

Analyzing Persistent State Interactions to Improve State Management

Chad Verbowski

Emre Kiciman

Brad Daniels

Shan Lu

Roussi Roussev

Yi-Min Wang

Juhan Lee

April 4, 2006

Technical Report

MSR-TR-2006-39

Microsoft Research

Microsoft Corporation

One Microsoft Way

Redmond, WA 98052

Analyzing Persistent State Interactions to Improve State Management

Chad Verbowski, Emre Kiciman, Brad Daniels, Shan Lu, Roussi Roussev, Yi-Min Wang

ABSTRACT

A primary challenge to building reliable and secure computer systems is managing the persistent state of the system: all the executable files, configuration settings and other data that govern how a system functions. The difficulty comes from the sheer volume of this persistent state, the frequency of changes to it, and the variety of workloads and requirements that require customization of persistent state. The cost of not managing a system's persistent state effectively is high: configuration errors are the leading cause of downtime at Internet services, troubleshooting configuration problems is a leading component of total cost of ownership in corporate environments, and malware—effectively, unwanted persistent state—is a serious privacy and security concern on personal computers.

In this paper, we analyze how computer systems dynamically interact with files and configuration settings in an attempt to gain insights into the problem of persistent state management. We analyze over 3648 machine days of these *persistent state interactions*, collected over an 8 month period from 193 machines. These machines are under real workloads and include Internet servers, corporate desktops, and home machines. We characterize the scope and magnitude of the persistent state management problem today, measuring not only the gross characteristics of persistent state, but also analyzing how it is used by applications, and when administrators and users modify it. We find that monitoring persistent state interactions provides important visibility and show how it can be used as a foundation for building better persistent state management tools.

1. Introduction

Misconfigurations and other persistent state (PS) problems are among the primary causes of failures and security vulnerabilities across a surprising variety of systems, from individual desktop machines to large-scale Internet services. One large MSN service found that, in their 7000 machine system, 70% of their non-reboot-curable problems were related to PS corruptions, while only 30% were hardware failures. In [13], 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 [13,4]. 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 these vulnerabilities because they do not know the state of their systems and thus cannot predict the impact of a change [2,16,22].

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. Managing the PS of computer systems poses many challenges. First is the sheer volume of data. Today, the unit of state management is individual files or registry entries. A typical Windows machine contains on average 70,000 files and 200,000 configuration settings [5,24]. Understanding the roles of these files and settings and their inter-dependencies requires a practically unattainable breadth of knowledge. The second challenge is the rate of change. Our traces indicate that the average Windows computer makes over 1 million changes to files (*e.g.*, file creations, writes and deletions) and 30,000 changes to configuration settings daily. While many of these changes are to temporary files and user data, we found that each machine saw critical security updates, software upgrades, and new application installations occurred, on average, once every 5 days. To effectively manage computer systems' PS, we have to track and evaluate the impact of these changes. The third challenge to PS management is that “correctness” is ever changing. Security patches, software upgrades and changing user requirements mean that the critical PS on a system must be updated frequently. Moreover, different systems, having different workloads, requirements and hardware configurations, will require different PS—there is no single “golden” version.

The first step to improving PS management is gaining a better understanding and characterization of how computer systems interact with their PS—how and when this state is created, read, written and deleted by the programs and users of a computer system. To this end, we collected over 3648 machine days of these *PS interactions* over an 8 month period from 193 machines operating under real workloads in a variety of environments, including Internet services, corporate desktops, experimental lab machines and home machines. In Section 3, we find that there is significant variation in how state is used and managed across our monitored environments, implying that “one-size fits all” approaches to PS management may not be appropriate. While our analysis is limited to Windows machines, many of the observations we make have a strong relationship with user workload and administrator behavior and thus may apply well to other systems with similar workloads and administrative policies.

While existing methods for managing PS are based on static analysis of the state itself (*e.g.*, malware scanners) or *a priori* descriptions of actions taken on the state (*e.g.*, installation manifests), we find that analyzing the state and actions together, as PS interactions, provides a more complete foundation for

¹ The other calls were related to hardware problems (17%), software bugs (15%), design problems (6%), “how to” calls (9%) and unclassified calls (12%).

managing PS. In Section 4, we characterize how processes use and share state. In Section 5, we present two case studies to show how monitoring PS interactions can improve PS management.

Finally, in Section 6, we explore how PS interactions can be naturally grouped into coarse-grained bursts of activity. We find that this grouping reduces the volume of events that must be recorded and analyzed by $O(10^3)$. Moreover, most of these activity bursts are repeated events and the amount of new activity is low—70% of applications generate no more than 5 previously unseen bursts each day.

2. Methodology

In this section, we describe our instrumentation package for monitoring and collecting PS interactions. We also describe the machines and environments from which we collected our traces. Throughout this paper, we use *PS* to refer to both the file system and the Windows Registry. *PS entries* refer to files and folders as well as their registry equivalents. A *PS interaction* is any kind of access, such as a read or write, to an entry.

2.1 Trace Collection

To collect our traces of PS interactions, we built a black box instrumentation tool for the Windows operating system. It consists of (1) a kernel mode driver that intercepts all PS interactions with the file system and the Windows Registry, along with process creation and binary load activity; and (2) a user mode daemon that manages the trace files and uploads them to a central server. Neither the kernel mode driver nor the user mode daemon requires any changes to the core operating system or the applications running atop it.

The kernel mode driver operates in real-time and, for each 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. We also record process tree information, noting when a process spawns another. The daily logs for a machine range in size from 348MB to 1.4GB, depending on the system’s workload. Idle machines generated 30MB of logs daily.

Once the user mode daemon on a monitored machine has uploaded a set of collected observations of state interactions to a central machine (or machines), these log files are processed and inserted into a database. Processing log files involves correlating registry and file activity with the applications making the request. In addition, we canonicalize some machine-specific differences in the fully qualified name of files and registry settings to improve cross-machine comparison. For example, the Windows file manager is `C:\WINDOWS\explorer.exe` on some systems, but `D:\WINNT\explorer.exe` on others. Our processing will canonicalize both instances to `WINDIR\explorer.exe`.

To date, our data collector has been tested and deployed reliably on over 300 Windows 2000, Windows XP, and Windows 2003 machines. The performance overhead imposed by this monitoring is not noticeable for common user tasks. At a large commercial

Table 1 Summary information about our collected traces

Environment	Number of Machines	Total Observed Machine-Days
Internet service machines	76	841
Research lab machines	72	1703
Corporate desktops	35	849
Home machines	7	169
Idle machines	3	86
Total	193	3648

web site, pre-production testing of our data collector was done in a lab setup of 4 identical servers, 1 running our data collector, 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 data collector was minimal, with less than 1% CPU overhead measured, and no measurable degradation in transaction rate or latency.

2.2 Monitored machines

For our experiments, we installed the data collector software and monitored over 193 machines. In total, we have collected 3648 machine days of PS interactions over an 8 month period from March to November, 2005. Our deployment of the data collector was gradual, so many machines were not monitored for the full period. We monitored machines from a variety of environments, as summarized in Table 1, with different applications, workloads, and management policies.

Internet server environment: We worked with a MSN to instrument 76 of their machines, across 5 different services with different workloads and management styles. Svc1 is a CPU-bound system with heavy disk workload. Svc2 is a critical back-end system with high workload and is “intensely managed.” The main distinction of Svc3 is that it is administered under a “follow the sun model,” with 3 operations teams around the world monitoring the system. Svc4 is a large storage service for external users. Svc5 is web notifications publish/subscribe system. All services are managed by the same operations organization, but different engineering teams.

Lab environment: We monitored laboratory machines in our research facility, used for various data collection, analysis and simulation experiments. These machines are managed by our research labs’ own IT staff. Software patches are automatically installed as needed through Windows’ auto-update facility.

Corporate desktop environment: We also monitored corporate desktops and laptops, used by researchers and engineers, primarily for activities such as software development and word processing. These machines are managed jointly by users, who can install and uninstall software, and by the corporate IT department, which installs critical patches as needed and manages security-critical settings, such as firewall and web browser configuration.

Home environment: We monitored home machines, used for entertainment and work-related activities by researchers, engineers and their families. These machines are not actively

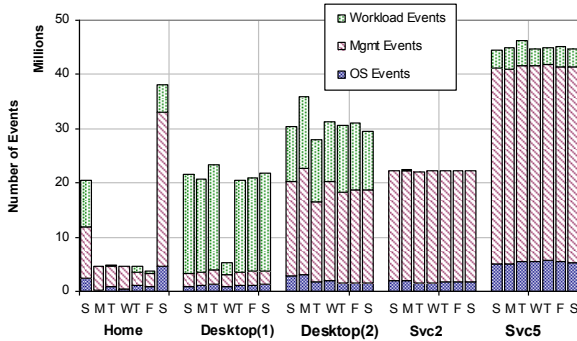


Figure 1 Week long samples of interactions from a home machine, two corporate desktops, Svc2, and Svc5.

managed by professionals, though they may be configured to accept critical Windows updates automatically.

As a control, we also collected traces from 3 idle systems, running within virtual machines. The first idle machine had no network access. The second had network access; and the third idle machine had network access as well as Windows’ auto-update functionality enabled. After our initial setup of these idle machines, we placed no user workload on them.

For some of our analyses, we categorize the processes running on a system into one of three broad functionalities: (1) an OS-level process, or a surrogate process such as cmd.exe, svchost.exe, cscript.exe, which primarily spawn other programs; (2) an installation program or a management tool, such as a configuration panel or antivirus program; (3) a workload application such as a word processor on a home machine or a web service on a server. We categorized processes based on our own knowledge of its functionality as well as information gleaned from the Internet.

When we analyze our traces to find daily statistics, such as the volume of activity per day, we exclude machine-days that do not cover a whole 24 hours, such as days with extended periods where a machine or our monitoring was turned off.

3. Characterizing PS

In this section, we quantify the three challenges of PS management, and explore how the state is used by the software and users on the system.

3.1 Scale

The first challenge to managing PS is the volume of data and volume of interactions with the data. Previous studies of file system contents [5,17,21] and the Windows registry [22,24] have noted that modern computer systems contain 70k files and approximately 200k registry settings. We examined file system and registry snapshots from 33 of our monitored machines, and found them to be consistent with prior findings. We do not further review the volume of data here.

Table 2 Summary of daily PS activity per machine across environments. All units are in millions.

Environment	Category			Type		Total
	Workload	Mgmt.	OS	Read	Write	
Svc. 1	39.75	26.52	3.59	68.31	1.55	69.86
Svc. 4	19.16	37.10	4.37	57.77	2.86	60.63
Svc. 5	2.29	23.10	3.67	28.31	0.76	29.07
Svc. 2	0.005	21.00	1.45	21.26	1.20	22.46
Svc. 3	1.63	14.93	2.18	16.90	1.83	18.73
Home	4.27	8.89	4.17	16.70	0.62	17.33
Desktop	2.74	5.02	1.62	8.94	0.44	9.38
Lab	2.52	5.99	0.74	8.73	0.51	9.25
Idle	0.005	0.25	0.10	0.34	0.02	0.36

In addition to the volume of data, the volume of PS interactions also challenges efforts to manage PS. The existing I/O load influences when we might be willing to add extra load on the system to manage state. Figure 1 presents a one week sample of activity for a representative home machine, corporate laptop, corporate desktop, Svc2 machine, and Svc5 machine. The variation of interactions from day to day matches our intuition of each machine’s workload: home machines are busiest on the weekend, the corporate laptop and desktop machines vary significantly from day to day, while the servers workloads are stable from day-to-day. Perhaps not surprisingly, our corporate desktops load did not drop over the weekends.

Figure 2 shows the average number of daily interactions across all our traces. Here, we see that Svc1 has the highest workload and variability of the servers—the other server environments have very stable workloads. The lab machines, especially the lab machines 60-78 show high variability in the number of interactions per day. Desktop and home machines have a high variability both across machines and from day-to-day on a single machine.

Table 2 presents the average volume of daily interactions, divided by the category of application generating the interaction and type of activity. One surprising result is that existing state management applications, such as hardware configuration tools and antivirus scanners, generate 38%-98% of the interactions across our environments. We examine this observation in detail in Section 4.1. Overall, we see that server machines generate considerably more interactions than desktop, home, and lab machines.

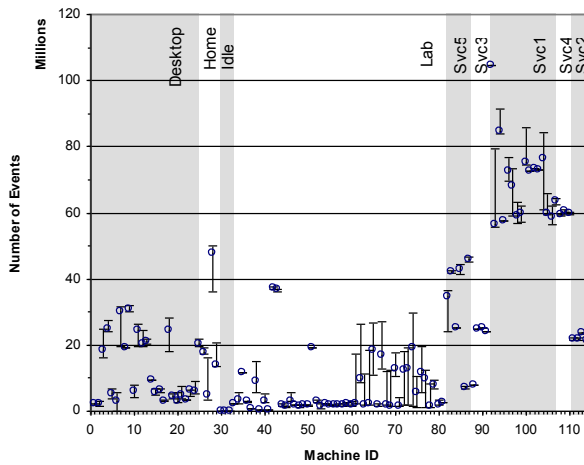


Figure 2 The median number of interactions per day for each machine. The vertical bars mark the 25th and 75th percentile of daily activity. The gray/white bands group our environments.

While millions of interactions occur every day on a typical machine, there are several factors that we can use to easily reduce the volume of events and state that we care about. First, we see that read activity is consistently an order of magnitude (or more) greater than write activity across all environments, consistent with prior findings [2]. While read interactions are important when trying to debug or explain software behavior, we do not have to worry about these interactions corrupting PS.

Furthermore, the number of distinct non-temporary files and registry entries read or written every day is much smaller than the total number of interactions, as shown in Figure 3—up to 2 orders of magnitude smaller. However, this is still a large number of entries: 10-15% of the 70k files and 5-10% of the 200k registry entries on a system are used on any given day. Although server machines have more total interactions per day, Figure 3 shows that they use an order of magnitude fewer distinct PS entries compared to the desktop and home machines. Also note that, in

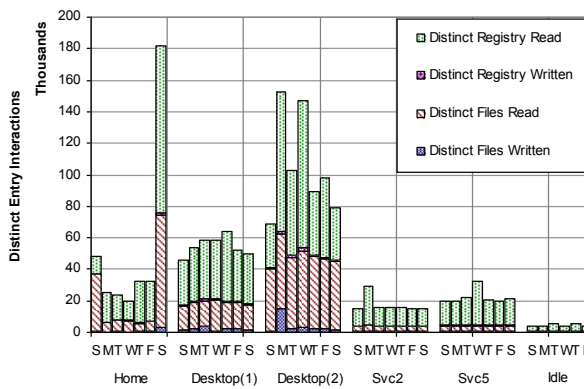


Figure 3 Number of distinct non-temp files and registry items read/written during 1 week of representative machine activity.

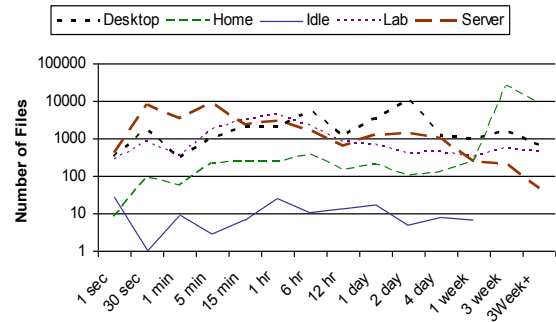


Figure 4 The average number of files per machine with given update frequency, for each environment.

terms of the number of distinct files, management activity is no longer dominant, indicating it is a very repetitive activity..

3.2 Update Frequency

The second challenge to managing PS is that it changes so frequently. The result is that it is difficult to track changes or distinguish between acceptable and problematic ones.

By identifying specific intervals at which most entries are updated, we might be able to automatically flag unexpected changes for closer inspection. To explore this potential, we measure the update frequency of the PS entries on our monitored machines. We calculate the average time between modifications for each entry modified at least twice. Figures 4 and 5 show that the update frequency distribution of files and registry settings ranges greatly, from 30 seconds to 3 weeks. We also see that the volume of entries that change at a given frequency varies across environments by orders of magnitude. Desktop, lab and servers have thousands of entries that change at a range of intervals, while home machines have hundreds. It is worth noting that even our idle machines have tens of entries that are modified regularly, for example, to update internal OS data and cryptographic random seeds. Overall, there does not appear to be a specific interval at which most entries are updated.

Another method of identifying potentially problematic PS changes might be to identify a deviation from the normal rate of change for each entry. Table 3 shows the regularity of PS updates by calculating the ratio of standard deviation to average for the time between successive updates to each distinct registry or file entry. We see that idle machine activity is quite regular, as we might

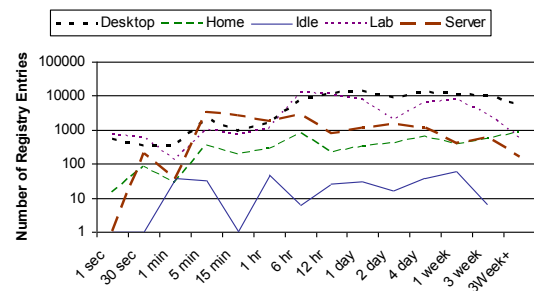


Figure 5 The average number of registry entries with a given update frequency, for each environment.

Table 3 Percentage of file and registry entries with change frequency standard deviation / average ratio <1.

Environment	File	Registry
Desktop	55.9%	17.6%
Home	43.6%	38.4%
Idle	98.0%	97.4%
Server	63.6%	16.3%

expect. Across environments, we find that files are updated more regularly than registry entries. Investigation found regular updates to log files, temporary files created by applications’ periodic auto-save feature, and memory mapped files. Many of the irregular updates to registry entries are associated with the startup and shutdown of end-user applications, and are thus tied to human behavior, rather than regular computer behavior. These registry entries hold information such as recent window coordinates, the last run time, and other application-specific information. Overall, it appears that monitoring the update frequency of entries is not a good indicator of potential problems because the false positives due to irregularly updated entries would be overwhelming.

3.3 Variety of PS

The third challenge to managing PS is that there is no single “correct” version of PS. “Correctness” varies across workloads and system configurations, as well as with the development of security patches, etc. One way to measure the diversity of “correct” state is to measure the cardinality of configuration settings: how many valid settings does a particular entry appear to have across machines? If we see that a registry entry always has one of a small number of values, we might be able to develop consistency checks on it. Alternatively, if we see that a registry entry always has a different value and changes frequently, we may assume that it is temporary or less important state.

Table 4 shows the cardinality for all registry entries we have seen accessed for each environment. We see that 97.7% of idle machine registry state has cardinality 1, not surprising given that there is no user activity and minimal differentiation between machines. We manually inspected all of the 153 (1.6%) registry entries, across the idle machines with cardinality between 2 and 5, and found that these entries were related to installation differences

Table 4 Summary of registry cardinality by environment showing the percentage with cardinality 1, 2-5, or >5.

Environment	1	2-5	>5
Idle	97.7%	1.6%	0.7%
Home	96.9%	2.7%	0.4%
Server	95.0%	3.4%	1.6%
Desktop	88.8%	10.4%	0.8%
Lab	87.7%	11.7%	0.6%
All machines	60.4%	27.0%	12.6%

(e.g., network settings) or were one of a small number of settings involving locally generated unique IDs. We see also that home machines at 96.9% and server machines at 95% also have very high degree of cardinality 1 entries. Interestingly desktop and lab machines have 10% fewer cardinality 1 entries, and 10% more cardinality 5 or fewer entries. This reflects that these systems are configured slightly differently, in many cases due to OS and application installation that differs from the default locations and settings. As we compare entries across all our environments, we find that the number of registry entries with a cardinality of 1 quickly decreases, indicating, not surprisingly, that the system’s environment and workload has a large influence on its specific configuration. However, most of the entries still have low cardinality. While it would not be a complete solution, this raises the possibility of identifying consistency checks on many entries.

4. Process Interaction with PS

In this section, we analyze how processes use the PS on a system, and the implications for PS management.

4.1 Running Processes

Before we analyze how processes use state, let us first look at what processes are running on our monitored machines. Figure 6 presents the number of distinct processes per day for each of our machines. As expected, idle machines have the fewest number of processes run per day (15-17). The corporate desktop environments show the highest variability, with some machines running over 100 different processes on a given day. The server environments show variability across services but show similar behavior within a single service. Looking at how often each of these processes is run per day, we find that 85% are run between 1 and 5 times per day across all our environments. We found that a small number of processes were running over 100 times every day on lab, desktop, and server machines. Upon inspection, we found that these were related to the workload of the system, such as build processes on developer desktops, and command line administrative tools on servers.

We found that a few processes in each environment account for a significant portion, sometimes a majority, of PS interactions. On

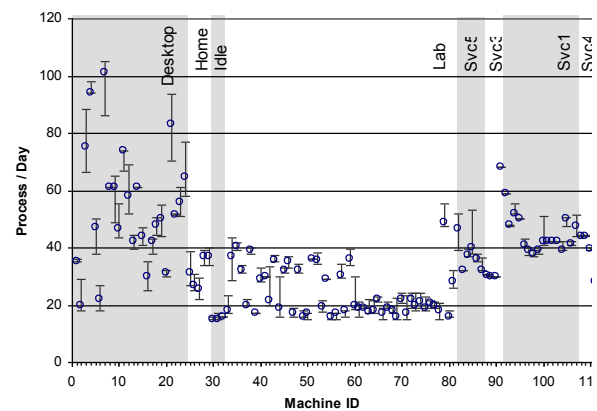


Figure 6 The median number of distinct processes run per day for each machine. The vertical bars mark the 25th and 75th percentiles. The gray/white bands group our environments.

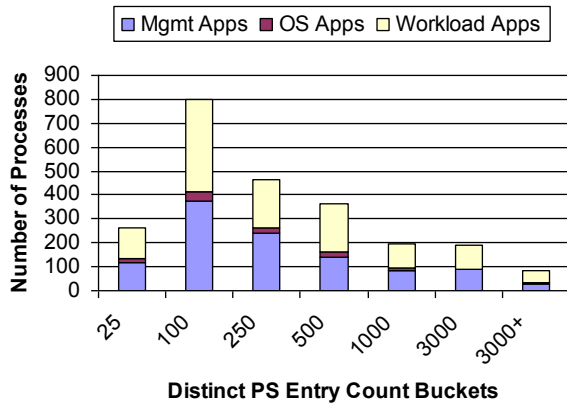


Figure 7 The number of processes with a given manifest of file and registry entry size.

Svc2, Svc3, Svc4, and Svc5, one management process, 'cqmghost.exe,' was responsible for 88-96% of daily interactions. This process enumerates all of the daemons installed on a system every 2 seconds. The only busier process is one found on Svc1: a server-side web application that reads the computer name from the registry approximately 19M times a day. On desktops we found that two management processes each accounted for 10% of daily interactions. The first process provides dynamic configuration support for a laptop modem and reads the same registry entry up to 400k times per day. The second is an enterprise management agent which scans the machine, generates log files, and installs patches. User workload accounted for the rest of the interactions on desktop and home machines.

4.2 Process state

Monitoring the distinct PS entries used by running processes can help us with several state management problems: when a process fails, knowing the entries it accesses narrows down the potential cause of the failure; knowing the processes that use every PS entry can help when evaluating the potential impact and risk of a software update; and finally, we might be able to take special care to secure or lock-down the state accessed by the most important processes on a system.

Figure 7 shows the distribution of processes by the volume of distinct PS entries that they access. We can see that application category does not appear to be correlated to the volume of data accessed. Overall, 80% of processes access 500 entries or fewer, with a third using less than 100 entries. A very small percentage of processes access more than 3000 entries. These processes include processes that access many entries because of their workload, such as state management tools, command shells, media players and compilers.

4.3 Shared PS Entries

PS entries that are shared across applications can be a serious source of problems. First of all, if many applications are modifying and reading an entry, then there must be agreement on locking or transactional semantics to avoid consistency errors. Furthermore, the more processes write to an entry, the greater the opportunity for the entry to become corrupted.

We found that all of the 1281 distinct processes we observed shared some file or registry entries with other processes. Looking

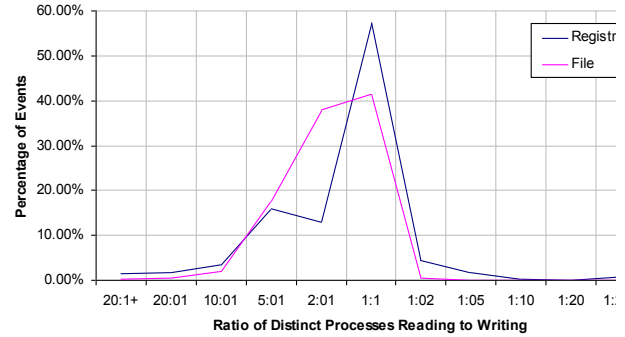


Figure 8 The distribution of files and registry entries bucketed by the ratio of their readers to writers.

at the distinct canonicalized entries, we observed that over half (54.9%) of 3.6M files are shared by more than one process, whereas only 25.6% of 1.7M registry entries are shared. An important special class of shared PS is the dynamically linked library (DLL). Across all our environments, we found 2020 distinct DLL files that had been used. Of these, 717 were loaded for execution by multiple distinct processes. This indicates that changes to many DLLs will impact multiple processes.

Figure 8 shows the ratio of the number of readers to writers for shared state. We see that most entries have exactly one reader and one writer. When entries are shared, they tend to have many more readers than writers. Very few files (0.6%) and very few registry entries (7.1%) are written to by more distinct processes than they are read from (a reader:writer ratio of 1:2 or higher).

We observed that 76.6% of files and 94.2% of registry entries are read but never written to by processes. To prevent accidental corruption, we might consider using access controls to prevent any writes to these files and entries. Interestingly, we observed 5235 registry entries and 263 files that were written to by multiple processes and never read. Presumably, this implies that these entries are not important during normal operation.

Given prior studies showing that most new files are deleted within seconds of creation [3,21], we might think that temporary entries also account for a significant portion of distinct file and registry entries. However, we found that only 16.2% of accessed files and 4.1% of accessed registry entries were temporary. We found that 0.9% of temporary registry entries and 7.2% of temporary files were shared. Process problems caused by sharing temporary entries would be difficult to reproduce and troubleshoot.

5. Case Studies in PS Monitoring

We believe that monitoring PS interactions is a fundamental building block in PS management solutions. In this section, we present two case studies that show how understanding PS interactions helps address PS management problems.

The first case study investigates the problem of unwanted software (malware) running in the background on systems by attaching as a plug-in to the OS or applications. The PS entries that control the dynamic loading and execution of these plug-ins are *extensibility points* (EPs). We analyze PS interactions to discover the extensibility points on a system, and quantify their importance in terms of system exposure. Based on our analysis of EPs, we make recommendations for eliminating or securing 81% of them.

The second case study uses PS interaction monitoring to evaluate the software installers used to add, update, or remove applications on a system. We were surprised that, on average, machines have a software installation every 5 days, and there appears to be no predictable pattern to these installs. We also found that software installers do a poor job of declaring the PS entries they add to a system, and orphaned PS entries are often left behind on a system when software is uninstalled. These abandoned PS entries accumulate, potentially causing software decay and reliability problems. Today, it often takes a system reinstallation to remove all of this abandoned state. However, by auditing PS interactions we directly associate PS entries with the software installers and applications responsible for them.

5.1 Managing Software Plug-ins

In this section, we discover the hooks used by malware to ensure they are automatically run by their host system, and make recommendations for securing them. While malware could modify existing executable files on machines, this is not effective because detecting them is deterministic. The file malware modifies may be signed, software updates could overwrite it, or a management tool can be used to compare a file's hash of the file with a known good value. Instead, malware can simply use the plug-ins exposed by the OS, daemons, and frequently run applications to be run. Detecting them then becomes the challenging problem of distinguishing malware from other plug-in extensions. One study found that a sample of 120 different spyware programs used a total of 334 EPs to integrate themselves into a system [5].

We present a method for analyzing PS traces to discover and rank security-sensitive PS entries that are used by processes as plug-ins. Unfortunately, the Windows OS and thousands of third party applications have many EPs whose usage is not controlled. Managing EPs is difficult because they are hard to identify since many were designed for internal use only and are therefore not documented. Discovering EPs through static analysis is difficult because of the general unavailability of system and application source code; and the lack of a standard location or format for EPs make it impractical to find them amongst the $O(10^6)$ PS entries.

5.1.1 Detecting Extensibility Points

To discover EPs we analyze the per process PS interactions preceding a new executable file loading into a process, to correlate the name of the executable file with prior reading of a PS entry containing that name.² We label these PS as *direct extensibility points*. We then identify first order *indirect extensibility points* by continuing to search through the history of PS events to discover references to the direct EP. This process is then repeated to identify EPs with the next order of indirection. For example, an indirect EP may reference an ActiveX class identifier that contains the ID of the COM³ object direct EP that ultimately contains the executable file name. Identifying indirect EPs may potentially yield false positives, however we filter them out through manual investigation of new EP instances.

² Our PS interaction tracing records the loading of a file for execution as a distinct activity from simply reading a file into memory.

³ 'Component Object Model' is a Microsoft standard for reusing and sharing software components across applications.

Many EPs have a similar name prefix that typically indicates that plug-ins using it follow a standard design pattern. We define a common EP name prefix as an *extensibility point class*, and the fully named EP as an *extensibility point instance*. For example, the EP class "HKLM\Software\Classes\CLSID\" contains 2396 EP instances. We identify new EP classes by manually examining newly discovered EP instances that do not match an existing EP class, and update our list with a new entry.

In our analysis we processed 912 daily traces from 53 Home, Desktop, and Server 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 extensibility points. COM objects are the most dominant EP instances with 2796 of 7227 (40%) of the total. The second most EP class is associated with the Windows desktop environment, and the third contains web browser plug-ins. The remaining popular EP classes are related to an Office productivity suite and a development environment, both of which support rich extensibility features.

We found that 470 of the 697 (67%) of distinct processes in our traces used EP instances, 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).

Overall we found that monitoring PS interactions and correlating them with process executable file loads is an effective method of discovering EPs in applications. Also with 67% of processes using EPs, and at least 364 EP classes available, there is a lot of opportunity for malware to run on a system.

5.1.2 Importance of Extensibility Points

Some EPs are more critical security hazards than others. We estimate an EP's criticality using three metrics: 1) the privilege-level of the loading process, where higher-privilege processes such as operating system or administrator-level processes, are more critical; 2) the lifetime of the loading process, where longer running applications provide higher availability for a malicious extension; 3) the sensitivity of the data accessed by the loading process and, hence, available directly to a malicious extension. The first two metrics are captured in our traces, while the third cannot be measured.

Figure 9 shows a scatter plot for all machines where each point is the percentage of EP instances that are loaded by processes that run for a given percentage of the overall system uptime. It shows that there are many EPs either used by short lived processes, or long lived processes and not many in the middle. 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 machines uptime. Furthermore we observed that one third of all EP instances were used by processes running in a process with elevated privileges. Overall, we found that a significant portion of EP instances are security hazards due to longevity and elevated privileges.

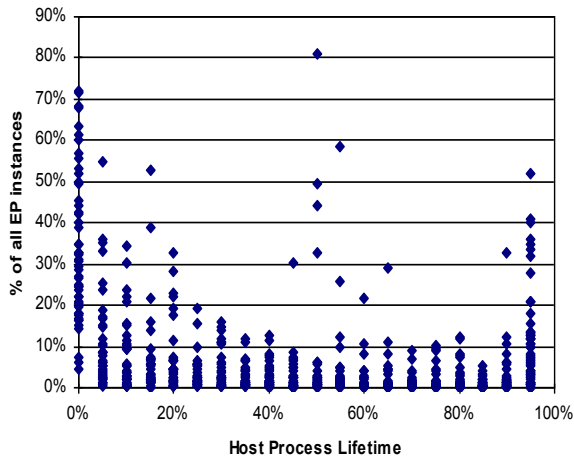


Figure 9 The distribution of EP instances to lifetime of their host processes as a percentage of machine uptime.

5.1.3 Lessons and Suggestions

We compared our list of EP classes that contained EP instances loaded by processes running for 80% of system uptime or greater, with the published gatekeeper list [22]. Gatekeeper identified 30 long life EP classes, compared to the 210 long life EP classes. Gatekeeper identified 6 EP classes that were not in our list because we did not observe processes that used them.

Apart from demonstrating the utility of PS interactions in discovering and documenting these extensibility points and their exposure, our results suggest that OS and application design changes may be possible to restrict or eliminate many EP instances.

Removal – We observed that 3197 of 7227 EP (44%) of the EP instances across all machines were not modified during the monitoring period. It may be possible to remove these EPs by converting them into static application data.

Lockdown – 5111 of 7227 (70%) of the EP instances, spanning 221 of 367 (60%) EP classes were used by a single process. For these privately used EP instances it may be possible to lock them down by implementing a policy mechanism to prevent interference or corruption from other applications, or to convert them into static application data. If the EP instances identified above are removed or locked down 88 (18%) of EP using applications will no longer use them.

We found that 1297 (19%) of EP instances, spanning 43 EP classes, were changed and shared by multiple applications and therefore are not candidates for removal or lock down. However the remaining 81% can potentially be locked down or removed, thus making 18% of current EP using applications more secure.

5.2 Tracking Software Ownership

Intuitively, we may think that managing PS should be easy because executable files and configuration are infrequently created or are modified by well-known software installers. In Section 5.2.1, we analyze our traces to find that installations actually occur quite frequently. Additionally, while most software installers provide *manifests*, listing the PS entries owned by the installed application to facilitate its clean uninstallation, these

Table 5 The distribution of installations by installation program type for all observed installations.

	User Setup	Scripted	Auto Update	Self Update	Developer
Home	7%	7%	51%	35%	0%
Desktop	5%	8%	58%	22%	7%
Lab	2%	5%	61%	32%	0%
Server	1%	17%	38%	44%	0%

manifests are often incomplete or incorrect [7]. In Section 5.2.2 we analyze PS entries across 70 machines and find that many entries cannot be accounted for via the machines’ static manifests. To better understand the origins of these orphaned PS entries, we describe point experiments with installing and uninstalling three popular applications.

5.2.1 How often is software installed?

To identify a software installation examined our PS traces to identify the creation or modification of files that were later loaded by a process as an executable file. We found that on average 20% of all machine days had at least 1 installation. However this varied significantly across each environment. 15% of Home and Lab machine days and 30% of desktop machine days had at least 1 install. Servers environments had a wide variance in the frequency of software installations ranging from 7%-80% of machine days having at least one install. This reflects the variation in change management policy for each Internet service.

While we might have thought that centralized administration of corporate desktops, or Windows auto-update service might cause synchronized updates, this does not appear to be the case. Overall we observed that most software installations, even in the server environment, occur in an unpredictable manner. Table 5 describes the distribution of observed installations across install types. We see that processes that update themselves (Self Update), predominantly anti-virus applications, account for a significant portion of installs. Also, enterprise software distribution applications and Windows auto-update account for the majority of software installs in most environments. As we expect, Servers have a large portion of scripted installs from administrators manually rolling out upgrades and in-house applications. Installations caused by users running install programs are infrequent. It is interesting to see that our analysis was able to distinguish binary files created and used on developer machines as ‘installed’ by the developer tools. Overall we found that a wide variety of software installers create and modify executable files on a machine, and that installations happen quite frequently.

5.2.2 Static Software Ownership Manifests

Unfortunately, a statically declared manifest is not always complete. Today’s manifests are not always correctly specified, nor do they account for PS created post-installation, such as user preference settings, log files, etc. This means that during software upgrades or removal entries can become orphaned on the machine. Furthermore, installation or removal of software can fail or be interrupted which often leaves registry entries and files in an inconsistent state. Over time the few orphaned files and registry entries accumulate on a machine causing a build up of unused

Table 6 Average file and registry entries that are specified in manifests, implicitly in manifests, user data, or temp entries. We do not have heuristics to recognize data and temporary registry entries.

		Manifest	Implicit	Data	Temp	Unknown
File	Desktop	18.5%	21.0%	20.6%	8.3%	31.6%
	Server	4.7%	52.6%	2.4%	3.4%	36.9%
	Lab	13.2%	5.8%	9.5%	1.4%	70.1%
Reg.	Desktop	28.2%	32.2%	N/A	N/A	39.6%
	Server	10.5%	36.4%	N/A	N/A	53.1%
	Lab	30.3%	31.7%	N/A	N/A	38.0%

entries that can lead to system problems.⁴ Because of this phenomenon, common advice is to reinstall your computer occasionally to return the system to a known state.

To quantify the significance of this problem, we wrote a tool to extract PS ownership manifests from within the Windows OS. The tool identifies the components installed on the system by analyzing the OS installation configuration files, enumerating the list of program that have registered with the Windows ‘Add/Remove programs’ component, the Windows Installer database (WI), the OS configuration for launching applications when a file of a given extension is run, and manifests of patch contents as described in the Microsoft Security Baseline Analyzer tool. We then enumerate all files and registry entries on a machine and compare them with these manifests to identify unaccounted for entries. We associate entries that are children of entries that are referenced in the manifests as implicitly associated with the manifest as well. Finally we filter the remaining list to exclude well known user data files by their extension, and remove entries from well known temporary folders. We define the remaining subset as *leaked* entries on the system. Table 6 contains the results of running this tool across 8 desktops, 20 Server, and 42 lab machines. It shows that 31-70% of files and 38-53% of registry entries could not be accounted for.

To further understand the prevalence of software leaks, we measured the leakage of 3 popular commercial applications. By running our data collector while installing the application, using it for a short period, and then uninstalling the application; we were able to measure the net increase of file and registry entries on the machine. The first application was the game ‘Doom3’, which left 9 files and 418 registry entries. The second was the common corporate desktop application suite Microsoft Office product suite which left no files, but 1490 registry entries. Additionally, it left 129 registry entries for each user that logged into the system and used the program while it was installed. The third example was the enterprise database application Microsoft SQL Server Yukon edition, which leaked 57 files and 6 registry entries.

6. Activity Bursts

In this section, we analyze *activity bursts*: groups of PS interactions occurring close together in time, within a single thread. We first discuss how we find activity bursts, and then explore how they can be used to reduce the $O(10^7)$ daily events

⁴ Examples can be found at <http://support.microsoft.com/> via the article IDs: 898582, 816598, 239291, 810932, 181008.

reported in Section 3.1 to $O(10^3)$ distinct daily activity bursts. Also, we show that 73% of observed processes produce only 2 never-before-seen bursts per day. We found that bursts may be the ideal unit for measuring PS activity, and discuss how they might be used to detect anomalous events.

6.1 Identifying Activity Bursts

To make our definition of an activity burst more concrete, we have to define how close together in time interactions have to be before we will group them together into a single activity burst—in other words, what *gap* should separate activity bursts? While we could group together interactions at any time scale, we are interested in analyzing activity bursts for the purpose of studying how software, administrators, and users of computer systems manage their PS, and thus are interested in relatively macro-scale activities. This implies that our activity bursts will also be relatively macro-scale, at the granularity of seconds or minutes, as opposed to a granularity of milliseconds that we might want if we were interested in studying, for example, the effects of activity bursts on disk caches.

Examining our trace logs, we find that over 99% of PS interactions within a single thread occur within 1 second of another interaction. Also, there are long periods of no state interactions as well. This seems to confirm our expectation that PS interactions do occur in bursts.

Formally, we define an activity bursts as a group of interactions $\{e_i | i \leq t \leq j\}$ occurring within a single thread, where $gap(e_i, e_{i+1}) < k$, for all $i \leq t < j$; $gap(e_{i-1}, e_i) \geq k$; $gap(e_j, e_{j+1}) \geq k$; $gap(x, y)$ is the time between two interactions x and y ; and k is the threshold gap between bursts. To inform our specific choice of the threshold gap k , we considered the distribution of the gap between PS interactions over the range [1-900] seconds. We found that the curve representing the distribution of the gap between all PS interactions per thread is relatively flat, implying that there is likely little difference between values of k within this range. The ideal gap size will produce activity bursts that closely match user and system activities while producing few overall distinct blocks, each of which containing a small number of events to enable manual investigation if needed. For our analysis, we choose $k=60$ sec. Examining the 55.6 million burst we identified with this gap size, we found that the average burst was 34 seconds long. Also, these bursts contained 34 distinct events representing 650 total events.

6.1.1 Exact Matching with Signatures

To identify repeating activity bursts we assign each burst a deterministic signature, calculated by hashing the file paths, names and activity types (read, write, etc) for all interaction within the activity. We considered using other attributes, such as interaction ordering or timing information, but found that, given the PS entries accessed; this extra information did not help.

Some interactions, such as temporary files and user documents, obscure the underlying similarity of activity bursts. For example, a program that uses a temporary file will usually generate a random file name for it. Two executions of such a program will generate activity bursts with different signatures, unless we recognize and compensate for the randomness of the temporary filename. In our analysis, we recognize temporary files, as described in Section 4.3, and replace their random filenames with

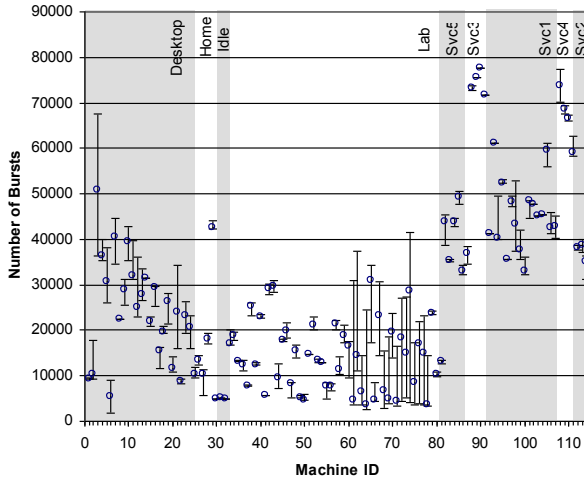


Figure 10 The median number of activity bursts we see on each machine, with bars marking the 25th and 75th percentiles. The gray/white bands group our environments.

a canonical one. Similarly, we use simple heuristics, such as file extensions, to recognize and canonicalize users’ documents, as well as unique identifiers, such as user and machine IDs.

6.2 Event Reduction

We expect that the number of activity bursts in a day will be much less than the number of PS interactions per day, given that the number of bursts we expect to see is bound by the number of gaps longer than our threshold $k=60$. Thus, with about 0.1% of interactions having $\text{gaps} > 60$, we would expect a $O(10^3)$ reduction in the number of events. Shown in Figure 10, the number of daily activity bursts is approximately 10^3 fewer than the number of daily PS interactions.

To gain assurance that these bursts are valid groupings of activities, we look to see how many of these bursts are repeated over time. If our grouping is arbitrary, we would expect to see very few repeated activities. However, we find that most of the activity bursts we observe in a given day are in fact repeated, and the number of distinct activity bursts we observe is on the order of 1000-4000 bursts/day for most machines, as shown in Figure 11.

6.3 New Activity Blocks over Time

In addition to the event reduction afforded by identifying coarse-grained activity bursts, we find that the number of new activity bursts occurring over time quickly stabilizes to a small number. That is, most applications appear to repeat the same activity bursts over time, and exhibit few new activity bursts in any given day.

To quantify this, we model the application’s creation of new bursts over time as a stochastic process, in particular, as two

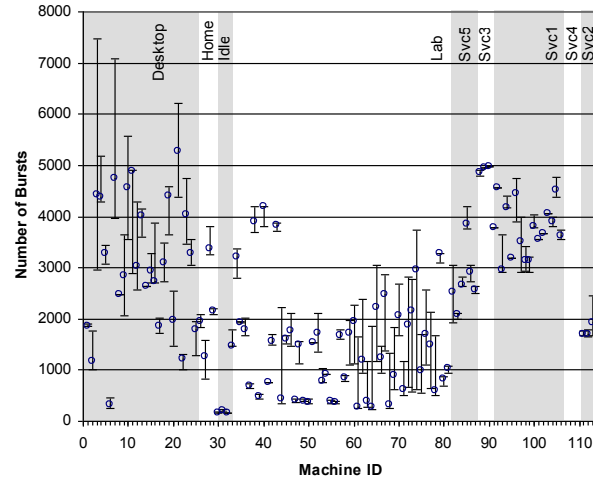


Figure 11 The median number of distinct activity bursts we see on each machine, with bars marking the 25th and 75th percentiles. The gray/white bands group our environments.

Gaussian processes with unit variance and a single transition time where the stochastic process transitions from the state associated with the first Gaussian process to the second Gaussian process. Our initial intuition is that this simple model might be sufficient to capture the behavior of an application which initially generates large numbers of new blocks, and then quickly settles into generating few or no new blocks for the remainder of our observations. The first Gaussian process corresponds to the former transient behavior, while the second Gaussian corresponds to the steady state behavior.

Given this model, finding the most likely setting of parameters is straightforward. Let the mean of the first Gaussian be denoted by λ_1 , the mean of the second by λ_2 , and the transition time by t . Then the most likely setting of these parameters is given by:

$$\arg \min_{\lambda_1, \lambda_2, t} \left(\sum_{x_i < t} (x_i - \lambda_1)^2 + \sum_{x_i \geq t} (x_i - \lambda_2)^2 \right)$$

For a fixed t , the minimizing values of λ_1 and λ_2 are given by the respective averages over the x_i ’s on either side of the transition t , allowing us to efficiently compute the above quantity as:

$$\arg \min_{\lambda_1, \lambda_2, t} \left(\sum_{x_i < t} (x_i - \text{avg}(x_i, I < t))^2 + \sum_{x_i \geq t} (x_i - \text{avg}(x_i, I \geq t))^2 \right)$$

The only pathological case we have observed is for applications with no observed transition. In this case, we choose the left-most t , and the number of new bursts generated in the application’s steady state (λ_2) remains a large number.

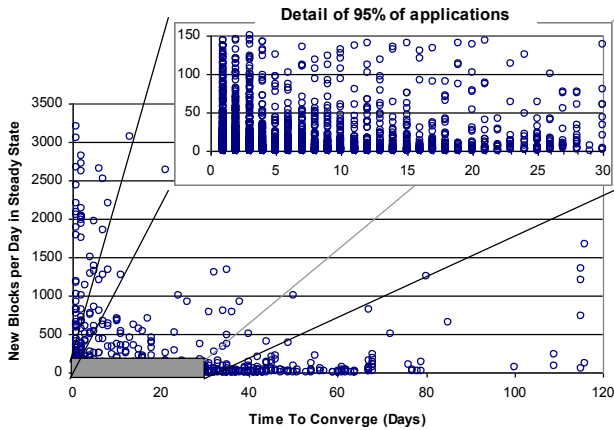


Figure 12 Scatter plot for each distinct process of number of days to converge and blocks at convergence.

Figure 12 shows the result of our analysis. For each application, we plot the steady state blocks generated by it (λ_2) against the transition time it takes for the application to stabilize into steady state (t). We see that most of the applications we have monitored do stabilize and generate a small number of new bursts per day. The median of λ_2 across our applications is 2 bursts/day, while the mean is 29.8 bursts/day. The median convergence time t is 1 day, while the mean is 4.5 days. The mean for both λ_2 and t is heavily skewed by the pathological applications that show poor convergence. These applications make up only 23% of the applications we observed. Overall activity bursts are an ideal unit for managing PS because they provide $O(10^4)$ reduction when considering PS interactions, and because they provide the additional contextual information of the process, user context, and related PS interactions for every PS change. This complexity reduction and additional information simplifies PS management.

7. Related Work

There have been many studies of file system workload traces with the goal of improving I/O system performance by optimizing disk layout, replication, etc. [3,5,7,9,15,17,18,19,21]. To our knowledge, we are the first to study file system accesses and registry accesses with the goal of characterizing and improving the management of PS.

Some studies of distributed file systems do not capture the complete logical file accesses of a system [7,18]. While these traces, captured through network sniffing of NFS or at the disk I/O layer, are sufficient for studying many performance and structural issues, it does not provide the usage information required for improving PS management. Other studies have instrumented system calls or drivers to capture complete logical traces of file system activity [12,17,21]. Our tracing strategy is similar to Vogel’s instrumentation in [20] and Lorch and Smith’s VTrace [11]. We share with VTrace the ability to associate process and user IDs with accesses to PS. However, Hau and Smith’s analysis of VTrace logs in [8] focuses on the I/O of low-level disk blocks. Roselli *et al.*’s comparative analysis of VTrace logs does not consider process or user IDs. In our tracing, we also add the ability to trace accesses to the Windows Registry. In [9], Kroeger and Long find that recent file system accesses can be used to predict the next file system access with high probability.

This is consistent with our findings of repeated activity bursts in Section 6.

Most current techniques for PS management use predetermined information, *e.g.*, manifests to track installed software and rules to detect missing dependencies and known symptoms of problems. Automated or semi-automated tools, such as software installers, anti-virus monitors and configuration error checkers use this predetermined information to effectively reduce the complexity of PS management by focusing on a narrow set of state. The problem with this approach is its reliance on the correctness and completeness of pre-determined information. If new software is added to the system or an unanticipated failure occurs then the predetermined rules will not notice a problem.

Recent work on black-box analysis of the Windows Registry for troubleshooting configuration problems avoids the above problem by replacing predetermined information with dynamically observed truth about software behavior and registry accesses and using administrator interaction to scope the tracing and analysis of events during troubleshooting [22,24]. The black-box analysis used in this paper avoids the problems of predetermined information and uses, for example, the structure of activity bursts to reduce the scale of the analysis problem. Also, there has been previous work on designing models for component interactions [10], and for tracking only the dependencies between executable files [19], where our approach is to focus on the managing all PS.

8. Conclusion and Future Directions

Our goal in performing this study was to better characterize the challenges of PS management, as well as to explore potential methods for tackling the problems facing it. To this end, we collected and analyzed several thousand machine-days of file system and Windows Registry accesses, across a range of server and client environments under real-world workloads. To our knowledge, this is the first such study characterizing how PS is used and managed across multiple environments. Our characterization and analysis provides several interesting observations:

Differences across environments: The variation we observed across environments was significant, and may imply that different state management solutions may be required in different environments. In particular, desktop systems and home machines exhibited widely varying behavior from day to day, whereas server behavior was relatively stable. Thus, for example, monitoring techniques that interpret anomalous behaviors as significant events might be appropriate within Internet services, but not in corporate desktop and home environments.

Management tool overhead: We were surprised to find that 60% of the PS interactions across our traces is overhead from PS management tools, such as configuration tools, anti-virus scanners and auto-update tools. Measuring the performance effects of this extra load on end-users is future work.

Extensibility points: Our case study on discovering extensibility points for the operating system and applications demonstrated how PS interactions can tie dynamic application behavior—in this case, security-critical loading of third-party plug-ins, extensions and malware—to the configuration settings that control the behavior. Exploiting this relationship in the context of other

security-critical yet poorly-understood configuration settings may also prove fruitful.

Poor manifest coverage: We found that current methods for tracking and managing software packages, through *a priori* declared ownership manifests, do a poor job of describing the total state of installed software processes. We believe on-line monitoring of PS interactions could generate more complete and accurate manifests.

Repeated structure: While there are tens of millions of daily accesses to files and registry settings on both server and desktop system under normal load, these file system accesses show a large degree of structure and repetition, in the form of activity bursts. Recognizing this structure enables the volume of events to be reduced by several orders of magnitude. This makes the on-line analysis of PS interactions more feasible.

Overall, we found that tracing the combination of PS interactions and identifying the responsible processes and users for interactions provided a powerful view into system behavior, and believe that incorporating on-line monitoring of this information will be a great aid to PS management. To further research in this area, we will be releasing our data collector instrumentation for general use. We are also investigating how we might anonymize our captured traces of PS interactions for wide distribution.

9. REFERENCES

- [1] W. Arbaugh, W. Fithen, and J. McHugh, "Windows of Vulnerability: A Case Study Analysis," In *IEEE Computer*, Vol. 33, No. 12, Dec. 2000.
- [2] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout, "Measurements of a Distributed File System," In *Proc. of the 13th ACM Symp. on Operating Systems Principles*, Pacific Grove, CA, 1991.
- [3] N. Brownlee, K. Claffy, and E. Nemeth, "DNS Measurements at a Root Server," In *Proc. of the 6th Global Internet Symposium*, San Antonio, TX, 2001.
- [4] J. Douceur and B. Bolosky, "A Large-Scale Study of File-System Contents," in *Proc. of the ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems*, Atlanta, GA, 1999.
- [5] J. Dunagan, R. Roussev, B. Daniels, A. Johnson, C. Verbowski, and Y.-M. Wang, "Towards a Self-managing Software Patching Process Using Black-box Persistent-state Manifests," in *Proc. of the IEEE Intl. Conf. on Autonomic Computing*, New York, NY, 2004.
- [6] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer, "Passive NFS Tracing of Email and Research Workloads," in *Proc. of the 2nd Usenix Conf. on File and Storage Technologies*, San Francisco, CA, 2003.
- [7] J. Hart and J. D'Amelia, "An Analysis of RPM Validation Drift," in *Proc. of the 16th USENIX Conf. on Systems Administration*, Berkeley, CA, 2002.
- [8] W.H. Hau and A. J. Smith, "Characteristics of I/O Traffic in personal compute and server workloads," in *IBM Systems Journal*, Vol. 42, No. 2, pp. 347-372, 2003.
- [9] T. Kroeger and D. Long, "Predicting Future File System Actions from Prior Events," in *Proc. of the USENIX Annual Technical Conference*, San Diego, CA, 1996.
- [10] M. Larsson and I. Crnkovic, "Configuration Management for Component-based Systems," in *Proc. of the 23rd Intl. Conf. on Software Engineering*, Toronto, Canada, 2001.
- [11] J. Lorch and A. Smith, "The VTRace Tool: Building a System Tracer for Windows NT and Windows 2000," in *MSDN Magazine*, Vol. 15, No. 10, Oct. 2000.
- [12] R. Mahajan, D. Wetherall, and T. Anderson, "Understanding BGP Misconfiguration," In *Proc. of the 2004 ACM SIGCOMM Conf.*, Pittsburgh, PA, 2002.
- [13] D. Oppenheimer, A. Ganapathi, and D. Patterson, "Why do Internet services fail, and what can be done about it?" in *Proc. of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, 2003.
- [14] K. K. Ramakrishnan, P. Biswas and R. Karedla, "Analysis of File I/O Traces in Commercial Computing Environments," in *Proc. of the ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems*, Newport, RI, 1992.
- [15] E. Rescorla, "Security Holes... Who Cares?" in *Proc. of the 12th USENIX Security Symp.*, Aug. 2003.
- [16] D. Roselli, J. Lorch, and T. Anderson, "A Comparison of File System Workloads," in *Proc. of the USENIX Annual Technical Conference*, San Diego, CA, 2000.
- [17] C. Ruemmler and J. Wilkes, "UNIX Disk Access Patterns," in *Proc. of USENIX Technical Conference, San Diego, CA, 1993*.
- [18] D. Santry, M. Feeley, N. Hutchinson, A. Veitch, R. Carton, and J. Ofir, "Deciding when to Forget in the Elephant File System," in *Proc. of the 17th ACM Symp. on Operating Systems Principles*, Charleston, SC, 1999.
- [19] Y. Sun and A. Couch, "Global Analysis of Dynamic Library Dependencies," in *Proc. of the 15th USENIX Conf. on Systems Administration.*, San Diego, CA, 2001.
- [20] W. Vogels, "File system usage in Windows NT 4.0," in *Proc. of the 17th ACM Symp. on Operating Systems Principles*, Charleston, SC, 1999.
- [21] H. Wang, C. Guo, D. Simon, and A. Zugenmaier, "Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits," in *Proc. of the 2004 ACM SIGCOMM Conf.*, Portland, OR, 2004.
- [22] Y.-M. Wang, R. Roussev, C. Verbowski, A. Johnson, M.-W. Wu, Y. Huang, and S.-Y. Kuo, "Gatekeeper: Monitoring Auto-Start Extensibility Points (ASEPs) for Spyware Management", in *Proc. of the 18th Large Installation System Administration Conf.*, 2004.
- [23] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. Wang, C. Yuan, and Z. Zhang, "STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support," in *Proc. of the 17th Large Installation Systems Administration Conf.*, San Diego, CA, 2003