Flight Data Recorder: Monitoring Persistent-State Interactions to Improve Systems Management

Chad Verbowski, Emre Kıcıman, Arunvijay Kumar, Brad Daniels,
Shan Lu[‡], Juhan Lee[*], Yi-Min Wang, Roussi Roussev[†]
Microsoft Research, [†]Florida Institute of Technology, [‡]U. of Illinois at Urbana-Champaign, [*]Microsoft MSN

Abstract

Mismanagement of the persistent state of a system—all the executable files, configuration settings and other data that govern how a system functions—causes reliability problems, security vulnerabilities, and drives up operation costs. Recent research traces persistent state interactions—how state is read, modified, etc.—to help troubleshooting, change management and malware mitigation, but has been limited by the difficulty of collecting, storing, and analyzing the 10s to 100s of millions of daily events that occur on a single machine, much less the 1000s or more machines in many computing environments.

We present the Flight Data Recorder (FDR) that enables *always-on tracing, storage and analysis* of persistent state interactions. FDR uses a domain-specific log format, tailored to observed file system workloads and common systems management queries. Our lossless log format compresses logs to only 0.5-0.9 bytes per interaction. In this log format, 1000 machine-days of logs—over 25 billion events—can be analyzed in less than 30 minutes. We report on our deployment of FDR to 207 production machines at MSN, and show that a single centralized collection machine can potentially scale to collecting and analyzing the complete records of persistent state interactions from 4000+ machines. Furthermore, our tracing technology is shipping as part of the Windows Vista OS.

1. Introduction

Misconfigurations and other persistent state (PS) problems are among the primary causes of failures and security vulnerabilities across a wide variety of systems, from individual desktop machines to large-scale Internet services. MSN, a large Internet service, finds that, in one of their services running a 7000 machine system, 70% of problems not solved by rebooting were related to PS corruptions, while only 30% were hardware failures. In [24], Oppenheimer *et al.* find that configuration errors are the largest category of operator mistakes that lead to downtime in Internet services. Studies of wide-area networks show that misconfigurations cause 3 out of 4 BGP routing announcements, and are also a significant cause of extra load on DNS root servers [4,22]. Our own analysis of call logs from a large software company's internal help desk, responsible for managing corporate desktops, found that a plurality of their calls (28%) were PS related.[1] Furthermore, most reported security compromises are against known vulnerabilities—administrators are wary of patching their systems because they do not know the state of their systems and cannot predict the impact of a change [1,26,34].

PS management is the process of maintaining the "correctness" of critical program files and settings to avoid the misconfigurations and inconsistencies that cause these reliability and security problems. Recent work has shown that *selectively* logging how processes running on a system interact with PS (e.g., read, write, create, delete) can be an important tool for quickly troubleshooting configuration problems, managing the impact of software patches, analyzing hacker break-ins, and detecting malicious websites exploiting web browsers [17,35-37]. Unfortunately, each of these techniques is limited by the current infeasibility of collecting and analyzing the complete logs of 10s to 100s of millions of events generated by a single machine, much less the 1000s of machines in even a medium-sized computing and IT environments.

There are three desired attributes in a tracing and analysis infrastructure. First is low performance overhead on the monitored client, such that it is feasible to always be collecting complete information for use by systems management tools. The second desired attribute is an efficient method to store data, so that we can collect logs from many machines over an extended period to provide a breadth and historical depth of data when managing systems. Finally, the analysis of these large volumes of data has to be scalable, so that we can monitor, analyze and manage today's large computing environments. Unfortunately, while many tracers have provided low-overhead, none of the state-of-the-art technologies for "always-on" tracing of PS interactions provide for efficient storage and analysis.

We present the Flight-Data Recorder (FDR), a high-performance, always-on tracer that provides complete records of PS interactions. Our primary contribution is a domain-specific, queryable and compressed log file

[1] The other calls were related to hardware problems (17%), software bugs (15%), design problems (6%), "how to" calls (9%) and unclassified calls (12%). 19% not classified.

format, designed to exploit workload characteristics of PS interactions and key aspects of common-case queries—primarily that most systems management tasks are looking for "the needle in the haystack," searching for a small subset of PS interactions that meet well-defined criteria. The result is a highly efficient log format, requiring only 0.47-0.91 bytes per interaction, that supports the analysis of 1000 machine-days of logs, over 25 billion events, in less than 30 minutes.

We evaluate FDR's performance overhead, compression rates, query performance, and scalability. We also report our experiences with a deployment of FDR to monitor 207 production servers at various MSN sites. We describe how *always-on* tracing and analysis improve our ability to do after-the-fact queries on hard-to-reproduce incidents, provide insight into on-going system behaviors, and help administrators scalably manage large-scale systems such as IT environments and Internet service clusters.

In the next section, we discuss related work and the strengths and weaknesses of current approaches to tracing systems. We present FDR's architecture and log format design in sections 3 and 4, and evaluate the system in Section 5. Section 6 presents several analysis techniques that show how PS interactions can help systems management tasks like troubleshooting and change management. In Section 7, we discuss the implications of this work, and then conclude.

Throughout the paper, we use the term PS *entries* to refer to files and folders within the file system, as well as their equivalents within structured files such as the Windows Registry. A PS *interaction* is any kind of access, such as an open, read, write, close or delete operation.

2. Related Work

In this section, we discuss related research and common tools for tracing system behaviors. We discuss related work on analyzing and applying these traces to solve systems problems in Section 6. Table 1 compares the log-sizes and performance overhead of FDR and other systems described in this section for which we had data available [33,11,21,20,40].

The tools closest in mechanics to FDR are file system workload tracers. While, to our knowledge, FDR is the first attempt to analyze PS interactions to improve systems management, many past efforts have analyzed file system workload traces with the goal of optimizing disk layout, replication, etc. to improve I/O system performance [3,9,12,15,25,28,29,33]. Tracers based on some form of kernel instrumentation, like FDR and DTrace [30], can record complete information. While some tracers have had reasonable performance overheads, their main limitation has been a lack of support for efficient queries and the large log sizes. Tracers based on sniffing network file system traffic,

Table 1: Performance overhead and log sizes for related tracers. VTrace, Vogel and RFS track similar information to FDR. ReVirt and Forensix track more detailed information. Only FDR and Forensix provide explicit query support for traces.

	Performance Overhead	*Log size (B/event)*	*Log Size (MB/machine-day)*
FDR	<1%	0.7	20MB
VTrace	5-13%	3-20	N/A
Vogel	0.5%	105	N/A
RFS	<6%	N/A	709MB
ReVirt	0-70%	N/A	40MB-1.4GB
Forensix	6-37%	N/A	450MB

such as NFS workload tracers [12,29] avoid any client-perceived performance penalties, but sacrifice visibility into requests satisfied by local caches as well as visibility of the process making a request.

Complete versioning file systems, such as CVFS [31] and WayBack [8] record separate versions of files for every write to the file system. While such file systems have been used as a tool in configuration debugging [39], they do not capture file reads, or details of the processes and users that are changing files. The Repairable File Service (RFS) logs file versioning information and also tracks information-flow through files and processes to analyze system intrusions [40].

In [33], Vogels declares analysis of his 190M trace records to be a "significant problem," and uses data warehousing techniques to analyze his data. The Forensix project, tracing system calls, also records logs in a standard database to achieve queryability [13]. However, Forensix's client-side performance overhead and their query performance (analyzing 7 machine-days of logs in 8-11 minutes) make it an unattractive option for large-scale production environments.

A very different approach to tracing a system's behavior is to record the nondeterministic events that affect the system, and combine this trace with virtual machine-based replay support. While this provides finer-grained and more detailed information about all the behaviors of a system than does FDR, this extra information can come at a high cost: ReVirt reports workload-dependent slowdowns up to 70% [11]. More significantly, arbitrary queries are not supported without replaying the execution of the virtual machine, taking time proportional to its original execution.

While, to our knowledge, we are the first to investigate domain-specific compression techniques for PS interaction or file system workload traces, there has been related work in the area on optimizing or compressing program CPU instruction traces [5,19], as well as work to support data compression within general-purpose databases [6].

3. Flight Data Recorder Architecture

In this section, we present our architecture and implementation for black-box monitoring, collecting, and analysis of PS interactions. Our architecture consists of (1) a low-level driver that intercepts all PS interactions with the file system and the Windows Registry, calls to the APIs for process creation and binary load activity, and exposes an extensibility API for receiving PS interaction events from other specialized stores; and (2) a user mode daemon that collects and compresses the trace events into log files and uploads them to a central server, (3) a central server that aggregates the log files and, (4) an extensible set of query tools for analyzing the data stream. Our implementation does not require any changes to the core operating system or applications running atop it. We provide detailed discussion of our domain-specific queryable log format in Section 4.

3.1 FDR Agent Kernel-Mode Driver

Our low-level instrumentation is handled by a kernel mode boot driver[2], which operates in real-time and, for each PS interaction, records the current timestamp, process ID, thread ID, user ID, interaction type (read, write, etc.), and hashes of data values where applicable. For accesses to the file system, the driver records the path and filename, whether the access is to a file or a directory and, if applicable, the number of bytes read or written. For accesses to the registry, the driver records the name and location of the registry entry as well as the data it contains. The driver sits above the file system cache, but below the memory mapping manager. This driver also records process tree information, noting when a binary module is loaded, or when a process spawns another.

The largest performance impact from the driver stems from I/O related to log writing, memory copies related to logging events, and latency introduced by doing this work on the calling application's thread. We mitigate this by only using the application's thread to write the relevant records directly into the user-mode daemon's memory space, and doing the processing on the user-mode daemon's thread. Caches for user names and file names that need to be resolved for each interaction also help to minimize lookup costs.

Our kernel driver is stable and suitable for use in production environments, and will be available for public use as part of Windows Vista.

3.2 FDR Agent User-Mode Daemon

The user-mode daemon is responsible for receiving records of PS interactions from the kernel driver, compressing them into our log format in-memory, and periodically uploading these logs to a central server.

To avoid impacting the performance of the system, we configure our daemon to run at lowest-priority, meaning it will be scheduled only if the CPU is otherwise idle. If the daemon does fall behind, the driver can be configured to either block until space is available or drop the event. However, in practice, we have found that a 4MB buffer is sufficient to avoid any loss on even our busiest server machines.

The daemon throttles its overall memory usage by monitoring the in-memory compressed log size, and flushing this to disk when it reaches a configurable threshold (typically 20MB to 50MB). The daemon will also periodically flush logs to disk to ensure reliable log collection in the event of agent or system failure. These logs are uploaded to a central server using a standard SMB network file system protocol. If a failure occurs during upload the daemon will save the log locally and periodically retry the upload.

The daemon also manages its own operation, for example, by automatically update its binaries and configuration settings when indicated on the central server, and monitoring its disk space and memory usage. Setting up FDR tracing on a new machine is simple: a user only needs to run a single binary on the machine and configure the log upload location.

3.3 FDR Collection Server

The collection server is responsible for organizing FDR log files as they are uploaded, triggering relevant query tools to analyze the files as they arrive, and pruning old log files from the archive. It also sets the appropriate access privileges and security on the collected files and processed data.

3.4 FDR Query Tools

The final pieces of our framework are the query tools that analyze log files as they arrive. Each query tool is specialized to answer a specific type of query for a systems management task. Simple example queries include "what files were modified today?", or "which programs depend on this configuration setting?" As all our log files are read-only, we do not require complicated transactional semantics or other coordination between our query tools. Each query tool reads the log files it is interested in scanning and implements its own query plan against the data within. While future work might investigate benefits of caching, sharing intermediate results across multiple concurrent queries, or other optimization techniques from the database literature, we found that allowing uncoordinated reads simplified the process of building new query tools as required.

4. Designing the Log Format

The key requirements we have for FDR's log format are that 1) logs are compact, so that their size does not

[2] A kernel-mode boot driver is the first code to run after booting and the last to stop if the system is shut down.

overly burden client resources, network bandwidth or server-side scalability; and 2) the log format efficiently supports common-case queries. To meet these requirements, we built a preliminary version of FDR with a straightforward, flat format, and collected 5000 machine-days of traces from a wide variety of machines. We can personally attest to the difficulty of collecting, storing and analyzing this scale of data without support for compression and queryability. Based on our analysis of these traces, and a survey of how previous work applies such traces to systems management tasks, we designed an optimized log file format that takes advantage of three aspects of PS interaction workloads that we saw across our collected traces.

First, most PS interactions repeat many times during a day—93-99% of daily activity is a duplicate of an earlier event. For queries that care only about what happened, rather than when or how often, we can improve query performance by separating the definitions of this small number of distinct interactions from the details of when they occur.

Secondly, we observe that PS interactions are highly bursty, with many interactions occurring almost simultaneously and long idle periods between bursts. This allows us to save significant storage space by amortizing timestamp information across a burst.

Finally, we find that sequences of PS interactions are also highly repetitious; if we see a sequence of PS reads and writes, we are very likely to see the same sequence again in the future. This leads us to apply standard compression schemes to the time-ordered traces of PS interactions, achieving a high compression rate.

In the rest of this section, we describe relevant attributes of common-case queries, present the results and implications of our survey of PS interaction traces, and then describe the details of our log format.

4.1 Common Queries

Today, systems administrators deal with large-scale, complicated systems. According to surveys [9,28,33,36], an average Windows machine has 70k files and 200k registry settings. Faced with the task of managing these systems, a systems administrator's job is often a problem of "finding the needle in the haystack." For example, troubleshooting is the task of finding the few configuration settings or program files that are causing a problem; and to test a software upgrade or patch, the administrator needs to know what subset of the system might be affected by the change. To be useful, FDR must help systems administrators quickly identify the small set of relevant state and events out of all the state existing and events occurring across the many machines of a computing or IT environment. We describe the details of how systems management tasks use PS interaction traces in Section 6. Here, we briefly describe the aspects of common-case queries that informed our log format design.

We find that most common systems management queries of PS interaction traces search for a subset of events, identified by the responsible process or user, the file or registry entry being accessed, or another aspect of the interaction ("Who changed this configuration?" or "What did I change yesterday?"). This means that, by organizing or indexing our log format around such attributes, we can quickly identify the subset of interactions of interest. Common queries are also often restricted by time range, looking only at events that occurred during a specific period, implying that our logs should support random access over time, not just sequential access.

Many systems management tasks only involve the existence (or absence) of a particular PS interaction, and not *when* or *how often* the interaction occurred. For example, finding all loads of a shared library, regardless of when they occurred, can identify the processes that depend on that library and help assess the impact of a software upgrade. Other times, queries do care about *when* a PS interaction occurred, but only need to know an interaction's relative-ordering vis-à-vis other PS interactions on a given thread, *e.g.*, to determine potential causalities like loading a binary after reading its name from the Registry. In both cases, the implication is that some queries need not read timestamps at all.

4.2 PS Workloads and Log Optimizations

For our survey, we monitored the PS interactions of over 324 machines during one year across a variety of computing environments and collected over 5000 machine-days of PS interactions in total. We worked with MSN to instrument 207 of their machines, across 4 different services with different workloads, including CPU-bound systems with heavy disk workloads, a large storage service for external users, and web notifications publish/subscribe service. In our own research lab, we monitored 72 laboratory machines used for various data collection, analysis and simulation experiments. We also monitored 35 corporate desktops and laptops, used by researchers and engineers, primarily for activities such as software development and word processing. Finally, we monitored 7 home machines, used for entertainment and work-related activities by researchers, engineers, and their families. As a control, we also collected traces from 3 idle systems, running within virtual machines with no user workload.

4.2.1 Scale and repeated interactions

The primary challenge to efficiently tracing the PS interactions of a machine is the volume of events that occur. In our survey, we found that the average number of daily PS interactions was $O(10^7)$ ranging from 9M on desktop machines to 70M on the busiest workloads,

Table 2: The average per machine daily total and distinct interactions, entries, and processes

Environment	Total PS Interactions	Distinct PS Interactions	Distinct PS Entries Accessed	Distinct Processes
Svc. 1	70M	0.2%	>0.1%	40-60
Svc. 4	29M	3.3%	0.4%	30-70
Svc. 2	22M	0.6%	0.1%	30
Svc. 3	19M	1.1%	0.1%	30-70
Home	17M	4.2%	0.6%	30-40
Desktop	9M	5.4%	1.0%	20-100
Lab	9M	1.5%	0.3%	17-40
Average	**25M**	**1.6%**	**0.2%**	**43**
Idle	*965K*	*2.1%*	*0.5%*	*14*

as shown in Table 2. Not surprisingly, servers tended to have a stable workload from day-to-day, while our lab, corporate desktop and home machines had varied workloads. The highest number of daily interactions we saw was 264M events, on an MSN server that collected application logs from 1000s of other machines.

However, we found several factors that mitigate the large volume of PS interactions in all these workloads. First, the number of distinct files and registry entries read or written every day is much smaller than the total number of interactions. Secondly, the number of distinct processes that run on each machine is very small, O(10^2) processes on the busiest desktops, and fewer on production servers. Overall, we found that most PS entries are only accessed by a small number of processes, and that the total number of distinct interactions (*i.e.*, distinct <user, process, operation-type, PS entry> tuples) was O(10^5), only 0.2% to 5.4% of the total interactions per day.

This implies that we can improve the performance of queries not interested in timestamps or counts of PS interaction occurrences by separating the unique definitions of observed interactions from the time-ordered traces of when they occur. Effectively, this allows many common queries to ignore 94.6-99.8% of the log. This also provides the possibility of compressing our logs, by replacing repeated descriptions of an interaction with a single unique ID.

4.2.2 Bursts of Activity

Several studies of I/O traffic and file system activities have shown that server and desktop I/O workloads demonstrate bursty or self-similar behavior [14,16]. We observe this in our traces as well, where it manifests as many interactions arriving together with long idle periods in between.

The primary implication of these bursts for our log format is that, when many events occur together, there is a clear opportunity to merge their associated time information, storing a single timestamp for all the events that occur during the same timestamp bucket. This is a significant observation because per-event timestamps are a major limiting factor to achieving high compression rates. To help us choose an appropriate bucket duration, we look to the requirements of common-case systems management queries. We find that fine-grained timestamps are rarely necessary, instead what is most important is the relative ordering of events and the ability to map event occurrences to human activities (*i.e.,* wall-clock time). This leads us to choose a relatively coarse-grained 48-bit or 6ms granularity timestamp. Note that this still provides a granularity finer than Windows' time-scheduling quantum of 10-15ms. While one might worry that a coarse-grained timestamp would mean every bucket would have at least one event in it, in practice, even our busiest observed machine-day, with 264M daily events, showed no PS interactions during 60% of its timestamp buckets. Of course, this does not mean the machine as a whole was idle—it might have been busy with CPU calculations during the times it was not doing PS interactions.

4.2.3 Repeated Sequences of Interactions

Our final key observation is that many sequences of PS interactions repeat over time. This is not a surprise, as we would expect that most of the file system and registry activities performed by a system are standard, repetitive tasks, including process start-up and shutdown, background activities, document auto-saves, and logging. We perform a detailed analysis of repeating "activity bursts," in [32] and, for space considerations, provide only a summary here.

In our analysis in [32], we define an "activity burst" as the set of PS interactions occurring in one thread, where each interaction occurs no more than some small time separation apart. Formally, we define an activity burst as a group of events $\{e_t | i \quad t \quad j\}$ occurring within a single thread, where $gap(e_t,e_{t+1}) < k$, for all $i \quad t < j$; $gap(e_{i-1},e_i) \quad k$; $gap(e_j,e_{j+1}) \quad k$; $gap(x,y)$ is the time between two interactions x and y; and k is the threshold gap between bursts. We call an activity burst a "repeat" if it is identical to an earlier activity burst in every attribute of its interactions except for timestamps. Otherwise, we call it a "distinct" burst. In our survey, we find that most activity bursts in a day are repeated bursts. On desktops, we see 2K-5K distinct bursts out of 20K-40K total and, on servers, we see 3K-4K distinct bursts out of 40K-70K.

This repetition of PS interaction sequences indicates that simple byte compression schemes, applied to the time-ordered event sequences, should detect and compress these repeating patterns. Since our analysis of activity burst repetition focuses on bursts within a single-thread, storing PS interactions in a system-wide timestamp sequence runs the risk of allowing

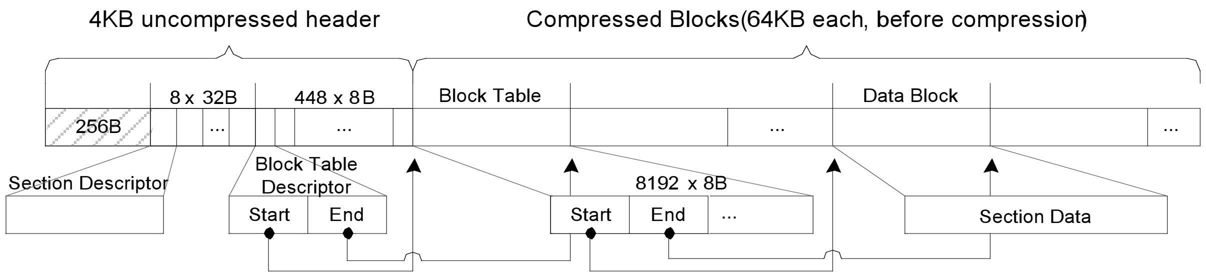

Figure 2: The physical layout of our log format

concurrent I/O from multiple threads to interfere with the compressibility of each other's patterns. However, because of the relatively large CPU time slice of 10-15ms on the Windows OS, and the knowledge that most PS interactions are handled quickly by file caches, we

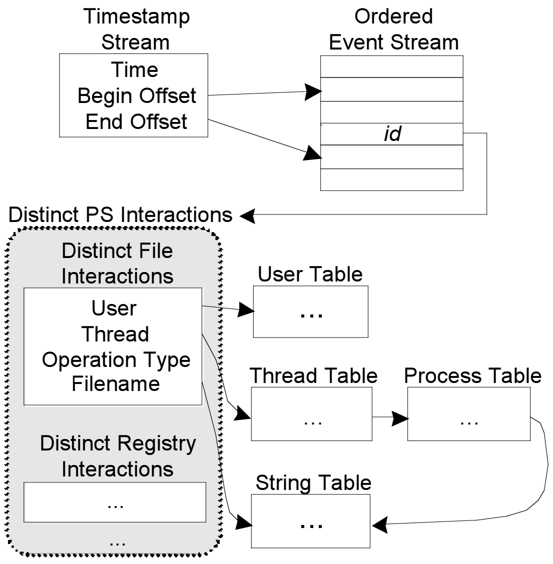

Figure 1: Logical design of our log format

still expect to gain significant compression of repeating patterns in a cross-thread trace of PS interactions.

4.3 Log Format Details

Based on the machines observed in our survey, our log format has two key facets to its logical design, shown in Figure 1. First, given the repetition of accessed files, observed processes, etc., we normalize the attributes of our persistent state interactions into separate sections, essentially following a standard procedure of database normalization.[3] We create one section for distinct PS interactions, which point to other sections containing distinct names, user context, process information, file data hashes and values. The primary benefit of this normalization is a reduction of repetitive information. In addition, we find that grouping attributes into their own sections improves the performance of byte compression algorithms as we compress these log sections later. This separation also improves query efficiency by transforming queries that involve multiple attributes and expensive string comparisons into inexpensive comparisons of integer IDs as we scan through traces of PS interactions.

Our second design choice in our logical log format is to represent the trace of events itself as two parallel, but connected, streams of data. The first stream is an ordered list of events as they are captured by our kernel driver and reported to the user daemon. The second stream contains timestamp information for groups of events. This amortizes the size of timestamp information across multiple events, reducing the overall size of the log, as well as improving byte compression of the event stream by better exposing patterns in the sequences of events. Both logical streams are stored in a single physical file to ease management of log archives.

We created a file structure that contains a large logical address space split into 32 sections. Each of the normalized attribute sections, the section of distinct PS interactions, as well as the ordered event stream and timestamp stream, are mapped to a section in our log file. Each section is composed of individually compressed 64k pages. Compressing in blocks, rather than using a streaming compression format allows random access within a data section.

To simultaneously optimize our log for random access and compression, our physical log file layout consists of a three-layer addressing scheme of block table, block number, and block offset, shown in Figure 2. This three-layer addressing scheme is important because we want to compress individual blocks and store the start and end offsets of each block in a table for fast lookup, and as a further optimization, compress these tables as well. With 448 block tables, 8192 blocks per table and a 64k uncompressed block size, this provides a maximum addressable storage size of 234 GB of uncompressed data within each log file. While we find this is many times more than a single machine-day of logs, this format gives us the flexibility of joining many days of logs into a single file for

[3] Database normalization is a process of organizing data to eliminate redundancy and reduce potential for error (33).

improved compression, and gives us flexibility if PS interaction workloads grow in the future.

Each log file starts with a 4k uncompressed header. The first 256 bytes consist of versioning and other miscellaneous information. Next are 32 section descriptions, each 8 bytes long. Each of the logical sections, described earlier, is laid out contiguously over our three-layer addressing scheme, aligned at block boundaries. These section descriptors provide the block table and number of the start and end of each section. The rest of the 4k header is filled by 448 block table descriptors, that point to the start and end offsets of a compressed block table. The block table, in turn, contains 8192 block entries, each pointing to the start and end offset of a compressed 64k block.

The timestamp section is maintained as a 16 byte entry containing 6 bytes (48 bits) to represent a 6ms time resolution, 2 bytes to count missing events within that region, and two 4 byte offsets pointing to the first and last consecutive event with that time resolution. While almost all events are received by the daemon in time sorted order, we correctly handle timestamp information for any events that appear out of order. This can happen when a context switch occurs just after an I/O activity completes, but before the kernel driver reports it, but this delays the reporting of an event by a few scheduling quantums, and never affect the intra-thread ordering of PS interactions.

The user daemon first creates log files in-memory. As it receives raw events from the kernel driver, the daemon normalizes the events, replacing attribute values with indexes into the appropriate sections. The normalized event is then compared to the table of distinct normalized events using an O(1) hash lookup, and added to the table if necessary. Finally, the normalized event is added to the ordered event stream section, along with new timestamp information, if necessary. Each of the log file's data sections is append-only in memory. When a log file is closed and flushed to disk the daemon writes each data section contiguously to disk while applying a standard compression algorithm to each 64K-byte block.

4.4 Querying Logs

When analyzing these log files, our tools tend to restrict their queries based on one or more attributes in the PS interaction record, based on a time-range of interest, or based on both. To restrict a query by attribute, a query tool scans the appropriate section, looking for all values matching the given criteria. From these values, the tool then generates a filter to apply against the section of distinct PS interactions, resulting in a set of unique IDs, one for each PS interaction matching the original attribute restriction. For example, to return only PS interactions that access a particular file, a tool would first scan the string section to find the ID of the filename, and then scan the section of distinct PS interactions to find the IDs of all distinct PS interactions that accessed this filename ID. If a tool is not interested in when or how many times an interaction occurred then it can stop here, without scanning the much larger event stream sections. Otherwise, the tool can scan through the ordered event list and timestamp stream to find the details of the occurrences of these PS interactions.

To restrict a query by a time range, a query tool applies a binary search to the timestamp stream, searching for the start of the desired time range. Once this timestamp is found, it can skip to the appropriate 64K block of the ordered list of events, and begin scanning the ordered list from that point on, until the end of the time range.

Common-case queries tend to extract sparse information from the extremely large data set of PS interactions. Our log format enables efficient queries by allowing query tools to focus on the relevant subsets of data, and expanding their scope to larger and larger portions of the data as necessary. For example, a query to find a list of all files modified during a day of 25M PS interactions requires only one pass over a distinct event table with 318Kentries to identify the string attribute id of modified files, and then scanning over the string attribute section with 100K entries to discover the full filenames of each modified file, avoiding ever scanning the full list of 25M events that occurred during the day.

5. Experimental Evaluation

In this section, we evaluate FDR and find that on average, our logs use only 0.7 bytes/PS interaction. We find that query performance scales linearly with the number of events in a log. All of our queries can be processed in a single pass against an entire machine-day of logs in just 3.2 seconds. FDR's client-side performance overhead is less than 1%, and calculating the load on bottleneck resources in our central server indicates that a single machine could scale to collecting and analyzing all the PS interactions from 4300 machines, keeping all logs for over 3 months.

5.1 Log File Compression

The efficiency of FDR's log file format affects both the runtime load on clients' memory and network, and the long-term storage costs of PS interaction logs.

In our survey of 5000 machine-days of logs, described in Section 4.2, the average raw event size is 140 bytes and daily per machine raw logs are 7GB, compressing to 700MB with GZIP. After converting our collected logs to our new format, we found, depending on workload, each PS interaction takes between 0.5 to 0.9 bytes of storage, and 0.7 bytes on average. One machine-day of PS interactions can be stored in 6 to 71MB, 20MB on average, with a

Table 3: Daily storage requirements for machines across environments

Role	Avg. Bytes/Event	Avg. MB/day	Max MB/day
Svc. 1	0.91	71	179
Svc. 4	0.78	57	103
Svc. 2	0.71	19	22
Svc. 3	0.66	10	53
Home	0.58	17	51
Desktop	0.80	13	21
Lab	0.47	6	43
Average	**0.70**	**20**	**49**
Idle	*0.85*	*0.76*	*0.88*

maximum observed machine-day size of 179MB. Table 3 shows the results across our environments.

In addition to PS interaction workload, the efficiency of our log format's compression is sensitive to the frequency at which the logs are uploaded to the central server. More frequent uploads of logs reduces the latency between when an event occurs and when it can be analyzed. However, uploading logs less frequently allows the logs to collect more repeated events and achieve better overall compression.

Figure 3 shows how storage size varies with the event collection period. It shows that 80% of the compression efficiency is typically achieved with log files an hour long, and that improvement ceases after 1 week of data. Based on this result, our current deployments upload log files every 2 hours, and our central server reprocesses and merges them into 1-week long log files to save long-term storage space. We envision that future versions of FDR will support latency-critical queries by moving them to the client-side agent.

In Table 4, we look inside the log files to see how much space is taken up by each section of our log file format across our different environments. We see that the ordered event stream and timestamps dominate,

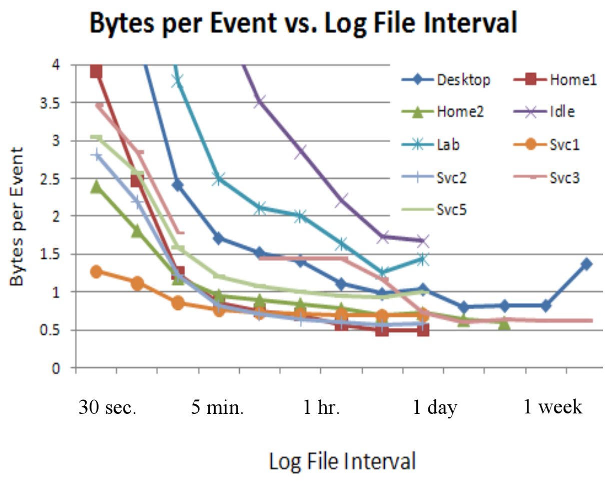

Figure 3: Compression Ratio variation with log file interval for each category of machines

Table 4: Average section size as a percentage of total log file size for each machine role

Role	Event	Time	Reg.	File	String	Data	Other
Svc 1	77%	19%	1%	0.3%	0.2%	1.9%	0.6%
Svc 4	57%	17%	7%	13.9%	1.8%	3.2%	0.1%
Svc 2	71%	15%	4%	1.4%	0.6%	7.9%	0.1%
Svc 3	69%	15%	7%	1.8%	0.9%	2.7%	3.6%
Home	46%	27%	9%	11%	4.0%	1.9%	1.1%
Desktop	47%	19%	14%	8.2%	5.2%	5.7%	0.9%
Lab	48%	31%	8%	3.5%	1.9%	7.0%	0.6%
Average	**53%**	**22%**	**9%**	**7.7%**	**2.8%**	**4.1%**	**1.4%**
Idle	*44%*	*35%*	*7%*	*4.2%*	*4.8%*	*3.9%*	*1.5%*

together taking 66-96% of the daily logs of non-idle machines. The definitions of distinct file and registry interactions are the next major contributor to log file size, taking 1.3-22.2% of daily logs from non-idle machines. Surprisingly, storing the values of every registry setting read or written (in the data section) takes only 1.9-7% of daily logs of non-idle machines.

5.2 Log File Query Performance

To evaluate FDR's query performance, we performed three types of queries against our collected daily machine logs: a single-pass query, a two-pass query, and a more complicated multi-pass query. Our first query is a single-pass scan of distinct PS interactions, and generates a *manifest* that identifies the distinct set of registry settings and files used by every process during the day. This query does not require scanning the time-ordered event streams. Our second query is a two-pass scan that searches for *stale binaries* (discussed in Section 6.1.2). This query scans through all PS interactions for files loaded for execution, and then scans the time-ordered event stream to see if the file has been modified on-disk since it was last loaded into memory, indicating that a stale copy exists in memory. Our third query looks for *Extensibility Points* (discussed in Section 6.2). This is our most complicated query, making multiple scans of the distinct PS interactions and time-ordered event stream.

Query performance was measured by evaluating our three queries against log files that ranged from 200k to 200M PS interactions across all categories of machines. For this experiment we used a single 3.2GHz processor. We found that the average machine-day can be queried in only 3.2 to 19.2 seconds, depending on query complexity. Figure 4 plots the count of items in scanned sections vs. the time to complete each query, and indicates that performance scales linearly with log size.

We found that our query performance was not I/O bound reading log files from disk, but rather CPU-bound, on decompressing log files. In fact, we found

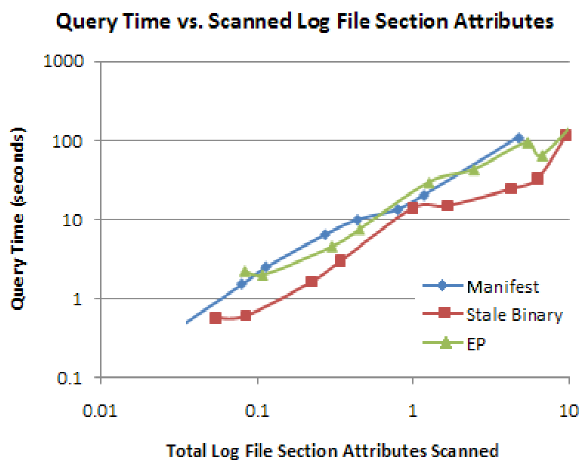

Figure 4: Time taken for each query compared with the number of attributes scanned

that query performance can be accurately predicted as a linear function of the number of items per section in the log file scanned by the query and the Intel Pentium CPU cycle cost to process each item. We measured the average per item CPU cycle cost to be 63k cycles for manifest queries, 27k for stale binary queries and 53k for extensibility point queries. Using Pearson's product moment correlation to compare our predicted and measured query times, we find the correlation to be from 0.923 to 0.998, indicating that query performance is a linear function of the size of the log items scanned.

5.3 Client-Side Overhead

Here, we evaluate the tracing and compression overhead of our kernel driver and user-mode agent. Through our initial survey of PS interactions, we have tested and deployed our driver and our user-mode agent without our compression tools on over 324 Windows 2000, Windows XP and Windows 2003 machines. At MSN, pre-production testing of our data collector was done in a lab setup of 4 identical servers, 1 running our agent, each receiving a copy of the current live production load. Measurements were made of volume and latency of workload transactions along with memory, network, CPU, and I/O overhead. The performance impact of our driver and agent was minimal, with < 1% CPU overhead measured, and no measurable degradation in transaction rate or latency. To further confirm this, we conducted an experiment where we placed a high and varied PS interaction workload, consisting of simultaneously creating 100,000 processes, scanning the entire registry and file system for non-existent entries, and making 10,000 copies of a 1KB file, 35 copies of a 1GB file, and 100,000 registry keys and values. Even under this load, we found no measurable performance degradation.

The next stage of our evaluation of agent performance focuses on the overhead of log format generation and compression. Because we configure our agent to run on a low-priority thread, we have not observed a noticeable performance impact, but do want to understand its limits. Therefore, we measure the CPU cost of processing a single PS interaction, and use this to evaluate the cost of processing our observed PS interaction rates.

Our measured CPU cost to process a PS interaction is, on average, 64k CPU cycles. This cost is constant with regard to the number of events already processed. Our highest observed spike from our collected data is 2400 interactions per second. The average peak burst every day is 1800 interactions per second. From this, we extrapolate that the highest 1 second CPU usage spike on a 3.2 GHz CPU is 64k x 2400 / 3.2 GHz = 4.8% CPU overhead. Our average peak is 3.6% CPU overhead. As a further test case, we created an artificially high PS interaction load by repeatedly accessing cached PS data, without pause, generating a rate of 15k interactions / second. Compressing these interactions at this rate produces 30% CPU overhead. Our average rate of 100-800 PS interactions per second requires only 0.2 – 1.6% CPU overhead.

5.4 Server Scalability

The scalability of a single-machine FDR server collecting and analyzing PS interaction logs from many other machines is potentially limited by several factors: network bandwidth for receiving logs, disk I/O bandwidth for storing and reading logs, CPU cost for analyzing logs, and the disk capacity for storing logs.

The single-machine configuration we consider is the one we use to collect logs from our agents today. It is a dual 3.2GHz CPU, 2GB of RAM, a 1Gbps network connection, and a 24 hard drives (400GB 7200RPM SATA) in two 12 disk RAID 5 sets with 1 parity drive, providing 8TB of storage. Assuming an average 20MB log file per machine-day, and using the performance of this system, we consider each of the potential scalability bottlenecks:

Network bandwidth: A 1Gbps network link, with an achieved bandwidth of 100 Mbps, could support 54,000 machines uploading per day.

Disk I/O bandwidth: Our RAID storage system provides 80 Mbps random access bandwidth. At this rate, we could support both writing logs and a single-pass query at a rate of 43,200 machines per day.

CPU for analysis: Following the analysis of query cost in Section 5.2, our dual processor 3.2GHz machine can support querying up to 54,000 machines per day, at a rate of 1.6 seconds per machine-day (3.2s per CPU).

From this analysis, not counting long-term storage requirements, the limiting factor to scalability appears to be disk bandwidth, supporting the centralized collection of data from 43,200 machines. Of course, this is not a sophisticated analysis, and there are likely to be issues and interactions which might further limit the scalability of a single-machine log collector. For this reason, we apply a safety factor of 10 to our analysis, and claim that FDR can allow a single server to provide centralized collection and analysis of the complete PS interaction logs of up to 4,300 machines.

Separately, analyzing the storage capacity of our single-machine server, we find that our 8TB RAID system can store a total of 400k 20MB machine-days. This would store the complete PS interaction logs of 13,000 machine for 1 month, 4,000 machines for over 3 months, or 1000 machines for 1 year.

6. Using FDR for Systems Management

In this section, we first review our use of early on-demand tracing prototypes to attack various systems management problems, then present one new case study in detail. Throughout, we describe how each technique is improved by one or more of FDR's benefits:

Completeness: FDR gathers a complete record of reads, writes, creations, etc. to the file system and registry, including details of the running process, user account and, when appropriate, data hashes, values, etc.

Always-on: FDR's always-on tracing means that users do not have to anticipate when they might need tracing information, do not need to reproduce problems to obtain traces, and enables analysis of long-term trends.

Collection and Query Scalability: FDR's scalability eases cross-machine analysis, such as PeerPressure [35], and allows administrators to centrally apply current PS analysis techniques rigorously and en masse to large computing and IT systems.

6.1 Management Scenarios

6.1.1 Troubleshooting Misconfigurations

When a problem, like a misconfiguration, happens, troubleshooting is the task of determining what has gone wrong and fixing it. The Strider Troubleshooter [36] used a precursor to FDR, called AppTracer, to capture on-demand traces of a program's registry interactions. Once a user notices a program error, they turn on AppTracer and reproduce the problem. Strider then asks the user when the program last worked, and uses this date to find a "known-good" Windows System Restore snapshot. Strider then searches for registry settings used by the program that have changed since the known-good snapshot. With some noise filtering and ranking heuristics, Strider produces a short list of settings likely responsible for the problem.

PeerPressure [35] improves on Strider by using knowledge of the configuration settings on other machines, stored in a central database. By assuming that most other machines are configured correctly, PeerPressure removes the need for users to identify the last known-good state.

Both Strider and PeerPressure suffer from similar limitations due to their use of an on-demand AppTracer. First, both tools require users to reproduce a problem so that the AppTracer can collect a trace—hard to accomplish if an error appears only transiently or is otherwise hard to reproduce. An always-on tracer will already have captured the trace at the time of the original problem. Secondly, both tools require the user to guess which process is failing and should be traced and the user must know to iteratively expand the scope of the on-demand tracer to debug a cross-application problem, where one process fails because of an error in another (*e.g.*, a word processor acting as an editor for an e-mail reader). An always-on tracer will already have captured traces of all the processes on a system, obviating the user's need to guess what part of the system to trace. Finally, [35] states that updating PeerPressure's central database of machine configurations as software and operating systems are upgraded is an open challenge. With the scalable log collection provided by FDR, collecting descriptions of new configurations and software to insert into PeerPressure's central database is trivial.

Furthermore, FDR's always-on tracer improves on Strider and PeerPressure's troubleshooting in a fundamental way: whereas these previous tools are only able to locate the misconfiguration, FDR's history of PS interactions can also help place responsibility for a misconfiguration by determining when and how the misconfiguration occurred. This can help identify the root cause of the issue and prevent future occurrences.

6.1.2 Detecting Known Problems

In addition to reactive trobuleshooting, we can use always-on tracing to proactively search for specific, known problems, such as common misconfigurations and old versions of software with known vulnerabilities. One common problem, the "stale binary problem," occurs when software upgrades fail to restart affected processes or reboot a machine after replacing its on-disk binaries. The result is that the system is continuing to execute the old program in-memory. This is an especially serious problem when patching security vulnerabilities. With complete, always-on tracing, we can periodically query for the last load-time of running programs and DLLs, and compare them to their last modification-time on disk, reporting inconsistencies to system administrators.

6.1.3 Change Management: Impact Analysis

Keeping systems up-to-date with the latest security and bug patches is critical for minimizing vulnerability to malicious adversaries, and loss of productivity to software bugs. At the same time, patches can destabilize existing applications. Today, unfortunately, even if a patch only updates a single shared library, the administrators do not know in advance which applications might be affected. Consequently, patch deployment is often delayed as it undergoes a lengthy (and expensive) testing cycle, and computer systems remain vulnerable to "fixed" bugs and security holes.

To help administrators focus their testing of software upgrades, we built the Update Impact Analyzer (UIA) [10] that cross-references the files and configuration settings being patched against always-on traces of PS interactions. The UIA generates a list of programs that interact with any state that is going to be updated. Any application not on this list can be placed at a lower-priority on the testing regimen. (An exception is when a given application interacts with an updated application via inter-process communication—in this case, both applications could still require thorough testing. See Section 7.3 for a discussion of other possible candidates for always-on logging, including inter-process communication and locks.)

A primary challenge faced by UIA, as reported in [10], is that patch testing and deployment is managed centrally by administrators, but application usage, for determining the dependencies between an application and various files and settings, is distributed across many computers. FDR's scalable tracing and collection of PS interactions enables administrators to easily gather the accurate information they need.

6.1.4 Malware Mitigation

The challenges to mitigating a malware infection, whether spyware, Trojan software, viruses or worms, are detecting its existence on a system and determining how the malware entered the system. With always-on PS interaction traces, identifying running malware is a matter of querying for the hashes of loaded executables and shared libraries. Any well-known malware signatures can be flagged, and unrecognized hashes can be reported to an administrator to determine whether or not they are malicious. Following the methodology of [17], always-on tracing of PS interactions can also be analyzed to discover how malware entered a system.

To further backtrack the "route of entry" of malware, the HoneyMonkey (HM) project analyzes the PS interaction traces collected with FDR of web browsers as they visit many websites [37], Using a farm of virtual machines running scripted web browsers, HM crawls the Internet. If HM notices a web browser writing to the file system outside of its browser sandbox (*e.g.*, writes other than temporary cache files), then it can be assured that a malicious website is exploiting a browser vulnerability to install malware. SpyCrawler, a concurrent research project, used a similar system to detect malicious websites [23]. Without FDR's detailed trace information stating which processes where making the changes, their browser monitoring system had a high false-positive rate for detecting exploits, reduced by using antivirus tools to check for known malware. New malware would not be detected.

FDR's log collection can be modified in several ways to harden it against malicious adversaries. Many malicious software programs, such as spyware bundled with otherwise legitimate downloaded software, must first be written to disk before executing. We can prevent these programs from tampering with FDR's logs after-the-fact by adding tamper-evident hash-chaining signatures [18] to our logs or by moving our user agent to a hypervisor or VM outside the accessibility of the monitored system. Malicious software that enters a system directly (*e.g.*, via a remote buffer overflow exploit) could corrupt our kernel driver before writing to disk. To avoid this attack, the file system itself would have to be moved to a separate VM or hypervisor. Detecting malware that never interacts with the disk is outside of FDR's scope.

6.2 Case Study: Exploiting SW Extensibility

Once spyware or other malware first infects a system, they often load themselves as a plug-in or extension to the operating system, daemon, or frequently used applications such as a web browser, ensuring their continued execution on the host system. One method to defend against such malware is to monitor the settings or *extensibility points* (EPs) which control software extensions, alerting the user to changes that might signify a malware installation.

By comparing snapshots of the Windows Registry before and after 120 different malware installations, GateKeeper [38] found 34 EPs that should be monitored for signs of malicious activity. Here, we show how we can take advantage of always-on tracing to detect *potential* malware-exploitable settings, even if they are currently used only by benign software extensions. Further, we show how PS interaction traces can help rank the importance of these EPs, based on the observed privileges and lifetimes of the processes that use these extensions.

6.2.1 Detecting Extensibility Points

To discover EPs we monitor application PS interactions for files being loaded for execution[4], and check for a previous PS interaction which contained the executable's filename. If we find such a setting, we assume that it directly triggered the executable file load and mark the setting as a *direct extensibility point*. In some cases, if we continue to search backward through the history of PS interactions, we will also find *indirect extensibility points*, where another configuration setting triggers the process to read the direct EP. For example, an indirect EP may reference an ActiveX class identifier that points to a COM[5] object's settings that contain the executable file name.

Many EPs have a similar name prefix, indicating that plug-ins using it follow a standard design pattern. We

[4] Our PS interaction tracing records the loading of a file for execution as a distinct activity from simply reading a file into memory.

[5] `Component Object Model' is a Microsoft standard for reusing and sharing software components across applications.

define a common EP name prefix as an *extensibility point class*, and the fully named EP as an *extensibility point instance*. We identify new EP classes by manually examining newly discovered EP instances that do not match an existing EP class.

To survey how significant a vulnerability EPs are, we processed 912 machine-days of traces from 53 home, desktop, and server machines. From these machines, we discovered 364 EP classes and 7227 EP instances. 6526 EP instances were direct and 701 were indirect. While 130 EP classes had only 1 instance, 28 had more than 20 unique instances. The dominant EP class consists of COM objects, and accounts for 40% of all EP instances. The next two largest EP classes are associated with the Windows desktop environment, and web browser plug-ins. Other popular EP classes are related to an Office productivity suite and a development environment, both of which support rich extensibility features. Overall, we found that 67% of the software programs observed in our traces accessed an EP instance, and those that did used 7 on average. `Explorer.exe`, responsible for the Windows desktop, used the largest number of EP classes (133 EP Classes), followed by a web browser (105 EP Classes) and an email client (73 EP Classes).

Comparing our list of EP classes with the 34 discovered by Gatekeeper, we found that our procedure detected all except 6 EPs used by programs not observed in our traces.

6.2.2 *Criticality of Extensibility Points*

The criticality of an EP can be estimated using 1) the privilege-level of the loading process, where higher-privilege processes such as operating system or administrator-level processes, are more critical; and 2) the lifetime of the loading process, where longer running applications provide higher availability for a malicious extension. We observed that on average a machine will have at least a third of EP instances (spanning 210 EP classes) loaded by processes that are running for 95% of the machine's uptime. We also observed that one third of all EP instances were used by processes with elevated privileges. This indicates that many EP instances are critical security hazards.

6.2.3 *Lessons and Suggestions*

This case study shows how FDR's traces of PS interactions can be analyzed to connect the security-sensitive behavior of loading dynamic code modules back to the critical configuration settings which control its behavior, and furthermore rank the criticality of each setting. Since this analysis requires that an EP be in use, whether by malware or by a benign software extension, FDR's scalability and always-on tracing is critical to analyzing a wide-breadth of computer usage and detecting as many EPs as possible.

Once we have discovered these EPs, we can continue to analyze their use and suggest ways to mitigate the threat from EP exposure. In particular, we observed that 44% of EP instances were not modified during our monitoring period. This suggests that system administrators could restrict write permissions on these EPs, or that application designers could transform them into static data instead of a configurable setting. Also, 70% of all EP instances were used by only single process: an opportunity for administrators or application designers to lockdown these EPs to prevent their misuse. In all, we found that only 19% of EP instances were both modified and shared by multiple applications, and thus not easy candidates for removal or lockdown. For these remaining EPs, we suggest monitoring for suspicious activities and, for critical EPs, we suggest that administrators and developers audit their usefulness vs. their potential to be misused.

7. Discussion

In this section, we discuss the implications of our work on systems management techniques, as well as limitations and opportunities for future work.

7.1 White-box and Black-Box Knowledge

Software installers use manifests to track installed software and their required dependencies, anti-virus monitors use signatures of known malware, and configuration error checkers use rules to detect known signs of misconfigurations. All of these automated or semi-automated tools use explicit prior knowledge to focus on a narrow set of state and look either for known problematic state or checking for known good state. This approach is fragile in its reliance on the correctness and completeness of pre-determined information. This information can become stale over time, might not account for all software and cannot anticipate all failures. Keeping this information complete and up-to-date is hard because of the long-tail of third-party software, in-house applications, and the continuous development of new malware. With FDR-collected traces, a black-box approach to systems management can help by augmenting predetermined information with observed truth about system behavior.

For example, today's software installers commonly fail to account for program-specific PS created post-installation, such as log files and post-install plug-ins, as well as interruptions or failures during installation or removal. These mistakes accumulate and lead to system problems.[6] Common advice is to occasionally reinstall your computer to clean up such corruptions. Black-box tracing helps avoid this by providing installers with complete, ground-truth about installed files.

To test this approach, we compared a black-box accounting of files created during installations of 3 popular applications to the files accounted for in their

[6] Examples of failed installations causing such problems can be found at http://support.microsoft.com/ via the article IDs: 898582, 816598, 239291, and 810932.

explicit manifests. By collecting FDR traces while installing the application, using it, and then uninstalling it, we measured the file and registry entries leaked on the system. The first application, Microsoft Office 2003, leaked no files, but did leave 1490 registry entries, and an additional 129 registry entries for each user that ran Office while it was installed. The second application, the game 'Doom3', leaked 9 files and 418 registry entries. Finally, the enterprise database Microsoft SQL Server leaked 57 files and 6 registry entries. These point examples validate our belief that predetermined information can be unreliable and that black-box analysis of FDR traces provides a more complete accounting of system behavior. Predetermined information does have its uses, however. For example, *a priori* knowledge can express our expectations of software behavior [27] and higher-level semantics than can be provided by a black-box analysis.

7.2 State Semantics

One of the limitations of a black-box tracing approach is that, while it can provide complete, low-level ground truth, it cannot provide any semantic guidance about the meaning of the observed activities. For example, FDR cannot tell whether the Windows Registry editor program (RegEdit) is reading a registry entry as a configuration setting to affect its own behavior, or as mere data for display. Similarly, FDR cannot tell whether any given file on disk is a temporary file, a user document, or a program binary (unless explicitly loaded for execution). Investigating how to best augment low-level tracing with heuristics and semantic, white-box knowledge is an important topic for continuing to improve systems management techniques. One option, discussed below, is to selectively log application-level events and information to expose the semantic context of lower-level PS interactions.

7.3 Logging higher-level events

The question we pose here, as a challenge for future work, is what classes of events, in addition to PS interactions, should be logged to help operators and administrators maintain reliable and secure systems?

One category, mentioned above, is the semantic context of PS interactions, such as the context we receive when software binaries must be explicitly loaded for execution. Perhaps we can receive similar context and benefit from knowing that the Windows Registry editor is reading configuration settings as data, and not to affect its own behavior. Similarly, explicitly recording whether a newly created file is a user document, temporary program state or a system file might help administrators improve backup strategies and debug problems.

A second category of higher-level events are those that help to track the provenance of data. While there has been research on how to explicitly track the providence of data, we might be able to gain some of the same benefit from simply logging a "breadcrumb" trail as new files are created. Even just integrating web browsing history with PS interactions would allow us to track the provenance of downloaded files and locate the source of malware installed via a browser exploit.

A third category of events that might benefit from FDR-style always-on logging are interactions and communications between processes, such as network connections and inter-process communication. While this category does not provide extra semantic information, these interactions are important for detecting software dependencies, fault propagation paths, and potential exposure to malware.

Altogether, extending always-on tracing to include more context and events could enable a gray-box approach to systems management, combining the benefits of black-box ground-truth and white-box semantic knowledge [2].

7.4 Using Traces to (Re) Design Programs

In this paper, we have focused on analyzing PS interactions to benefit systems administrators and operators as they attempt to understand the state of the systems they manage. However, these PS interactions might be just as useful, though on a different time-scale, for developers interested in improving the applications and systems they have built. One obvious benefit is when PS interactions expose an otherwise difficult-to-track software bug. We already discussed an analysis to detect "stale binaries" after software installations (a bug in the installer). Tracing PS interactions has uncovered other bugs in several server management programs as well. Other benefits to application designers can come from specific analyses of a system's reliability and security, such as our analysis of extensibility points in Section 6.2.

The bottom-line is that always-on tracing of PS interactions improves our understanding of a system's dynamic behavior in production environments, and understanding this behavior is the first step towards improving it.

8. Conclusion

We built FDR, an efficient and scalable system for tracing and collecting a complete, always-on audit of how all running processes read, write, and perform other actions on a system's persistent state, and for scalably analyzing the enormous volume of resultant data. Thus, FDR addresses significant limitations faced by prior work in using PS interactions to solve systems management problems. We achieved our goal by designing a domain-specific log format that exploits key aspects of common-case queries of persistent state interaction workload: the relatively small number of daily distinct interactions, the burstiness of interaction occurrences, and repeated sequences of interactions.

For the last 20 years, systems management has been more of a black-art than a science or engineering discipline because we had to assume that *we did not know* what was really happening on our computer systems. Now, with FDR's always-on tracing, scalable data collection and analysis, we believe that systems management in the next 20 years can assume that we *do know and can analyze* what is happening on every machine. We believe that this is a key step to removing the "black art" from systems management.

9. Bibliography

[1] W. Arbaugh, W. Fithen, and J. McHugh. Windows of Vulnerability: A Case Study Analysis. In *IEEE Computer,* Vol. 33(12)

[2] A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *SOSP*, Banff, Canada, 2001

[3] M. Baker, et al. Measurements of a Distributed File System. In *SOSP*, Pacific Grove, CA, 1991

[4] N. Brownlee and K. Claffy and E. Nemeth. DNS Measurements at a Root Server. In *Global Internet Symposium.* San Antonio, TX, 2001

[5] M. Burtscher. VPC3: A Fast and Effective Trace-Compression Algorithm. In *ACM SIGMETRICS.* New York, NY, 2004

[6] Z. Chen, J. Gehrke, F. Korn. Query Optimization In Compressed Database Systems. In *ACM SIGMOD.* Santa Barbara, CA, 2001

[7] E. A. Codd. A relational model for large shared databanks. *Communications of the ACM*, Vol. 13(6)

[8] B. Cornell, P. A. Dinda, and F. E. .Bustamente. Wayback: A User-level Versioning File System for Linux. In *Usenix Technical.* Boston, MA, 2004

[9] J. Douceur, and B. Bolosky. A Large-Scale Study of File-System Content. In *ACM SIGMETRICS.* Atlanta, GA, 1999

[10] J. Dunagan, et al. Towards a Self-Managing Software Patching Process Using Blacpk-box Persistent-state Manifests. In *ICAC.* New York, NY, 2004

[11] G. W. Dunlap, et al. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *OSDI,* Boston, MA, 2002

[12] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS Tracing of Email and Research Workloads. In *USENIX FAST*, San Francisco, CA, 2003

[13] A. Goel, et al. Forensix: A Robust, High-Performance Reconstruction System. In *ICDCS Security Workshop.* Columbus, OH, 2005

[14] S. D. Gribble, et al. Self-similarity in File Systems. In *ACM SIGMETRICS,* Madison, WI, 1998

[15] W. H. Hau, and A. J. Smith. Characteristics of I/O Traffic in Personal Compute and Server Workloads. *IBM Systems Journal.* , 347-372, Vol. 42(2)

[16] W. W. Hsu, and A. J. Smith. Characteristics of I/O Traffic in Personal Computer and Server Workloads. *UC Berkeley Computer Science Division Technical Report*, UCB/CSD-02-1179, 2002

[17] S. King, and P. Chen. Backtracking Intrusions. In *SOSP*, Bolton Landing, NY, 2003

[18] L. Lamport. Password authentication with insecure communication. In *Communications of the ACM*, 24(11):770-772, Nov. 1981

[19] J. Larus. Whole Program Paths. In *PLDI,* Atlanta, GA, 1999

[20] J. Lorch, Personal communication. April 11, 2006

[21] J. Lorch, and A. J. Smith. The VTrace Tool: Building a System Tracer for Windows NT and Windows 2000. *MSDN Magazine.* , Vol. 15, 10

[22] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP Misconfiguration. In *ACM SIGCOMM,* Pittsburgh, PA, 2002

[23] A. Moshchuk, T. Bragin, S. D. Gribble, and H. A. Levy, Crawler-based Study of Spyware in the Web. In *NDSS,* San Diego, CA, 2006

[24] D. Oppenheimer, A. Ganapathi and D. Patterson. Why do Internet services fail, and what can be done about it? In *USITS.* Seattle, WA, 2003

[25] K. K. Ramakrishnan, B. Biswas, and R. Karedla. Analysis of File I/O Traces in Commercial Computing Environments. In *ACM SIGMETRICS.* Newport, RI, 1992

[26] E. Rescorla. Security Holes... Who Cares? In *USENIX Security Symposium.* Washington, DC, 2003

[27] P. Reynolds, et al. Pip: Detecting the unexpected in distributed systems. In *NSDI,* San Jose, CA, 2006

[28] D. Roselli, J. Lorch, and T. A. Anderson. Comparison of File System Workloads. In *USENIX Technical,* San Diego, CA, 2000

[29] C. Ruemmler, and J. Wilkes. UNIX Disk Access Patterns. In *USENIX Technical,* San Diego, CA, 1993

[30] Solaris Dynamic Tracing (DTRACE) http://www.sun.com/bigadmin/content/dtrace/

[31] C. Soules, G. Goodson, J. Strunk, G. Ganger. Metadata Efficiency in a Comprehensive Versioning File System. In *USENIX FAST,* San Francisco, CA, 2003

[32] C. Verbowski, et al. Analyzing Persistent State Interactions to Improve State Management, *Microsoft Research Technical Report*. MSR-TR-2006-39, 2006

[33] W. Vogels. File System Usage in Windows NT 4.0. In *SOSP,* Charleston, SC, 1999

[34] H. Wang, C. Guo, D. Simon, and A..Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *ACM SIGCOMM.* Portland, OR, 2004

[35] H. Wang, et al. Automatic Misconfiguration Troubleshooting with PeerPressure. In *OSDI.* San Francisco, CA, 2004

[36] Y.-M. Wang, et al. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *LISA.* San Diego, CA, 2003

[37] Y.-M. Wang, et al. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites that Exploit Browser Vulnerabilities. In *NDSS.* San Diego, CA, 2006

[38] Y.-M. Wang, et al. Gatekeeper: Monitoring Auto-Start Extensibility Points (ASEPs) for Spyware Management. In *LISA,* Atlanta, GA, 2004

[39] A. Whitaker R. Cox, S. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In *OSDI,* San Francisco, CA, 2004

[40] N. Zhu, T.-C. Chiueh. Design, Implementation, and Evaluation of Repairable File Service. In *ICDSN,* San Francisco, CA, 2003