

Understanding One's Systems: A New Role for Auditing

Alva Couch

Marc Chiarini

Josh Danziger

May 30, 2006

Abstract

Auditing at the kernel level has frequently been used to debug the performance of operating systems and applications [27, 8] or to detect intrusions by finding anomalies in program behavior [20, 16]. This paper explores an alternative use of auditing to enhance understanding of system behavior and improve the processes of diagnosing and responding to failures. We describe our experiences with the auditing subsystem built into the Linux 2.6 kernel and present methods for correlating and interpreting the collected data in useful ways.

1 Introduction

Most highly experienced system administrators can remember a time in their career when they were virtually clueless about the configuration of their systems. Whether learning on the job as a junior sysadmin or walking into a brand new infrastructure, none of us is ever handed a comprehensive guide to “the way things work around here.” Instead, sysadmins must slowly develop a *mental model* of the systems in our care [21]. They study existing documentation and Internet sources, solicit expert advice, explore component interactions, and much more. While this process is valuable in the long run, it is also time-consuming and error prone, and competes with the efficiency of whatever task is at hand, e.g. tracking down and fixing the root causes of problems. Additionally, mental models are developed on an as-needed basis, resulting in large gaps and inaccuracies. Some form of automated system documentation would help to mitigate these drawbacks in much the same way that clear, concise, comprehensive documentation eases the addition of new members to a software development team.

In the sections that follow, we describe the task of dependency analysis, justify its necessity, and explain our approach in detail.

2 Dependency Analysis

Efficient system administration requires mental models accurate enough to suggest proper courses of action. One part of a good mental model is a map of dependencies between the various components in a system. At a high level, *components* can be thought of as subsystems, e.g. “the web subsystem depends upon the DNS subsystem.” At the lowest level of abstraction, components consist of programs and their individual configuration parameters. At this level, a good mental model maps how parameter changes affect a program's dependencies.

For the purposes of this research, we loosely define *dependency* as external information that must be transmitted to a component for it to function correctly. For example, when a process loads a library, functions necessary to the core behavior of the process are transmitted to it from a file. The process is *dependent* on the library being loaded into some part of memory and being made accessible.

Likewise, when Apache starts, it reads necessary parameters from an external source of information, e.g. `httpd.conf`. Further, Apache depends upon its runtime environment to properly specify the location of `httpd.conf`.

The process of identifying dependencies for every component in a system naturally generates a *directed dependency graph*. Each vertex in the graph represents a component, and each edge represents observed information flow from an antecedent component to a dependent component. We refer to each such antecedent as a *first-order dependency*. Further, two vertices that are connected by some path through the graph may exhibit *transitive dependency*, e.g. if *A* depends upon *B*, and *B* depends upon *C*, then *A* may depend upon *C*.

We define *dependency analysis* as the generation and study of dependency graphs and related statistical information extracted from one or more systems. Working with graphs proves advantageous because they are amenable to a wide variety of well understood algorithms that can reveal useful and previously hidden facts. It is clear that dependency analysis will never be able to reveal all dependencies or suggest the root causes of *every* problem; however, such an analysis will in many circumstances result in a model that is “good enough” for efficient, accurate diagnosis.

3 The Case for Automated Dependency Analysis

As suggested earlier, a clear and accurate system model is paramount to troubleshooting. Although sysadmins already troubleshoot in the absence of such models, their efforts have been significantly hindered by complexity. When something fails in a system, knowing where to look first is usually a “gimme”. Under progressively greater pressure, knowing where to look second, third, fourth, and so on, takes experience, perserverance, and nerves of steel. Dependency analysis will not obviate the need for these characteristics, but in most cases it will lower the bar that must be met to succeed in these tasks.

In the absence of formal documentation, sysadmins have few resources for determining the dependencies of a program. Although one may assume that documentation is available for general-use tools, many organizations develop in-house solutions. When these solutions are intended for internal use only, there is little economic incentive to create polished user interfaces or comprehensive documentation; tools must simply be “good enough.” As the number of internal libraries, scripts, and programs increases, making changes to the system becomes increasingly difficult. For example, deleting old libraries becomes virtually impossible when sysadmins have no knowledge of what programs call which libraries. The complexity of these poorly understood systems will continue to grow without bound as long as they are actively developed. Sysadmins in this situation would benefit greatly from a comprehensive view of component dependencies.

The final area where we expect dependency analysis to shine is in preparation for system migration. At many sites, sysadmins must frequently transfer services from legacy or failing systems to new hardware or software platforms. Ideally, everything on the old system that is not being used would be left behind while important existing functionality would be reincarnated on new machines. Even with reams of documentation and a methodical plan, this is an arduous task. One is always worried that something will be overlooked. Auto-documentation of dependencies will contribute to the stability and completeness of this property.

4 Auditing as a Foundation

Generally speaking, all I/O performed by a process is mediated by the operating system (files, shared pipes, signals, sockets, shared memory), and almost all communication with the operating system is

```

type=SYSCALL msg=audit(1140945727.543:14126959): arch=40000003 syscall=5 success=yes exit=3
a0=22afba a1=0 a2=1b6 a3=8157240 items=1 pid=7235 auid=4294967295 uid=0 gid=0 euid=0 suid
=0 fsuid=0 egid=0 sgid=0 fsgid=0 comm="sh" exe="/bin/bash"
type=PATH msg=audit(1140945727.543:14126959): item=0 name="/etc/mtab" inode=460869 dev=fd:00
mode=0100644 ouid=0 ogid=0 rdev=00:00
type=SYSCALL msg=audit(1140945727.618:14127510): arch=40000003 syscall=5 success=yes exit=3
a0=872fba a1=0 a2=1b6 a3=941a240 items=1 pid=7242 auid=4294967295 uid=0 gid=0 euid=0 suid
=0 fsuid=0 egid=0 sgid=0 fsgid=0 comm="sh" exe="/bin/bash"
type=PATH msg=audit(1140945727.618:14127510): item=0 name="/etc/mtab" inode=460869 dev=fd:00
mode=0100644 ouid=0 ogid=0 rdev=00:00

```

Fig. 1: Sample log entries from the audit subsystem showing accesses of `/etc/mtab`. Audit records contain a wealth of information about the `open` call, including filename, process id/name, mode, etc.

performed via system calls. Therefore, passive interposition at the system call level provides sufficient data to generate complete dependency graphs.

There exist many utilities designed to record process activity at the kernel level. Command-line tracing programs like `strace` and `truss` work with the kernel to stop a single traced program and obtain information upon syscall entry or exit. Dynamic tools such as Intel’s PIN [24] and Sun’s Dtrace [9] perform customizable in vivo instrumentation of programs, can collect virtually any type of information, and can even modify program behavior. Other custom solutions utilize system call interposition [15, 35] in the form of a loadable kernel module or modifications to the C library. Lastly, there are auditing facilities for the Linux 2.4 kernel (SNARE [2] requires a kernel patch, `syscalltracker` [4] kernel module), the Linux 2.6 kernel (LAuS [14]), and Solaris (BSM [7]). These are designed to deliver system call info from the kernel to a daemon residing in user space.

4.1 Current Approach

Our strategy for collecting dependency information was guided by several important goals. First, the chosen method must be efficient, as system administrators are not likely to use something which significantly hampers performance. Second, the method must be unobtrusive, altering system behavior as little as possible and requiring few configuration changes to the systems of interest. Finally, we desired a method which was easy to learn and use; we wanted to focus our energies on the unique issues posed by automated dependency analysis without worrying about the technical details of implementing a complex tool.

The Linux Audit System (LAuS) proved to be a nearly ideal candidate. The Linux 2.6 kernel, now present in a plethora of Linux distributions, includes the LAuS by default. Thus, enabling auditing on a Linux 2.6 system only requires modification of a few configuration files. The interface is also incredibly simple; plain text messages generated by the kernel are buffered until the user-space daemon `auditd` fetches and writes them to disk. The ASCII format employed by the kernel produces information-rich messages that are easily parsed (Figure 1). Finally, because log entries are stored on disk, they can be processed when machines would otherwise be idle.

Our prototype implementation focuses on the auditing of component dependencies expressed in the filesystem. All `open()` calls are intercepted and recorded in an audit log. A script reads log entries and generates a database which contains program names, file names, open mode (read/write/append/create), and access frequency. This information can be used to generate rudimentary dependency graphs, which can be enhanced via several methods as described in Section 4.3.

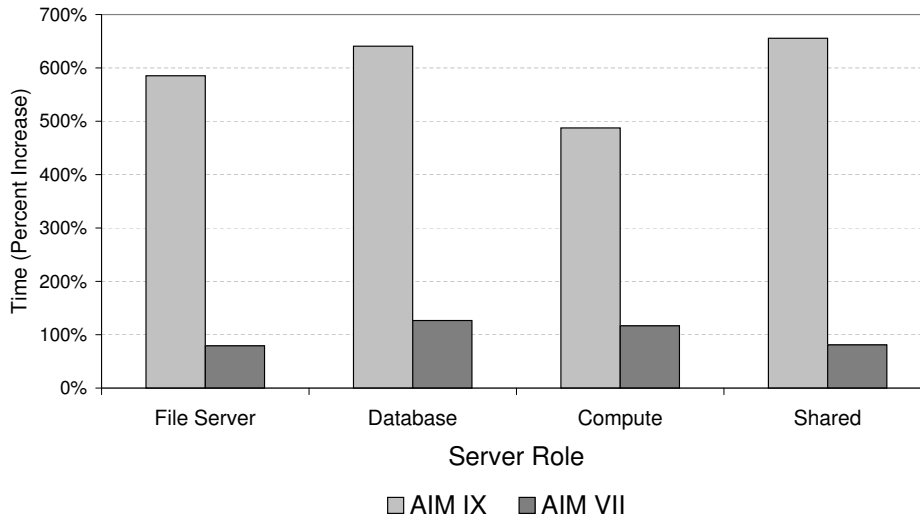


Fig. 2: Auditing performance in the `open()` call as measured by AIM.

4.2 Performance

For the typical program, system call interposition is not likely to degrade performance. Compared to `read()` and `write()` operations, `open()` calls are rare. Thus, even substantial decreases in `open()` call performance should scarcely affect overall performance. Unfortunately, there are classes of programs which frequently call `open()`, e.g. web servers and filesystem monitors. For these programs, `open()` call overhead can dominate execution time and degrade performance system-wide.

In tests performed with the AIM VII (multi-user) benchmark [1], the time spent in the `open()` call roughly doubled with the introduction of auditing. However, the AIM IX (single-user) benchmark showed more dramatic increases in the time spent per call. Further research must be conducted to ascertain the reason for this disparity. Nonetheless, the results of AIM VII are encouraging (Figure 2).

Auditing may generate data at extremely high rates, depending upon the type of system; compiling *glibc* created 5GB of data in approximately one hour. Post-processing audit data can reduce space requirements by several orders of magnitude, but current implementations of our post-processing utility require approximately 40 minutes to process 1GB of data. This is a bit disheartening; however, we expect that an optimized post-processor will be able to achieve significantly better results.

4.3 Reasoning About the Data

By now the reader may have noticed that what we refer to as a *dependency graph* is really a graph of *potential* dependencies; for many programs, there are no dependencies whatsoever on the vast majority of files that they open. For example, `cat`, which reads the contents of an input stream, only relies upon the C library (and technically, `/etc/ld.so.cache`), yet the graph captures a potential dependency for *every* distinct file that `cat` opens. Though the absence of these files may cause a *script* to fail, none of them is essential to the core behavior of `cat` itself. A similar result holds for many other programs; almost every file that is necessary for them to function properly is loaded with their image or shortly thereafter. There are notable exceptions: programs like Apache and PERL frequently load modules on-demand; daemons may reload their configuration files when a HUP signal is received, but will rarely reload a library; and shell scripts frequently defy all notions

of predictability.

It would appear that the generated graph contains too much extraneous information. In the absence of code and dataflow analysis, we need a way to determine whether an edge truly represents a dependency. We can use several simple observations to guide us:

- If a program opens the same file one or more times on (nearly) every invocation, there is a high likelihood of dependency.
- The first-order dependencies of many programs are known a priori, either via direct experience, documentation, or technical detail, e.g. statically-linked programs.
- Files residing in well-known configuration directories such as `/etc` can be labeled with a high probability when all other indicators are (nearly) equal.
- Files that are created by and opened for reading and writing in short intervals or across multiple invocations by a single program might be safely excluded.

Acting intelligently on these observations will greatly reduce the size and density of the graph. Certain edges of which we are unsure may of course be left in the graph until we are better able to classify them. Once we have a graph of reasonable size and density, we can utilize elementary graph algorithms, statistical techniques, and even machine learning to answer interesting queries.

4.4 Building Tools

In this section, we describe two types of tools that will be essential to using dependency analysis both as an educational framework and diagnostic aid. Instead of getting caught in a GUI vs. CLI conflict, we take the advice of those smarter than ourselves and work toward making all of our tools “fast, truthful, scalable, and scriptable [*or at any rate, programmable*].” [3].

4.4.1 A Command-Line Interface

Good CLI’s are held in high regard by many system administrators primarily because of their flexibility and the fact that they allow tool use to be automated via scripting. CLI’s for dependency analysis should in no way hinder the process of exploration. This means that they must present a familiar interface that even the least-experienced sysadmins will find comfortable. Our approach is to lay out dependencies in a virtual filesystem, and use a standard directory paradigm to navigate them. We start in the root directory, which represents the file `ld.so.cache`. The reason for this is that every program accesses this file when it is loaded into memory and run. Executing `ls` in this directory will list as subdirectories, all programs (and scripts) that have ever been the subject of an audit. Changing into one of these directories and listing its contents will show all files on which the program depends. Thus, exploration of the graph can be continued indefinitely. Standard commands can be issued to discover more information about any file or program, e.g. obtaining its type, showing its contents, etc. Identical names are disambiguated via a unique user-customizable prefix. Scripts can easily be written to traverse this unique filesystem and process its contents.

4.4.2 Visualization

Command-line interfaces suffer from a serious weakness: they are unable to efficiently utilize the high-bandwidth visual conduit and strong pattern-recognition skills of human beings. To discover the necessary and useful characteristics of a visual dependency analysis tool, we have experimented with the Prefuse Visualization Toolkit for Java [19]. Prefuse provides a highly versatile API for building

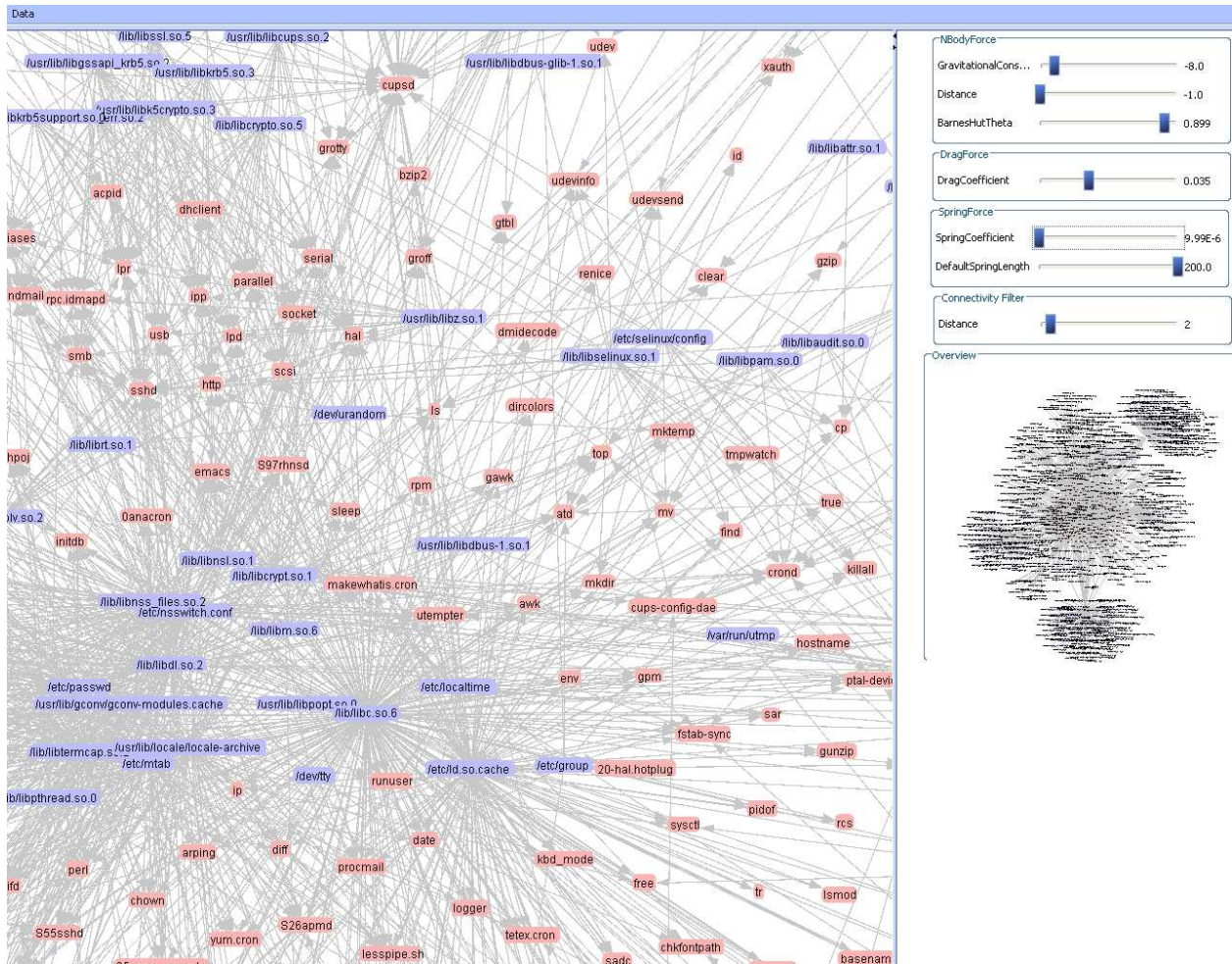


Fig. 3: Screenshot of a dependency graph visualization tool built with Prefuse

visualizations from almost any type of data. It is particularly well-suited to representing graphs in different types of layouts, e.g. force directed, radial, and balloon tree, tree map, etc [18]. Additionally, graph vertices and edges can be labeled, colored, weighted, and positioned in programmatic fashion or by user control. When working with smaller graphs (less than 1000 edges), the force directed layout enables a sysadmin to immediately see “dependency clusters” (files on which many programs depend) as well as programs with many dependencies. Figure 3 shows a screenshot of our prototype.

5 Related Work

In recent years, there has been exciting research on improving failure diagnosis and impact analysis capabilities for system administrators. These approaches generally fall into several categories for which we cite examples here:

- Path-tracing of requests through distributed system infrastructures [10, 11].
- Static extraction of service (or program) dependencies by analysis of package management repositories [22] and program images [28].

- Automatic construction of operational dependency models via active perturbation [6].
- Using visualization to help operators rapidly detect and diagnose problems [29, 5].
- Using comparison against golden state configurations (generated via statistical analysis of machine populations [32, 33] or known good disk states on a single machine [34]) to pinpoint misconfigurations.
- Using event correlation in log-file analysis to identify extant and potential problems [17, 26].
- Using service grammars (high-level descriptions of requirements) and model-finding to diagnose configuration errors [25, 23].

It is important to note that many of these approaches are tailored to specific environments and in fact may not be applicable in the general case. In contrast, we feel that our own work begins from a generic base and will be useful at any site that utilizes open-source or COTS operating systems.

We end this section by acknowledging a recently discovered project whose philosophy is strikingly similar to our own. InDependence was a 1999 USENIX Student Grant Project directed by Crispin Cowan whose aim was to “make system configuration easier by systematically automating the tracking of dependence relationships between various system components.” [13]. This is the only other research of which we are aware that used system-call tracing specifically for this purpose. Compared to our work, InDependence was much more limited in scope: its runtime trace facility worked on only a single program at a time.

6 Technical Challenges and Future Work

Thus far, we have shown that auditing is a good first step toward generating accurate dependency graphs. However, much work remains to render our approach useful in practical settings. Specifically, we must make system call auditing more efficient and develop high quality, intuitive tools for analysis of collected data.

Because our approach must constantly monitor system behavior, audit overhead must be made minimal. As discussed in Section 4.2, the post-processing of audit logs is currently suboptimal. We expect that our post-processing algorithms can be improved by taking advantage of spatio-temporal locality and other a priori knowledge. Performance in the kernel itself is also a mixed bag; the AIM benchmarks show that in the best case, time spent in the `open()` call roughly doubles when auditing is enabled. We plan to make auditing run even faster by reworking the kernel code to filter unnecessary data at the source or, even better, prevent its generation in the first place.

Our current implementations only examine dependencies as expressed via the Linux filesystem. Unfortunately, many dependencies are expressed via other information vectors, e.g. pipes, shared memory, signals, environments, and the network. As a result, dependency graphs generated by our tools are not comprehensive. We believe that analysis of network I/O would prove to be a powerful technique. By auditing, e.g. `connect()/accept()` pairs, we can identify dependencies that span physical machines. For example, a network-aware tool would be able to identify dependencies between the web server and the DNS server. Expanding the collection and analysis phases in this way will require considerable effort.

Finally, we plan to incorporate ideas from machine-learning, not only to help generate automatic analyses of dependency graphs, but to augment graphs with information gleaned from `sysadmin` interactions with our tools [12].

7 Conclusions

In our introduction, we made the claim that accurate mental models are necessary to most tasks performed by system administrators, including troubleshooting and maintenance. As such, any tool which aids in the timely development of accurate mental models would be a big win for sysadmins at both the junior and senior level. In this paper, we have demonstrated that data for dynamic analysis can be gathered via system call auditing.

Although we have developed proof-of-concept tools to perform dependency analysis through the Linux filesystem, much work remains to make analysis practical for real systems. Specifically, tools must be optimized and altered to capture all dependency vectors. Despite our initial findings, we are confident that dynamic dependency analysis will emerge as an effective utility in everyday system administration.

References

- [1] Aim benchmark. <http://aim.sourceforge.net>.
- [2] T. I. Alliance. SNARE (system iNtrusion Analysis and Reporting Environment). <http://www.intersectalliance.com/projects/index.html>.
- [3] R. Barrett, E. Kandogan, P. P. Maglio, E. M. Haber, L. A. Takayama, and M. Prabaker. Field studies of computer system administrators: analysis of system management tools and practices. In J. D. Herbsleb and G. M. Olson, editors, *CSCW*, pages 388–395. ACM, 2004.
- [4] M. Ben-Yahuda. Syscalltracker. <http://syscalltrack.sourceforge.net/>.
- [5] P. Bodic, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A. Fox, M. I. Jordan, and D. Patterson. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 89–100, Washington, DC, USA, 2005. IEEE Computer Society.
- [6] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment, 2001.
- [7] Sunshield basic security module. <http://docs.sun.com/app/docs/doc/806-4078?q=BSM>.
- [8] A. N. Burton and P. H. J. Kelly. Performance prediction of paging workloads using lightweight tracing. In *IPDPS*, page 278. IEEE Computer Society, 2003.
- [9] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track* [31], pages 15–28.
- [10] M. Y. Chen, A. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E. A. Brewer. Path-based failure and evolution management. In *NSDI*, pages 309–322. USENIX, 2004.
- [11] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. A. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN*, pages 595–604. IEEE Computer Society, 2002.
- [12] S. J. Cunningham, I. H. Witten, and J. Littin. Applications of machine learning in information retrieval. *Annual Review of Information Science*, 34:341–384, 1999.

- [13] C. C. et al. InDependence project. <http://www.usenix.org/publications/login/1999-12/student.html>.
- [14] R. Faith and S. Grubb. The linux audit system. <http://people.redhat.com/sgrubb/audit/>.
- [15] D. P. Ghormley, D. Petrou, and T. E. Anderson. SLIC: Secure loadable interposition code. In *1998 USENIX Technical Conference*, 1998.
- [16] A. K. Ghosh, A. Schwartzbard, and M. Schatz. Learning program behavior profiles for intrusion detection. In *Workshop on Intrusion Detection and Network Monitoring*, pages 51–62. USENIX, USENIX, 1999.
- [17] B. Gruschke. Integrated event management: Event correlation using dependency graphs. In *Proceedings of the 9th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management*, 1998.
- [18] J. Heer. Prefuse API documentation. <http://www.prefuse.org/doc/api/prefuse/action/layout/graph/package-summary.html>.
- [19] J. Heer. Prefuse: Interactive information visualization. <http://prefuse.sourceforge.net>.
- [20] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.
- [21] D. G. Hrebec and M. Stiber. A survey of system administrator mental models and situation awareness. In M. A. Serva, editor, *SIGCPR*, pages 166–172. ACM, 2001.
- [22] G. Kar, A. Keller, and S. B. Calo. Managing application services over service provider networks: Architecture and dependency analysis. In *Proceedings NOMS-2000 (IEEE)*, Hawaii, unknown 2000.
- [23] S. Narain. Network configuration management via model finding. In *LISA*, pages 155–168, 2005.
- [24] Pin: A dynamic binary instrumentation tool. <http://rogue.colorado.edu/Pin/index.html>.
- [25] X. Qie and S. Narain. Using service grammar to diagnose bgp configuration errors. In *LISA* [30], pages 237–246.
- [26] J. P. Rouillard. Real-time log file analysis using the simple event correlator (SEC), 2004.
- [27] Y. Ruan and V. S. Pai. Making the "box" transparent: System call performance as a first-class result. In *USENIX Annual Technical Conference, General Track* [31], pages 1–14.
- [28] Y. Sun and A. L. Couch. Global impact analysis of dynamic library dependencies. In *LISA*, pages 145–150. USENIX, 2001.
- [29] T. Takada and H. Koike. Mielog: A highly interactive visual log browser using information visualization and statistical analysis. In *LISA*, pages 133–144. USENIX, 2002.
- [30] USENIX. *Proceedings of the 17th Conference on Systems Administration (LISA 2003)*, San Diego, California, USA, October 26-31, 2003. USENIX, 2003.
- [31] USENIX. *Proceedings of the General Track: 2004 USENIX Annual Technical Conference, June 27 - July 2, 2004, Boston Marriott Copley Place, Boston, MA, USA*. USENIX, 2004.

- [32] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *OSDI*, pages 245–258, 2004.
- [33] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. Strider: A black-box, state-based approach to change and configuration management and support. In *LISA* [30], pages 159–172.
- [34] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *OSDI*, pages 77–90, 2004.
- [35] K. Yaghmour and M. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *USENIX Annual Technical Conference, General Track*, pages 13–26. USENIX, 2000.