

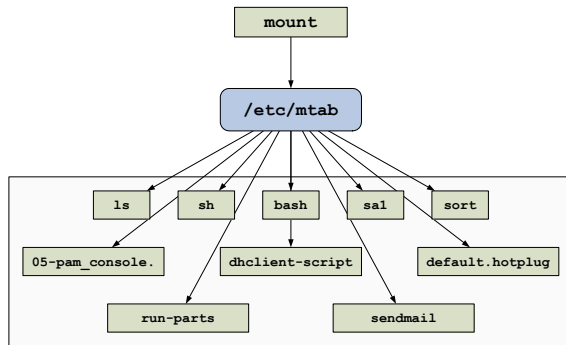
Improving Troubleshooting: Dynamic Analysis of Dependencies

Research Objectives

In computing systems, effective troubleshooting requires a good mental model of how components (programs, services, files, etc) depend upon and interact with each other. Our research aims to improve system administrators' understanding by automating the collection, correlation, and presentation of dependency data.

Dependency Analysis

For the purposes of this research, dependency analysis is the process of examining models of system behavior to determine the root cause of errant behavior. The quality of any analysis is greatly determined by the accuracy and completeness of the model. We argue that automatically generated dependency graphs will help sysadmins verify and/or enhance their mental models with observed data.



A sample graph showing dependencies expressed through the `/etc/mtab` file. Square boxes represent programs, rounded boxes represent files. We say that there is a transitive dependency between `mount` and the programs that read `/etc/mtab`; if `mount` corrupts the `/etc/mtab` file, then all of those programs may behave unexpectedly.

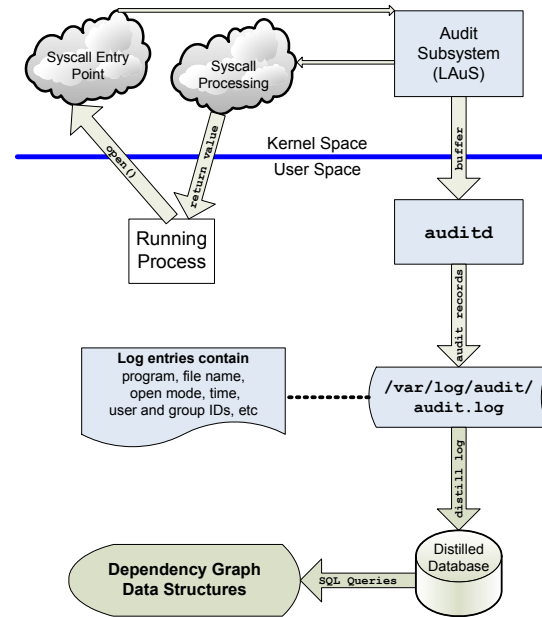
Using Dependency Graphs

- Dependency graphs are amenable to both visual exploration and algorithmic analysis.
- Visualization quickly transmits a high volume of useful information.
- Algorithmic analysis can reveal many useful but normally hidden facts.

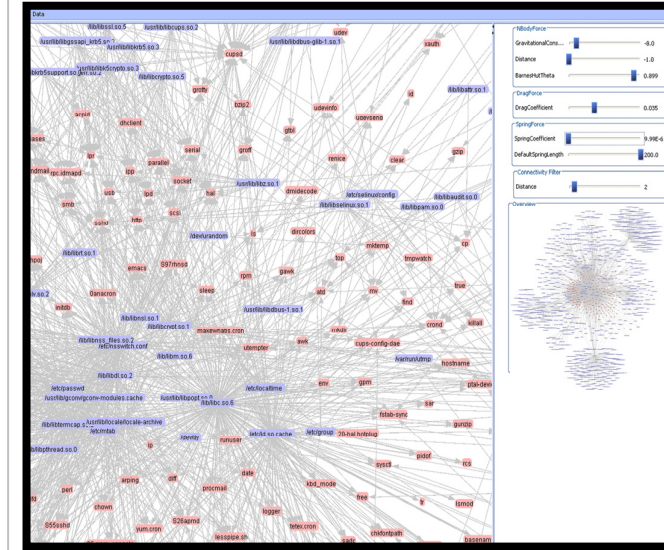
Gathering Dependency Information

- Statically:** Tools, such as `sowhat`, gather information about library dependencies by statically analyzing a program's image.
- Dynamically:** System call tracing reveals dependencies on libraries, network services, and configuration files in real-time but requires more data and processing power.

Our Approach



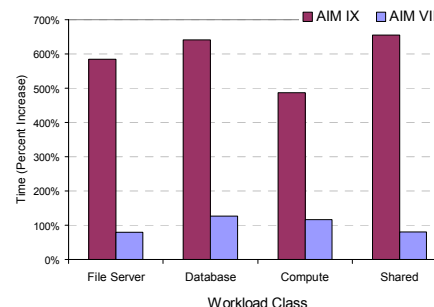
Visualization



Technical Challenges and Future Work

- Large amounts of data are generated in a short period of time. Modifying the kernel to filter records at the source (using, e.g. computed goto's) is likely to improve performance.
- Apply machine-learning techniques to extract useful patterns.
- Build a scriptable command-line interface tool.
- Make the visualization tool more interactive and customizable.
- Expand scope of analysis to include other types of I/O, e.g. network services and user environment.

Performance



- The audit subsystem generates system call information in the kernel and relays it over a netlink to a daemon residing in user space.
- We used `strace` to record times spent in the `open()` call during a set of AIM IX (single-user) and AIM VII (multi-user) benchmarks with various synthetic workloads.
- With auditing enabled, a substantial, but manageable overhead was observed in the multi-user benchmark (~100%).