

C Compiler Test 14.6 lpm (60 secs, 3 samples)

Shell scripts (1 concurrent) 13.0 lpm (60 secs, 3 samples)

Shell scripts (2 concurrent) 7.7 lpm (60 secs, 3 samples)

Shell scripts (4 concurrent) 4.0 lpm (60 secs, 3 samples)

Shell scripts (8 concurrent) 1.6 lpm (60 secs, 3 samples)

Dc: sqrt(2) to 99 decimal places 925.1 lpm (60 secs, 6 samples)

Recursion TestTower of Hanoi 508.1 lps (10 secs, 6 samples)

INDEX VALUES

TEST BASELINE RESULT INDEX

Arithmetic Test (type = double) 2541.7 2951.0 1.2

Dhrystone 2 without register variables 22366.3 30512.0 1.4

Execl Throughput Test 16.5 37.6 2.3

File Copy (30 seconds) 179.0 139.0 0.8

Pipebased Context Switching Test 1318.5 643.9 0.5

Shell scripts (8 concurrent) 4.0 1.6 0.4

=========

SUM of 6 items 6.5

AVERAGE 1.1

</HR>

BYTE UNIX Benchmarks (Version 3.11)

```
System HPUX mcsexp B.08.00 B 9000/385 08000917228b

Start Benchmark Run: Fri Jul 8 14:23:57 EDT 1994

1 interactive users.

Dhrystone 2 without register variables 30516.4 lps (10 secs, 6 samples)

Dhrystone 2 using register variables 30515.4 lps (10 secs, 6 samples)

Arithmetic Test (type = arithoh) 1452569.6 lps (10 secs, 6 samples)

Arithmetic Test (type = register) 3235.4 lps (10 secs, 6 samples)

Arithmetic Test (type = short) 3509.7 lps (10 secs, 6 samples)

Arithmetic Test (type = int) 3234.9 lps (10 secs, 6 samples)

Arithmetic Test (type = long) 3235.9 lps (10 secs, 6 samples)

Arithmetic Test (type = float) 2740.8 lps (10 secs, 6 samples)

Arithmetic Test (type = double) 2949.2 lps (10 secs, 6 samples)

System Call Overhead Test 5106.8 lps (10 secs, 6 samples)

Pipe Throughput Test 1505.9 lps (10 secs, 6 samples)

Pipebased Context Switching Test 649.0 lps (10 secs, 6 samples)

Process Creation Test 60.7 lps (10 secs, 6 samples)

Execl Throughput Test 44.5 lps (9 secs, 6 samples)

File Read (10 seconds) 52.0 KBps (10 secs, 6 samples)

File Write (10 seconds) 2200.0 KBps (10 secs, 6 samples)

File Copy (10 seconds) 214.0 KBps (10 secs, 6 samples)

File Read (30 seconds) 10442.0 KBps (30 secs, 6 samples)

File Write (30 seconds) 2266.0 KBps (30 secs, 6 samples)

File Copy (30 seconds) 139.0 KBps (30 secs, 6 samples)

C Compiler Test 14.6 lpm (60 secs, 3 samples)

Shell scripts (1 concurrent) 14.0 lpm (60 secs, 3 samples)

Shell scripts (2 concurrent) 8.0 lpm (60 secs, 3 samples)

Shell scripts (4 concurrent) 4.0 lpm (60 secs, 3 samples)

Shell scripts (8 concurrent) 1.6 lpm (60 secs, 3 samples)

Dc: sqrt(2) to 99 decimal places 1116.2 lpm (60 secs, 6 samples)

Recursion TestTower of Hanoi 508.0 lps (10 secs, 6 samples)

INDEX VALUES
```

TEST BASELINE RESULT INDEX

Arithmetic Test (type = double) 2541.7 2949.2 1.2

Dhrystone 2 without register variables 22366.3 30516.4 1.4

Execl Throughput Test 16.5 44.5 2.7

File Copy (30 seconds) 179.0 139.0 0.8

Pipebased Context Switching Test 1318.5 649.0 0.5

Shell scripts (8 concurrent) 4.0 1.6 0.4

=========

SUM of 6 items 6.9

AVERAGE 1.1