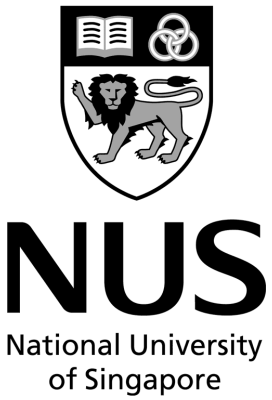
Operating System Auditing and Monitoring

Yongzheng Wu
B.Comp.(Hons.), National University of Singapore



A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2011

Acknowledgments

I would like to use this opportunity to thank all the people who have helped me make this thesis possible.

I thank my supervisor, Dr. Roland Yap, who has advised my research ever since my honours year project. I feel privileged to be led into research of operating system and to work with him. His broad range of knowledge in many areas has inspired me to look at problems from different angles.

I thank my coauthors of research papers for their great contributions. They are Dr. Chang Ee-Chien, Dr. Sufatrio, Felix Halim, Rajiv Ramnath, Dr. Lu Liming and Yu Jie. It was a pleasant experience working with them. I thank my thesis examiners for the valuable and detailed comments.

I thank my family for their support throughout my Ph.D. study. Special thanks to my wife Long Xue for her love; my father Wu Yong for his unconditional kindness; and my son Wu Jien for the joys brought to me.

I acknowledge the support of Temasek Laboratories through the VISCA research grant; and the SELFMAN research project. The excellent research facilities of School of Computing, National University of Singapore are also greatly appreciated.

Contents

Acknowledgments **i**

Summary **v**

1 Introduction **1**
1.1 Motivation 2
1.2 Main Contributions 6
1.3 Thesis Organization 10

2 Background and Related Work **11**
2.1 Windows Issues 11
2.1.1 Closed Source 11
2.1.2 Super User Account 12
2.1.3 Software Management 13
2.1.4 Binaries 14
2.1.5 Other Issues 16
2.2 System Monitoring 17
2.2.1 `printf`, Casual Debugging 19
2.2.2 Traditional Syslog 19
2.2.3 `ptrace` and `/proc` 19
2.2.4 Linux Auditing System 20
2.2.5 Windows Sysinternals 20
2.2.6 Solaris DTrace 21
2.2.7 SystemTap 21
2.2.8 Binary Instrumentation 22

3 Monitoring Infrastructure **23**
3.1 LBox 26
3.1.1 The Monitor Framework 27
3.1.2 Security and Monitor Interactions 33
3.1.3 Using Monitors 37

3.1.4 Implementation Issues . . . 38
3.1.5 Comparing to DTrace . . . 40
3.1.6 Experimental Evaluation . . . 41
3.1.7 Conclusion . . . 45
3.2 WinResMon . . . 46
3.2.1 Motivation and Applications . . . 47
3.2.2 System Design . . . 49
3.2.3 Implementation . . . 55
3.2.4 Writing Custom Analyzers . . . 58
3.2.5 Using WinResMon . . . 60
3.2.6 WinResMon Overhead . . . 61
3.2.7 Related Work . . . 64
3.2.8 Conclusion . . . 65

4 External Monitoring 66
4.1 Introduction . . . 66
4.2 The Framework . . . 69
4.3 Applying the Framework to Malware Detection . . . 71
4.3.1 Methodology . . . 72
4.3.2 Detecting Malware which Sends Spam Email . . . 75
4.3.3 Detecting DDoS Zombie Attacks . . . 79
4.3.4 Detecting Misuse of Compute Resources . . . 83
4.3.5 Handling Exceptions . . . 86
4.3.6 Security Discussion . . . 87
4.4 Application to Access Control and Rate Control . . . 88
4.4.1 Access Control . . . 88
4.4.2 Rate Control . . . 89
4.5 Related Work . . . 90
4.6 Conclusion . . . 92

5 Visualizing System/Software Traces 93
5.1 Comprehending Module Dependencies and Sharing . . . 95
5.1.1 Related Work . . . 96
5.1.2 Visualizing Software Dependencies . . . 96
5.1.3 Explaining the Visualizations . . . 103
5.1.4 Implementation . . . 104
5.1.5 Comprehending Module Dependencies in Real Software . . . 105
5.1.6 Conclusion . . . 113
5.2 Visualizing Windows System Traces . . . 117

5.2.1 Related Work . . . 117
5.2.2 System and Visualization Design . . . 118
5.2.3 VDP Implementation and Scalability . . . 123
5.2.4 Case Studies . . . 129
5.2.5 Conclusion . . . 137

6 Binary Integrity **139**
6.1 BinAuth: Secure Binary Authentication . . . 141
6.1.1 Windows Issues . . . 142
6.1.2 Related Work . . . 144
6.1.3 BinAuth and Software IDs . . . 145
6.1.4 Testing . . . 153
6.1.5 Conclusion . . . 158
6.2 BinInt: Usable System for Binary Integrity . . . 159
6.2.1 Normal Usage versus Malicious Attacks . . . 159
6.2.2 Related Work . . . 161
6.2.3 The BinInt Security Model . . . 163
6.2.4 Implementation . . . 166
6.2.5 Security Analysis . . . 167
6.2.6 Evaluation . . . 168
6.2.7 Conclusion . . . 171

7 Conclusion **172**
7.1 Summary of the Thesis . . . 172
7.2 Future Work . . . 174

Summary

Operating system monitoring is an essential method of obtaining information on running operating systems. The information can be used to understand programs or the operating system kernel. It can be used to verify correctness of the execution or discover problems such as performance bottlenecks and security flaws. This thesis presents our monitoring infrastructures and uses them to solve various problems on software comprehension, software diagnostics and system security.

We first present two monitoring infrastructures, LBox and WinResMon. LBox is a monitoring infrastructure on UNIX variants such as Linux. It features novel user-level monitoring and recursive monitoring, which make LBox safe to be used by unprivileged users in a multi-user environment. It is light-weight as it can be implemented with very little kernel patching; while its performance is comparable to state of the art monitoring systems such as Solaris DTrace. Our second infrastructure, WinResMon, monitors resource usage in Windows. The closed source nature makes Windows internals obscure. Traditional system call based monitoring would not make sense because the semantics of system call names and parameters are not generally understandable. Resource-based monitoring, in contrast, monitors software behaviour on its resource usages such as file/registry, network and process/thread operations. As an infrastructure, WinResMon supports APIs which can be used to build tools for system administrators. Our benchmarking shows that WinResMon is reliable and is comparable to other popular tools.

Our two infrastructures are host-based, i.e. the monitoring system and the monitored software run in the same host. If the kernel of the host is compromised, which is the case for Rootkit, the information from the monitor cannot be trusted. We propose external monitoring which obtains information from entities, such as network routers and environment sensors, that are outside the host. We use the sensors to monitor human user presence and correlate this information with network traffic to detect malware in the host. Moreover, we mitigate the impact of malware by limiting its resource usage, which is done by adapting WinResMon from resource usage monitoring to resource usage control.

With the large amount of information obtained by our system monitor, we have developed techniques to visualize it. We use system traces together with function call trace to

visualize software module dependencies. As the number of modules can be very large, we developed a number of "zooming in" techniques including grouping of modules; filtering by causality; and the "diff" of two dependencies. Our second visualization, named `lviz`, discovers patterns and anomalies. It is highly configurable to suit different purposes. As shown in our case studies, it can be used for software failure diagnostics, analysing performance issues and other strange behaviours.

Many of the system security problems such as malware stem from the fact that untrusted binaries are executed. Since the WinResMon monitoring infrastructure monitors file system related information flow, we can tackle the binary trustworthiness from the information flow point of view, similar to the Biba Integrity Model. In short, low integrity process should not modify high integrity binary and high integrity process should not load low integrity binary. We achieve this goal in two steps. We first implement a secure and efficient *binary authentication* system which only allows binaries in a white-list to be loaded. We then apply it on our *binary integrity* security model. The security model prevents binary related attacks such as DLL planting, drive-by downloading and phishing attacks; while it is usable under typical usage scenarios including software running, installation, updating and development.

Many parts of the thesis is implemented in Windows because of the great variety of software and number of users which also attract many attacks. The closed source nature also makes the monitoring challenging and demanding. However, the ideas can be applied on other operating systems.

List of Tables

2.1 Classification of Monitoring Systems. "Sec.", "transp.", "disc.", "mand.", "instru.", "Lin." and "Win." are abbreviations of Section, transparent, discretionary, mandatory, instrumentation, Linux and Windows respectively. 18

3.1 `open(2)` micro-benchmark on Linux. All times are in seconds. 42
3.2 `open(2)` micro-benchmark on Solaris 10 43
3.3 `connect(2)` micro-benchmark 43
3.4 Macro-benchmarks 44
3.5 Intercepted system calls 58
3.6 Performance comparison on file and registry access (n operations in seconds) 63
3.7 Performance of process creation (in seconds) 64
3.8 Performance of macro-benchmarks (in seconds) 64

4.1 Overview of malware detection rules using changepoint detection. 73
4.2 Detection time of different spam worms. (Detection threshold $N = 120$ emails in $t = 6$ hours at user presence, and $N = 1$ during user absence.) . 76
4.3 Detection time of spam worms, using rate based detection, moving average detection, and changepoint detection 77
4.4 Rules for email detection 78
4.5 Detection time of DDoS attacks of different attack patterns 82
4.6 Detection time of CPU intensive activities, using rate based detection, moving average detection, and changepoint detection (The upper bound of normal CPU temperature is $a = 38.5°C$, and the detection threshold $N = 2400$ in $t = 30$ mins). 85

6.1 Benchmark results showing times (in seconds) and slowdown factors. The worst slowdown factors for each benchmark scenario are shown with underline, whereas the best are in bold. We define $slowdown_x = (time_x - time_{clean})/time_{clean}$. 157
6.2 Performance overhead 170

List of Figures

1.1 Overview of the Contributions 8

2.1 Binaries loaded when running `notepad.exe` in Windows XP 16

3.1 A Simple Monitor 34
3.2 A Tree of Cascaded Monitors 36
3.3 WinResMon overall system architecture 49
3.4 Example of Log Priorities for Trace Compaction 54
3.5 A sample installer wrapper 56
3.6 Overview of how the logger works 57
3.7 A sample analyzer 59

4.1 The components of the framework 69
4.2 False detections caused by email rate based spam detection 75
4.3 Samples of user email rate 76
4.4 Difference in the outgoing packet rate and the net outgoing packet rate (in packets per second) 81
4.5 Distribution of the maximum net outgoing packet rate p_{net} with 13,620 TCP and UDP flows, each flow is observed for 10 minutes during user presence and absence 82
4.6 Net outgoing packet rate of the DDoS attack flow in different attack patterns 82
4.7 Correlation of CPU load and CPU temperature 83
4.8 CPU temperature variation when user is absent and present. The user is absent from 0 to 64,000 second; and present from 64,000 second onwards. The user is absent left of the vertical dotted line and present to the right of the line. 84
4.9 CPU temperature variation during various activities. 84
4.10 Correlating attack intensity and CPU temperature. 85

5.1 Dependency graph without (left) and with (right) grouping of programs A and B with other DLLs D1 to D5 98

5.2 EXE dependency graph of three browsers: IE, Firefox, Opera 99
5.3 DLL dependency graph of `wget` without grouping 101
5.4 Each function is in its own DLL 102
5.5 EXE dependency graph of `wget` 103
5.6 DLL dependency graph of `wget` grouped by functionality 103
5.7 EXE dependency graph of the whole system 107
5.8 Software dependency graph of Microsoft Word and OpenOffice Writer . . 109
5.9 DLL dependency graph of Gimp grouped by functionality 110
5.10 DLL dependency graph of Gimp grouped by software vendor 111
5.11 DLL dependency graph of Firefox grouped by software vendor 112
5.12 Diff of DLL dependency graph of Internet Explorer with Flash and without 114
5.13 Projection of the DLL dependency graph of Internet Explorer on Flash . 115
5.14 Two examples of events 119
5.15 Elements of VDP: axis histograms (Region 1,2); barcodes (3,4); and extended DotPlot (5). This figure is same as Figure 5.16 with the added annotation. 120
5.16 Self-comparison event-ordered VDP of `xcopy` copying 8 files of different sizes with the following configuration rules: 121
5.17 The alternate zoomed-in view of a blue region in Figure 5.16 showing reading (magenta) and writing (cyan) operations. 124
5.18 Clockwise from top-left: histogram equalization, $\gamma = 1$, $\gamma = 1/4$ and $\gamma = 4$. 124
5.19 Event-ordered VDP comparing `cp` (x-axis) and `xcopy` (y-axis) copying the same files. The configurations are the same as in Figure 5.16. 130
5.20 Time-ordered VDP comparing `cp-64k` (x-axis) and `xcopy` (y-axis). The configurations are the same as in Figure 5.16. 130
5.21 Event-ordered VDP comparing a successful (x-axis) software build process and a failed (y-axis) one. 132
5.22 Program point event-ordered VDP of project build: pseudo program point trace (y-axis). 133
5.23 Changing the DP matching rule of Figure 5.21. Left side DP matching rule is operation; Right side is program name. 134
5.24 Time-ordered VDP comparing two idle systems. **a.** (left) comparing one hour interval between two machines; **b.** (middle) zoom in of Region *a2*; **c.** (right) zoom in of *a3*. The different DP color intensity in the zoomed views is caused by histogram equalization. 135
5.25 Time-ordered VDP comparing boot of a clean (Y axis) and a dirty (X axis) system. 136
5.26 Time-ordered VDP comparing IE7 (x-axis) and Chrome (y-axis) performing the SunSpider JavaScript benchmark. 137

6.1 SignatureToMac: Deriving the MAC 148
6.2 Verifier: The Verifier in-kernel authentication process 148

Chapter 1

Introduction

Software is increasingly complex with many interactions among software components and the operating system. The complex interactions make the software ecosystem hard to understand. This is further multiplied by the fact that many software products are closed source. Without proper understanding, many problems arise, such as maintenance problems, performance problems, software bugs and vulnerabilities. The complexity and software bugs also increase the *attack surface* [56], which measures the bugs or features exploitable by malicious attacks. Monitoring is an effective way to understand these interactions. Software monitoring is the process of obtaining useful information from running software. It is used for different purposes and the information obtained is different according to the purposes:

- **Software Comprehension**
  Monitoring can help software comprehension such as studying the control flow and module dependency. Monitoring running software can be used to study the dynamic behaviour which cannot be achieved from static analysis.

- **Inspection**
  We can inspect the expected behaviours in order to verify the correctness of the execution. Sometimes it is not enough to determine the correctness from the output in the narrow sense. For example, the correctness of a web server program not only includes the web page sent to the client, but also the files and databases accessed and the timing. The correctness of the whole web server host is even more complex. This can be verified if we know the expected behaviour and can formalize and monitor it. Similarly, we can look for unexpected behaviours in order to discover problems.

- **Diagnosis**
  In the other direction, if we have a specific problem such as software failure, incorrect output or performance problems, we can diagnose the problem by studying the behaviour and locate the root cause. Software log is probably the most useful

resource for software diagnosis. It is considered as one kind of monitoring, which we call discretionary monitoring. However, the log may not be available or the interested information may not be logged. Mandatory monitoring can obtain the information in this case.

- **Security**
  The access of files and interactions of processes are needed by many security models such as the Biba Integrity Model [23]. A reliable underlying monitoring system is essential to implement these security models. Some intrusion detection systems also work by monitoring malicious behaviours in the operating system.

In the rest of the introductory chapter, we discuss the motivation and challenges in Section 1.1. We then summarize the main contributions of our research in Section 1.2. Finally, we outline rest of the thesis in Section 1.3.

1.1 Motivation

We motivate our research by first giving some examples of problems and then showing how monitoring can help solving them.

1. *"DLL Hell"*

   A software product usually consists of many software modules. A module may depend on another module in the same software product or a different one. This dependency is very complex because firstly there are many software products which include a large number of modules. Secondly, the dependency is not explicitly specified by the module because it may depend on the configuration and input of the software. Lastly, different software products may include duplicate or conflicting modules, which make them overwrite the modules of each other.

   Without proper management of the dependencies, many problems arise. For example, we cannot determine whether a module can be removed when we uninstall a software product. The depended modules may not be available or may be in a incompatible version. The problem is more serious in Windows because of the large number of software products, which are not properly coordinated. This is known as "DLL Hell", where the most common modules in Windows are Dynamic Link Libraries (DLL).

   With a monitoring system that keeps track of module creation and usage, we can heuristically answer the question whether a module can be removed. In addition, when software stops working because of a missing or incompatible module, we can look at the module updating history to identify which software uninstall or update causes it.

2. *Software works yesterday, but not now.*

   We often encounter situations where a program suddenly stops working after configuration changes, software updates or some unknown operations. A similar situation is that the program works in one computer but not in another computer. For example, a program may execute very slowly in one of the computers, but not in others. One way to diagnose this problem it to compare the log or execution trace of the program. The root cause is probably located at the point where the two trace deviates.

3. *How to tell if a host is compromized?*

   If a host (including the operating system kernel) is compromized, information from the host cannot give the answer because the information cannot be trusted. An analogy is asking a crazy person, "are you crazy?" To solve this problem, we have to use information outside the host. This is where the idea of *external monitoring* comes in. We use the information from network routers and sensors which monitor CPU temperature, keyboard typing sound and human user presence, etc. in order to study the host behaviour as a black-box.

4. *Which files are infected by virus?*

   After a user realized that his computer has virus, he wants to know which files or software are infected by the virus. Anti-virus software commonly looks for infected files by matching the virus signatures. This technique usually only finds the main executable of the virus. Other infected files, such as text data files or configuration files, cannot be identified. The user may also want to know whether files containing his confidential information are accessed by the virus.

   We can monitor the access of files, including creation of executables and reading/writing of files in order to track the propagation of the virus. There are two caveats for this monitoring. Firstly, the monitoring has to be always-on because it would be too late to monitor if the files are already infected. This brings the challenge of maintaining the growing log. Secondly, the infected files are not only the files directly modified by the main virus executable. The virus may create additional executables or use shared libraries to hijack other software, which in turn hijacks more software. This brings us the idea of information flow tracking in the system.

5. *How to prevent untrusted program from running?*

   Perhaps we should first ask the question of how to tell if a program can be trusted. We can apply the solution of the previous question (i.e. based on the source of the program) and answer it recursively. If all code (machine code, not source code)

used by the program comes from trusted programs, we consider the program trusted. There are other practical considerations with this simple definition. For example, how to get the initial trusted programs? Is code sufficient? How about data? Trusted programs can be exploited and behave maliciously.

After identifying trusted and untrusted programs, we can use an access control system to prevent untrusted programs from running. The access control system can be implemented in a similar way to our monitoring system, except that the former prevents the access and the latter reports the access.

Although system monitoring helps solving various of problems, there are many challenges.

- The interactions may not be well defined or understood, especially when the source code or proper documentation is not available in operating systems such as Windows. However, we can still discover behaviours such as repeated patterns. Even when the source code is available, it can be difficult to understand because of its large size and dependency with other software.

- In quantum physics, the observer changes the system it observers. Software monitoring also have the same problem. The monitoring system itself can inevitably affect the monitored system in an undesirable way.

- The monitoring system cannot be trusted if the host on which it runs is compromised. Reliable monitoring is always based some assumptions. Most existing monitoring systems rely on the integrity of the operating system kernel. These systems cannot be used to detect kernel malware such as Rootkit.

- Depending on the level of detail, the system call level trace can be several megabytes per second and the instruction level trace can be several gigabytes per second. Moreover, problems such as tracking origin of files require keeping the trace over sufficiently long period. The huge amount of information is hard to maintain and analyze.

There are many monitoring systems for UNIX-like operating systems, but very few for Windows. This is partially because the Windows NT operating system is rather complex and different from other operating systems. It has many unique features and mechanisms which impact on understanding, monitoring and security. We briefly introduce them here and the details are shown later in Section 2.1.

The Windows operating system is a closed source system. This can be seen from three aspects: Firstly, the kernel is closed source, which makes kernel monitoring very difficult. Dynamic instrumentation tools like DTrace [26] and SystemTap [73] are not

relevant because their probes are specific to code points or functions in the kernel. Without understanding the purpose of each function, probes are meaningless. It makes kernel extension difficult as well. The lack of kernel APIs makes anti-virus developers use undocumented internal functions which is not officially supported by Microsoft and may cease to work after a Windows update. Unfortunately, there is no officially supported technique to achieve this. Secondly, the semantic of system calls is closed. Unlike UNIX, programs do not directly invoke systems in Windows. They call higher level APIs, which may call some other APIs, which make the system call. The association between higher level APIs and the systems is complex and again closed. Thirdly, the interaction among the components is closed. Windows has microkernel operating system features which make some tasks, such as networking, printing and graphical interface be partially handled by user space services. In other words, a process can perform tasks on behave of another process. This feature can be exploited to circumvent monitoring or security mechanisms.

Windows users typically use the administrator account to perform all tasks. This is caused by its single-user operating system history and the backward compatibility of the current version. However, this is against the least privilege principle [79] and makes malware capable of performing critical operations. Although User Account Control (UAC) is introduced in recent version of Windows, there are limitations with it.

We use the term *binary* to denote a file that contains native executable code and can be directly loaded by the operating system kernel. There are many types of binaries and they can be loaded and executed in many ways. There are even several version of the same library kept at the same time in the system for the purpose of backward compatibility. The different ways of binary loading increases the "attack surface". For example, the "DLL planting" attack exploits the DLL search order so as to hijack benign DLLs with malicious ones. It is surprising that Microsoft consistently releases fixes and similar attacks consistently reappears [62].

Windows lacks a consistent software management system to manage the installation, update and removal of software. In any case, other systems may not have a mandortary software management system. Most software products have their own installers, which perform installation in different ways. The dependencies and conflicts make binaries in windows rather "chaotic". Firstly, it is not possible to systematically tell which software a binary, or file in general, belongs to. Secondly, the software dependencies are unknown.

There are other features that make monitoring in Windows special. Windows has a central database called the *registry* to store all kinds of configurations including operating system settings, per-user configurations and per-software configurations. There is and API to access the registry. This enables the monitoring of configuration related behaviour.

1.2 Main Contributions

A general monitoring infrastructure needs to be *correct*, *secure*, *transparent*, *flexible*, and *efficient*. By *correct*, the monitored events must be sound and complete, i.e. no events should be missed, duplicated or invented. The monitoring infrastructure needs to be *secure* in both design and implementation. For example, it should not leak confidential information to low privilege users. It should be carefully implemented so that malicious monitored software would not exploit the infrastructure. By *transparent*, the monitored software does not need to be changed. Moreover, its execution including output should be consistent with and without monitoring. By *flexible*, the infrastructure should be sufficiently general to handle different problems. For example, an API can be used to extend the monitored events for future software. A filter language can be used pre-process events. By *efficient*, the infrastructure should not introduce too much overhead on the monitored software. In quantum physics, an observer changes the system it observes. Similarly, a monitor can bring side effects to the monitored program. Too much overhead not only slows the system down, but may also make it incorrect.

We have design and implemented two monitoring infrastructures, LBox and WinResMon. LBox [104] is a monitoring infrastructure on UNIX variants such as Linux. It features novel user-level monitoring and recursive monitoring. User-level monitoring means it is safe to be used by unprivileged users in a multi-user environment. Most traditional monitoring infrastructures are super-user based, mainly because they are system-wide. User-level monitoring requires the monitoring system to have user separation, i.e. a user should not monitor private information of another user. LBox allows hierarchical monitoring. For example, program B monitors program A and that the same time, program C monitors program B. We have implemented LBox in Linux. It is light-weight as it can be implemented with very little kernel patching; while its performance is comparable to state of the art monitoring systems such as Solaris DTrace.

Our second infrastructure, WinResMon [76], monitors resource usage in Windows. The closed source nature makes Windows internals obscure. Traditional system call based monitoring would not make sense because the semantics of system call names and parameters are not generally understandable. Resource-based monitoring, in contrast, monitors software behaviour on its resource usages such as file/registry, network and process/thread operations. As an infrastructure, WinResMon supports APIs which can be used to build tools for system administrators. Our benchmarking shows that WinResMon is reliable and is comparable to other popular tools.

Our two infrastructures are host-based, i.e. the monitoring system and the monitored software run in the same host. If the kernel of the host is compromised, which is the case for Rootkit, the information from the monitor cannot be trusted. We propose external monitoring [29] which obtains information from entities, such as network routers and

environment sensors, which are outside the host. We use the sensors to monitor human user presence and correlate this information with network traffic to detect malware in the host. Moreover, we mitigate the impact of malware by limiting its resource usage, which is done by adapting WinResMon from resource usage monitoring to resource usage control.

With the large amount of information obtained by our system monitor, we have developed techniques to visualize it. Our first visualization [108] investigates the dependencies between programs and binaries. As discussed earlier, software often lives in a complex software eco-system with many interactions and dependencies between different modules or components. This problem is exacerbated both by the overall system complexity and its closed source nature in Windows. Even when the source code is available, there are still interactions with modules which are only in binary form. The visualization uses system traces from WinResMon and program traces from binary instruction, thus it does not need to rely on source code. We use the following scenarios to explain how our visualizations can be used to investigate various aspects of software dependencies: (i) visualizing whole system software dependencies; (ii) visualizing the interactions between selected modules of some software; (iii) discovering unexpected module interactions; and (iv) understanding the source of the modules being used. Because of the large number of modules and their complex dependencies, we developed a number of "zooming in" techniques including grouping of modules; filtering by causality; and the "diff" of two dependencies.

Our second visualization, `lviz` [107], is a visualization tool for many different purposes including software failure diagnostics, analyzing performance issues, anomaly discovery, etc. The visualization is based on DotPlot, which compares two traces and plot the common (or different) items. It was early used for analyzing similarities in DNA sequences [55]. `lviz` extends the traditional DotPlot through a number of visual elements so that we can easily associate the visual representation with events in the trace and identify the key events. As we will see in a number of case studies, `lviz` is highly customizable can be used to look at problems across a large spectrum.

Many of the system security problems such as malware stem from the fact that untrusted binaries are executed. Since the WinResMon monitoring infrastructure monitors file system related information flow, we can tackle the binary trustworthiness from the information flow point of view, similar to the Biba Integrity Model [23]. In short, low integrity process should not modify high integrity binary and high integrity process should not load low integrity binary. We achieve this goal in two steps. We first implement a secure and efficient *binary authentication* system [43, 103] which only allows binaries in a white-list to be loaded. We then apply it on our *binary integrity* security model [105, 106]. The security model prevents binary related attacks such as DLL planting, drive-by downloading and phishing attacks; while it is usable under typical usage scenarios including software running, installation, updating and development.

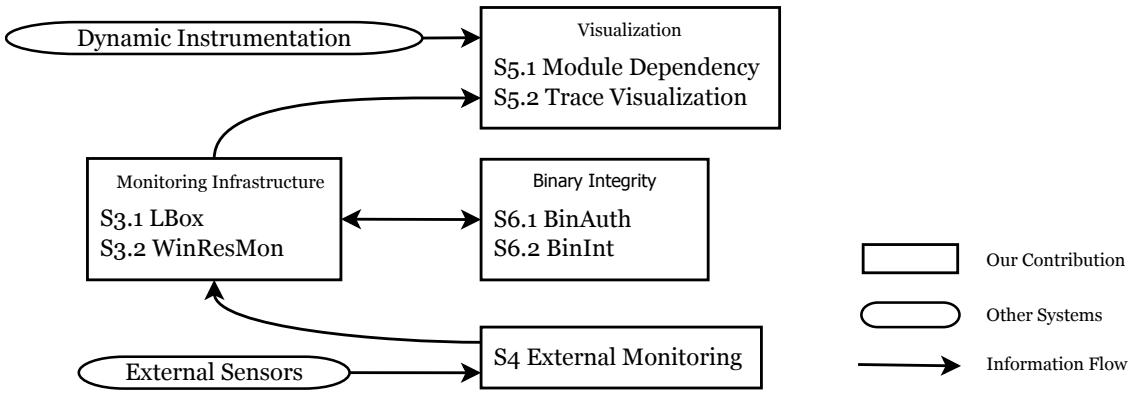


Figure 1.1: Overview of the Contributions

Figure 1.1 visualizes the contributions and relationships between the work in this thesis. The monitoring infrastructures serve as the base in our research. Traces collected by the monitoring infrastructure along with other information is used in various visualizations. External sensors gather information which is used to manage and control resources within and outside a host machine in our external monitoring work. The monitoring infrastructure records the binary related information flow which is used in our binary integrity security model.

Many parts of the thesis are demonstrated in Windows with system prototypes because of the great variety of software and number of users which attract many attacks. The closed source nature also makes the monitoring challenging and demanding. However, the ideas can be applied on other operating systems.

The published works included in this thesis are listed below in chronological order.

1. Yongzheng Wu and Roland H.C. Yap. A user-level framework for auditing and monitoring. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 95–105. IEEE Computer Society, 2005. (in **Section 3.1**)
2. Rajiv Ramnath, Rajiv Sufatrio, Roland H.C. Yap, and Yongzheng Wu. WinResMon: a tool for discovering software dependencies, configuration and requirements in Microsoft Windows. In *Proceedings of the 20th Conference on Large Installation System Administration (LISA'06)*, pages 175–186. USENIX Association, 2006. (in **Section 3.2**)
3. Felix Halim, Rajiv Ramnath, Yongzheng Wu, and Roland H.C. Yap. A lightweight binary authentication system for windows. *Trust Management II*, pages 295–310, 2008. (in **Section 6.1**)
4. Yongzheng Wu, Sufatrio, Roland H.C. Yap, Rajiv Ramnath, and Felix Halim. Establishing software integrity trust: A survey and lightweight authentication system for windows. In Zheng Yan, editor, *Trust Modeling and Management in Digital En-*

vironments: from Social Concept to System Development, chapter 3, pages 78–100. IGI Global, 2009. (in **Section 6.1**)

5. Ee-Chien Chang, Liming Lu, Yongzheng Wu, Roland H. C. Yap, and Jie Yu. Enhancing host security using external environment sensors. In *Proceedings of the 6th International ICST Conference on Security and Privacy in Communication Networks (SecureComm 2010)*, volume 50, pages 362–379. Springer, 2010. (in **Chapter 4**)
6. Yongzheng Wu and Roland H.C. Yap. The problem of usable binary authentication. In *Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement Companion (SSIRI'10)*, pages 34–35. IEEE Computer Society, 2010. (in **Section 6.2**)
7. Yongzheng Wu, Roland H.C. Yap, and Felix Halim. Visualizing Windows system traces. In *Proceedings of the 5th International Symposium on Software visualization (SOFTVIS'10)*, pages 123–132. ACM, 2010. (in **Section 5.2**)
8. Yongzheng Wu, Roland H.C. Yap, and Rajiv Ramnath. Comprehending module dependencies and sharing. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, volume 2, pages 89–98. ACM, 2010. (in **Section 5.1**)
9. Yongzheng Wu and Roland H.C. Yap. Towards a binary integrity system for Windows. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS'11)*, pages 503–507. ACM, 2011. (in **Section 6.2**)
10. Ee-Chien Chang, Liming Lu, Yongzheng Wu, Roland H. C. Yap, and Jie Yu. Enhancing host security using external environment sensors. In *Special Issue in Intentional Journal of Information Security (IJIS)*, Springer, 2011. (to appear) (in **Chapter 4**)

The following are other published works by the author during his doctoral candidature, that are not related to this thesis.

1. Felix Halim, Yongzheng Wu and Roland H.C. Yap. Security Issues in Small World Network Routing. In *Proceedings of the 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2008)*, pages 493–494. IEEE Computer Society, 2008.
2. Felix Halim, Yongzheng Wu and Roland H.C. Yap. Small World Networks as (Semi)-Structured Overlay Networks. In *Workshops Proceedings of the 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO Workshops 2008)*, pages 214–218. IEEE Computer Society, 2008.
3. Felix Halim, Yongzheng Wu and Roland H.C. Yap. Wiki credibility enhancement. In *Proceedings of the 2009 International Symposium on Wikis (WikiSym'09)*, pages

17:1–17:4. ACM, 2009.

4. Felix Halim, Yongzheng Wu and Roland H.C. Yap. Routing in the Watts and Strogatz Small World Networks Revisited. In *Workshops Proceedings of the 4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO Workshops 2010)*, pages 247–250. IEEE Computer Society, 2010.
5. Felix Halim, Roland H.C. Yap and Yongzheng Wu. A MapReduce-Based Maximum-Flow Algorithm for Large Small-World Network Graphs. In *Proceedings of the 2011 IEEE 31th International Conference on Distributed Computing Systems (ICDCS'11)*, pages 192–202. IEEE Computer Society, 2011.

1.3 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 gives some background knowledge on operating system monitoring and Windows. We also show and some existing monitoring systems and tools. Chapter 3 presents our monitoring infrastructures LBox and WinResMon. Chapter 4 shows our research on external monitoring. Chapter 5 presents our two trace visualization works. Chapter 6 shows the binary authentication system and the binary integrity security model. Finally, Chapter 7 concludes the thesis and points out directions for future work.

Chapter 2

Background and Related Work

In this chapter, we give some background knowledge on operating systems and monitoring. In particular, since several parts of the thesis are related to the Windows operating system, we discuss the issues that are related to monitoring in Windows. After that, we show some related work on monitoring.

2.1 Windows Issues

The Windows NT operating system is rather complex and different from other operating systems. It has many unique features and mechanisms which impact on understanding, monitoring and security. We now discuss some of the these which are related to the thesis.

2.1.1 Closed Source

The Windows operating system is a closed source system. Firstly, the kernel is closed source. This makes kernel monitoring very difficult. Dynamic instrumentation tools like DTrace [26] and SystemTap [73] are not relevant because their probes are specific to code points or functions in the kernel. Without understanding the purpose of each function, probes are meaningless. It makes kernel extension difficult as well. The lack of kernel APIs make anti virus developers use undocumented internal functions in a hacking way. For example, the Kaspersky is known [84] to patch internal kernel functions, which makes it only work on 32-bit but not 64-bit systems. In our WinResMon (Sec 3.2) work, we monitor system calls by hooking the kernel dispatch table, which is a well-known system call monitoring technique, but not officially supported by Microsoft and may cease to work after a Windows update. Unfortunately, there is no officially supported technique to achieve this.

Secondly, the semantic of system calls is closed. Unlike UNIX, programs do not directly invoke systems in Windows. They call higher level APIs, which may call some other APIs, which make the system call. The association between higher level APIs and

the systems is complex and again closed. For example, to open a file in UNIX, one may call the `open(2)` system call directly. In Windows, one should call the officially documented API `CreateFile()`. `CreateFile()` calls `CreateFileA()` which calls the system call `ZwCreateFile()`. One may think this is not a problem because we can just monitor the documented API layer and ignore the system call layer. However, not all programs follow the documented API. To make reliable monitoring, system call have to be monitored.

Thirdly, the interaction among the components is closed. Windows has microkernel operating system features which make some tasks, such as networking, printing and graphical interface be partially handled by user space services. In other words, a process can perform tasks on behave of another process. This feature can be exploited to circumvent monitoring or security mechanisms.

2.1.2 Super User Account

The early versions of Windows (Windows 95, Windows 95 and Windows Me) are single user operating systems, thus do not distinguish normal and super user accounts. Windows NT introduced the multiple user operating system, which separates user configurations and introduces normal/super user account. The super user account (also known as administrator) has higher privilege and is supposed to only perform administrative tasks following the least privilege principle [79]. However, in practice, most users choose to use the super user account, because some software written for older Windows do not work if running using normal account. Furthermore, the first account created during Windows 2000 and XP installation is by default administrator, thus running normal account is an opt-in feature and many users are even not aware of using administrator account.

In modern multi-user operating systems, (i) separation of kernel and user context; (ii) separation of different processes' address space; and (iii) separation of different users' configurations are very important concepts of security. When programs run under the super user account, all these separations are invalidated because the super user is able to load kernel drivers, modify arbitrary process's state and arbitrary files. As a result, the recent versions Windows Vista and Windows 7 introduced User Account Control (UAC) in order to mitigate the security problems of super user account and promote the use of normal user account. When a program running in super user account performs administrative operations (listed below), a UAC prompt is displayed and the user can choose to authorize or prevent the operation. It is designed to prevent malware from *automatically* perform these operations.

- Installing and uninstalling applications
- Installing device drivers

- Installing ActiveX controls
- Installing Windows Updates
- Changing settings for Windows Firewall
- Changing UAC settings
- Configuring Windows Update
- Adding or removing user accounts
- Changing a user's account type
- Configuring Parental Controls
- Running Task Scheduler
- Restoring backed-up system files
- Viewing or changing another user's folders and files

There are a few problems with UAC. Firstly, UAC only cares about administrative operations, which are to do with system settings, but not user settings. Thus, this is aimed at protecting the system but not the user, i.e. malware which do not modify system resources are not affected. This is alright from a multi-user security perspective, which focuses on preventing a user from interfering other users. However, in a single-user situation, which is mostly the case for Windows PC, it is more relevant to prevent an application from interfering with other applications of the same user. For example, UAC cannot prevent malware from stealing web browser cookies or modifying Word documents. Some software, such as Google Chrome, are by default installed in the user's home directory instead of the `Program Files` directory and is consequently not covered. UAC does not protect their binaries from being modified. Secondly the protection from UAC may be illusory — a common complaint is that frequent UAC prompts leads to users blindly allowing UAC queries [64]. Lastly, the UAC prompt does not give much information to make a decision, which is essentially whether a particular executable is trusted for an operation. Most users (including technical ones) would not be able to decide if the operation should be allowed.

2.1.3 Software Management

A software management system controls the installation, updating and removal of software in the operating system. Open source OS, such as Linux, commonly uses a package management system. Examples are the Redhat Package Manager (RPM) for Redhat and Fedora Linux, the Debian package management system for Debian and Ubuntu Linux.

Mobile operating systems such Apple's iOS and Google's Android have similar application managers.

There is no consistent software management in Windows. Most software products have their own installers, which perform installation in different ways. They have their updaters as well. Some check online for update at each execution. Some perform regular checks, which involves running a service in the background. Probably the only common thing is that they all register their software removal programs so that the "Add/Remove Program" tool knows how to remove them.

Without a consistent software management system, binaries in windows is rather "chaotic". Firstly, it is not possible to systematically tell which software a binary, or file in general, belongs to. There is no database to record this relationship as in RPM. The directory structure is not reliable to figure this out because binaries can be installed anywhere in general. For example, some software install binaries in the `C:\windows\system32` directory which is used to store core system files. Even worse, a software installer can overwrite binaries installed by another software. Secondly, the software dependencies are unknown. Package managers such as RPM keeps detailed and accurate package dependencies and conflicts so that when installing a software, it installs its dependencies as well. Because Windows lacks this knowledge, software tends to bundle its dependencies. This introduces a lot of duplicates in the system and makes software update difficult.

Even with software management systems, the problems cannot be fully solved because the systems do not provide mandatory protection. They can only maintain the software installation under the assumptions that (i) the software package is centrally created and signed by a single distributer, e.g. RedHat; (ii) software is installed properly using their tools; and (iii) the installed files are not modified outside the software management system. However, any of the assumption can be false. There is often some software that is not included by the distributor, either because it is new or it does not satisfy the distributor's requirement. Third party software packages are then made. Examples are the Google Chrome web browser for most Linux distributions and the Cydia application repository for Apple's iOS. These packages can conflict with existing or future first party packages. Package signature can partially solve the problem, but users usually ignore the signature. The software management systems do not provide mandatory protection mechanisms to prevent installed files from being modified. Once the root privilege (or any software installation equivalent privilege) is acquired, any file from any software can be modified.

2.1.4 Binaries

Parts of the thesis, including module dependency (Section 5.1) and binary integrity (Chap. 6) focus on binaries in Windows. Throughout the thesis, we use the term binary to denote a file that contains native executable code and can be directly loaded

by the operating system kernel. There are generally three types of binaries in modern operating systems. An *executable* file is the main and firstly loaded binary of a process. Executable files in Windows are conventionally (not necessarily) given the `.exe` file extension. A process loads a single executable file. A *dynamic linked library* (or DLL) contains native code which is designed to be shared by different software. A process can load many DLLs and a DLL can be loaded by many processes. In Windows, DLLs are usually given the extension `.dll`, but other extensions such as `.ocx`, `.cpl` and `.ime` exist as well. A *kernel driver* can be loaded by the kernel and its code executes in kernel space. They usually have the `.sys` extension. We list some of the common binaries and their conventional extensions below (this list is not intended to be exhaustive):

- Applications (`.exe`) — the executable associated with a process
- Command files (`.com`) — legacy executables for the MS-DOS environment
- Dynamic linked libraries (`.dll`) — libraries loaded implicitly by dependency or explicitly by `LoadLibrary` function at runtime
- ActiveX controls (`.ocx`) — software components which implement the Microsoft Component Object Model (COM). ActiveX controls are most commonly used by Internet Explorer. Browser helper objects(BHO) are also an example of ActiveX controls.
- Device drivers (`.drv` and `.sys`) — kernel loadable modules
- Screensavers (`.scr`) — executables used by Windows to display screensavers
- Control Panel applets (`.cpl`) — applets for the Control Panel
- Input Method Editors (`.ime`) — used by the Windows On-screen-keyboard to support different languages
- Codecs (`.acm` and `.ax`) — bundled into Windows and also can be installed by 3rd party software. The codecs are used for playing audio and video media.

We briefly discuss the binaries loaded when we run `notepad.exe`, the simplest text editor of Windows, as shown in Figure 2.1. Binaries with different extensions — `.dll`, `.drv` and `.ime` — are loaded by the simple text editor. We highlight that `avgrsstx.dll`, which is one of the DLLs of the anti-virus software AVG, is "injected" into every processes. This hacker-style DLL injection technique is not officially supported and is also used by many malware. This makes some anti-virus software behaves like malware. The `comctl32.dll` with long pathname is version 6.0.2600.5512 of the common control DLL, which keeps several different versions for backward compatibility. The `winspool.drv` is the user space component of the printer driver.

```
c:\windows\apppatch\acgenral.dll
c:\windows\system32\avgrsstx.dll
c:\windows\system32\imm32.dll
c:\windows\system32\lpk.dll
c:\windows\system32\msacm32.dll
c:\windows\system32\msctf.dll
c:\windows\system32\msctfime.ime
c:\windows\system32\shimeng.dll
c:\windows\system32\usp10.dll
c:\windows\system32\uxtheme.dll
c:\windows\system32\winmm.dll
c:\windows\system32\winspool.drv
c:\windows\winsxs\x86_microsoft.windows.common-controls_
6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83\comctl32.dll
```

Figure 2.1: Binaries loaded when running `notepad.exe` in Windows XP

Loading of binaries is the most common way to lead code execution. The complexity of binaries in Windows brings challenges to software understanding and security. We will see this in more detail in our visualization (Chap. 5) and binary authentication (Chap. 6) work.

2.1.5 Other Issues

In Windows, the configuration is stored in a central database named the *registry*. This includes all kinds of configuration, such as operating system settings like TCP buffer size; per-user configurations like desktop background; and software configurations like default font size for Adobe Photoshop. The centrally managed configuration database is a unique[1] feature of Windows. UNIX variants usually manage configurations in a per application basis. Each application keeps its own configuration file (typically a text file in `/etc/` or the home directory). Our WinResMon (Section 3.2) monitors the registry related system calls. It can monitor program accessing *individual* settings. If a text based configuration file is used, this is much more difficult and may require data flow analysis. In this sense, WinResMon benefits from this feature. In Section 5.2.4, we will use our LViz to study the software behaviours of accessing configurations by visualizing the log generated by WinResMon.

Windows has a different way in organizing the file system hierarchy. While UNIX variants organize the file system in a single tree, Windows adopts the notion of drive (or volume). Each drive contains a separate file system tree, thus all drives form a file system forest. Historically, file and directory names are constrained by the 8.3 standard, i.e. maximum eight letters name with three letters extension. Later versions of Windows

[1]There are other central configuration systems such as GConf for GNOME applications. However, they are not as widely adopted as the registry in Windows.

allow longer name, but for backward compatibility, each long file name is automatically assigned by kernel a 8.3 name, so that it can be accessed by legacy software. For example, `C:\Program Files` is equipment to `C:\PROGRA~1`. This feature is often exploited to circumvent security systems with file blacklist mechanisms. In the kernel, files are named in a different way. For example, the user space pathname `C:\foo.txt` is equivalent to the kernel space pathname `\Device\HarddiskVolume1\foo.txt`, depending on the underlying hard disk and partition layout. Similar to most UNIX flavoured file systems, symbolic and hard links are supported by the Windows NT file system (NTFS). All these features make a challenging task to obtain unique identifiers to files. This problem is answered in our binary authentication work (Section 6.1).

2.2 System Monitoring

Related work of monitoring can be classified in a number of ways. From the enforcement point of view, we have discretionary and mandatory monitoring. Discretionary monitoring requires that the monitored software actively report to its monitor. The traditional UNIX `syslog` is an example of discretionary monitoring. A log entry is generated when the monitored software calls `syslog(3)`. The naive `printf()` debugging technique is also discretionary monitoring. In contrast, mandatory monitoring systems enforce that logs entries are always generated when certain actions are performed by the monitored software. The `ptrace(2)` interface and Solaris Basic Security Module (BSM) Auditing are examples of mandatory monitoring. Mandatory monitoring is more suited for security purpose because of its enforcement. Discretionary monitoring may give more friendly output since the monitored software knows which pieces are more important.

A correlated classification is transparent/opaque monitoring. In transparent monitoring, the monitored software does not need to be adapted and sometimes is not aware of being monitored; whereas in opaque monitoring, the monitored software need to be either rewritten and recompiled or transformed manually. The two types of classification are usually correlated because transparent monitoring is usually mandatory as well, and opaque monitoring is usually discretionary.

From execution environment point of view, the monitor can be executed in a number of environments.

- The monitor can be executed in the process (i.e. same address space) of the monitored software. The naive `printf()` debugging technique belong to this category. A common technique to monitor API or DLL function calls is to rewrite the to-be-monitored function with a wrapper function which does the logging and then calls the actual function. This type of monitor is usually used for debugging purpose, and not for security, because the monitoring can be circumvented.

System	Enforce	Transp.	Level	Alter	OS	Sec.
`printf`	disc.	opaque	user	log	all	2.2.1
`syslog`	disc.	opaque	user	log	POSIX	2.2.2
`ptrace(2)`	mand.	transp.	kernel	alter	POSIX	2.2.3
`/proc`	mand.	transp.	kernel	alter	Solaris	2.2.3
Linux audit	disc.	opaque	kernel	log	Linux	2.2.4
FileMon RegMon ProcMon	mand.	transp.	kernel	log	Windows	2.2.5
DTrace	both	opaque	kernel	log	Solaris	2.2.6
DProbes SystemTap	both	opaque	kernel	log	Solaris	2.2.7
Pin DynamoRIO	mand.	transp.	instru.	alter	Lin, Win.	2.2.8
valgrind	mand.	transp.	instru.	alter	Linux	2.2.8

Table 2.1: Classification of Monitoring Systems. "Sec.", "transp.", "disc.", "mand.", "instru.", "Lin." and "Win." are abbreviations of Section, transparent, discretionary, mandatory, instrumentation, Linux and Windows respectively.

- In order to prevent circumvention, the monitor can be executed in the kernel (assuming the kernel is authentic). The `strace` utility uses the kernel `ptrace(2)` interface to monitor system calls. Most of the related work that we are going to introduce are kernel based.

- In the same vein, to securely monitor kernel events, the monitor should execute in a lower level than the kernel, i.e. the hypervisor. Examples are the virtual machine monitors. The recent Intel and AMD processors support hardware virtualization features which can virtualize and monitor an unmodified kernel with almost no performance penalty.

- The *instrumentation* technique is used to get instruction level monitoring such as memory load, memory store, and branch events. Section 2.2.8 will discuss this in detail.

- To get lower level information such as TLB or cache miss rate, hardware monitoring need to be used.

Another classification is whether the monitoring system is able to alter the execution of the monitored software. Logging systems such as `syslog` only record the events, but do not alter the execution (except perhaps performance overhead). `ptrace(2)`, on the other hand, can be used to filter system calls or change system call's arguments.

Before discussing each related work in detail, Table 2.1 lists the classification of them.

2.2.1 `printf`, Casual Debugging

Directly printing debugging message to console is probably the mostly widely used debugging technique for simple programs, because of its portability and simplicity. In C, `printf` is most commonly used for this purpose. Other languages and environments have similar representatives such as `System.out.println` in Java and `printk` in Linux kernel. However, this technique is not used in more sophisticated software. Despite the reason that the debugging message can mess up with the actual output, `printf` lacks the separation between the monitor and the monitored program. This means that monitoring output can be modified or removed by the monitored program, thus a bug in the program can mess up the monitoring output.

2.2.2 Traditional Syslog

Seeing the problem of `printf`, people developed the syslog framework, which is the most widely supported logging framework for UNIX-like systems. `syslog` separates log generation program and log recording program. It works by letting the log generation program call the `syslog(3)` function, which talks to a dedicated daemon `syslogd`, which receives and records the log. Kernel messages generated by `printk` are send to `syslogd` through the middle man `klogd` in a similar way.

Although syslog protect the log from log generator, we consider it as discretionary monitoring because it requires the monitored program (log generator) to actively call `syslog(3)` in order to generate a log message. More specifically, if the monitored program is compromised, it cannot modify already logged messages, but can suppress or spoof new log messages.

2.2.3 `ptrace` and `/proc`

System call, the interface between user and kernel space, is often monitored for various purpose. The UNIX `ptrace` and Solaris `/proc` [37] are commonly used for system call monitoring because of their portability. To use `ptrace`, the monitor calls the `ptrace(2)` system call and specifies the process ID of another process to be monitored and waits. When the monitored process makes a system call, the monitoring process is waked up. The monitoring process can then check or modify system call parameters or return values. The Solaris `/proc` works in a similar way except that instead of calling `ptrace(2)`, it performs IO controls on the `/proc/[pid]/ctl` file. A subset system calls can be specified in `/proc`, while `ptrace` must monitor all system calls.

`ptrace` and `/proc` are mandatory monitoring systems because the monitored program cannot evade the monitoring as long as it make the system call. They are transparent monitoring systems because in general, the monitored program is not aware of the monitoring. Because of this, systems like Janus [93] or Alcatraz [53] use them to do system

call monitoring. However, this usage is problematic because it is not meant to be a secure monitoring mechanism, e.g. `ptrace` was meant to support debuggers. In the Solaris manual pages, `ptrace` is described as being "unique and arcane". These kinds of problems and common pitfalls with user-level system call interposition are discussed by Garfinkel [40], such as: (i) race conditions between time of check and time of use (TOCTOU), i.e. a buffer can be modified by another thread; (ii) non-inheritance of tracing, i.e. special `strace` hacks in Linux; and (iii) not transparent with respect to setuid/setgid executables and signals, i.e. `ptrace` and `/proc` disable tracing on setuid/setgid executables. In both `ptrace` and `/proc`, when a traced process calls `setuid(2)`, the call will fail because the tracing process would have insufficient privileges to the `setuid` process. Because of their subtleties and intrinsic difficulties, `ptrace` and `/proc` are not suitable for general purpose user-level monitoring although they may be useful in specific situations.

The other serious drawback of `ptrace` or `/proc` is that the overhead is considerable, incurring at least two context switches per traced system call. Our micro benchmarks in Section 3.1.6 show that this can lead to an order of magnitude slowdown on system call intensive programs.

2.2.4 Linux Auditing System

The Linux auditing system (also known as lightweight auditing framework) is used to monitor kernel events such as system calls and file system operations. The system consists of the kernel space event record producer and the user space event record consumer (i.e. the audit daemon `auditd`). At compile time, kernel developers insert audit code into the kernel. At run time, system administrators control which event and what information to record using the `auditctl` tool. All event records are transmitted through netlink sockets to `auditd`. The event records are stored in a custom database which can be queried using the `ausearch` and `aureport` tools. The auditing system is incorporated into Linux kernel since version 2.6.4, and is available in almost all Linux distributions.

The auditing system is a discretionary monitoring system to the kernel because monitoring code is manually inserted by the kernel developer and can be circumvented by kernel code. However, when used for monitoring system calls of a user program, it can be considered as mandatory if the kernel is assumed to be authentic. The system only performs logging and does not alter the execution, thus buffering of event records can be used to reduce context switches and improve performance.

2.2.5 Windows Sysinternals

FileMon [4] and RegMon [7] are file and registry monitoring tools for Windows, respectively. They monitor operations taking place on the registry or specified file system. A graphical interface is used to filter and display monitored events in real time. A later

tool named Process Monitor combines features from both tools and adds thread/process related event monitoring and event filtering. The tools are closed source, so we study them by monitoring them using our monitoring tool WinResMon (in Section 3.2). We found that they work by intercepting system calls and making use of the kernel file system filter API. Upon execution, a kernel driver is created in a temporary directory. The driver is then loaded into the kernel and start to intercept the kernel operations. A named pipe is used to transmit event records.

The monitoring tools are standalone GUI programs, which do not provide API to be used by other software. We have observed that when events are rapidly generated, all their tools can drop events. The details are covered in in Section 3.2.6.

2.2.6 Solaris DTrace

DTrace [26] is a dynamic tracing framework created on Solaris 10, for troubleshooting kernel and application problems on production systems. Software developers insert probes into the code of the software (kernel or user space program) at compile time. System administrators or users monitor the execution by writing a script in the *D language* and associating them with the probes, so that when the software executes over the probes, the script is executed. There are about 48,000 probes[2] in almost all aspects of the kernel. The D script runs in the kernel and thus reduces the context switch. For example, to count the number of `write(2)` system call of a process, an integer variable is declared and a script which increments it is associated with the `syscall::write:entry` probe. Only one context switch is needed to output the final count. To do this using `ptrace(2)`, a pair of context switch is needed for each `write(2)` system call.

Having monitoring code dynamically (That is where the D comes from) generated at runtime and executed in kernel is the key feature of DTrace. This poses a security threat as well however. To prevent D script from running into infinite loops, loops (or backward branch in general) and user defined functions are not supported.

2.2.7 SystemTap

SystemTap [73] is developed to bring the idea of dynamic instrumentation in DTrace into Linux and possibly other UNIX variants. Before we describe SystemTap, let us first look at its predecessor DProbes [63, 22] and the underlying KProbes [58], which was introduced earlier than DTrace.

KProbes is a Linux kernel framework which allows dynamic instrumentation of kernel code. The KProbes API consists of two components, the probe registration functions and

[2]Since both Solaris kernel and DTrace are in active development, our information is based on its current status in June 2011. The number of probes is counted by executing "`dtrace -l | wc -l`" in Solaris 10u9 x86.

the probe activation functions. The probe registration functions mark points in the code that can be instrumented. What it actually does is inserting a few `nop` instructions. The probe activation functions associate registered code points with probe handlers, which are called when the code points are executed. What it actually does is rewriting the `nop` instructions with a jump instruction targeting to the handler. There are some optimization techniques, such as return address rewriting, but the basic idea is the same.

KProbes is not convenient to use because its API are solely in kernel, thus only kernel developers can use it. DProbes is developed to allow user space program to make use of the probe activation functions. The way DProbes works is similar to DTrace, where a compiler is used to compile a script which defines the probe handler, and the compiler feeds the compiled script to the kernel to execute. What is different is that instead of compiling into intermediate byte code as in DTrace, DProbes compiles directly into native machine code which is feed to KProbes' activation functions. The DProbes language is rather simple comparing to DTrace. It is written in an assembly-like language, based on the Reverse Polish Notation. Logic, arithmetic and control flow operations are supported. To prevent infinite loops, the number of branches is capped.

SystemTap also uses KProbes as the underlying kernel mechanism. It uses a more advanced C-like language, where functions are supported and a collection of library functions are provided.

2.2.8 Binary Instrumentation

The above mentioned monitoring systems are targeting at specific code points, which are usually software specific. Sometimes we need to monitor the instruction level behaviour. For example, in order to study the control flow of a program, we need to monitor all branching instructions. We consider there to be three ways to achieve this. The first way is to *emulate* the CPU, i.e. implement the CPU in software. The advantage is that cross architecture emulation is possible, thus it is quite portable. However, emulation is very slow. The second way is static binary instrumentation. The monitored binary is translated to add the monitoring code. The resulting binary is executed natively in the CPU, thus is much faster than emulation. The problem is that static disassembly in not reliable, especially in the case of variable opcode size and dynamic generated code. The third way is dynamic binary instrumentation. Each basic block (contiguous instructions without branches in the middle) is translated just before execution. This is also known as just-in-time translation. There are a number of dynamic instrumentation systems available, such as Pin [54], DynamoRIO [25] and Valgrind [68]. In our module dependency visualization work (Sec 5.1), we monitor all function calls in a program using Pin.

Chapter 3

Monitoring Infrastructure

System monitoring is an important task on ensuring a correct running system. It can be used to confirm or verify the correctness of a running system; diagnose system failure; identify performance problems; and find security problems. As system grows larger and more complicated, these tasks become more challenging.

A general monitoring infrastructure needs to be *correct*, *secure*, *transparent*, *flexible*, and *efficient*. By *correct*, the monitored events must be sound and complete, i.e. no events should be missed, duplicated or invented. In some situation, events can be generated faster than the monitor can handle. In this case, a choice must be made to either discard the events or suspend the monitored program. When we monitor for security purpose, the latter is preferred. However, this could affect the monitored software and sometimes may even cause dead lock. The Solaris DTrace (Section 2.2.6) adopts the former for the reliability and performance of the monitored software. We believe that a monitoring system should let the user make the choice, because different scenarios may have different trade-off.

The monitoring infrastructure needs to be *secure* in both design and implementation. For example, it should not leak confidential information to low privilege users. It should be carefully implemented so that a malicious monitored software would not exploit the infrastructure.

There are many definitions on *transparency*. An early definition can be traced back to the Popek and Goldberg virtualization requirements [71] on equivalence and efficiency of virtual machines. Here, we give two definitions, a weaker one and a stronger one. (i) The monitored software does not need to be adapted (e.g. rewritten or recompiled) for the monitoring. In other words, the monitoring should work even if the author of the monitored software is not aware of it. `syslog` requires the monitored program to call the `syslog(3)` function, thus `syslog` is not transparent in this definition. Monitors such as DTrace and DProbes require the monitored software to call their probe API. When they are used to monitor the kernel, they are not considered transparent, because

the kernel has to be rewritten to call their probe API. However, when they are used to monitor the system calls made by a program, they are considered transparent, because the program does not need to be rewritten. In this case, DTrace and the kernel as a whole is considered as the monitor. (ii) The monitoring is undetectable by the monitored software. The monitor may change the execution environment, which can be detected. For example, some system call monitors are implemented by patching the user space dispatching table. They can be detected by examining the table. Other monitors can be detected by timing analysis. These monitors are not transparent under this definition. We believe that the former definition is enough for general purpose monitoring. The latter is too costly, because it either incurs large performance overhead if implemented in software; or requires special hardware. Moreover, study [75, 41] has shown that existing software and hardware virtualizers can be easily detected.

By *flexible*, the infrastructure should be sufficiently general to handle different problems. For example, an API can be used to extend the monitored events for future software. A filter language can be used to pre-process events.

By *efficient*, the infrastructure should not incur too much overhead on the monitored software. An observer is part of the system and changes the system, similarly, a monitor can bring side effects to the monitored program. Too much overhead not only slows the system down, but may also make it incorrect.

In this chapter, we start by giving some background of monitoring techniques and show some related work. We then propose two general monitoring infrastructures. The LBox addresses the problem of *user-level* monitoring. Most traditional monitoring infrastructures are super-user based, mainly because they are system-wide. With user-level monitoring, LBox can be used by all users in a multi-user system, moreover, LBox allows monitor to be cascading. However, this poses several new challenges. Allowing all users to do monitoring changes the adversary model because users, unlike administrators, can be untrusted. If not carefully designed, the monitoring infrastructure can be exploited by malicious users to obtain confidential information such as other users' password. Cascade monitoring allows monitors to be monitored by other monitors. The monitoring infrastructure has to prevent infinite message loop-back, which can be caused by, for example, two monitors generating events for each other.

Our second monitoring infrastructures, WinResMon addresses the problem of extensible resource-based monitoring in Windows. In open source operating systems such as Linux, both the internal design and system call API are understandable by the developer, thus system based monitoring makes sense. However, as we discussed in Section 2.1, in Windows, the native calls are not documented and continuously changing. Though it is possible to monitor native calls, the output would not be generally understandable. WinResMon addresses the problem from a resource usage point of view. It monitors resource usage of all processes in the system. Its main use is to inspect resource access, software

dependency and maintaince issues. As an infrastructure, it can be used to build tools for custom queries for system administrators. WinResMon differs from LBox as it provides whole system monitoring, because the software maintenance problems usually require global view of the system, and some problems require always-on monitoring for a long period. LBox is designed to study a single process or a group of processes launched for a single task, thus the monitoring can be usually isolated to the related processes and the particular run. Our benchmarking shows that WinResMon is reliable and is comparable to other popular tools.

3.1 LBox

Logging and auditing are important operating system facilities used to help monitor correct system operation and to detect potential security problems. In Unix systems, logging is traditionally *application based*. The application itself controls what is being logged through the system logging mechanism `syslog` (Section 2.2.2), e.g. security audit log messages generated by `login`, `su`, etc. The drawback of application logging is since it is under the control of an application which may be compromised or malicious, no security guarantees are possible. More secure versions of Unix have finer grained auditing mechanisms to satisfy the Trusted Computer System Evaluation Criteria (TCSEC) or Common Criteria (CC) security requirements. The Solaris Basic Security Module [69] for example defines kernel auditing events which can serve to log certain system calls. Such auditing is typically system-wide on all processes and requires administrator privileges.

Traditional auditing mechanisms are designed mainly for system audit trail purposes. As such, they are not sufficient for the needs of more demanding security monitoring applications such as intrusion detection systems (IDS), determining correct application behavior, detecting improper system usage, etc. In this section, we present an approach to auditing and monitoring which is sufficiently flexible for a variety of applications. We provide a kernel extension which enables easy programming of user level (as opposed to kernel level) monitors for observing the effects of system calls made by specified processes of interest. Our philosophy is to separate mechanism from policy. A kernel-level mechanism provides transparent, secure and efficient monitoring, while the core logic and functionality is encapsulated in a user-level monitor. Having a user-space monitor means that we do not have to worry about code safety issues unlike a kernel-level one. As user-level monitors do not have to be privileged, ordinary users can create/run their own monitoring tools. We show that general purpose user-level monitors are easy to write without requiring any knowledge of kernel programming. In the remainder of this section, we will refer to monitoring as encompassing the concept of auditing and logging.

We provide a number of security guarantees: (i) the selected processes (which can include their children) cannot circumvent monitoring, we call this *mandatory monitoring*; (ii) none of the operations/events of interest from the set of monitored processes are missed, we call this *reliable monitoring*; and (iii) the monitor cannot escalate its privileges, only exactly the operations/events of processes at the same privilege level can be monitored. The mandatory and reliable properties are necessary to ensure that a monitor can be used for security purposes. The last property is important since the user-level monitors can be unprivileged. Finally we also require that the monitor be *transparent* to the monitored processes — thus the act of being monitored has no side effects to the monitorees. We remark that traditional Unix mechanisms such as `ptrace` and `proc` (Section 2.2.3) do not provide these guarantees.

A key objective is that the monitoring mechanism be efficient and scalable. By efficiency, we mean that fine-grained monitoring is possible with low overheads. Scalability means that the cost of monitoring should be dependent on how much is being monitored and the amount of information desired. The cost should be controllable by the monitor so that overhead is commensurate with need. In the end, we want to be able to have several fine-grained root-level and unprivileged monitors to be permanently running without paying too high a price. On one end of the spectrum, we allow for global monitors which log all interesting events across all processes to disk like an audit log; and on the other end, the monitor might only be concerned with writes to particular system files from particular processes and then perform sophisticated analysis.

Consider the following motivating example. Suppose we want to monitor whether a web server has been attacked, perhaps as part of an IDS. The web server logs cannot be used since either the server or the logs could be compromised. A traditional auditing facility like a disk based log would have a number of problems. Firstly, there may be confidentiality issues in giving the system log to the IDS, the IDS may gain access to confidential information (assuming it isn't running as root). Another question is what happens if the disk log causes the filesystem to run out of space? Add a network IDS to this scenario will further strain the audit log! One could use `ptrace` to monitor the web server but this can have a significant performance penalty and may not ensure mandatory or reliable auditing.

Our prototype implementation shows that it is possible to to get all these desirable features in a user-space monitor without requiring special privileges. Furthermore, we demonstrate an efficient implementation which has low overhead even though the monitors are in user-space.

3.1.1 The Monitor Framework

A monitor is a user-space process which audits the behavior of other processes. Monitors are described by two specifications: (a) a process specification defining which processes to monitor; and (b) event specifications which define what operations to monitor from those processes. In what follows, we describe the design of our monitoring framework and portions of the API. The API is actually a user library which provides a convenient interface to the kernel monitoring interface. Instead of documenting the underlying details, we illustrate by examples.

3.1.1.1 The Monitor Process Specification

An arbitrary collection of processes, not necessarily related by parent-child relationships can be designated for mandatory monitoring. To allow for flexibility and dynamic process creation (including children), we use an API for constructing boolean expression in a

functional lisp-like style which allows easy creation in C. The boolean expression is built from the following predicates using the following usual boolean operators, `AND`, `OR` and `NOT`:

1. *true/false*: For example, a global specification to monitor all processes is simply the boolean expression *true*.

2. *uid/euid/suid/fsuid* (user id): These user identities are used to identify the owner of a process in different contexts. For example, the *fsuid* is used during file system operations. These predicates are true if and only if the user id of the process is same as the user id specified. Similar predicates are also used for group ids.

3. *pid* (process id): This predicate is true if and only if the pid of the process is same as the pid specified. This is used to include or exclude existing processes.

4. *childof*: This predicate is true if and only if the process specified by the pid is an ancestor of the current process. Note that we do not distinguish direct child processes and grandchild processes - so childof can specify a subtree in the process hierarchy. This can be used to include or exclude both existing processes and processes which are not yet created.

5. *executable*: This predicate is true when the executable of the process is the same as the given pathname. This can be used to include or exclude both existing processes and processes which are not yet created.

An example of the API (see also Section 3.1.1.4) is to monitor all processes owned by the user Bob except for process 1468 and its child processes.

```
proc_spec = lbox_AND(
 lbox_UID("bob"),
 lbox_NOT(
   lbox_CHILDOF(1468)));
```

Thus, the monitor can be targeted to observe only the activities of particular processes of interest, ignoring other processes. This helps to reduce monitoring overhead.

3.1.1.2 The Event Specification List

An event specification defines which behaviors of the monitored processes is of interest to the monitor. Suppose a monitor event expression is S and event e happens. Then S is triggered when e is an object which matches S and the operation is one which is compatible with S. The notion of matching and compatibility is specific to the type of object.

A monitor defines a list of all the events of interest. The event specification also defines the appropriate information to return when an event is triggered. One monitor might specify that a file event should consist of the inode, canonical pathname and operation type. Another monitor might only need the operation type, perhaps because the file is unambiguous and known to the monitor. This helps achieve scalability. In the previous example, it avoids the need for constructing a canonical pathname and reduces the event size.

Our events are specified on system resources or objects together with their associated operations. We feel this is more declarative than a system call approach and makes it easier to specify the events of interest. We will focus on file objects and briefly mention other important system objects. Common information which can be specified for all types of events are:

1. *time*: This gives the wall clock time of the event.
2. *pid*: This is the process which performs the operation.
3. *type*: The operation type which is specific to the object.

All event specifications can specify whether the event is asynchronous and can be buffered up, or if it is synchronous and needs to be delivered directly to the monitor. Synchronous events also flush any earlier events which have been buffered. This is analogous to the `PUSH` flag in the TCP packet and allows timely delivery of important events to the monitor.

An event can also specify that reliable monitoring is not needed — this means that the kernel can choose to drop the event if the event buffer is currently full. This would mean that the monitored processes do not have to be suspended and thus higher throughput is obtained at the cost of losing some events.

File Objects: A file event specification consists of:

1. *file pathname*: The pathname is translated into inode number and device number pair. For efficiency, this pair is used internally by the kernel as the identifier of the file.
2. *inclusion flag*: For directories, we can specify whether we are monitoring the directory itself, or all files in its subdirectories. For example, one can specify that all files under `/etc` are included by using the SELF+SUBDIRECTORIES flag. Alternatively, one can specify that only the `/etc` directory itself is included by using the SELF flag only.

   Another flag, IGNORE, means that we are not monitoring this directory and its subdirectories. Thus IGNORE can be used for removing files in the existing file

definition. This is useful because sometimes we want to audit file access *outside* some directory. For example, we are interested in the file access *outside* `/var/www` by the web server process. We can combine two file event specifications to achieve this.

(a) `(/, SELF|SUBDIRECTORIES, R|W|X)`
(b) `(/var/www, IGNORE, R|W|X)`

The way multiple file specifications are treated is that when more than one file event specifications matches the file access, the more specific event – the deeper file event specification takes precedence.

3. *operations*: It specifies which operations (read/write/execute) we are interested in. For example, with a read operation, a file read access event is generated when a process reads from the file but not for write operations. Note that the execute operation differs from the executable predicate in the process specification as the latter is used for process selection.

   Operations on the file meta-data such as permissions and access times can be monitored. The same holds for directory operations such as file creation, deletion or renaming.

When a file is removed (i.e. its reference count becomes 0), all the corresponding file event specification are also removed. This implies that the number of event specifications may decrease at run time. For example, suppose initially a monitor has 4 file event specifications, later there may be only 2 file event specifications. Removing the file event specification is necessary because an inode number is reusable (like a pid). When a process A creates a monitor with a read-only specification on the file `/tmp/a`. Later, another process B deletes file `/tmp/a` and creates another file `/tmp/b` which may have the same inode number of previous `/tmp/a`. If the corresponding file event specification is not removed, the monitor would get an incorrect event.

File events can specify the following information to be returned:

1. *inode number and device number*: This uniquely identifies a file in Unix systems.
2. *operation*: This is the type of operation. For example, read, execute or delete.
3. *data*: the data which is read or written — which allows the monitoring of actual I/O.
4. *canonical pathname*: A file can have many possible (absolute) pathnames because of hard links, symbolic links, the ".." component and different mount points. We thus return a canonical pathname not containing symbolic links with the same semantics as `realpath(3)`.

Socket Objects: Socket specifications allow the monitoring of network operations and interprocess communication. An address specification of IP address and port applies for TCP and UDP sockets with ranges and masks. Special operations specific to sockets apply such as listen, accept, etc. The return information includes also the file descriptor and address.

The socket event specification specifies whether the process can use interprocess communication. It can take the following parameters.

1. *network family*: This can be tcp and udp. Note that Unix socket is not included here because the address space of Unix socket is quite different from IP address.

2. *operation type*: This can be create, connect, listen, accept, send and recv.

   (a) `create`: Socket creation event.
   (b) `connect`: Client socket connection event.
   (c) `listen`: Server socket listen event. Note that local address range can be specified, thus this operation type monitors both `bind(2)` and `listen(2)` system calls.
   (d) `accept`: Server socket `accept(2)` event. Remote address can be retrieved in the event.
   (e) `send` and `recv`: Can be used for monitoring network traffic.

3. *address range*: This can be an address range of ipv4 ip address + port number. This only applies to tcp and udp socket and the connect, listen, and accept operation.

In return, besides the common information, the following information can be included. Note that most of them are specific to operations.

1. *file descriptor* can be used by the monitor to identify sockets.

2. *address* is specific to `connect()`, `bind()` and `accept`.

Process Objects: The process event specification defines what kinds of operations on processes by the monitored processes should be audited. This includes process creation, signals, setuid operations, prctl. The return information includes the target process id and signal number for signal operations.

This type of event specifies whether a process may signal other processes or affect process. Operations include System V IPCs, prctl, ptrace, setuid and fork. The process policy can take the following parameters.

1. *operation type*: E.g. fork, signal or prctl.

2. *target process*: It applies to signal, prctl and ptrace, because these operations have a target process. However, the fork and setuid operations do not have a target process. This parameter is essentially a signed integer.

   (a) zero means all processes.
   (b) positive number n means a specific process with $pid = n$
   (c) negative number $-n$ means all child processes of a process with $pid = n$

In return, besides the common information, the following information can be included. Note that most of them are specific to operations.

1. *target pid*: Note that, for `fork()`, this is the pid of newly created process.

2. *signal*: It is specific to `kill()`.

3.1.1.3 The Monitor Operational Model

In our framework, an auditing monitor is simply a user-level process which is the user-level monitor. It can be thought of as a *logging box* in an analogy to a sandbox, hence we abbreviate this as *lbox*. The `lbox_create(proc_spec, event_spec, n)` is used to designate the current process to be the lbox monitor which will receive the events generated from the processes defined by `proc_spec` which trigger the set of events defined in `event_spec`. One caveat is that we disallow self-referential monitoring (see Section 3.1.2.3).

In our framework, asynchronous events are stored in a kernel buffer of size `n` defined at monitor creation time. Asynchronous events can only be received by the monitor when the buffer becomes full — this can be thought of as flushing the buffered events. To ensure mandatory monitoring which guarantees that no events are lost, when the monitor buffer is full, the process generating the event is blocked until the monitor has emptied the buffer. Processes which are blocked because of the full buffer are also unblocked if the lbox is deleted explicitly by the monitor, `lbox_delete()`, and also implicitly when the monitor terminates. Synchronous events also cause the buffer to be flushed. The monitor can also choose to disable and then re-enable events from the event specification.

We provide a convenient API to read events one at a time, `lbox_next_event()` which abstracts out the low level details. This is simply a library function (in an analogy to `stdio`) which actually reads an event buffer up to size `n` which can contain one or more events. `lbox_next_event()` behaves like a blocking read, it blocks till either the kernel buffer has filled up and can now be read in one go or if it is flushed. A timeout can also be specified. Thus, assuming the default of asynchronous events, there are only a small number of context switches to the monitor with the monitor mostly sleeping until there are sufficient number of events. However the use of synchronous and timeouts allows the

monitor to have finer control and more timely event delivery but can mean more system overhead.

The monitor API described in this section is actually a user-space library which uses the monitoring kernel extension. Thus, this particular API is chosen to be which is reasonably easy to write specifications in a declarative fashion in C. One could of course build a different user-space library. Even nicer would be a scripting language to make monitoring even easy, i.e. even a D-like scripting language.

3.1.1.4 An Extended Example

Figure 3.1 illustrates how a monitor is written in our system. For simplicity, we have used C-like pseudo code to hide some details and have not given the complete program. Error handling also has been omitted for the most part.

In part 1, two events are to be monitored. Any access to any files starting from the current working directory including its subdirectories (the `SUBDIRECTORIES` flag is used to mean this directory including any file with a path starting from here). We also monitor whether `bash` is being executed.

Part 2 defines the process specification which determines which processes are to be monitored by this monitor. Here we want to monitor process given by `pid` and all its children but not if it happens to be the process `inetd_pid`. Also monitor any processes which happen to have an effective user id which is the same as the user name "`apache`". It should be obvious that this specification specifies some existing processes as well as potential future processes which are to be monitored.

Part 3 creates a lbox with the process and event specifications. A buffer of size 4096 is specified. Part 4, just collects events for processing by the monitor. It returns when there is one matching event and doesn't use timeout.

3.1.2 Security and Monitor Interactions

We now turn to security considerations. Earlier we have looked at mandatory and reliable (and non-reliable) auditing. There are two other important security components in our framework.

3.1.2.1 Confidentiality Considerations

As we allow non-privileged users to run their own lbox monitors, it is important to ensure system confidentiality. We do not allow a non-root user to monitor processes owned by other users. A corollary of this constraint is that a non-root user u cannot monitor a setuid (setgid) executable which is setuid (setgid) to user w as long as the effective user (group) id w is different from u. The dual to this is that a monitor process belonging

```
/* open the /proc/lbox file descriptor */
lbox_open(); // implicitly used by library

/* PART 1: create events spec: current directory and exec.
   Also need to specify the information to be retrieved. */
lbox_addevent_file(event_spec, ".", SELF|SUBDIRECTORIES,
   F_R|F_W|F_X, I_INO|I_DEV|I_PID|I_ACC|I_PATH);
lbox_addevent_file(event_spec, "/bin/bash", SELF, F_X,
   I_INO|I_DEV|I_PID|I_ACC|I_PATH);

/* PART 2: create a process specification */
pid = (get root pid of process tree from argv)
proc_spec =
   lbox_OR(
      lbox_AND( lbox_OR( lbox_PROC(pid), lbox_CHILDOF(pid)),
                lbox_NOT(lbox_PROC(inetd_pid))),
      lbox_PROC(lbox_EUIDNAME("apache"))
   );

/* PART 3: now create the lbox */
lbox_create(proc_spec, event_spec, 4096);

/* PART 4: read and process events. */
for (;;) {
   /* -1 is the timeout value which means to block forever */
   event = lbox_next_event(-1);
   switch (event->type) {
   lbox_FILE_EVENT:
      event_file = (struct lbox_event_file *)event;
      printf("pid=%d, dev=%lu, ino=%lu, access=%d, path=%s\n",
         event_file->pid, event_file->dev,
         event_file->ino, event_file->access,
         event_file->path);
   }
}
```

Figure 3.1: A Simple Monitor

to w can monitor such a process from user u when its effective userid is u. This allows monitors to have separate privileges from the monitored processes.

In contrast, `ptrace` takes a different approach to maintaining security. Setuid and setgid programs are not allowed to be traced by `ptrace`. Thus, `ptrace` cannot be used for monitoring while our mechanism can still allow monitoring while maintaining confidentiality.

Consider a process p with original userid u is being monitored in a lbox where the monitor has userid u. Suppose p changes its effective userid to w, then confidentiality prevents the events in p to be monitored. To ensure mandatory monitoring, once p (or its children, if the lbox contains the children) returns its userid back to u, the monitor can now receive monitored events.

3.1.2.2 The Lbox Policy Daemon

It may be the case that for non-root users, one would like to impose further restrictions on monitoring. This might be to disallow certain kinds of events from being monitored or certain processes from being monitors. It could limit monitor usage, e.g. a monitor quota. Monitor creation policy can be policed by an *lbox daemon* which is simply another user-space process. This is an extension of the same philosophy of user-level monitors but now for the construction of arbitrarily complex monitor access policies.

When a process tries to create a lbox, the kernel passes the lbox specification to the lbox daemon, and at the same time suspending the process. The lbox daemon then decides whether or not to allow or deny the request. Furthermore, the lbox daemon is able to modify the lbox specification. The suspended process can then be woken up, if the lbox is successful, it now becomes a lbox monitor with possibly a modified specification. The lbox daemon can also be turned off nor does it need to be present, in which case, a default policy is applied.

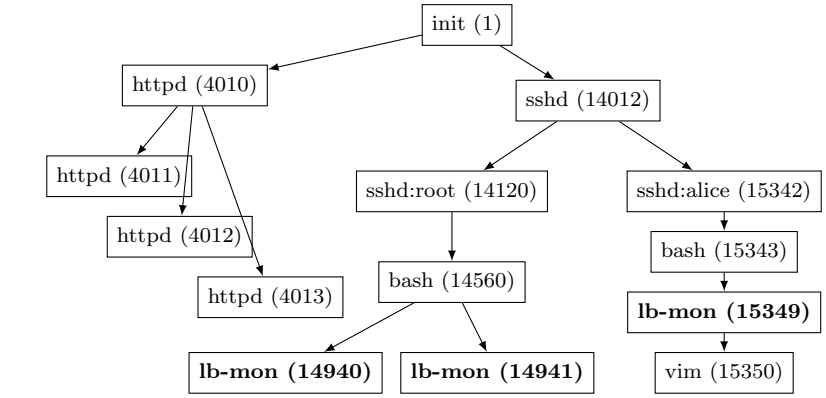
3.1.2.3 Cascading Monitors

It may be the case that one of the processes being monitored is itself a monitor, we call this cascaded monitors. Cascading auditing gives the maximum flexibility to create collections of independent monitors as well as an ecology of monitors. For example, the system administrator can have a global monitor to audit any processes (except itself) which tries to execute `/sbin/su`. A user might run a monitor to log all writes from some of his processes to a particular file.

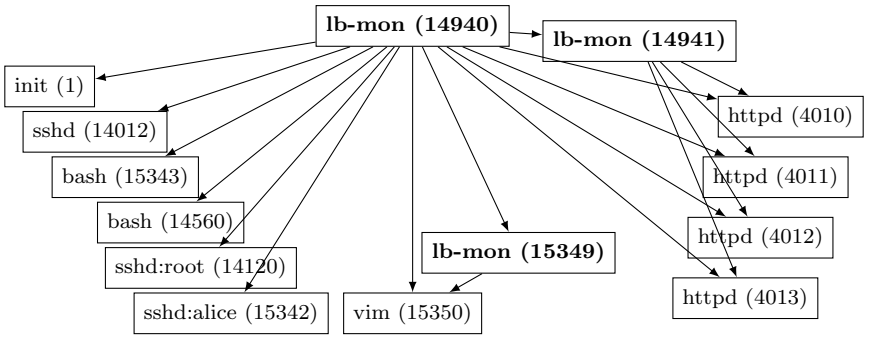
A collection of monitors can be viewed as a directed acyclic graph (DAG) where the edges indicate which the monitoring and monitoree relationship. Suppose that instead of a DAG, we have a cycle, thus a circular monitoring chain. The self-referencing monitoring will now lead to an ever increasing cascading chain of monitored events which will not

terminate. For example, suppose monitors A and B are watching each other, then event a_1 triggers monitor B which will generate event b_1 for monitor A which will trigger yet another event a_2 and so on. Thus, we require as a system safety condition no self-referential monitoring. It is interesting to note that DTrace does not prevent this, so for example, a D program which traces all the `write` I/O will also monitor the DTrace process itself. This appears to cause Solaris to freeze in the kernel.

Rather than disallowing the creation of a monitor which may lead to a cycle in the monitor DAG, we instead remove the monitors which would otherwise create a cycle from the process specification. This makes sense since a new monitor may not wish to care about monitors in the system. Furthermore, it can be easier to write a process specification which is a little looser than to write one which guarantees no cycles. For example, global monitoring using the process specification, `TRUE`, should strictly also exclude the monitor pid.



(a) Parent-Child Process Tree



(b) Monitor DAG

Figure 3.2: A Tree of Cascaded Monitors

The following example shows a snapshot of a system where a web server is running and user Alice is using `vim`. Figure 3.2(a) shows the process tree induced by the parent-child relationship while Figure 3.2(b) shows the monitor DAG induced by the monitoring relationship. The following sequence describes the creation of the snapshot given in

Figure 3.2:

1. The master web server process 4010 is created.
2. Process 4010 creates 3 worker processes: 4011, 4012, 4013.
3. The root user is logged in through ssh, creating shell(14560).
4. Root first creates a global monitor 14940 to audit file access on `/etc/passwd` for all processes (except 14940).
5. Root then creates a monitor 14941 to audit the web server processes.
6. Alice logs in and a shell 15343 is created.
7. Alice creates a monitor 15349 to audit her `vim`(15350) editor because she want to know which files are accessed by `vim`(15350).

Here, two monitors 15349 and 15340 are monitoring the `vim`(15350) process. Suppose the `vim` (15350) process accesses the `/etc/passwd` file, two file events will be generated to the two monitors, 15349 and 15340. If `vim` (15350) accesses another file, then only monitor 15349 will receive the event.

3.1.3 Using Monitors

We now give some simple examples of using the framework. This examples are only meant to exemplify the ease of creating a customized auditing monitor and are not meant to be full blown applications.

3.1.3.1 Testing a possibly untrusted program

This example checks to see if a process is not stealing your PGP keys. The monitor process first creates the lbox and then a child process which executes the program to be audited.

```
proc_spec = lbox_CHILDOF(getpid());
lbox_addevent_file(event_spec,
  "/home/alice/.gnupg/secring.gpg",
  SELF, F_R, I_ALL);
// .... create the lbox
if (fork()) {
        // .... do the event processing
} else execl(....);
```

3.1.3.2 Monitoring the web server

In this example, we want to see whether the web server is working correctly. More precisely, we want to make sure it is only accessing files inside `/var/www`.

The process specification can be described as follows:

```
proc_spec = lbox_CHILDOF(apach_master_pid);
```

The event specification is an example of monitoring files *outside* some directory. We first make all files to be monitored, then we make all files under `/var/www` *not* to be monitored. Note that the order of the two specifications is not important.

```
lbox_add_event_file(&event_spec1, "/",
  SELF|SUBDIRECTORIES, F_R|F_W, I_ALL);
lbox_add_event_file(&event_spec2,
  "/var/www", IGNORE, F_R|F_W, I_ALL);
```

3.1.3.3 Global Monitoring

Sometimes we need to monitor all actions on a single process; and other times we need to monitor a single action on all processes. In this example, we want to see if any process is sending packets to the network `137.132.0.0/255.255.0.0`. The process specification is simply,

```
proc_spec = lbox_TRUE();
```

The event specification is the following network event,

```
lbox_net_connect_ipv4(&event_spec, "137.132.0.0/255.255.0.0");
```

3.1.3.4 Other Applications

LAFS [98] is a logging and auditing file system where accesses not conforming to a policy are logged. The policy specification includes a time interval, userid specification, application and operation type. A LAFS-like monitor could easily be written. Note that it is harder to use `dtrace` to do this LAFS-style tracing because of its system call orientation. Here we can make use of inheritance in the directory structure.

Finke [38] proposed a monitoring system, which reports when applications did not run as they were supposed to, e.g. due to some failure. Again, it is easy to write a monitor which can detect whether tasks ran as scheduled.

3.1.4 Implementation Issues

3.1.4.1 The Hooking Mechanism

We have implemented our prototype as a kernel module for Linux 2.6 kernel. There are several ways of implementing the monitor mechanism:

1. **`ptrace()` or `/proc`** As discussed earlier, this has many problems and does not support mandatory or reliable monitoring. A system-call based mechanism is also not suitable for our object-based orientation.

2. **In-kernel system call interposition** This can suffer from TOCTOU (time of create time of use) problems. For example, if we are monitoring the `write()` system call, a process can change the actual file which the file descriptor points to between the time when the file descriptor is looked up and the actual file descriptor is used. This would lead either to missing an event (unreliable) or reporting an incorrect event (the wrong file). As before, it also does not suit our object-based orientation.

3. **Linux Security Module (LSM)** We currently use this approach in our prototype which has a good match with kernel objects. Its also much simpler as LSM hooks are mostly in the right places. It guarantees mandatory monitoring, and does not suffer from TOCTOU. However, it does not cover every possible system calls. As the present prototype focuses initially on events on files, this is sufficient but it does mean that a full solution cannot be just a module.

3.1.4.2 Matching The File Specification

Matching file event specifications, requires the deeper matching file event specification to take precedence. To achieve this, the algorithm for checking of a file access is as follows.

1. If the current file is marked SELF, report it (generate event), done.

2. If the current file is marked SUBDIRECTORIES, report it (generate event), done.

3. If the current file is marked IGNORE (neither SELF nor SUBDIRECTORIES), ignore it, done.

4. If it is the root directory, ignore it, done.

5. Set parent directory as current file.

6. Goto 2.

Suppose we have the following two file event specifications from the previous example:

1. `(/, SELF|SUBDIRECTORIES, R|W)`

2. `(/var/www, IGNORE, R|W)`

For the file /var/www/a/b/c, the checking is stopped at `/var/www` at the third step. For the file /etc/passwd, the checking is stopped at `/` at the second step.

To do the upward directory traversal, we make use of the `dentry` kernel structure. The upward directory traversal is fast, because in Linux, during the path resolving phase, all the parent `dentry` nodes are stored in the `dcache`.

3.1.4.3 Performance Tuning Methods

To reduce the overhead of of task switching caused by our auditing system, we introduce an event buffer so that the monitored process can continue its execution even if it generates a auditing event. Timeouts and events with a flushing property are used to allow the monitor to get events even if the buffer is not full. In this sense, the monitor can choose a trade-off between efficiency and timeliness.

When a system call takes place, we first check whether the process is being monitored. This check only requires a few instructions. Thus there is very little impact on non-monitored processes. For monitored processes, the monitor can also be found efficiently. Thus there is very little impact on a process as long as the number of monitors of the process is not too large.

3.1.5 Comparing to DTrace

There are some similarities with Solaris DTrace (Section 2.2.6), which was published at about the same time as this work. Both systems aim to bring minimal side effects to the monitored program in order to be used on production systems. Both systems have a flexible in-kernel event filter mechanism. However, there are some key differences.

Firstly, although DTrace does integrate in privileges, it does not have the same security concerns. DTrace has a all-or-nothing security configuration, i.e. a user is either able to trace all processes and the kernel (thus obtaining confidential information), or not able to use DTrace at all. This is because DTrace is designed for system administrators. LBox is designed for all users in a multi-user environment, thus has to prevent one user's information from being monitored by another user.

Secondly, the DTrace implementation requires extensive changes to Solaris, in the words of the developers, "*it was a hell of a lot of work* – Bryan Cantrill, Solaris Kernel Development". While it is difficult to have portable kernel enhancements, we feel our approach should be less of a massive change to the kernel type solution and in this sense is more lightweight and amenable to implementation in different UNIX variants.

Thirdly, as DTrace is meant to minimize impact, it does not provide reliable event monitoring and can drop events. We believe that both reliable event monitoring and minimal side effect are important. LBox gives the users the choice in a per-event basis on this trade off.

Lastly, DTrace allows user supplied code written in the D language to be executed in kernel. Although there is constrains on what are allowed, this increases the complexity of the kernel thus poses security threats to the operating system. The sole purpose of having monitoring code executing in the kernel is to reduce context switches and eliminate information traffic between user space and kernel space thus improving efficiency. LBox achieves this by having event buffer to reduce context switches; and flexible event filters

to reduce number of events.

3.1.6 Experimental Evaluation

Our prototype Linux implementation makes use of LSM [102] and version 2.6.10 of the kernel, and thus is convenient to install as it consists of loadable kernel modules. The interface to the kernel is done through `ioctl` to a pseudo file system in `/proc/lbox`. The lbox API library in Section 3.1.1 gives a more convenient interface that using `ioctl` system calls. The PC used here is a Pentium IV 3.0GHz PC with 1G memory.

We want to demonstrate that the framework and prototype system leads to rather efficient monitoring. Although our prototype is not yet fully optimized, the results do show that the overheads of monitoring can be very low.

We first use a simple micro-benchmark which gives an indication of worst case overheads. The program performs 1000000 open(2) and close(2) calls with a monitor watching for the file access. We compare the auditing framework in each of the following scenarios:

1. *clean kernel*: A clean stock kernel.

2. *proc miss*: The lbox module is loaded in the kernel but no monitors are present.

3. *file miss 1/2*: A monitor is monitoring the file open but on a different file specification. Thus no event is generated. We compare two scenarios. *File miss 1*, uses a directory specification without the SUBDIRECTORIES property, thus no directory traversal is needed. *File miss 2* has a file specification which is some non-matching directory with the SUBDIRECTORIES set. This requires traversing up ancestor directories up to the root.

4. *0 dir level*: The monitor is monitoring a regular file. The benchmark is accessing the same regular file.

5. *1/2 dir levels*: The monitor is monitoring a directory which is 1 or 2 levels deep containing the file which is being used. For example, the monitor uses the file specification `/1` with SUBDIRECTORIES set. The 2 dir level benchmark opens the file `/1/2/foo`. This test investigates the time for directory traversal.

6. *no buff*: The file event is synchronous which means 1000000 context switches to the monitor are needed. The benchmark process suspends until the monitor has read the event

7. *ptrace*: The comparison is with `strace` which uses the traditional `ptrace` mechanism in Linux.

The open micro-benchmark results are given in Table 3.1 with all times in seconds. The average and standard deviation is given over 10 runs. As the timings are only measured

scenario	real	user	sys
clean kernel	1.99 ± 0.01	0.21 ± 0.01	1.77 ± 0.02
proc miss	2.04 ± 0.02	0.21 ± 0.01	1.82 ± 0.03
file miss 1	2.17 ± 0.01	0.22 ± 0.01	1.95 ± 0.02
file miss 2	2.27 ± 0.01	0.22 ± 0.01	2.04 ± 0.02
1 dir level	2.29 ± 0.01	0.21 ± 0.01	2.03 ± 0.02
2 dir levels	2.52 ± 0.01	0.21 ± 0.01	2.23 ± 0.03
3 dir levels	2.56 ± 0.01	0.20 ± 0.02	2.31 ± 0.02
no buff	8.70 ± 0.04	0.99 ± 0.08	4.57 ± 0.16
ptrace	59.04 ± 0.15	12.90 ± 0.32	46.11 ± 0.39

Table 3.1: `open(2)` micro-benchmark on Linux. All times are in seconds.

with the Unix user/system and real-time mechanism, they are only approximate and are meant to give an indication of the overheads of monitoring under the different scenarios.

The kernel inspection implementation (proc miss) has negligible overhead ($\sim$ 2%) over the clean kernel. The *file miss 2* test has ($\sim$ 14%) overhead, this is because that the system needs to travel to the root directory to make sure the file is not monitored. *file miss 1* has a smaller overhead ($\sim$ 9%) because directory traversal is not needed. When monitoring is synchronous, so events are not buffered (no buff), overhead jumps substantially to 337%. It is interesting to note that this is still much smaller than `ptrace` which has 2866% overhead or about seven times slower than the no buffering case. Using asynchronous events with buffering (0 dir level) drops the overhead to 15%.

One point of comparison with other systems on Linux would be with Systrace [74]. A pure comparison is not valid since in our system any event which must be monitored is eventually passed to the monitor. Systrace, on the other hand, does access control first can choose between a kernel-level policy or the user-level policy daemon. The kernel-level policy should entail less work for Systrace simply because the decision is handled only within the kernel, while the user-level policy is more expensive simply due to the increased number of context switches. The objectives of the two systems are also different. With those caveats in mind, the Systrace open micro-benchmark shows a 6.25% overhead over no monitoring. Our overhead for *file miss 1* is a little more at 9%. This makes sense since for auditing, an event has to cross in two directions: first inwards when it is recorded; and later back to user-space when it is sent to the monitor. Our implementation is not yet optimized and we expect also to be able to reduce the overheads further.

Our directory file specifications can be compared against the pathname normalization of Systrace. The "0-2 dir levels" scenario shows that the overhead of an inherited file specification is small, the initial overhead of checking a directory is reasonable at 26% and then a rather small overhead for the next component. The Systrace results show that each directory component in their specification adds about 1665% additional overhead over the original `open()` system call. This is because of the filename normalization done

in user-space which is significantly more expensive.

scenario	real	user	sys
direct run	3.84 ± 0.01	1.04 ± 0.01	2.80 ± 0.01
truss	100.32 ± 0.85	10.12 ± 0.05	56.71 ± 0.80
DTrace	8.41 ± 0.02	1.08 ± 0.01	7.33 ± 0.01

Table 3.2: `open(2)` micro-benchmark on Solaris 10

Table 3.2 shows the test on Solaris for the same hardware with the same benchmark program. While the baseline for Solaris is slower than Linux, it is reasonable to make relative comparisons. `Truss` which uses the `proc` adds 2510% overhead to the `open()` system call and DTrace is significantly more efficient with 119% overhead. The use of `proc` allows only the `open()` system call to be traced rather than all system calls. Even so, we see that the overhead is considerably higher than in Linux where `ptrace` traces every system call. In the DTrace test[1], we have allowed DTrace more advantage as the D program does not return any information to user space, whereas in our system, all the relevant `open()`'s are being returned to the user space monitor. The absolute running time in the DTrace test is 3.67 times of that in our system. The overhead added to the original system call in DTrace is 7.9 times of that in our system. This is slightly surprising since DTrace uses dynamic instrumentation (which is hardware dependent) while we use a simpler static instrumentation which is more expensive but portable. We suspect it is partly a case of the Linux kernel having less overhead than Solaris.

setting	real	user	sys
clean kernel	0.27 ± 0.01	0.01 ± 0.01	0.20 ± 0.01
monitored	0.28 ± 0.01	0.01 ± 0.01	0.20 ± 0.01

Table 3.3: `connect(2)` micro-benchmark

Since the open micro-benchmark reuses the same file over and over again, the open itself becomes quite fast in Linux. We use another micro-benchmark on socket operations to show the effect of a more expensive system call. The benchmark creates a TCP socket and connects to a local port. This is repeated for 4000 times and the results are in Table 3.3. We can see that in an expensive system call, the overhead is not noticeable.

The `open` micro-benchmark gives a measure of worst-case overhead since the file is totally cached in memory and in that case, the Linux `open` code is heavily optimized. It is also useful to look at applications (macro-benchmarks which do more than just system calls) which may have moderate to moderately heavy system call usage. Obviously, there is little point benchmarking CPU bound applications which only have low system call use. In the macro-benchmarks, we also look at the cost of monitoring I/O (reading and writing), this is meant to simulate a monitor which might want to examine precisely what

[1]The D program is "syscall::open:entry { @[execname]=count(); }"

	web server (request/sec)	make bash (sec)	install Mozilla (sec)
Linux clean	8903.1 ± 21.8	34.3 ± 0.2	1.8 ± 0.1
lbox no I/O, file miss	$8773.1 \pm 19.4(1.5\%)$	$34.4 \pm 0.2(0.17\%)$	$1.8 \pm 0.0(0\%)$
lbox no I/O, file hit	$8762.0 \pm 25.6(1.6\%)$	$34.4 \pm 0.2(0.17\%)$	$1.8 \pm 0.0(0.56\%)$
lbox with I/O	$8728.9 \pm 18.4(2.0\%)$	$34.4 \pm 0.2(0.29\%)$	$1.7 \pm 0.1(3.9\%)$
strace no I/O	$3577.6 \pm 8.5(148.9\%)$	$39.6 \pm 0.1(15.3\%)$	$2.2 \pm 0.1(21.1\%)$
strace with I/O	$1981.4 \pm 6.9(349.3\%)$	$100.7 \pm 0.1(293.2\%)$	$23.2 \pm 0.1(1189.4\%)$
Solaris clean	6889.1 ± 42.6	43.8 ± 0.1	1.6 ± 0.1
DTrace no I/O	$6776.2 \pm 53.6(1.7\%)$	$44.4 \pm 0.1(1.4\%)$	$1.6 \pm 0.1(0\%)$
DTrace with I/O	$6326.9 \pm 52.4(8.9\%)$	$44.5 \pm 0.0(1.6\%)$	$1.6 \pm 0.0(0.6\%)$
truss no I/O	$1382.0 \pm 2.0(398.5\%)$	$70.9 \pm 0.0(61.8\%)$	$7.7 \pm 0.0(372.2\%)$
truss with I/O	$1126.7 \pm 4.4(511.5\%)$	$74.6 \pm 1.5(70.2\%)$	$13.4 \pm 0.13(724.7\%)$

Table 3.4: Macro-benchmarks

an application has done. We have chosen three application benchmarks which are realistic applications

- apache web server: this is chosen because web server performance is important. For apache, we measure the average number of requests served per second using the apache `ab` benchmark and I/O monitoring is for the http requests (reading) and http responses (writing).
- `make bash`: this builds the bash shell.
- install Mozilla: this installs Mozilla. The Linux and Solaris differ. The Linux one has a custom install program while the Solaris version is simply untar. This means the results are not comparable except in a relative to each operating system alone.

The last two benchmarks, bash and Mozilla, have also been chosen because these applications have been used in Alcatraz [53]. For I/O monitoring, we have monitored all the I/O from the application. The results given in Table 3.4. The results for `strace` and `truss` are only meant to illustrate the additional overhead of I/O since both programs do reprocess the data and thus do have additional overhead but serves to bound the cost of `ptrace` and `proc`. We see that in all cases, lbox monitoring has very low overheads, below 2% without I/O and less than 4% with complete I/O monitoring of the applications. The Mozilla install cannot be directly compared since the actual install is different. Even though these are macro-benchmarks, overhead for `ptrace` without I/O is significant, at least 15%. With I/O monitoring, the overhead shoots up. Alcatraz which uses `ptrace` has rather high overheads for their system call interception, 43% for a `make` and 79% for Mozilla. Their isolation overhead which is a measure of write I/O adds a further 20%. Our overheads are much smaller ($< 4\%$) but we don't do isolation.

It is a little surprising that our user-level auditing framework can outperform DTrace which is an in-kernel tracing mechanism. This is very encouraging and furthermore, we believe we can still optimize our prototype.

3.1.7 Conclusion

We show a user-level auditing framework suitable for general purpose auditing and security monitoring. We achieve transparent auditing at a fine-grained level of applications while providing guarantees on the security of the auditing process. We are careful to avoid problems with privilege escalation and access to information beyond the user's privileges. Furthermore, we avoid problems associated with denial of service which can be caused by self-referential monitoring or tracing.

As our monitors are user-level, the auditing is expressed in terms of operations on operating system objects and resources. This makes it easy to write a custom monitor since the semantics is close to that of user code rather than having to understand kernel internals.

A user-level monitor is desirable given it is safer than an in-kernel one. The question is whether a user-level monitor is sufficiently efficient. Our framework is designed so that the cost of monitoring is commensurate with the amount of events and information needed. Our experiments are very encouraging showing that the overhead is comparable to in-kernel mechanisms such as DTrace.

Future work includes further system optimizations to make the framework even more scalable and efficient. We would like to increase the expressivity of the in-kernel mechanisms without incurring more cost and also avoiding executing monitor code in the kernel.

3.2 WinResMon

The tasks of software maintenance and configuration require a precise understanding of system resources, the individual requirements of each piece of software, and interdependencies between every software program on the system. We use the term *software maintenance* to describe the system administration task of ensuring that software on a system is configured and maintained correctly over time. Examples of software dependencies include:

- File sharing: including dynamic libraries and external data storage.
- Sharing of software configurations: usually in the form of registry keys in Microsoft Windows.
- Interprocess communication and synchronization.

Although software maintenance tasks might seem conceptually trivial, they can be time consuming and difficult, especially in large or complex environments. System administrators often rely on documentation and on-line information such as FAQs or forums, but such information is often incomplete.

As discussed in Section 2.1.3, in various distributions of Linux, software dependency issues are partly addressed by the use of a package manager. The Red Hat Package Manager (RPM) [36], for example, records the currently installed packages and the files required and provided by these packages in a centralized database. RPM checks dependencies before removing a package to ensure that files required by other installed packages are not removed. Similar checks are done to prevent installing a package which contains files that conflicts with other existing packages.

In Microsoft Windows, however, software installation can be complex, and the exact dependencies between different software programs might not be clear. The confusion is further compounded by many implicit software interdependencies, e.g. registry keys which are part of shared software configurations. As a result, it is difficult to know whether to remove a file when uninstalling an application. File removal might lead to problems with another piece software or might create security vulnerabilities.

When installing two or more programs that share files, one program may cease to function correctly because the installation of the second blindly overwrote shared libraries (DLL files). There is also the question of when to perform a major software upgrade. System administrators may delay upgrading for fear of breaking existing software; yet such a choice has its own risks.

We present *WinResMon*, a discovery and system debugging tool for determining a program's resource usage as well as the resource usage interactions between multiple

programs. Our work focuses on Microsoft Windows NT-based operating systems (e.g. Microsoft Windows XP, Microsoft Windows 2000, Microsoft Windows 2003).[2]

WinResMon differs from LBox as it provides whole system monitoring, because the software maintenance problems usually require global view of the system, and some problems require always-on monitoring for a long period. LBox is designed to study a single process or a group of processes launched for a single task, thus the monitoring can be usually isolated to the related processes and the particular run.

3.2.1 Motivation and Applications

We believe that the key to solving the software maintenance problem is to understand the life cycle of the system and programs therein. We also wish to empower ordinary users, removing the requirement of knowing every minutiae of Microsoft Windows. Although WinResMon is not tailored specially for system security, it can also be utilized as a security auditing tool.

We designed WinResMon to act as both an infrastructure or framework and a system utility. As a framework, it is extensible, and one can therefore add new functionality and build customized tools. As a tool, it comes with pre-built modules to answer typical questions about resource usage and dependencies.

When used as a debugger, WinResMon investigates the current system state, i.e. which program uses which registry keys, and determines how the system has arrived at that state. WinResMon accomplishes this by recording information about the evolution of system software dependencies and resource usage over time. To solve general problems in software maintenance, WinResMon monitors: files, the registry, and interprocess communication and synchronization. However, it is not feasible to continuously and permanently record all changes to the system since the required space would be prohibitive. WinResMon employs a reasonable compromise by maintaining detailed usage records over the current time period and a subset of information that can be maintained over the lifetime of all software in the system.

We illustrate the software maintenance problem with some simple examples. One attack vector for spyware is to register itself as a start-up program, thereby hiding itself from the end user. Also consider a music player which may require some sound decoding libraries. This application only functions correctly with certain versions of the libraries. Thus, replacing a library can lead to software failure. Various pieces of software may also conflict, e.g. two mail transfer agents (MTAs) usually do not co-exist.

Some common system administration questions and tasks which WinResMon can assist with include:

[2] While an appropriate version of a tool similar to WinResMon could also be of use in UNIX, its value is much greater in a Microsoft Windows environment.

1. Can we safely remove a particular DLL file?

   Some applications provide shared libraries (DLL files) for use with other applications. When the system administrator uninstalls an application, she can also choose to remove the DLLs. Removing a DLL can cause other programs which still use it to malfunction. On the other hand, blindly retaining all DLLs will cause the system to keep growing and may create security vulnerabilities. The system administrator generally lacks adequate information to determine whether another program uses a shared library. WinResMon can be used to record the utilization of each DLL so that the system administrator can determine which programs use which DLLs.

2. Why does a program need administrator privilege to run?

   Running programs with administrator privilege is discouraged because malware such as viruses/spyware or poorly written applications can damage the system. However, some programs may need to run as the administrator without an obvious reason. WinResMon can detect whether a program needs administrator access. The idea is to understand the reasons for elevated privileges and configure the system to limit the use of privileges. If it finds applications that require certain administrator privileges to function correctly, the system administrator can set up a policy that restricts the administrator privileges to the needed resources (files, registry keys, etc). We remark that this approach can be contrasted with confinement systems such as Systrace [74] in Unix. WinResMon is an auditing tool, it does not confine system calls, but provides useful input for system administrators to create policies on resource access which can then be used to limit privileges.

3. Monitoring sensitive registry locations to detect spyware.

   Managing the Microsoft Windows registry is difficult due to its complexity. Spyware often takes advantage of this complexity to bury itself in the registry, making it difficult for the user to remove it completely. Wang et al. [97] have listed the most common entry points for spyware to enter a Windows NT system. The following are some of the configuration settings WinResMon can monitor:

   - *Autostarts*: monitor which programs load on startup.
   - *Internet explorer hooks*: track hooks which define the default search page, toolbars and browser helper objects (BHO), etc.
   - *Winlogon*: look for applications that hook into system resources.
   - *Services*: monitor services such as automatic startup services (e.g. task scheduler) or drivers which are installed as services.
   - *DLL injection*: monitor DLL injection attacks (any application that uses `user32.dll` can be hijacked by having a DLL injected into its process space).

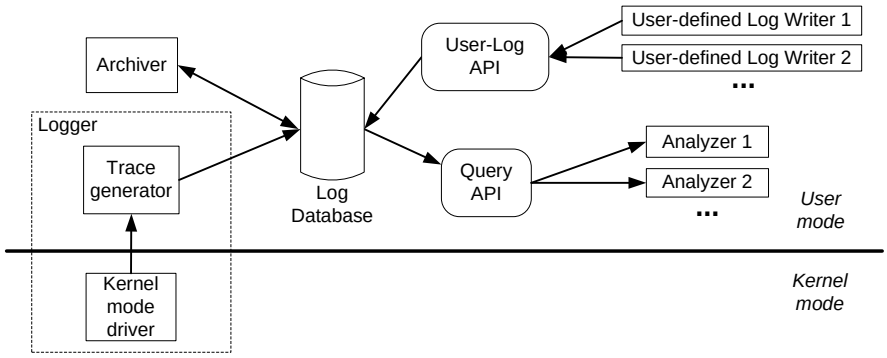


Figure 3.3: WinResMon overall system architecture

- *File associations*: monitor the registration of file extensions with applications. For example, `.DOC` is registered to Microsoft Word.

3.2.2 System Design

The WinResMon system infrastructure shown in Figure 3.3 consists of the following components: logger, archiver, query API, and user-log API. The logger generates resource-access traces which are later used by the analyzer. The archiver performs log compaction/summarization of old traces. Query and user-log API provide the interface to the trace database.

3.2.2.1 The Logger

The logger consists of a system call interceptor and a trace generator module. The system call interceptor is an in-kernel driver which monitors system calls made by each process and sends the monitored event information to the trace generator. The trace generator is a user-space service (daemon) which collects event information sent from the system call interceptor and generates resource access traces. Ideally, the logger is meant to run *all the time* so as to record the entire life cycle of how resources are used by software.[3]

A log trace file consists of a list of records, each representing an access operation on one of the following types of resource:

1. *File*: covering both directories and regular files
2. *Registry*
3. *Process*: mainly to record process creation

[3]One might only run the logger at selected times instead, but this means WinResMon could miss critical information.

4. *Synchronization objects*: which records information on inter-process synchronization mechanisms provided by Microsoft Windows (mutex, semaphore, event and waitable timer)

5. *IPC*: including named pipes and mailslots.

In addition to these five basic resource types, WinResMon also captures *system event* information such as: process termination, system boot and shutdown, user login and logout. Moreover, it also provides a user-log API (described in Section 3.2.2.5) for users or applications to insert user/application-defined milestone events. One potential usage of custom events is to demarcate and distinguish the software installation and uninstallation portions of the log.

A record entry contains common information and specific information relevant to that record type. The common information consists of: *record-type*, *start-time*, *end-time*[4], *process-id*, *thread-id user-id* and the *error-code* (in the case of failure). We can optionally record the user space stack trace, which is a list of function return addresses who directly or indirectly make the system call. This stack trace feature is used in our trace visualization (Section 5.2). The specific information recorded by different record types are:

1. **File:**

   - absolute path of the file
   - file operation: *open*, *read*, *write*, *delete* or *move*
   - operation-specific information. For open: the access flags. For read and write: the number of bytes read or written. For move: the new path.

2. **Registry:**

   - absolute path of the registry key
   - operation: *open-key*, *query-value*, *set-value*, or *delete-key*
   - operation specific information. For *open-key*: the access flags. For *query-value* and *set-value*: the type, size, and value of the registry key.

   Due to the importance of the registry in software maintenance, WinResMon logs the actual data changes made to the registry; whereas for file I/O, it is not practical to record the data.

3. **Process:**

   - absolute path of the executable corresponding to the newly created process
   - command line arguments

[4]The timing is measured in CPU cycles, thus it is very accurate

- process id of the newly created process.

4. **Synchronization objects:**

   - type of the object: *mutex*, *semaphore*, *event*, or *waitable-timer*
   - name of the object
   - operation: *create*, *open*, or *delete*.

   We are interested only in objects which have names in the system-wide namespace, because anonymous and process-wide named objects will not interfere with other programs, thus they are unrelated to software dependencies.

5. **IPC:**

   - type of the IPC: *named-pipe*, or *mailslot*
   - name of the IPC
   - operation: *create*, *open*, *delete*, *send*, or *receive*.

   WinResMon records the synchronization objects and IPC operations listed above since they involve global (system-wide) namespace which could be used by different programs to interact with each other. One can use WinResMon to uncover the causes of the following problems:

   (a) If process A has created a semaphore s, and process B, which is unaware of the existence of A, is trying to create a semaphore with the same name s, B's operation will fail.

   (b) If process A fails to run correctly and semaphore s is not created, then process B, assuming the existence of A, will fail trying to use s.

6. **System event:**

   - system event type: *process-termination*, *boot*, *shutdown*, *user-login*, or *logout*
   - any event specific information: path of the executable of the process in the case of process termination, user name in the case of user login/logout, etc.

7. **User defined event:**

   - the path of the executable of the process generating the event
   - a binary string describing the event.

Not all operations need to be logged into the database as some resources are not significant to record, e.g. the temporary directory, `C:\temp`. A filter can therefore be used in the logger to prevent logging resource access of no interest to the system administrator.

3.2.2.2 The Log database

The log database consists of a number of log files and one distinguished *active log* to which the logger records current resource access activity. To maintain reasonable log file sizes, WinResMon performs a "log switch" to create a new active log for recording subsequent entries. The old log files are then subject to the log compaction/summarization process by the archiver. Any of the following conditions can trigger a log switch:

- The size of the current log file reaches the specified `Max_log_size`.
- The number of entries in the current log file reaches `Max_log_entries`.
- The user manually initiates a log switch process.

3.2.2.3 Archiving Old Traces

As WinResMon logs system activities over a long period of time, the trace becomes very large. If old traces are discarded, some early yet potentially valuable information, such as identifying which program first created a file, is lost. An example of important information is identifying which program first created a file. Since it is necessary to eventually prune some information to avoid excessively large log files, WinResMon summarizes old trace files into a "*compacted trace database*". This maintains a balance between compactness and the ability to answer important questions about system resource access.

There are two main issues in performing trace compaction, determining when to initiate the compaction and how to perform compaction on old trace entries. WinResMon uses a module called the archiver which runs based on a specified `Archive_interval_check` time interval. Upon activation, the archiver compacts previously uncompacted entries which are older than the specified `Old_log_age`. WinResMon employs the following strategies in performing the compaction:

1. Log entry summarization/aggregation

WinResMon can summarize multiple entries with similar information to produce an aggregated entry in the compacted trace database. It applies the following policies on various resource types:

- **File:** multiple read or write operations on the same file are summarized by recording the time of the first and last operations and the total number of time the operations were done.
- **Registry:** multiple query-value operations on the same registry key are aggregated. Multiple set-value on a key are aggregated only if the values written are identical.

- **IPC:** send and receive operations of one IPC object are aggregated by recording the time of the first and last operations.

When matching a query (described later in Section 3.2.2.4) on an aggregated entry which records a time interval, WinResMon considers that the entry satisfies the specified time constraint if *one* of the values matches the constraint.

2. Selective priority-based entry removal

One strategy to reduce the old traces involves removing entries deemed to be of little value for answering future questions. WinResMon implements selective entry removal strategy based on a user-supplied configuration file. The configuration file assigns a priority to log entries. For example, a log entry for writing a registry key might be considered more important to keep than one for reading a registry key.

A fragment of an example configuration is shown in Figure 3.4. This example uses priority values ranging from 1 (least important) to 5 (most important). For each resource type, configuration entries are matched in a sequential order, and mappings are listed from most to least specific. It is possible to omit some arguments, i.e. with wildcards. To simplify the priority assignment, WinResMon classifies file open operations in Microsoft Windows into one of three modes: RO (read-only), RW (read-write) and WO (write-only/append). The archiver translates the semantics of each operation from file flags in the raw log entries to the appropriate values for matching against the configuration.

The existing applications and anticipated usage, together with some general principles, can be used to derive the priority assignment for the configuration. One general principle is that "transient information," such as that those on synchronization objects and IPC, becomes less relevant after system shutdown and can be given low priority. During the removal process, all log entries with priority lower than the `Lowest_priority_retained` value will be purged.

3. Auto deletion of old log files

To maintain reasonable storage usage, WinResMon eventually needs to remove entries deemed too old. The log file whose newest entry timestamp exceeds `Max_log_lifetime` is deleted. If necessary, it is also possible to additionally provide an API function (in a secure manner) to perform deletion on selected trace entries based on a specified selection condition.

3.2.2.4 A Query API and Analyzer

We want to support trace analyzers which answer queries solving particular software maintenance problems. WinResMon therefore provides a query API which can be used to obtain information from the trace database. One can view the trace as a database and the provided query API as the query language by which the analyzers extract relevant information from the database.

```
# File Section
# Format:
# Type  Action  Object                                          Mode      Priority
FILE    Read    *                                               *         1
FILE    Write   *                                               *         1
File    Open    C:\Windows\Temp\*                               *         1
File    Open    C:\Windows\System32\*                           RW|WO     5
File    Open    C:\Windows\System32\*                           RO        4
File    Open    C:\Windows\*                                    RW|WO     4
File    Open    C:\Windows\*                                    RO        3
File    Open    C:\Program Files\*                              RW|WO     3
File    Open    C:\Program Files\*                              RO        2
File    Open    C:\Documents and Settings\Local Settings\
                {Temp\*|Temporary Internet Files\*}             *         1
File    Open    *                                               RW|WO     2
File    Open    *                                               RO        1
File    Delete  C:\Windows\Temp\*                               *         1
File    Delete  C:\Windows\System32\*                           *         5
File    Delete  C:\Windows\*                                    *         4
File    Delete  C:\Program Files\*                              *         3
File    Delete  C:\Documents and Settings\Local Settings\
                {Temp\*|Temporary Internet Files\*}             *         1
File    Delete  *                                               *         2
....
```

Figure 3.4: Example of Log Priorities for Trace Compaction

The main query API, analogous to an SQL Select command, is `trace_select([`*selection condition*`], [`*field projection*`], [`*time interval*`], [`*output order*`])`, where the caller specifies the matching condition(s) and fields to return. The *selection condition*, specified on string types such as program name and registry key path, takes the form of a regular expression. Logical operators can be used to combine matching conditions.

Continuing the database analogy, the caller uses projection to specify which fields to return. For example, suppose one only wants to know the list of programs accessing a certain file. In other cases, one wishes to know all access operations on the file (i.e. the order and access time matter). In the former, the analyzer only returns a set of program names. In the latter, it returns the sequence of all access operations.

We specify *time* in the format below:

1. "`YYYY-MM-DD hh:mm:ss`": an explicit timestamp
2. "`-[count][m|h|d]`": a relative time earlier than the current time
3. "`OLDEST`": the time of the oldest available entry in the log
4. "`NOW`": the current time.

The *time interval* is then specified as: "[*start time*] `TO` [*end time*]".[5] Some examples of time interval definitions are:

- "`2006-01-25 11:47:51 TO 2006-08-21 13:41:16`"
- "`-1d TO NOW`" (in the last 24 hours)
- "`OLDEST TO -8h`" (everything except the past 8 hours).

Output order controls the ordering of query results. It can be either: `FORWARD`, to list the oldest entry first or `BACKWARD`, to show the most recent entry first. The `BACKWARD` option is useful to get the k most recent operations.

After obtaining the *trace_handle* from `trace_select()`, one can call `trace_next (`*trace_handle*`)` to retrieve the records and `trace_close(`*trace_handle*`)` to finish retrieval. Some sample analyzer applications using this query API are described in Section 3.2.4.

3.2.2.5 The User-Log API

The user-log API lets applications add their own entries into the log database. As applications are not allowed to write directly to the log file, custom events are generated using an `ioctl` interface to the kernel driver. One use of user-entries includes marking events related to software installation, making it easier to determine what files and registry keys are created/modified during installation. This feature also provides a general purpose logging facility.

The API for user-entries is `winresmon_userlog(`*logdata*`, `*length*`)`. It signals the trace generator to record the *logdata* to the log database in binary form. The ownership information of a user log entry (i.e. the program pathname and process) is always recorded.

Figure 3.5 shows a simple wrapper for an installer program which generates custom installation events. In this example, the *logdata* consists of the name of the software and path of the installer program. Invoking the wrapper as "`C:\ins-wrapper.exe photoshop_cs2 H:\setup.exe`", generates *logdata* containing "`INBGH:\setup.exe|photoshop_cs2`" and "`INEDH:\setup.exe|phot`

3.2.3 Implementation

Figure 3.6 gives an overview of how the logger works. The information flow is as follows:

1. A sample program `iexplore.exe` calls `CreateFile("C:\WINDOWS\system32\Macromed\Flash\Flash8.ocx", READ, ...)`.
2. This is intercepted by our system call interceptor which passes the parameters to the original system call handling routine.

[5] Although an analyzer can include constraints on log's timestamp as conditions in the *selection condition*, an explicit time interval specification is less complex.

```
#define MAGIC_INSTALL_BEGIN "INBG"
#define MAGIC_INSTALL_END   "INED"

int main (int argc, char **argv)
{
  char buff[256];

  if (argc != 3) {
    printf("example: %s software_name c:\...\installer.exe", argv[0]);
    exit(1);
  }

  buff[sizeof(buff)-1] = '\0';
  _snprintf(buff, sizeof(buff)-1, MAGIC_INSTALL_BEGIN "%s|%s",
      argv[2], argv[1]);
  winresmon_userlog(buff, strlen(buff));
  system(argv[2]); // fork and wait
  _snprintf(buff, sizeof(buff)-1, MAGIC_INSTALL_END "%s|%s",
      argv[2], argv[1]);
  winresmon_userlog(buff, strlen(buff));
  return 0;
}
```

Figure 3.5: A sample installer wrapper

3. The system call handling routine returns the file handle.

4. The handle, together with the system call parameters are then sent to the trace generator.

5. As the call is successful, the trace generator updates the file handle/path lookup table. Since `"foo"` is a relative path, the trace generator concatenates `"foo"` with the current directory and add it to the trace.

6. The handle is returned to `iexplore.exe` and operation continues as per normal.[6]

3.2.3.1 System call interception

The overview of the logger in Figure 3.6 shows that resource usage is captured by intercepting system calls. As discussed in Section 2.1, this is more complex in practice. Rather than defining and using the actual system calls, the Microsoft Windows API is described at a level higher than the operating system using the Win32 API. Although most programs use Win32, they can use the native API [67] directly. The native system call interface API is unfortunately not well documented and supported. Furthermore, the

[6]Step 6 and 4 are independent. Thus, depending on the scheduler, step 6 is not necessarily executed after step 4.

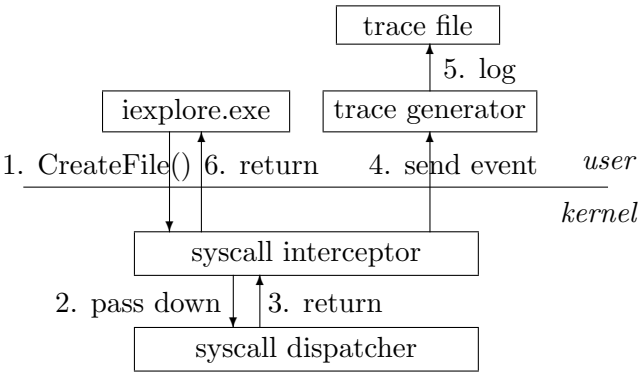


Figure 3.6: Overview of how the logger works

view of the operating system at the native API level is not quite the same as at the Win32 level. This means that intercepting system calls at what regular programs might think of as the Microsoft Windows API is problematic, and there could be discrepancies and mismatches between API use at the Win32 level and native level. To ensure accuracy, WinResMon intercepts system calls at the native API level.

WinResMon implements the system call interceptor by means of a kernel driver. The driver captures system call requests made by a process by "hooking" the native system calls [9]. Table 3.5 lists the system calls intercepted by WinResMon. We found these to be the most common system calls arising from file, registry, IPC, synchronization, and process operations.[7]

To gather information about processes, WinResMon uses a simpler method. Microsoft Windows NT exports a set of process callback functions in the kernel space [6]. WinResMon makes use of `PsSetCreateProcessNotifyRoutine()` and `PsSetLoadImageNotifyRoutine()` which notifies the call back function during process creation or termination.

3.2.3.2 Event handling

When sending the data from the kernel to the user space, it is inefficient to send every event as it arrives. WinResMon's implementation therefore makes use of double buffering. When the size of a buffer reaches a threshold, WinResMon sends the contents of the buffer to the user space and switches to another buffer to continue logging in the kernel space. This strategy tries to ensure that most, if not all, events are captured and reduces the system overhead due to context switching. Again, due to the undocumented nature of the kernel, we cannot claim to capture all events.

In the prototype implementation, the kernel and user space communicate through an ioctl mechanism. Ioctls are used to define a protocol to synchronize data transfer

[7]Since Microsoft Windows is closed source and the native API is only partially documented, it is difficult to make any guarantees about completeness.

File	
ZwCreateFile	opens or creates a new file
ZwOpenFile	opens an existing file
ZwDeleteFile	deletes a file
ZwReadFile	reads from an open file
ZwWriteFile	writes to an open file
ZwQuerySystemInformation	queries for information internal to the system
Registry	
ZwCreateKey	opens an existing key or creates it if it does not exist
ZwDeleteKey	deletes a key
ZwOpenKey	opens an existing key
ZwQueryKey	provides metadata of a key
ZwQueryValueKey	provides the value of a registry key entry
ZwSetValuekey	creates or replaces a registry key's value entry
ZwDeleteValueKey	deletes a registry key's value entry
Process	
ZwTerminateProcess	terminates a process and all its threads.
Synchronization Object (Mutex)	
ZwCreateMutant	creates a mutex or opens an existing mutex
ZwOpenMutant	opens an existing mutex
Synchronization Object (Semaphore)	
ZwCreateSemaphore	creates a semaphore or opens an existing semaphore
ZwOpenSemaphore	opens an existing semaphore
Synchronization Object (Event)	
ZwCreateEvent	creates a new event or opens an existing event
ZwOpenEvent	opens an existing event
Synchronization Object (Waitable Timer)	
ZwCreateTimer	creates a new timer object or opens an existing one
ZwOpenTimer	opens an existing timer object
IPC (Named pipe)	
ZwCreateNamedPipeFile	creates a named pipe
IPC (Mailslot)	
ZwCreateMailslotFile	creates a mailslot

Table 3.5: Intercepted system calls

between the driver and user level trace generator. WinResMon can also be extended to as a remote-monitoring tool.

3.2.4 Writing Custom Analyzers

This section demonstrates how to write custom analyzers on top of the WinResMon framework by means of examples.

- An analyzer to show all the programs which read `C:\foo.txt` after 2005/1/1 could use the following query:

```
trace_select("file.path == \"C:\\foo.txt\"",
      "prog_path",
```

```
      "2005-1-1 00:00:00 TO NOW",
      FORWARD);.
```

- A more complicated example asks, "What's the most recent execution of `msnmsgr.exe` before this boot?" First determine the last *shutdown_time* with the query:

```
trace_select("sysevent.type == \"shutdown\"",
      "time",
      "OLDEST TO NOW",
      BACKWARD);
```

The next query gets the whole process creation event for `msnmsgr.exe`:

```
trace_select("proc.childname =~
      \"/^.*\\msnmsgr.exe$/\"",
      NULL,
      "OLDEST TO last_shutdown_time",
      BACKWARD);
```

- Figure 3.7 shows a code fragment from a simple analyzer which searches for all the registry keys opened by Internet Explorer.

```
struct trace_struct *handle;
struct trace_entry *entry;

trace_handle = trace_select(
    "type==\"registry\" && "
    "prog_path=~\"/iexplorer.exe$/\"",
    "time, registry.path",
    "-1d TO NOW",
    FORWARD);
if (trace_handle == NULL)
   exit(1);
while ((entry = trace_next(trace_handle)) != NULL) {
  printf("time=%s, registry=%s",
      entry->fields[0], entry->fields[1]);
}
trace_close(trace_handle);
```

Figure 3.7: A sample analyzer

WinResMon issues the query with selection `type == "registry" && prog_path =~ "/iexplorer.exe$/"` and projection on `time` and `registry.path`. After correctly obtaining a `trace_handle`, it iterates over all selected records by using `trace_next()`. It prints all the fields, `time` and `registry.path`, for each selected trace entry. After iterating over all relevant records, it closes the handle.

3.2.5 Using WinResMon

The extended example below shows the use of WinResMon to solve software maintenance problems.

Yahoo Toolbar adds tabbed browsing to Internet Explorer (IE) and adds various icons and links from within IE to different Yahoo services. The following walk-through illustrates monitoring the Yahoo toolbar throughout its entire life cycle. There are three stages:

1. Installation

   The provided installer refuses to run under a standard user account as it needs administrator privileges. Upon successful installation under an administrator account, it creates the following DLLs and keys:

   DLLs:
   `C:\Program Files\Yahoo!\Companion\Installs\cpn\yt.dll`
   `C:\Program Files\Yahoo!\Companion\Installs\cpn\YTabBar.dll`
   `...`

   Registry keys:
   `HKEY\_LOCAL\_MACHINE\SOFTWARE\Yahoo`

   From our list of sensitive registry locations, we note the following.

   Before Installation:
   Search Page:
   `http://www.microsoft.com/isapi/redir.dll?prd=ie&ar=iesearch`
   Search Bar: -

   After Installation:
   Search Page:
   `http://us.rd.yahoo.com/customize/ycomp/defaults/sp/*http://www.yahoo.com`
   Search Bar:
   `http://us.rd.yahoo.com/customize/ycomp/defaults/sb/*http://www.yahoo.com/search/ie.html`
   Yahoo! Toolbar Helper:
   `02478D38-C3F9-4EFB-9B51-7695ECA05670 - C:\Program Files\Yahoo!\Companion\Installs\cpn0\yt.dll`

   Among the changes made, Yahoo replaced MSN search as the default search engine.

2. Program usage

   Since the database is persistent, WinResMon logs all the events associated with Yahoo and preserves the information even if the system reboots. This helps analyze the behavior of Yahoo toolbar over a period of time.

3. Uninstalling

   When uninstalling Yahoo toolbar, WinResMon observes that all the files have been removed. However, the registry settings it made are left unchanged. As a result, Yahoo remains the default search engine for the system.

3.2.6 WinResMon Overhead

To measure the performance overhead resulting from constant monitoring of systems with WinResMon, we first look at some worst case scenarios using micro-benchmarks consisting of only repeated system calls.

Our micro-benchmarks comprise of: seven benchmarks on file access, five on registry access, and two on process creation. All of these micro-benchmarks run on a Pentium 4 2.4GHz machine with 512MB running Microsoft Windows XP with SP2. The benchmarking procedure consists of first running the benchmarks on a clean Microsoft Windows XP (with SP2) to get the baseline performance. We then run with WinResMon loaded. Finally, we run with FileMon [4] loaded for file access benchmarks and RegMon [7] loaded for registry access benchmarks. Each micro-benchmark repeats an operation n times. We performed each benchmark four times to get the average execution time. Table 3.6 and 3.7 show the average and standard deviation of the execution time in seconds.

The file access benchmarks consist of:

(F_1) Open an existing file. The same filename is used every time.

(F_2) Create a new file and delete it. A different filename is used every time.

(F_3) Read 1 byte from a file. We ensure that the file is large enough so that EOF is never met for multiple reads.

(F_4) Read 4,096 bytes from a file. The file size is a multiple of 4,096. When we reach EOF, we rewind to the beginning of the file.

(F_5) Write 1 byte to a file. We start with an empty file.

(F_6) Write 4,096 bytes to a file. When we reach EOF, we rewind to the beginning of the file.

(F_7) Create a new directory and delete it. A different filename is used every time.

The benchmarks for registry access are:

(R_1) Open an existing registry key. The same key is used every time.

(R_2) Create a new registry key and delete it. A different name is used every time.

(R_3) Create a new volatile registry key and delete it. `RegCreateKeyEx` is used with the `REG_OPTION_VOLATILE` option.

(R_4) Query the value of a registry key. The type `REG_DWORD` is used.

(R_5) Set the value of a registry key. The type `REG_DWORD` is used.

The benchmarks for process creation are:

(P_1) Create a dummy console process and wait for its termination.

(P_2) Create a dummy GUI process and wait for its termination.

Table 3.6 shows the execution time (in seconds) for the file and registry access benchmarks. In order to avoid any extraneous overhead from the FileMon and RegMon GUI, the window is always minimized during the experiments. It appears that both FileMon and RegMon do not capture all the operations during the performed micro-benchmark, though. For example, during the "Read 1 byte" test, 10M events occurred, but FileMon only captured about 15K. During the "Create a new file" test, 600K events occurred, but only about 18K were actually captured. During the "Query value" test, 1M events occurred, but only 993,938 were actually captured by RegMon. We observe that WinResMon, by contrast, captured all operations in all the tests.

Note that FileMon, RegMon and WinResMon address different goals. FileMon and RegMon are meant for short term monitoring while WinResMon is designed for long term use and is therefore always running in the background. These two benchmark comparisons merely give us a baseline on how WinResMon compares with other, similar monitoring software.

Table 3.7 shows the results of the process creation benchmark. The console process in the benchmark creates a dummy child process using the `CreateProcess()` function and waits for its termination using the `WaitForSingleObject()` function. The difference between a console program and a GUI program is that the GUI program uses the `WinMain()` entry function and is linked using the `/SUBSYSTEM:WINDOWS` option, while the console program uses `main()` and `/SUBSYSTEM:CONSOLE`. The measured overhead is quite small because process creation is a slower operation than file or registry access.

We would expect normal programs to have much smaller overhead than that of the micro-benchmarks because the micro-benchmarks are very system call intensive. Normal programs, such as our macro-benchmarks, typically make significantly fewer system calls, spending more time in the application rather than the kernel. Table 3.8 gives some macro-benchmark results which show the impact of WinResMon on the following normal programs: `WinRAR`, `gcc`, LaTeX and `Lame`. The benchmarks perform the following:

- `WinRAR`: compress a 150MB file.

File Operation	n	Clean	WinResMon	FileMon
(F_1) Open an existing file	1M	20.457 ± 0.240	46.266 ± 3.271 (126.2%)	44.168 ± 0.279 (116.0%)
(F_2) Create a new file	100K	53.004 ± 2.532	67.539 ± 0.469 (27.4%)	73.117 ± 0.265 (37.9%)
(F_3) Read 1 byte	10M	14.277 ± 1.084	278.175 ± 14.282 (1848.4%)	107.414 ± 4.765 (652.4%)
(F_4) Read 4096 bytes	10M	41.207 ± 0.133	328.203 ± 34.869 (696.5%)	138.816 ± 0.793 (236.9%)
(F_5) Write 1 byte	10M	49.824 ± 1.160	388.050 ± 0.837 (678.8%)	172.422 ± 1.114 (246.1%)
(F_6) Write 4096 bytes	10M	116.355 ± 0.716	448.933 ± 2.192 (285.8%)	212.828 ± 2.950 (82.9%)
(F_7) Create a new directory	100K	46.546 ± 0.344	57.750 ± 9.565 (24.1%)	56.395 ± 0.282 (21.2%)
Registry Operation	n	Clean	WinResMon	RegMon
(R_1) Open an existing key	1M	10.378 ± 0.039	35.324 ± 0.080 (240.4%)	361.438 ± 40.504 (3382.7%)
(R_2) Create a new key	100K	8.980 ± 0.037	13.769 ± 0.041 (53.3%)	134.879 ± 10.778 (1402.0%)
(R_3) Create a new temp key	100K	7.832 ± 0.045	12.750 ± 0.082 (62.8%)	142.961 ± 12.811 (1725.3%)
(R_4) Query value	1M	1.461 ± 0.009	27.203 ± 0.061 (1761.9%)	166.301 ± 4.406 (11382.7%)
(R_5) Set value	1M	22.890 ± 0.153	46.379 ± 0.090 (102.6%)	182.473 ± 7.272 (697.1%)

Table 3.6: Performance comparison on file and registry access (n operations in seconds)

Process Operation	n	Clean	WinResMon
(P_1) Create a console process	10K	35.488 ± 0.071	37.855 ± 0.150 (6.7%)
(P_2) Create a GUI process	10K	34.641 ± 0.044	36.938 ± 0.097 (6.7%)

Table 3.7: Performance of process creation (in seconds)

Test Case	Clean	WinResMon
WinRAR	224.443 ± 0.542	226.524 ± 3.502 (0.9%)
gcc	26.265 ± 1.219	26.973 ± 0.968 (2.70%)
LaTeX	27.211 ± 0.473	27.498 ± 0.981 (1.1%)
Lame	45.631 ± 0.538	45.662 ± 0.534 (0.6%)

Table 3.8: Performance of macro-benchmarks (in seconds)

- `gcc`: compile a 500K-line C program.
- `latex`: compile a 2,000-page LaTeX file into PDF using the `pdflatex` program.
- `Lame`: encode a 100M wave file into a mp3 file.

The macro-benchmark results demonstrate that running WinResMon all the time is quite reasonable under typical usage.

3.2.7 Related Work

From a high level perspective, WinResMon differs from previous systems/tools in that it is:

- *integrated*: since it monitors accesses on files and registry under one infrastructure
- *extensible*: system administrators can write their own custom modules to utilize the generated log
- *geared for log management*: system administrators can view resource access activities generated over time, and inspect their relationships with respect to software configuration and dependencies.

We briefly mention some other tools/systems below, and highlight the important differences with WinResMon.

WinResMon shares the basic monitoring functionalities with FileMon and RegMon tools (Section 2.2.5). However, WinResMon's infrastructure is integrated, and its log database is designed to assist system administrators in inspecting software configuration and dependencies.

Strace (Section 2.2.3) is a Linux/UNIX tool used to intercept and log system calls invoked by a process. There is also a Microsoft Windows NT port of strace with similar functionality. WinResMon differs from strace in that is focused more on resource

usage (files, registry, etc) rather than system calls. In Windows, a system call viewpoint can be confusing since there are multiple levels of APIs which translate into the poorly documented native API.

Systrace [74] is a UNIX tool for sandboxing untrusted code. Unlike Systrace, which examines system-call sequences issued by the monitored processes and applies a specific security policy, WinResMon is meant as a monitoring tool to inspect resource usage and interactions among programs in a system.

DTrace (Section 2.2.6), SystemTap (Section 2.2.7) and LBox (Section 3.1) are auditing and instrumentation systems on various UNIX operating systems. They are all event based auditing systems, performing a specific action only on a specific event. DTrace and SystemTap allow administrators to dynamically execute supplied code in the kernel when certain event occurs. LBox allows the kernel to notify a user space program when certain events happen. Both LBox and WinResMon are designed for monitoring resource usage, while DTrace and SystemTap are designed for general system call instrumentation.

3.2.8 Conclusion

We presented the motivation, design, implementation and usage of WinResMon. Its main use is to inspect resource access and software dependency issues in Microsoft Windows environments. As WinResMon is extensible, system administrators can also build tools using WinResMon for custom queries and system analysis. Benchmarking shows that WinResMon is reliable and is comparable to other popular tools.

Future work is to increase the usability and robustness. We would also like to ensure that logging is as comprehensive as possible taking into account the undocumented and unsupported nature of the APIs in the Microsoft Windows NT kernel. We would also like to further increase the efficiency of the logging mechanism.

In Chapter 5, we will show how WinResMon's log can be explored through visualization. While we have shown how our query API can be used to answer software dependency questions, in Section 5.1, we will show how we can visualize a particular type of dependency — module dependency. WinResMon's log not only can be considered as a database recording the program-resource relation, but also can be considered as a trace recording program behaviours. In Section 5.2, we will visualize WinResMon's log in order to study program behaviours, including repeated patterns, abnormal events and behaviour differences, which are useful to discover various of problems.

Chapter 4

External Monitoring

Chapter 3 shows software monitors that execute in the *host*, which is shared with the monitored software. We have showed their robustness under the assumption that the kernel is authentic. However, when the kernel is compromised, the *host-based* monitoring cannot be trusted. In this chapter, we propose *external monitoring*, where information is collected from monitors external to the host, so that the information can be trusted even in a compromised host. We show that this can be used to detect malware in a host if we correlate user interaction with host behaviour, both of which are observable by our external monitor. In addition, the information of user interaction can be used to guide resource allocation, thus limiting the resource utilization of malware.

4.1 Introduction

Securing computers against malware is becoming increasingly difficult today. Anecdotes abound that the survival time of an unpatched PC running Windows XP connected to the Internet is in the order of minutes [5, 8]. Worms can be quite effective in infecting systems, e.g. the Conficker worm [82] is estimated to have infected 6% of computers on the Internet.

Often the goal of the attackers is to infect a host to make it part of a botnet. The malware may be mostly dormant until it is activated, as such, it can be difficult to detect that the host is infected. The detection problem is made worse since malware can exploit rootkit techniques to hide its presence and also any activity. For example, the Mebroot [14] rootkit infects the master boot record of the hard disk allowing it to infect the Windows kernel during boot. After that it hides the changes to the master boot record to make it difficult for antivirus software to detect its presence.

Many security mechanisms are host-based – either part of the operating system or interact with it. A primary exception is network-based security mechanisms which analyze network data and traffic. While there are some successes in detecting the presence or

activity of malware by network-based security mechanisms, there are many other sources of information outside the host that are also useful for making detection and security mechanisms more robust.

We propose a framework which fuses external (with respect to the host machine) environment information collected by external sensors in decision making to have more robust and enhanced security mechanisms. We employ the framework to: (i) enhance intrusion detection; (ii) monitor and control resource usage; and (iii) enhance access and rate control policies with user presence and out-of-band feedback. The advantage of our framework is that by using the external sensors, we are able to give reliable security guarantees even when the host is compromised, i.e. by a rootkit. Our focus will be on protecting stationary hosts (e.g. workstation) which users work on rather than servers controlled remotely or mobile laptops.[1]

We take advantage of the growth in pervasive computing and sensor technology which provide (relatively) cheap sensors which can take a variety of physical measurements. These sensors are used in a variety of ways. Firstly, the sensor can be used to measure resource usage on the host itself, e.g. correlating CPU usage (the resource) with CPU temperature (the sensor measurement). Secondly, the sensors can be used to measure the interaction of the host with the environment, e.g. network usage which also includes sending emails. Thirdly, the sensors can be used to measure environmental information which is orthogonal to the host, the most important being using sensors to determine the presence or absence of the user on the host as well as physical user activity such as keyboard usage. By "user", we mean the human using the host and although there may be more than one user, we simply say user.

We are also concerned with maintaining user privacy. While it is possible to obtain comprehensive user activity information, e.g. user identity and key strokes from surveillance cameras, we focus more on sensors that only provide binary information like the presence of a user or keyboard activity or other coarse-grained indirect data. However, in some applications, privacy may not be an issue and part of the security policy, e.g. access control policies, and we will also present such applications.

Another source of environment information could be from physical security information management systems. Such systems are already in-place in many organizations and could provide relevant environment information whereby user activities can be derived, e.g. the door entry control system gives evidence that a particular user is in the machine room. Green buildings may have many sensors which are designed to detect human presence or activity.

A natural application of our framework is to enhance the security of intrusion detection

[1]We remark that it is feasible to deal with mobile devices but the framework will need to be more complex and will need to incorporate authentication features, thus, for simplicity and clarity, we focus on non-mobile hosts.

systems (IDS). This is because the IDS may be running on the host, as such, it could be compromised by malware. External sensors, on the other hand, are difficult to be accessed or compromised by either malware or remote intruders.[2] The results of an environment sensing based malware detector can be correlated with alerts from an IDS to reduce false positives. While existing IDS may incorporate network traffic information in gateways which are external to the host, the difference is that we make further use of other tamperproof sensors and fuse it with user presence and activity.

The following example shows the difference from the network intrusion problem. Consider the case of a single email being sent. At the network level, there is insufficient information to be able to identify whether it is a spam email generated by malware or a legitimate email sent by the user. Whereas, in our framework, suppose that the email is sent in the absence of the user and user activity, it is reasonable to infer that the email has been sent automatically. Furthermore, in the absence of any additional information, a default rule to classify this email as spam activity will be reasonably effective and accurate. In this section, we present some application scenarios of malware detection where a botnet is using malware on the host for: (i) sending spam email; (ii) contributing to distributed denial of service (DDoS) attacks; and (iii) using the host as a compute engine for offline dictionary attacks.

A different set of security applications is where the environment information is used on the host as part of a security mechanism. It can be used to increase the expressivity of access control and rate control policies by requiring privileged actions on a host to be correlated with physical user presence and physical activity. This prevents remote attacks which escalate privileges, e.g. privileged actions can only be performed if the administrator is present and using the console.

An important property of our framework is that it can be immune to attacks which compromise the host. Without the fusion of sensor data from the environment surrounding the host, the attacker can simply hide inside the host or erase traces of an attack or intrusion. Some malware may even be able to shut down IDSs deployed in the host. A limitation of our framework is that the sensor data obtained is coarse grained and possibly noisy. Nevertheless, our experimental evaluations show such coarse data is already useful in identifying certain mismatches between the host's and user's physical activities.

The rest of this chapter is organized as follows. Section 4.2 introduces the framework of integrating the data from external sources to the access control or intrusion detection logic. We apply the framework to malware detection in Section 4.3 to demonstrate that information external to the host enhances the detection of malware activity. The framework is extended to rate and access control applications in Section 4.4. Related work is discussed in Section 4.5 and Section 4.6 concludes.

[2] Our focus is on remote attacks rather than attacks with physical system access.

4.2 The Framework

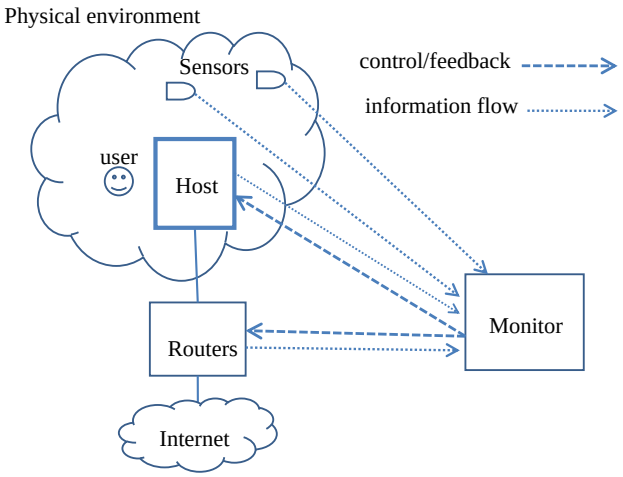


Figure 4.1: The components of the framework

Our framework is composed of the following entities illustrated in Figure 4.1: (i) the host; (ii) user; (iii) external environment; (iv) external sensors; and (v) external monitor. In this section, the *host* is a stationary computer which typically includes keyboard, mouse, hard-disk, CPU, monitor, etc., and is operated through devices such as keyboard and mouse.

The host, router(s), sensors, monitor and user are situated within an *external environment*. The host interacts with the external environment through its network which is controlled by some network *routers*. We assume that all network traffic in and out of the host is channeled through these routers. Within the external environment, there are external *sensors* for collecting measurements from the environment and also the host.

Users in our framework are persons who are directly accessing the computing resources. In order for a user to use the host/computing resources, the user needs to be in the proximity of the host and interacting with it directly through the keyboard, mouse and display. We remark that attackers could be accessing the host remotely through a network connection. Finally, there is an external *monitor* which collects information collected by the sensors and performs various security functions to make decisions/control for a security application.

We consider all information processed and stored in the host as *internal* information. Potentially, internal information can be manipulated by an adversary if the host is compromised. As such, we are more interested in the *external* information as that is immune to tampering by an adversary on the host. The external information collected by sensors can be varied. For example, an infra-red sensor can gather information in the external environment. It could be used to detect whether a user is sitting in-front of the host's

display, whereas a sensor installed in the router is an example where the sensor measures data due to host activity. The three types of information collected by sensors are:

1. Sensors which measure activity on the host: A sensor can measure externally resource usage on the host. The host CPU usage can be measured by using the correlation between CPU workload which leads to higher power consumption thus causing increases in the processor temperature. For example, the temperature can be measured using a sensor near the motherboard/processor. As disk drives also make noises when they are used, a microphone can listen to sounds from the drive; alternatively, a sensor can measure the disk activity light.

2. Sensors which measure interaction between host and environment: The main interaction we consider between the host and external environment is network traffic to/from the router. The router can itself be utilised as a sensor to measure network traffic to/from the host.

3. Sensors measuring user presence/activity: We utilise sensors to detect the presence of a user at the host. For example, infrared sensors can detect user presence, alternatively, pressure or motion sensors can be fitted to the user's chair. A more reliable method to measure user presence is to use video captured by a camera [50, 51].

   User activity can also be measured by sensors, e.g. a pressure sensor to measure keyboard typing.

Although we do not exclude the use of surveillance cameras as the environment sensors in our framework, the issue of user privacy must be taken into consideration in an implementation of the framework. (A recent webcam spying issue highlights the necessity of dealing with privacy [3]). A microphone not only can detect keyboard activity, but it can also record conversation among the users. A camera recording the user or display can also violate workplace privacy policies. For this reason, we do not make use of video in our experimental evaluation. Hence, we mainly consider binary information on user activities, such as whether a user is present, or detecting keyboard activities. Such information can be captured by sensors that are designed or can be converted to give binary output or other coarse information, which alleviates privacy concerns, e.g. an infra-red sensor that detects user presence, or a camera that only outputs the detection outcomes. There are many commercial wireless multi-sensor boards which fit our purposes, e.g. SBT80 from the EasySen [2] contains a number of sensors including infra-red, temperature and acoustic.

We remark that privacy may not always be an issue. Study [51] shows that users found continuous video sent to a monitor to be an acceptable trade-off. We will consider user presence measurements used for access control in Section 4.4.

All information gathered is channeled to a monitor which makes decisions for particular security applications. The monitor may also have the ability to control external resources, e.g. control the network traffic at the router. It could also control some aspects of the host. The channel between sensors and monitor should be secure, e.g. it should be not accessible from the Internet to prevent tampering by remote attackers.

Besides wireless sensor networks, many physical security systems can provide relevant information for our monitor. For example, the door entry control system can give information about persons who gain access to a machine room, while surveillance cameras can reliably detect the presence of users near a console or other devices like scanner and printer. Although physical security systems are costly, an organization could already have such systems in-place, together with a management system that collects and processes the data.

4.3 Applying the Framework to Malware Detection

We apply our framework to three malware detection problems by giving prototype malware detection implementations which detect email spamming, DDoS attack and password cracking. These are then evaluated with some experiments.

For the proof of concept implementation, we use a simple setup with two kinds of environment sensors. One sensor detects user presence at the host. Another sensor records the temperature near the CPU as a proxy for CPU usage. Network traffic of the host is also monitored at the router.

In our experiments, the malicious activities are carried out by a modified Agobot worm (also known as Gaobot) [80]. The worm is programmed with three tasks: sending spam email to other email accounts using the SMTP protocol; carrying out a DDoS attack by flooding a target with UDP packets; and using the CPU resources on the host to perform password cracking by hashing a dictionary of possible passwords.

The basic idea behind our detection rule is simple: the patterns of legitimate resource usage when the user is interacting with the host, is different from that when no user is present.[3] If the malware does not have/use user presence information, its behavior will not be correlated with the user presence. We divide the time into intervals, in each interval, the user is either present or absent. A detection algorithm is then applied to each interval to detect malicious activity. The algorithms we have used are rate based detection, moving average based detection, or changepoint detection [94, 21]. Other detection algorithms such as variations of rate based detection algorithms are also applicable [28].

[3] In this application, we do not distinguish between different users.

4.3.1 Methodology

The methodology for each malware detection application is as follows. For each application, we will first present the malware detection model, including the relevant external sensors and rules to the anomaly detection; then we perform experiments and analysis to validate the feasibility of our models.

4.3.1.1 Experimental Setup

Our experimental setup consists of 5 hosts in a 100Mbps LAN which is connected to the Internet through a router. All 5 hosts are running 32-bit Windows XP SP2 with Pentium 4 2.4GHz Processor and 2GB memory. First, we gather normal data from three uncompromised hosts for a period of 12 days. Next, we "compromise" one host by installing the modified Agobot on it and control it from another machine. The compromised host carries out different activities on demand: generate spam email, start a DDoS attack and perform password cracking. More experimental details are described in the relevant section.

Three types of sensor data are collected on the hosts: (*i*) user presence; (*ii*) CPU temperature; and (*iii*) network traffic. User presence is represented by a binary-valued time series, indicating whether the user is sitting at the host. We use infrared sensors to detect user presence. This is more reliable compared to using acoustic sensors to detect keyboard typing or mouse clicks. One reason is that a user can be simply looking at the monitor and not doing anything. Another reason is that malware can fool the sensor by playing back clicks.

The CPU temperature measurement is represented by a real-valued time series giving the temperature at the processor, which correlates to CPU load. In principle, the CPU temperature should be measured with a sensor between the CPU heat sink and fan, but we had difficulty in doing so with our sensor. As a proof of concept, we used the CPU temperature obtained from the host's operating system as a proxy.[4] We remark that the sensor data should be measured and sent to the monitor via a secure channel, and thus our temperature readings is only a simulation to simplify the experiments. For network traffic, headers of all packets and partial payload passing through the router are logged.

4.3.1.2 Detection Methods

We use three detection methods for each application in this section. We then demonstrate that the execution of malware can be detected using any of these methods using our framework. The three detection methods are rate based detection, moving average detection, and changepoint detection (our modified version of the CuSum algorithm).

[4] We use the system call *IDebugDataSpaces–>ReadMsr*() in Windows.

Malware Threat	Rules for triggering alarms
Email Spammer	(i) No user is present, and at least one email is sent; or (ii) A user is present, and changepoint detection decides that the cumulative sum of email sent exceeds a threshold.
DDoS Zombie	Changepoint detection detects the cumulative sum of the net outgoing packet rate exceeds thresholds for user present and absent
Password Cracker	Changepoint detection detects the cumulative sum of CPU temperature exceeds a certain threshold when user is absent

Table 4.1: Overview of malware detection rules using changepoint detection.

Rate Based Detection The general idea is that the allowable usage rate is dependent on user presence. When the user is present, the resource usage rate is higher than when the user is absent.

This metric limits the resource usage rate and uses an application specific threshold for detection. The specific threshold for a particular application can be determined from normal usage profiles.

The main advantage of rate based detection is its simplicity. However, the drawback is that it is very sensitive to the instantaneous rate. Thus, it has potential in having a high false positive or false negative rate. In addition, the detection threshold is prone to tampering if it is adaptive.

Moving Average Detection The moving average of x_i is defined by the formula below

$$\overline{x_i} = \begin{cases} (\sum_{j=1}^{i} x_j)/i, & 1 \leq i < n \\ (\sum_{j=i-n+1}^{i} x_j)/n, & i \geq n \end{cases} \tag{4.1}$$

where n is the number of readings taken into an average measurement. An anomaly is detected at time i if $\overline{x_i}$ exceeds a chosen threshold that bounds the normal values.

Arguably, moving average is less sensitive to the instant rate surge, thus it reduces the amount of false positives as compared to rate based detection. If the anomaly starts at time 1 or prior to the detection phase, the anomaly can be detected in a similar time to the rate based detection. However, if normal behavior is observed initially for some time, the alert of abnormal behavior is delayed due to the averaging behavior.

Changepoint Detection The problem of changepoint detection [21] is to detect changes from normal behavior. We use a customized version of CuSum [70] to do such detection. In our model, behaviors are described as a real-valued time series $x_1, \ldots, x_n$. Normal behavior is estimated to have a value a or smaller. To detect an increase from the normal

behavior, the following cumulative sum S_n is computed.

$$\begin{cases} S_0 = 0, \\ S_n = S_{n-1} + \max(0, x_n - a), \quad n \geq 1, \end{cases}$$

where x_n is the observation at time n from the sampling process. The value of a can either be determined by the system administrator, or by observing samples of normal behaviors to select the smallest acceptable normal upper bound for x_n. If S_n exceeds the threshold N, a change point is detected at time n and the alarm will be triggered. Note that a can be exceeded when the normal behavior is violated, so a is a lower detection threshold for abnormal behavior, S_n is the cumulative sum of the actual signal and N is the changepoint detection threshold. CuSum can be shown to be optimal under certain conditions [21]. As it is simple and efficient, it is suitable for real-time processing of streaming sensor data.

We incorporate a limit t on the accumulation time. This is similar to the moving average computation, in the sense of monitoring a moving time quantum. Yet it differs from moving average or rate based detection in that the signal which changepoint detection monitors is the excess of the rate over the normal upper bound, instead of the rate or the average rate themselves.

Changepoint detection relies on empirical or administrative thresholds, smart attackers may be able to obtain information on user presence and adapt their behavior accordingly. Our mechanism cannot fully stop such malware, but it effectively mitigates the malicious activity. Thus, it will still be effective even when the malware uses the similar information to the sensor.

The specific changepoint detection rules for each of the three applications are summarized in Table 4.1.

The Difference between Changepoint Detection and Rate Based Detection We use spam detection to show the difference between rate based detection and changepoint detection. Figure 4.2 illustrates situations where rate based detection causes false positives and false negatives. Occasionally, the user email rate exceeds the spam detection threshold, causing false alarms, even though the high rate could be due to a transient spike. On the other hand, spam malware can control its email rate to just below the threshold, by spreading the operation over a long period of time. Rate based spam detection do not report such cases correctly, while changepoint detection can. With changepoint detection, we can set the upper bound of normal email rate as a baseline, e.g. one email per minute, and then accumulate the excess amount of emails over a time period. Thus, it computes within a time interval, the area in the graph enclosed below by the baseline and bounded above by the email rate curve. With changepoint detection, occasional high email rates

do not cause false alarms, and constant medium email rates cannot evade detection.

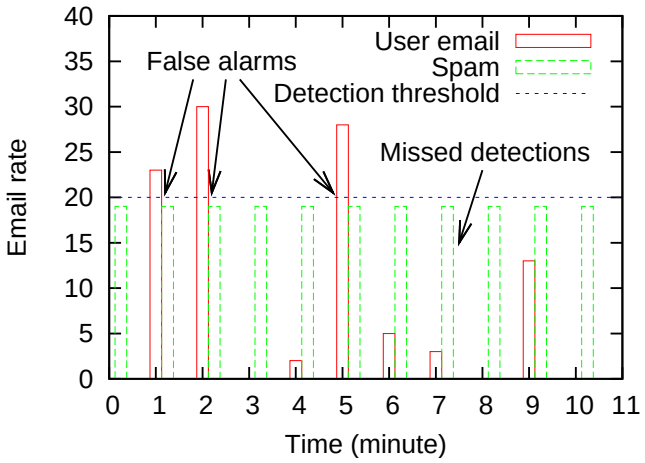


Figure 4.2: False detections caused by email rate based spam detection

4.3.2 Detecting Malware which Sends Spam Email

It is common that botnets are used to send spam email. Spam can be sent out at different rates. Some worms like Storm and Rustock have been reported to send at lowish rates of 20 and 33 emails/min, while others can be high, e.g. Srizbi's rate is 1800 emails/min [45]. While 20 emails/min is probably too high for humans, malware could also send at lower rates. We remark that humans might also send email at high rate, e.g. when using a script to send emails to a group of recipients such as a mailing list.

Our experiment is meant to show spam detection is reliable when the monitor uses data from external sensors. We also experiment on different modes of sending email, namely, SMTP-based clients and also webmail clients.

To detect bots that send spam, using our modified CuSum detection, we apply the following rules: (i) when the user is absent, any outgoing email flags a spamming activity; and (ii) when the user is present, our detection algorithms are applied on the number of emails sent. Essentially, if over N emails are sent within a time interval of length t, the algorithm flags it as spamming activity, where N and t are the two tunable parameters and carries slightly different meanings in each detection algorithm.

The first rule relies on the fact that emails are usually directly sent to the mail servers when the user performs "send" in the email client. Any scheduled delay in delivery is based on the server itself. The actual value of N and t in the second rule can be learned from the normal traffic for each host. Although the rules are simple, no matter how slow the spam rate is, we can detect spam when the user is absent. So if a stealthy spam program sends out only a single email at night, this could slip out unnoticed by normal

Spam worm	User present	User absent
Storm	6.1 min	immediate
Rustock	3.6 min	immediate
Srizbi	5 sec	immediate

Table 4.2: Detection time of different spam worms. (Detection threshold $N = 120$ emails in $t = 6$ hours at user presence, and $N = 1$ during user absence.)

email rate based spam detection while we would detect it.

Figure 4.3 shows the typical email activity of a user in a day. About 10 to 20 emails are sent daily. It shows that our hypothesis that all emails are sent with the user present is reasonable.

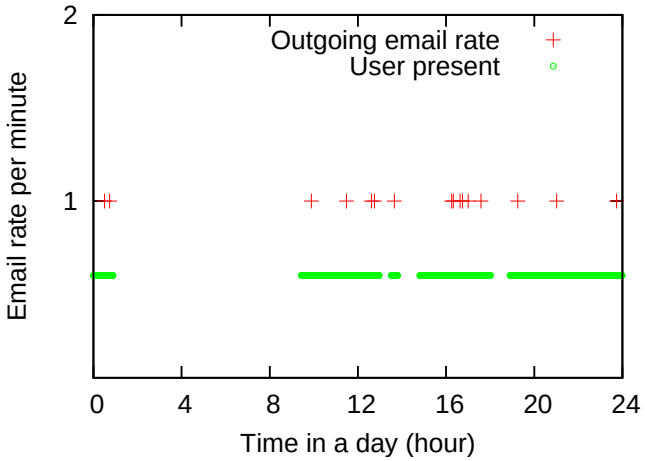


Figure 4.3: Samples of user email rate

When we apply CuSum to detect changes in the amount of emails sent when the user is present, we incorporate a limit on the accumulation time t, and set $a = 0$, $N = 120$ and $t = 6$ hours. The effect is that whenever there is an outgoing email, we accumulate the count, and no more than $N = 120$ emails can be sent in 6 hours. If the accumulated sum exceeds 120, we raise a spam alarm for the host. The allowable average email rate is three minutes per email. This is already high for a human user, since users do not consistently send an email every 3 minutes for 6 hours. Using a lower threshold will make it even easier to detect the spam. When the user is absent, we set $N = 1$, meaning any outgoing email indicates that it is spam.

In our experiment, spam is sent from the monitored host using the modified Agobot. We tested the detection of spam at rates corresponding to Srizbi, Rustock and Storm worms. We assume the spam emails are sent out at uniform rate. Table 4.2 shows the detection time when the user is present and absent. Note that threshold N is set on

Detection Algorithm	Rate	Moving Average	Changepoint	User absent
	User present (a=5 emails/min)			(a=0)
Storm	16 sec	18 sec $\sim$ 96 min	8 min	immediate
Rustock	10 sec	11 sec $\sim$ 58.2 min	4.3 min	immediate
Srizbi	1 sec	1 sec $\sim$ 1.1 min	4 sec	immediate

Table 4.3: Detection time of spam worms, using rate based detection, moving average detection, and changepoint detection

the parameter of the accumulated email amount. It is easy to see the detection time is inversely proportional to the spam rate. Since the spam rate of Srizbi is much higher than Rustock or Storm, it takes least time to detect. When the user is present, the detection time of Srizbi is within 5 seconds and about 4 and 6 minutes for Rustock and Storm respectively. When the user is absent, all three spam worms are detected instantly, because the detection threshold $N = 1$ at user absence.

To detect spam using rate based or moving average detection algorithms is straightforward. Table 4.3 compares the detection time using rate, moving average and changepoint detection. The detection threshold equals to 1 during user absence for all three algorithms. When user is present, the acceptable normal email rate is $a = 5$ emails per minute, and the detection threshold for changepoint detection is $N = 120$ emails in $t = 6$ hours, the threshold $T_r = a + N/t \approx 6$ email per minute for rate based detection, and the threshold for moving average detection is $T_m = 1920$ email per 6 hours or 16/3 email per minute on average.

Table 4.3 shows that using the user presence information, spam detection is significantly reduced for all three detection algorithms in the user absence case. In general, the more aggressive the spam worm, the easier it becomes to detect it. Srizbi is detected in much more quickly than Rustock or Storm. Comparing the detection time using the three detection algorithms, we can see that rate based detection is the quickest. However, what the detection time does not show is that rate based detection is sensitive to the instantaneous email rate. Thus, it is more likely to have false alarms than the other two detection algorithms. If the threshold of rate based detection is set too loose so as to reduce false positives, this will risk increasing the probability of false negatives. The moving average detector has a wide variation in detection time which depends on the initial state when the spam program starts execution. In the worst case, it takes about 1.5 hours to detect the Storm worm. Changepoint detection can detect all three spam worms in a reasonable time, and the detection is not sensitive to the instantaneous email rate.

4.3.2.1 Email Detection

We detect the sending of outgoing email is done by matching packets at the router with a list of signatures. We experiment with SMTP and various webmail protocols. Email sent

SMTP	the SMTP request command is `MAIL`.
`hotmail.com`	The destination IP is in the set of `hotmail.com` server IPs, the protocol is HTTP, the HTTP request method is `POST`, and the HTTP request URI starts with `/mail/SendMessageLight.aspx?`
`netease.com`	The destination IP is in the set of `netease.com` server IPs, the protocol is HTTP, the HTTP request method is `POST`, and the HTTP request URI ends with `&func=mbox:compose`
`gmail.com`	The destination IP is in the set of `gmail.com` server IPs, the protocol is HTTP, the HTTP request method is `POST`, and the HTTP request URI contains `&view=up&act=sm`
SquirrelMail	A particular HTTP request immediately after an email is sent.

Table 4.4: Rules for email detection

using SMTP protocol is relatively easy to detect, since an SMTP command `MAIL` indicates the user is submitting email. Webmail interfaces are more complex. Table 4.4 summarises the signatures we used to identify emails sent using the following four webmail protocols: Hotmail, Gmail, NetEase, and SquirrelMail.

For webmail protocols, the destination IP is first matched with a set of possible known servers relevant for that protocol. Next, we examine the first few bytes of packet payload to look for a HTTP request method `POST`. Existence of such a request indicates that the client is performing some webmail interaction such as submitting email, logging into the mail server, making a request to retrieve email, requesting for the email listing, or requesting housekeeping operations. To determine whether the client is sending an email, different checks are carried out for different mail services. For Hotmail, we look for a URI that starts with:

`mail/SendMessageLight.aspx?`.

For NetEase, the request URI ends with:

`&func=mbox:compose`.

For Gmail, the request URI contains:

`&view=up&act=sm`.

We remark that Gmail defaults to using HTTPS from late Jan. 2010 but our experiments were conducted prior to that. However, Gmail can still be configured to simply use HTTP. We discuss HTTPS webmail later.

The check to detect sending email needs to read the following data inside the HTTP message for Hotmail, Netease and Gmail: 33 bytes for Hotmail; 64 bytes for NetEase; and 80 bytes for Gmail. Hence during packet captures, the packet payload to the relevant webserver IP addresses needs to be logged partially. Detecting email sent with SquirrelMail is less straightforward as the payload is encrypted. We found that immediately after an email is sent, there is an HTTP GET request in plain text of a large size to fetch the listing of the current folder. These signatures enable us to identify the sending of email

reliably. Compared to a typical spam filter that inspects the email content, our method has less privacy concerns.

To detect email when encrypted webmail is used, we examine the HTTPS traffic destined to the webmail server. Although we cannot inspect the content of such traffic, we know that each interaction (check for new mail, compose ...) with the webmail server requires a one or more packets to be sent. Thus, the outgoing traffic rate, is also an upper bound on the email sending rate. The same reasoning can also be used to rate limit the number of emails sent.

We investigate this approach on the HTTPS traffic of Gmail. All of Gmail's web components (including Javascript, HTTPXML request, images, and Flash) are from `https://mail.google.com`. There are no third party resources except DNS traffic. When we interact with the Gmail interface, packets from `mail.google.com` are observed. We found the packet sizes to vary from a minimum of 40 bytes to a maximum of 1500 bytes, but each TLS record can be as large as $\approx 2^{14}$ bytes.

Exceptionally large TLS records are likely part of some email. However, not all emails contain exceptionally large packets, and piece of email may contain more than one large packet. We did not find any clear indication of emails sent, in the Gmail's HTTPS traffic as sending email and certain other operations, such as checking the inbox, display similar behavior in the HTTPS traffic. This shows that the detector might be limited to approximating the email sending rate by the HTTPS traffic rate if no distinguishing features can be used. We remark that the detection limit on the outgoing data rate of HTTPS traffic will need to be more generous than the limit when unencrypted email is used, but it nevertheless can be used as an upperbound on the email spam rate.

4.3.3 Detecting DDoS Zombie Attacks

This experiment deals with detecting zombies which carry out UDP packet flooding for a DDos attack. In this attack, the compromised host sends out UDP packets to a victim to consume the victim's network bandwidth. When the user is absent, legitimate network traffic rate is low and thus we can potentially detect the malicious UDP packet flood by observing the traffic rate. However, there could be background processes that generate network traffic, e.g. automated updates. Another scenario in our experiment is a legitimate P2P program. Unlike the email application, a P2P program can generate network traffic constantly even when the user is absent. Hence, it is desirable to derive another feature, instead of the overall rate, to distinguish the UDP flood attack.

We observe that the UDP flood generates one-way traffic, whereas typical legitimate processes generate two-way traffic. This motivates us to consider the net outgoing packet rate p_{net},

$$p_{net} = \max(p_{out} - p_{in}, 0),$$

where p_{out} and p_{in} are the outgoing and incoming packet rate respectively. We apply CuSum on p_{net}, but with a different threshold a when the user is absent and present.

In addition to p_{net}, we also monitor the ratio r of outgoing packets that are not responded to,

$$r = p_{net}/p_{out}.$$

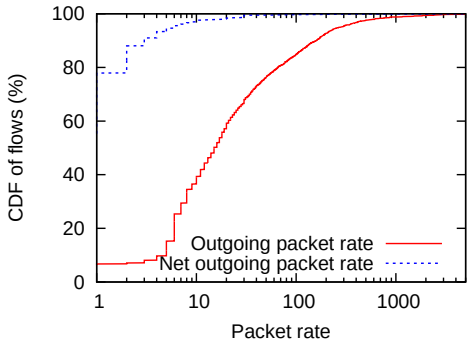
We have $0 \leq r \leq 1$ where $r = 1$ if the flow has only outgoing packets; and $r = 0$ if $p_{out} \leq p_{in}$. The excess of p_{net} over a is accumulated only if $r \approx 1$.

Figure 4.4 compares the maximum net outgoing packet rate p_{net} and the outgoing packet rate p_{out} of TCP and UDP network flows when the user is present and absent respectively. A flow is identified by a tuple consisting of $\langle$*local IP*, *remote IP*, *transport protocol*$\rangle$. We do not differentiate port numbers as attackers may open multiple ports to flood the same victim. Each flow is monitored for 10 minutes. We found this 10 minute observation window to be sufficiently long, because short lived flows last on average only 2 to 3 minutes, while flows that last for a long time are likely to have their traffic rate stabilized within this time. Figure 4.4(a) shows the distribution of p_{net} and p_{out} of 2,351 non-attack TCP flows active during user presence. Over 60% of the flows have the maximum $p_{out} > 10$ packets a minute; whereas less than 5% of the flows have the maximum $p_{net} > 10$. For example, in a sample flow, the maximum p_{out} is 2,443 packets per minute, but the maximum p_{net} is as low as 10 packets. Figure 4.4(b), Figure 4.4(c) and Figure 4.4(d) show the distribution of p_{net} and p_{out} of 980 TCP flows active during user absence, of 9484 UDP flows active during user presence and 1506 UCP flows active during user absence, respectively. They show that p_{net} is more stable a quantity than p_{out} among different flows, regardless of the transport protocol or user presence.

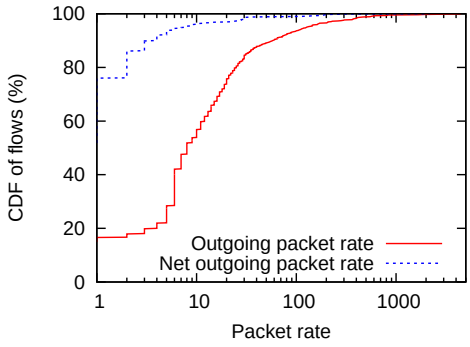
Figure 4.5 shows the distribution of p_{net} for 13,620 flows. Figure 4.5(a) shows the net outgoing packet rate is close to 0 for most flows. When the user is present, 35% of the flows have $p_{net} = 0$; and 80% flows have p_{net} less than 10 packets a minute. When the user is absent, 90% of the flows have $p_{net} = 0$. Figure 4.5(b) shows that the difference in p_{net} when user is present and absent, it can be as large as 600 for some flows, so different upper bounds of normal p_{net} should be used for user presence and absence and for each flow.

From Figure 4.5(a), initializing the upper bound of normal p_{net} to be $a = 60$ packets per minute is sufficient for most flows. Parameter a can be lowered by examining the online traffic. To accumulate the excess of p_{net} over a, r must be close to 1 and we set it at 0.95. The threshold to trigger an alarm is $N = 800$ packets, accumulated over 20 minutes. It ensures an attack flow can be detected if at least 2 UDP packets are sent a second on average.

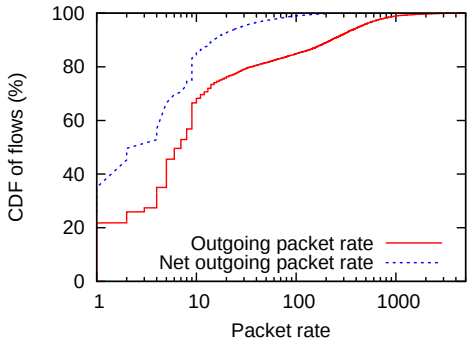
Some non-attack flows have large net outgoing packet rate. However, they do not cause false alarms, because the outgoing packets are responded to, or r is not close to 1.



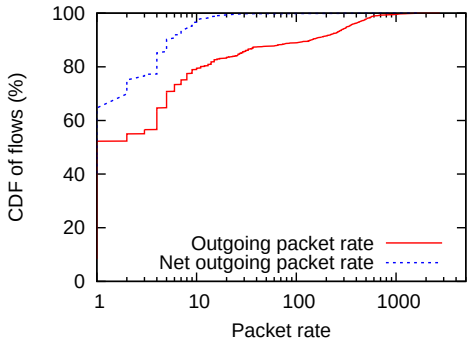
(a) 2351 active TCP flows during user presence



(b) 980 active TCP flows during user absence



(c) 9484 active UDP flows during user presence



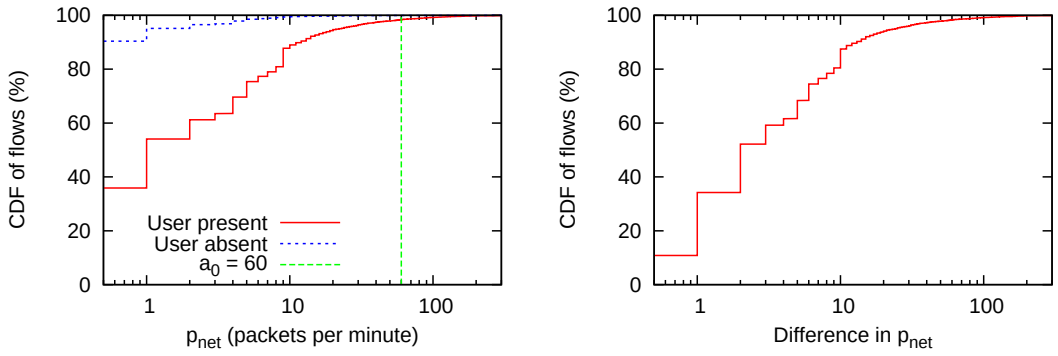
(d) 1506 active UDP flows during user absence

Figure 4.4: Difference in the outgoing packet rate and the net outgoing packet rate (in packets per second)

In terms of absolute value, $p_{net} = 1278$ is high, but its $p_{out} = 2,812$, making $r = 0.45$, so there is 1 response in about 2 packets, the ratio is reasonable and hence the high net outgoing packet rate is also accepted.

Figure 4.6 shows the net outgoing packet rate p_{net} of the attack flow under 3 different attack scenarios. The upper bound a of normal p_{net} is 10 and 2 packets a minute for user present and absent respectively. These bounds are determined from analyzing the training data of the flow. The attack starts at the 125th minute. For the continuous attack, p_{net} sharply increases to around 600. After 1.36 minutes, the alarm is triggered. The increasing rate attack increases its attack intensity very slowly to delay detection. This attack might not be detected if the detector adapts to the flow rate in near real time. Our approach detects the attack in less than 18 minutes, long before the attack reaches its peak rate. The fluctuating attack in Figure 4.6 constrains the attack intensity, and releases attack traffic in pulses. This is to avoid detection if average traffic rate or peak rate is used by the detector. Our approach detects the fluctuating attack within 3 minutes.

The DDoS detections using rate and moving average are also set to be relying on



(a) Distribution of flow net outgoing packet rates (b) Difference in the flow rates when user is present and absent

Figure 4.5: Distribution of the maximum net outgoing packet rate p_{net} with 13,620 TCP and UDP flows, each flow is observed for 10 minutes during user presence and absence

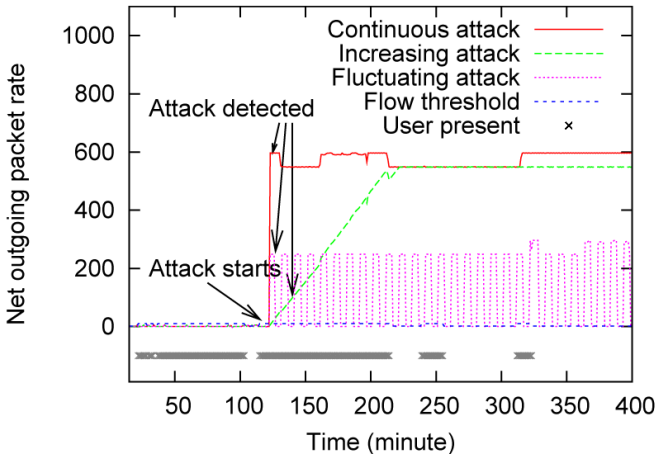


Figure 4.6: Net outgoing packet rate of the DDoS attack flow in different attack patterns

Detection Algorithm	Rate	Moving Avg (min)	Changepoint (min)
Continuous DDoS	1 sec	1.7	1.4
Increasing DDoS	8.3 min	17.8	17.5
Fluctuating DDoS	1 sec	3.6	3

Table 4.5: Detection time of DDoS attacks of different attack patterns

the net outgoing packet rate p_{net}. The threshold for rate based detection is $T_r = a + N/t$, which is 50 packets per minute or equivalent to 5/6 packet a second, when user is present. The corresponding threshold for moving average detection is $T_m = 1000$ packets per 20 minutes. Rate based detection detects continuous and fluctuating DDoS attacks immediately, since p_{net} of the attack flow is high. It detects the increasing rate DDoS attack in 8.3 minutes. Depending on the initial state, moving average detects in seconds

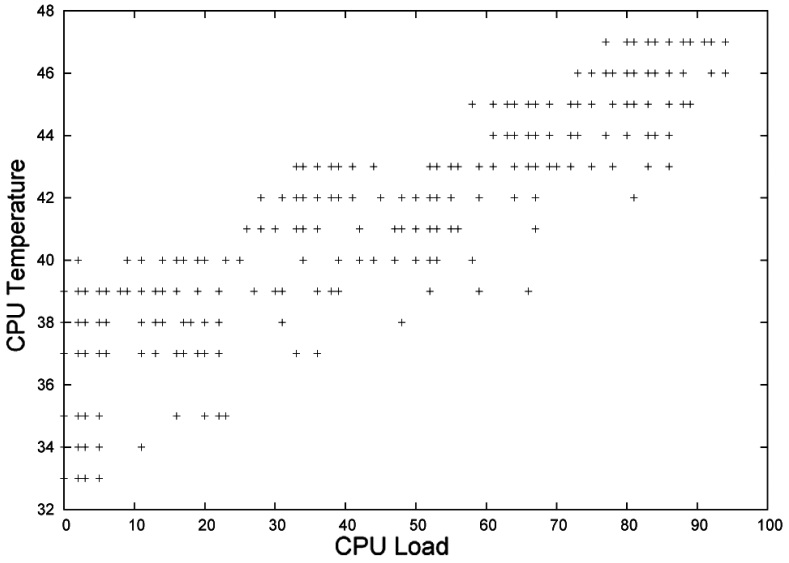


Figure 4.7: Correlation of CPU load and CPU temperature

to minutes the DDoS attacks of different rate patterns. For continuously high rate DDoS attack, the detection time ranges from 1 second to 1.7 minutes. For increasing rate attack, the detection time is the longest, ranging from 15.7 minutes to 17.8 minutes. Whereas for fluctuating rate attack, the detection time ranges from 1 second to 3.6 minutes. The result presented in Table 4.5 shows the worst case. Because in our experiments, the detection has observed a substantially long time of normal behavior before the commence of DDoS attack, the detection is delayed due to the averaging behavior.

The effect of user presence information in assisting DDoS attack detection is that it differentiates traffic rates according to user presence, giving a tighter bound on legitimate traffic rate when user is absent, and it makes the detection more effective.

4.3.4 Detecting Misuse of Compute Resources

Bots are often recruited to invert cryptographic functions to break passwords. Inverting such functions requires extensive computations which are distributed to the bots. To detect such activities, we look for increases in the CPU load when the user has been absent for a while. As CPU load is internal to the host, we rely on CPU temperature as an indirect measurement which can be obtained by an external sensor.

Figure 4.7 shows the correlation between CPU load and temperature. The correlation is good but we can see that there is noise and other fluctuations in the measurement. Thus, this is an example of a sensor measurement which is imperfect and has noise. Figure 4.8 shows the temperature when a user is present and absent. Note when the user is absent, the temperature is around 37°C, with less variability. We measured the temperature during automated software update and found the temperature increase of

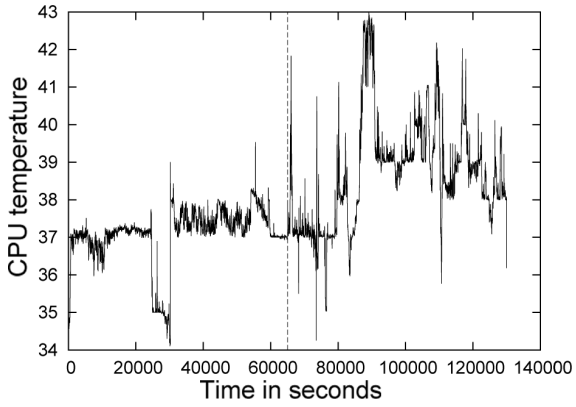


Figure 4.8: CPU temperature variation when user is absent and present. The user is absent from 0 to 64,000 second; and present from 64,000 second onwards. The user is absent left of the vertical dotted line and present to the right of the line.

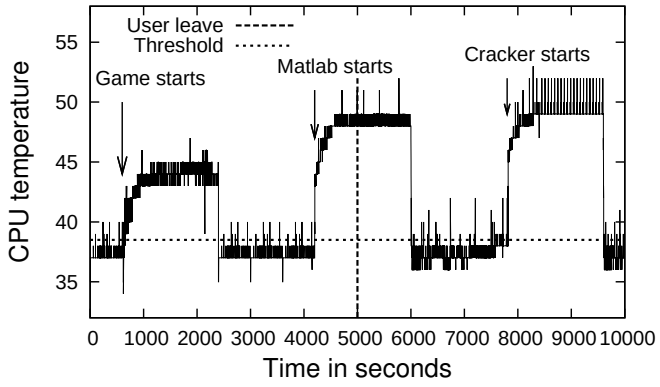


Figure 4.9: CPU temperature variation during various activities.

2-3°C.

When we employ CuSum to detect the increase in CPU temperature, we can determine the thresholds a, N and t from CPU temperatures in normal usage. Figure 4.8 shows normal temperature variations during user presence and absence. From the observation of Figure 4.8, we set the upper bound of system idle temperature to be $a = 38.5$°C. We set the threshold for the accumulated increase in temperature as $N = 2400$ in 30 minutes. This threshold N is chosen to allow software update that increases the CPU temperature by 2°C and lasts for 20 minutes.[5] We set a grace period of $g = 10$ minutes at the transition from user presence to absence, that is, if the changepoint is detected within 10 minutes after a user leaves, the alarm is not activated.[6] For comparison, the thresholds of CPU temperature for rate based detection is set as $T_r = a + N/t = 39.83$°C, and the threshold

[5]Windows Update may run when the user is absent.

[6]The grace period is a window for the bot to perform its computation but it is only a small fraction of the time available to the bot, so it still significantly limits the amount of computation which can be performed.

Detection Algorithm	Rate (sec)	Moving Average (min)	Changepoint (min)
Game	14	11	10.4
Matlab	4	8.5	5.2
Password Cracker	2	7.1	4.5

Table 4.6: Detection time of CPU intensive activities, using rate based detection, moving average detection, and changepoint detection (The upper bound of normal CPU temperature is $a = 38.5°C$, and the detection threshold $N = 2400$ in $t = 30$ mins).

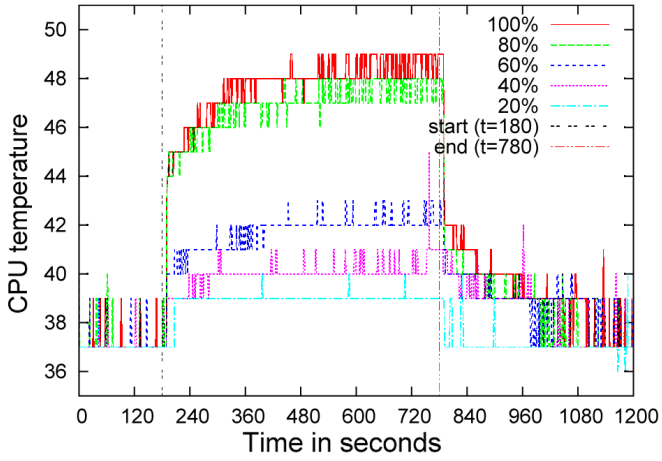


Figure 4.10: Correlating attack intensity and CPU temperature.

for moving average detection is $T_m = 40°C$. We note that the instantly measured CPU temperature that our sensor outputs has granularity of degrees.

Figure 4.9 shows the temperature during various computation tasks. Table 4.6 shows the detection time of computation intensive activities, using different detection methods. Rate based detection detects the computation activities in seconds; moving average takes 7 to 11 minutes; while changepoint detection has a detection time that is in between. The changepoints in temperature for playing games and running Matlab are detected 10.4 minutes and 5.2 minutes respectively after they start. Since the user is present or has just left when the changepoint occurs, the activities are considered initiated by the user and accepted as normal. The password cracker activity triggers an alarm 4.5 minutes after it starts. Since the changepoint is detected when user is absent, it is considered as malicious computation. If the malicious computation executes during user presence, it can be noted by the user since there could be an overall system slowdown.

Although rate based detection is very quick, it is highly sensitive to false positives and negatives. When the computation intensive programs start, this clearly causes a substantial increase in CPU workload which leads to a sudden increase in CPU temperature. Hence, rate based detection is quick in detecting CPU activity. However, very often, non-compute intensive programs can temporarily raise the CPU temperatures by 2°C upon

process startup and return to normal after that. This will cause false positive for a rate based detector. On the contrary, a moving average detector will treat sudden changes of CPU temperature as a gradual transition of the average temperature. Thus, the amount of time it takes to detect can vary significantly, depending heavily on the initial states. Detection using changepoint is usually more efficient than moving average, and is more reliable than rate based detection.

We also conducted an experiment which varies the intensity of the CPU usage. This investigates if the CPU usage can be detected when the malware throttles its computation at a lower rate. CPU usage is controlled as follows: for each one-second interval, the bot computes at full speed for x seconds and then sleep for $1 - x$ seconds, where $x < 1$. We denote the CPU usage to be $100x\%$ in Figure 4.10. We carry out the attack for 600 seconds (starts at 180 seconds and ends at 780 seconds). Note that after the attacks stop, the temperature gradually returns to normal. For the password computation running at 100% usage, we detect the change after about 4.5 minutes. For the computation at 20% usage, we can also detect the change using changepoint detection, if the computation lasts for more than 20 minutes. To evade detection, the malicious computation can only further reduce its CPU usage or shorten its execution time. Thus, our system effectively limits the amount of malicious computation. In contrast, at 20% CPU usage, rate based or moving average based detectors fail in detecting the misuse of computation resource. The increase in temperature is so subtle that it is covered up by non CPU intensive programs, even though it occupies the CPU consistently for a long time.

4.3.5 Handling Exceptions

It is important to also deal with cases where there is exceptional behavior which may otherwise cause false positives.

4.3.5.1 Scheduled Outgoing Email

If user has a large amount of email to send, and he opts to send them while going for a coffee break, our system supports that if the user has obtained sufficient email tokens. The tokens to send email are obtained by directly interacting with the monitor, such as by solving a CAPTCHA [12] or graphical Turing test to validate that it is a human request. Several tokens can be obtained in a batch, and the user only needs to solve one puzzle for validation.

Such a mechanism employing tokens to send emails can be adopted even when the user is present, as a special way of requesting for additional email quota. The tokens obtained for sending email when the user is present expire when the user goes away. Similarly, the tokens allowing the user to send email while he is away expire when the user comes back. The expiration mechanism reduces the opportunity for spam worms to hijack leftover

tokens.

4.3.5.2 Scheduled CPU Intensive Programs

We need to handle carefully the case of non-malicious programs that commonly run when no user is present. Besides system auto-updates, these programs include nightly backup, nightly virus scan, remote desktop and user scheduled computation. Backup and virus scan are regular tasks. Their resource consumption can be profiled, in terms of the start time, duration, and the increase in CPU temperature. Based on our measurement, system backup only increases the CPU temperature by 1-3°. Virus scan usually increases the CPU temperature by 2-3°. These host profiles are basis of a whitelist kept by the monitor to suppress false alarms. Even if attacker tries to hide the malware execution by running it at the same time as these tasks, the excess increase in temperature or duration will still trigger the alarm.

User scheduled computations may cause irregular behavior. As the execution of scheduled computation can be planned ahead, the user simply declares in advance to the monitor, the estimated process start time, duration and CPU load (converted to temperature increases by the monitor). The monitor suppresses any false alarm within the specified resource consumption. The user is advised to give a conservative estimation, for it is better for the user to intervene if excess resources are consumed, than allowing the attacker to free ride.

4.3.6 Security Discussion

We further discuss two security considerations.

Communication Channel In view that malware may reside in the hosts, it is important that external sensors are able to communicate with the monitor securely so that the hosts are unable to compromise the integrity, authenticity and confidentiality of the communication. One solution is to have a separate private network for the sensors, e.g. a wireless sensor network with the external sensors as the nodes. A cheaper implementation could tunnel encrypted communication through the host but that would be susceptible to DOS attacks if the host is compromised. The privacy requirements and the need for a separate private network fit well with wireless sensor networks equipped with multi-modality sensors.

Knowledgeable Malware Let us consider the scenarios where (i) the hosts are compromised by malware, (ii) the malware knows the system architecture and the detection algorithms, including the threshold values, and (iii) the external sensors, routers and the monitor are not compromised. Since the malware resides in the host, it may able to

accurately derive user presence using information from, for example, the keyboard and mouse. Based on the derived information, the malware can attempt to carry out malicious activities while mimicking normal activities, in order to evade detection. However, note that to evade detection, the malware is constrained by the detection threshold and can only utilize resources at a low rate when the user is not present. When the user is present, although the threshold is higher, the malware risks of being detected by the user. For example, the user may notice the interactive response is slow on the machine. Furthermore, user presence detection based on information accessible by the host is arguably less accurate than through specifically designed external environment sensors which add more constraints on the malicious activities.

4.4 Application to Access Control and Rate Control

Here we investigate how environment information can be useful in controlling and allocating resources. A security policy may require administrators to be physically inside the machine room to access server consoles and administrative tools. In order to mitigate malware from sending spam email, we could implement rate control in the router or the mail server, and the rate limit is based on user presence. In both cases, environment information is used as a condition to access certain resources. Note that the first case is on access control, while the second is on rate control.

4.4.1 Access Control

Our framework can be used to implement location-based access control. One location-based access control scheme was proposed by Ardagna et al. [17] to restrict access to certain resources based on physical location of the user. In their work, the physical location is obtained from mobile devices, such as mobile phones carried by users. Our framework also implements location-based access control, but in a different way. Their work adopts a user-centric approach where the device is attached to the user and user's location is measured in order to figure out which resources can be accessed. We adopt a resource-centric approach where the device is attached to the resource and user presence is observed near the resource in order to decide whether to grant the access.

Both approaches have advantages and disadvantages. When there are more resources than users, the resource-centric approach incurs more cost for the sensors. However, in the scenarios we discuss, there are likely fewer resources than users or it is closer to a 1:1 ratio. A user-centric approach may not be able to determine location reliably, e.g. GPS would not work indoors, whereas a resource-centric approach does not have to determine location.

Our access control policy not only incorporates user presence information but also

user activity information. For example, the user activity information can be "the user has typed some keys". Enforcement is implemented both on the host and router depending on the type of the resource. Enforcement implemented on the host implicitly assumes that malware is not in control of the host and the host decides the access based on environmental information gathered by the monitor, but the malware cannot affect access control policies at the environment level. Thus, the monitor and host cooperate, which is different from the earlier IDS applications where the monitor operates independently from the host. We give two access control policies to illustrate environment aware access control:

1. *The user can execute the* `/usr/bin/sudo` *program only if he is sitting at the host.* This policy is used to mitigate the problem of remote attacks - it requires that there is a human present before the sudo operation is allowed. This policy has to be enforced on the host. A remote attacker who does not yet have control of the host would be prevented from performing actions which could be used to infect the operating system.

2. *The user can send email only if he is sitting at the computer and has mouse or keyboard activity.* This policy is used to prevent malware from sending email while the user is away. Since user activity is enforced in the policy, it also prevents sending email while the user is idle, e.g. watching a movie. The intuition is that the user must have performed some typing or mouse action in order to send an email. Unlike user presence, which can be thought of as being a continuous signal, mouse and keyboard input are discrete events which may happen at time points which do not overlap with the email sending time interval. Thus, the precise meaning of the policy is "An email can be sent at time t if user presence is observed at that t and there is mouse or keyboard input between $[t - \Delta, t + \Delta]$". This policy is enforced in the router or the mail server instead of the host and thus provides enforcement even when the host is compromised.

4.4.2 Rate Control

Previously we showed that abnormal resource usage when the user is absent can be easily detected under several scenarios. This can be easily extended to controlling the resource usage rate in the case of activities involving external resources. Two natural scenarios of external rate control are:

- **Shaping Network Traffic**
  To mitigate computers being used as bots to perform DDoS attacks, the router can shape network traffic based on user presence information. By limiting the traffic on flows, this becomes a form of inverse quality of service, providing reduced quality

when the user is not present. If P2P programs are being used, this would save some network bandwidth, e.g. if the host uses Skype and becomes a Skype supernode, it could lose its supernode status under the reduced network flow.

- **Sending Email**
  Similarly, the emails could be simply denied when the user is absent. If the threshold is above zero, then for both webmail and SMTP, the router can simply rate limit the protocol. Alternatively for an SMTP server, it could quarantine email beyond the threshold. In both cases, outgoing spam emails are rate limited possibly to zero.

  The email rate can also be rate limited when the user is present. The idea is that email is based on typing and/or mouse clicks. This data can also be recorded using environmental sensors. The email rate can then be based on a function of the detected keystrokes and/or mouse clicks.

The framework can make resource rate limiting policies more flexible and usable. The idea is to have feedback if a usage is too high for a resource. External sensors can then be used as a secure channel to request a higher resource rate. For example, a monitor on the host can display the resource usage and warn the user if he is reaching the limit. The user can have something as simple as a button which is pressed to function as the secure request channel to obtain more network bandwidth or more emails. A low resolution camera could also be employed as an out-of-band channel to the monitor where the low resolution may mitigate privacy concerns. For example, it would be easy for the user to give hand signals to communicate to the monitor. On the other hand, implementing such a policy securely is difficult without the help of external environment information.

In the access and rate control application discussed above, to control the resource utilization, essentially we need to verify whether an entity is "human". Instead of using external environment to infer user presence, alternatively, a CAPTCHA could be used. Using the environment information has the advantage that the users are not interrupted by the challenges issued by the graphical Turing test.

4.5 Related Work

If we consider a computer host to include the host, the user channel and the network channel, then host security can be divided into: (*i*) software security, which ensures the software running in the host is authentic, e.g. anti-virus [72], system call filtering [74] and binary authentication [43]; (*ii*) user security, which ensures the user is authentic, e.g. password/biometric authentication, physical perimeters and surveillance camera monitoring; and (*iii*) network security, which ensures the network communication is authentic, e.g. personal firewalls [44]. Our approach to enhance host security is substantially different from the existing designs in three aspects. Firstly, the model we propose fuses data from

a few channels of external environment sensors to monitor the host activity. Secondly, as our model does not require controlling or modifying the host operating system or software, it is able to provide some level of security even when the host is compromised, rather than no security. Thirdly, our system detects outbound or on-host malware execution, which complements intrusion detection.

There are a few works which correlate information from different channels to improve host security. BINDER [31] correlates user events (user input), process events (process creation and process termination), and network events (connection request, data arrival and domain name lookup) to detect malware. Both our malware detection system and BINDER correlate user presence information and system behaviour to detect malware. BINDER uses information collected by the user's machine, which potentially could be manipulated by the compromised host.

The systems proposed by Gu et al. [42] and Yen et al. [111] detect botnets by analyzing and correlating network traces. The two systems and our malware detection system use information from the network router rather than the host in question so as to be able to deal with the case when the host is compromised and gives false information. The difference with our malware detection system is that we have user presence and activity information in addition to network information.

Kumar et al. proposed a system [50, 110, 51] which continuously monitors user's biometric identity and locks up the computer if it cannot detect the correct user. Both their system and our access control system use physical user information to provide additional factor for authentication. There are two main differences between the two. Firstly, our system only detects user presence information, while their system detects user's biometric information which is much stronger but gives less privacy. Secondly, their system runs entirely in the user's machine, thus it cannot guarantee the authentication once the machine has been compromised.

There are many works on location-based access control (LBAC) wireless networks [17]. LBAC models assume that the user devices are mobile, their locations are tracked for service continuity, or verified before granting access. The problem we address is different in that we are not focused on mobile devices, instead our focus is on utilizing a combination of environment data to enhance security, such as to detect malware on the host and to regulate the usage of resources by the host.

There are also theoretical models which fuse information from multiple channels for intrusion detection. Siraj et al. proposed a method [83] to fuse information using artificial intelligence techniques. Thomas et al. proposed a probabilistic model [89] to address this problem. Our work, on the other hand, addresses the physical sensor framework. In terms of information fusion, we have employed a simple rule based method, however, other methods could be adopted as well.

4.6 Conclusion

In this section, we proposed a framework that incorporates environment information in securing host computers in a few ways: by using the environment information as an additional source of information for malware detection, or by integrating the environment information with existing conditions in rate-control mechanisms and access control policies. We argued that, since the sensors are "external" with respect to the host, they are difficult to be accessed and tampered by a compromised host. Furthermore, we present three prototype intrusion detection applications which show that coarse information on user activities and resource usages is sufficient to provide intrusion detection even with a compromised host. It is also useful in expressing certain rate-control and access control policies. We have also identified a few important requirements of the sensors, in particular, the concerns of user-privacy and the need of a secure channel. Thus, we have proposed a simple and effective framework for security enhancement which is arguably safe against compromise by attackers.

The framework also takes advantage of the growing popularity of pervasive computing and sensor networks. As the trend in cost of wireless multi-modality sensors is decreasing, applications of our framework are feasible for cost-effective deployment in the near future.

Chapter 5

Visualizing System/Software Traces

The previous chapters have showed our monitoring systems. They can generate traces which contain valuable information on the execution of the software. The information include internal control flow of the program as well as external interaction between the operating system and other programs. However, the traces can be very large and also difficult to analyze. We observed that an idle Windows system generates about 800K events/hour using our WinResMon (Section 3.2). A busy system can generate more than 100M events/hour. Statistics [91] also show that Windows server workloads can generate as much as 70M events/day, while an idle machine can generate about 1M events/day. Although much of the interactions between the processes and software in the system is contained in the trace, in practice, the detail can be overwhelming.

One way to understand very large amount of data is through visualization, which abstracts information into a graphical form so that it can be more easily perceived by human. In this chapter, we show two trace visualizations which visualize the traces generated by our WinResMon prototype. Throughout the chapter, we work on two types of traces, The *system trace* records the interaction among the operating system and different software. The *program trace* records the execution of a single program, whose level of detail can be system call only as in WinResMon, or instruction trace (Section 2.2.8).

Section 5.1 shows our first visualization, which investigates the dependencies between programs and binaries. Software often lives in a complex software eco-system with complex interactions and dependencies between different modules or components. As we have discussed in Section 2.1, this problem is exacerbated both by the overall system complexity and its closed source nature in Windows. Even when source is available, there are still interactions with modules which are only in binary form. The visualization uses system traces from WinResMon and program traces from binary instruction, thus it does not need to rely on source code. We use the following scenarios to explain how our visualizations

can be used to investigate various aspects of software dependencies: (i) visualizing whole system software dependencies; (ii) visualizing the interactions between selected modules of some software; (iii) discovering unexpected module interactions; and (iv) understanding the source of the modules being used. Because of the large number of modules and their complex dependencies, we developed a number of "zooming in" techniques including grouping of modules; filtering by causality; and the "diff" of two dependencies.

Section 5.2 shows our second visualization, `lviz`, which is a visualization tool for many different purposes including software failure diagnostics, analysing performance issues, anomaly discovery, etc. The visualization is based on DotPlot, which compares two traces and plot the common (or different) items. It was early used for analysing similarities in DNA sequences [55]. `lviz` extends the traditional DotPlot through a number of visual elements so that we can easily associate the visual representation with events in the trace and identify the key events. As we will see in a number of case studies, `lviz` is highly customizable can be used to look at problems across a large spectrum.

5.1 Comprehending Module Dependencies and Sharing

Software is often made up of a mix of different modules; from system components, to software from various third parties. One's software is then part of an eco-system of software modules which forms a rich and complex web of relationships. System components (usually in the form of dynamic link libraries) are also crucial and are usually needed for a program to function.

As discussed in Section 2.1, Microsoft Windows has a very complex software module eco-system. In addition, the usage dependency between APIs with the actual software components delivered as binaries is usually not properly documented. Microsoft software often makes extensive use of Microsoft software components without much documentation. While source code may be available, it would not be available for all the relevant software modules. Some modules are so extensively embedded in the system that they are hard to study or understand.

This "web of dependencies" between software, means that there can be fragile and implicit dependencies between all the software installed (past, present or future) on a system. In Windows, this often leads to software failures when installing or uninstalling some application. This is colloquially known as "DLL hell" [15].

In this section, our objective is to understand the usage and dependencies between software modules (executables and DLLs) in Windows. We work purely with binaries since source is not available for all the components. Furthermore, some software may be written in more than one language. Even when there is source, it will still interact in possibly complex ways with other modules which are only in binary form. We propose two visualizations: EXE dependency graphs and DLL dependency graphs. The first is useful for comparing several programs against each other, possibly this may include the involvement of other processes not from those executables. The second is useful for investigating the interactions between an executable and the DLLs used, as well as, interactions between the DLLs themselves.

We deal with potential complexity of the visualizations in a number of ways. The visualizations are simplified either through a compressed dependency graph or through the simple notion of grouping by function or vendor (but other mappings are possible). Projection gives a simpler graph giving the interactions which only affect selected modules; while the diff operation gives the difference between two dependency graphs.

It turns out that both visualizations are simple, yet effective for achieving comprehension of module interactions and dependencies of realistic software on Windows. For example, we give scenarios involving whole system usage from boot to shutdown, understanding strange interactions which can occur when software is installed (TortoiseSVN modules affecting arbitrary software), sharing of modules between different browsers, understanding module usage in terms of source of the module (vendor perspective), visu-

alizing usage of different versions of modules for networking, etc. We believe that it is a good idea for software developers (and even users) to understand the interaction between modules on a system. This is useful for developing and maintaining robust software. Furthermore, the complexity of the Windows environment is such that many unexpected interactions and dependencies can occur without the knowledge of software developers, administrators and users.

5.1.1 Related Work

Most of the related work on software dependencies makes use of analysis of the source code. Some of it also makes use of static and dynamic analysis of binaries. Several works [86, 65] have been proposed to do software testing and analysis on Windows Server 2003 using software dependencies. Zimmermann et al. [112] use graph network analysis to predict software defect rates on Windows Server 2003. Other tools [99, 100] construct dependency graphs based on source code.

Salah et al.[77, 78] use dynamic analysis to extract views of software at different levels. The purpose of their tool is more likely suitable for developers since their use-cases are tailored for helping them locate areas to be changed. Their method relies on access to the source code since they rely on instrumentation by adding code to intercept function entry and exit points.

Our work focuses on understanding interactions of software modules, in terms of the binaries, as a whole in Windows through various dependency graphs. We deal with binaries and not with source code. We also deal with the more complex environment of Windows, e.g. kernel can call user space functions, a form of callback.

5.1.2 Visualizing Software Dependencies

As discussed in Section 2.1.4, there are many kinds of binary files in Windows. We use the term program to denote the executable, which is usually an EXE file in Windows. Unlike Unix, Windows has many kinds of binary files containing other software modules. A non-comprehensive list of the binary files other than EXE is as follows: DLL (dynamic link libraries), ocx (ActiveX controls, these are commonly used by Internet Explorer), sys (device drivers, these may be in user or kernel mode), cpl (control panel applets), acm/ax (multimedia codecs), etc. In this section, we use the term EXE to denote the single main executable, and the term DLL to denote binaries containing software other than the EXE. We consider both EXE and DLL files used to be software modules.

The module dependencies are naturally visualized using a directed graph where a directed edge from a node representing binary A to another node representing binary B indicates that there is a dependency, namely that code in A makes use of code in B. (We see later why there is the phrasing "node representing a binary" in Section 5.1.2.1). Given

this classification of binaries, there are three possible types of dependencies:

- **EXE-DLL**:
  This type of dependency represents the fact that a program uses a DLL. For example, `iexplore.exe` depends on `mshtml.dll`.

- **DLL-DLL**:
  DLLs can have dependencies among themselves. For example, one of Mozilla's UI libraries `xul.dll` depends on the SQLite database library `sqlite3.dll`.

- **EXE-EXE**:
  Programs may also depend on other programs. This arises when a process which is an instance of program P_a creates another process which is an instance of program P_b. We also visualize this with an edge from P_a to P_b (thicker blue edge). The label of an edge indicates the number of times P_a creates P_b.

In this work, we are concerned with determining and visualizing these dependencies at runtime, when the software (and there may be multiple EXEs) run. I.e. the dependencies are based on dynamic runtime behaviours instead of the static code structure. Instead of using a single visualization to show the three kinds of dependency together, we propose two visualizations which are meant to serve different purposes. The first type of visualization, called EXE dependency graph, shows the EXE-DLL and EXE-EXE dependencies. The second type of visualization, called DLL dependency graph, shows the EXE-DLL and DLL-DLL dependencies. The EXE-DLL dependencies are different between the EXE dependency graph and the DLL dependency graphs, i.e. they have different definitions and visualizations.

5.1.2.1 EXE Dependency Graph

We first define what we mean by EXE-DLL and EXE-EXE dependency in an *EXE Dependency Graph*.

Definition 1 *An EXE-DLL dependency in an EXE dependency graph is when process* a *running executable x loads binary y. We say that x has an EXE-DLL dependency on y.*

In Windows, a binary can be loaded explicitly by the `LoadLibrary()` API, or implicitly by the `IMPORTS` file header. Both of these lead to EXE-DLL dependencies.

Definition 2 *An EXE-EXE dependency in an EXE dependency graph is when process* a *running executable x creates a new process* b *which is running executable y. We say that x has an EXE-EXE dependency on y.*

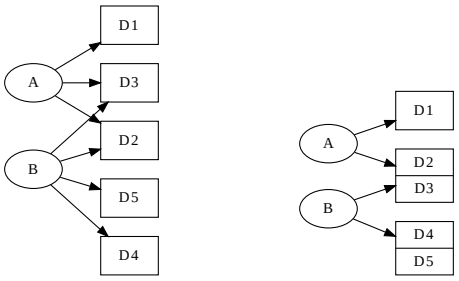


Figure 5.1: Dependency graph without (left) and with (right) grouping of programs A and B with other DLLs D1 to D5

EXE-EXE dependencies give a form of parent-child relationship. However, is not the same as the process hierarchy graph since the process hierarchy is about processes and the EXE-EXE dependency is on files. Furthermore, as we will see in the visualization, there is only one node for every program and there can be cycles.

We are mostly concerned with dependencies between different executables, hence, we omit self-loops if there is an EXE-EXE dependency between x and itself. We remark that in Windows, process creation is achieved in the WIN32 library with the `CreateProcess()` API (there is an implicit dependency that `kernel32.dll` relies on `ntdll.dll`), whereas in Unix, a combination of the `fork()` and `execve()` system calls are used.

The EXE dependency graph visualization is motivated by scenarios when we want to understand: (i) what various software modules have in common; and (ii) where they are different. Furthermore, a program may depend on a number of other programs to do its work.

The EXE dependency graph has two kinds of nodes, files which are EXEs, are denoted by ellipses, while files which are DLLs, are denoted by rectangles (Figure 5.1). EXE-EXE and EXE-DLL dependencies are represented by blue and black directed edges respectively. In general, the EXE dependency graph forms a general graph rather than a DAG since programs can mutually depend on one another (this is a generalization of a call graph to the process level).

In a direct visualization of EXE-EXE and EXE-DLL dependencies, a directed edge between node x and y denotes that there is the dependency between file x and file y, e.g. see the left graph in Figure 5.1. This straightforward direct visualization is usually too complex to be manageable as it often has too many nodes and edges simply because there are many dependencies in Windows. For example, in a fresh Windows install, the `C:/windows/system32` directory alone can contain more than a thousand DLLs; while in a Windows system with more software installed, there can be several thousand DLLs. A simple program such as `notepad.exe` loads more than 30 DLL files.

Thus, a more effective visualization is needed. We propose a compressed dependency graph with fewer nodes. This is obtained by grouping the DLL nodes according to the

EXEs which lead to that dependency, i.e. incoming edges to the binary file. More specifically, Let the set of all EXEs with a dependency on DLL x be the program set of x. We group all binaries whose program sets are the same. Figure 5.1 shows the dependency graph without grouping (left) and with grouping (right).

Since files often come from particular directories which are either system directories or program directories and there are different binary types, we further group files so that the files with the same directory and same file type are in one node, e.g. label 4 in Figure 5.2 which shows the dependencies from DLLs which match with the pathname `/windows/system32/*.dll` (note that we have used the forward '`/`' for pathnames).

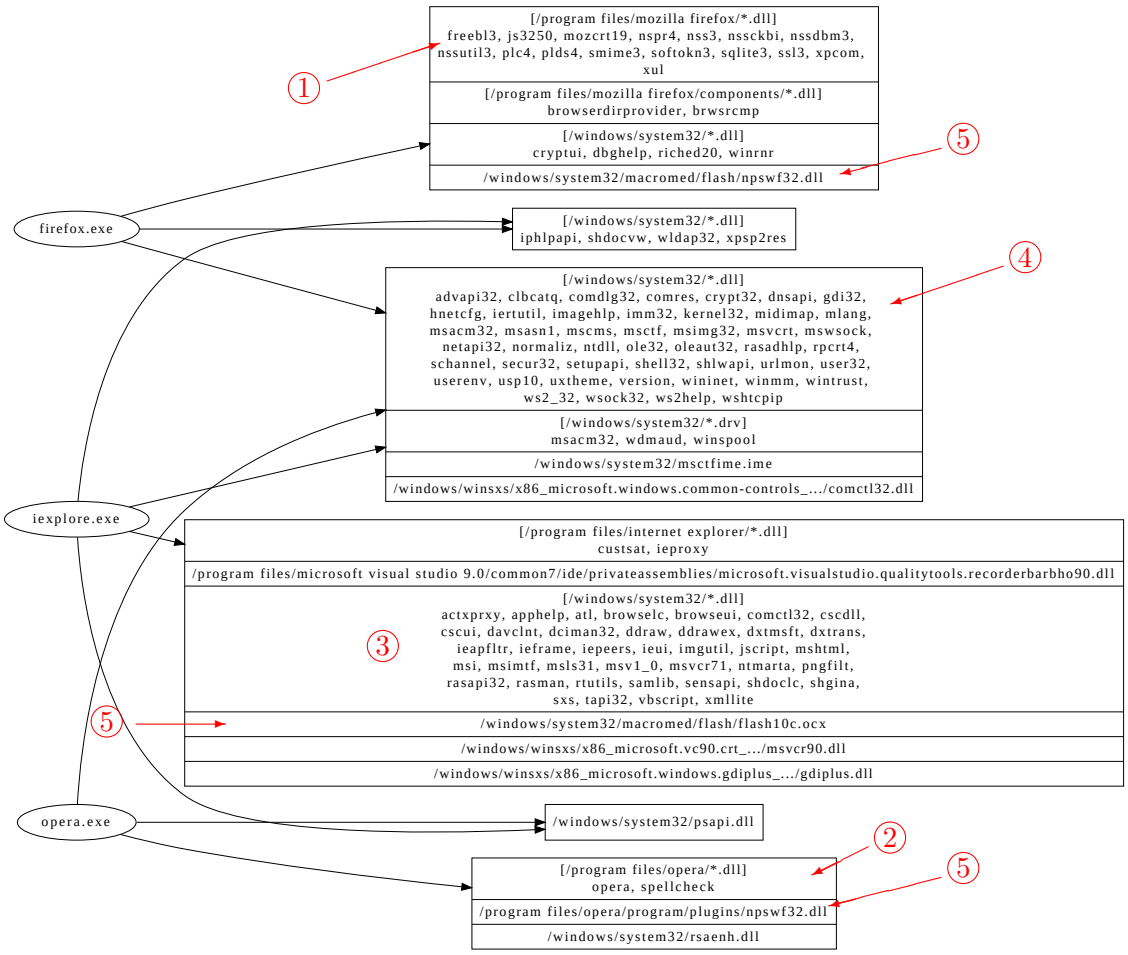


Figure 5.2: EXE dependency graph of three browsers: IE, Firefox, Opera

DLL Dependency Graph We define what we mean by EXE-DLL and DLL-DLL dependency in a *DLL Dependency Graph*. The DLL dependency graph is for a single process, so in the definitions below we can assume that process a is running executable x.

Definition 3 *An EXE-DLL dependency in a DLL Dependency Graph is when there is a control transfer from code in executable x to code in DLL y. We say that x has an EXE-DLL dependency on y.*

Definition 4 *A DLL-DLL dependency in a DLL Dependency Graph is when there is a control transfer from code in DLL x to code in DLL y. We say that x has a DLL-DLL dependency on y.*

A DLL dependency graph shows how the code in DLLs depend on each other. The motivation is that a DLL often has a certain functionality, so the graph also illustrates the use of that functionality in an abstract fashion. Furthermore, DLLs can depend on each other in a complex fashion which is shown by this graph. For example, DLL dependency graph can show how a DLL can have a hidden dependency because it relies on other DLLs for certain functionality.

The DLL dependency graph shows module dependencies during execution of a program. In the graph, the nodes are either the single EXE or DLLs. An edge from node x to node y represents a module or executable in x calling some code in module y. We also call this as x uses y. Typically this means x calls a function in y. The label of an edge (weight) represents the number of invocations, i.e. the number of calls from x to y. Note that it is possible for DLL x to use DLL y and also for y to use x during the execution. In this case, the edge between node x and y is bidirectional. In the visualization, we use a thick line without any arrow. The edge weight is the sum of number control transfers in both directions.

A direct visualization of the DLL dependency graph is often too complex, because there can be too many binaries used during an execution. For example, Figure 5.3 shows the graph for a small program `wget`. The graph has too many nodes and edges to be read. Larger programs generate even more complex graphs, thus we need a way to reduce the size of the graph. We use a simple idea that the visualization can be abstracted in two ways by grouping. One way of grouping is to group binaries which have a related function in the same node, for example, we can group Windows DLLs which deal with networking together. Another way of grouping is to think of the source of the binary, so this could be a way of grouping by software vendor. For example, we can also group binaries which are from Adobe together.

The DLL dependency graph shows actual control flow dependency. The EXE dependency graph, on the other hand, only shows what DLLs are loaded. Obviously, in a DLL dependency graph, if there is a path from executable x to the node representing DLL y (due to grouping), then there will be a direct edge in the EXE dependency graph between x and the node representing y (due of grouping). However, an edge in the EXE dependency graph does not imply that there is such a path in the DLL dependency graph, simply because a DLL might be loaded but not called. It is useful to be able to tell that

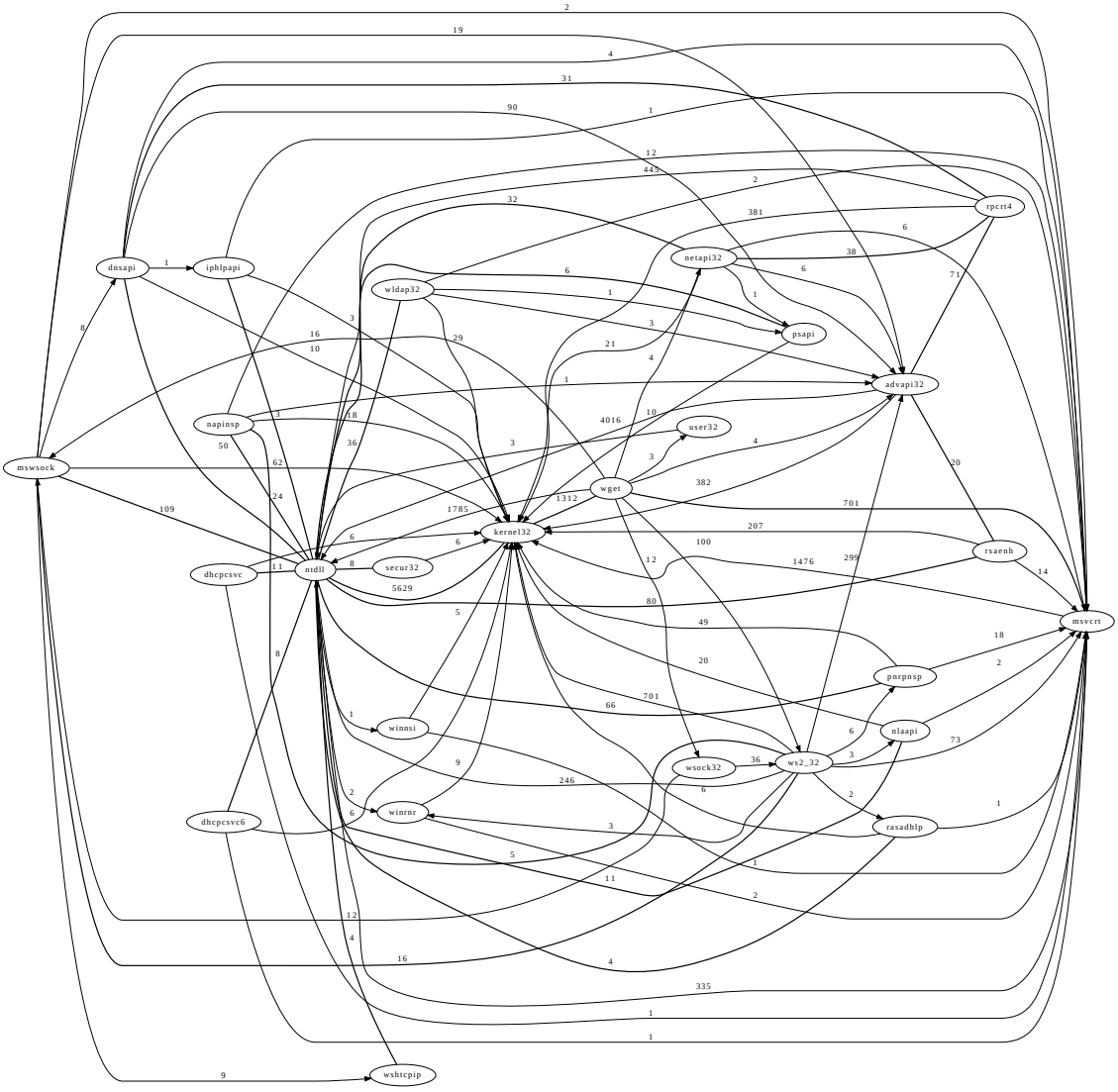


Figure 5.3: DLL dependency graph of `wget` without grouping

a DLL y is not actually called, although y is loaded. Such a DLL y is labeled with the notation *y* in the DLL dependency graph and is not connected to any other nodes.

To further simplify the visualization, DLLs can have initialization code. So control flow between EXE and DLLs due to initialization increases the complexity of the graph. We allow for the visualization to ignore dependencies which arise due to initialization which can reduce some of the edges in the graph. Note that this can lead to some nodes being disconnected from the graph, and in some cases, those nodes will be treated as nodes which are not called, i.e. labelled as *y*. Our visualization can choose to either ignore DLL initialization or to take it into account.

```
void main (void) {
   A();
   B(1);
}
void C (void) {}
void D (void) {}
```

```
void A (void) {
   B(0);
}
void B (int i) {
   if (i) D();
   else C();
}
```

Figure 5.4: Each function is in its own DLL

5.1.2.2 Other DLL Dependency Graphs

In addition to the basic DLL dependency graph, we developed two types of DLL dependency graphs obtained through *diff* and *projection* operations.

Sometimes we want to compare two executions of the same executable or module. For example, we want to identify which additional binaries are used when we browse a web page with Adobe Flash versus browsing without Flash. We can then conjecture that those additional binaries are related to the Flash component. We also want to know which additional module invocations/dependencies are incurred by Flash. To visualize this, it is useful to define the graph difference between the two graphs, which we call *diff*. The diff of graph A and B, denoted by $A - B$, is essentially subtracting graph B from A. The result is that only edges that appear in A but not in B appear in the resulting *diff* graph. A DLL x which only appears in A but not in B is annotated as $+x+$.

Sometimes we want to focus on a specific binary - or a group of binaries. For example, we want to know which binaries are used by the cryptography library `crypt32.dll`, and correspondingly, which binaries use `crypt32.dll`. We want to determine not only which binaries are directly used, but also those which are indirectly used. We propose the *projected* DLL dependency graph for this purpose. In the graph projected on a set of binaries S, only binaries that are used by S or binaries that use S either directly or indirectly are shown. Note that this is not the same as finding connected components containing S in the dependency graph. Instead, the projection looks at the control flow chain during execution and selects all the DLLs which have S in its path. Since projection removes edges from the original graph, we have the option to keep DLLs which are not in the projection but are used (this makes sense under grouping) and we label the node as $\#y\#$

The example in Figure 5.4 illustrates the difference between projection and connected components in the graph. The program has five functions which are in DLL modules of the same name. Suppose we project on `A.dll`. Only `main.exe` uses `A.dll` directly. `A.dll` uses `B.dll` (directly) and `C.dll` (indirectly). `D.dll` is used by `B.dll` but never by `A.dll` at runtime. Thus, the projection on `A.dll` does not contain `D.dll` even though it is reachable from `A.dll` in the DLL dependency graph. The resulting graph is "main→A→B→C".

5.1.3 Explaining the Visualizations

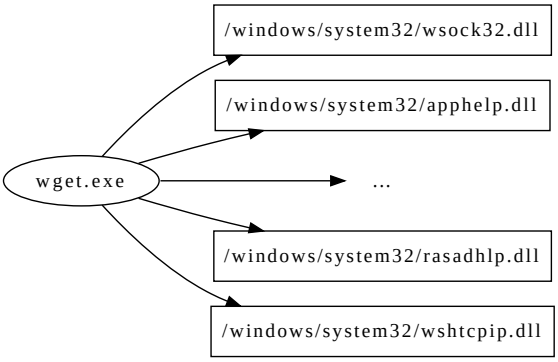


Figure 5.5: EXE dependency graph of `wget`

We illustrate the two dependency graphs on a small example, namely, the `wget` program downloading an `https` page. Figure 5.5 shows the EXE dependency graph for `wget`. `wget` loads 34 DLLs, but the figure only shows 4 of them for simplicity. The EXE dependency graph is meant for comparing multiple EXE's to show grouped dependencies. When there is only a single EXE in the graph, it reduces to a single level tree. Figure 5.7, 5.2 and 5.8 are EXE dependency graphs with multiple EXEs.

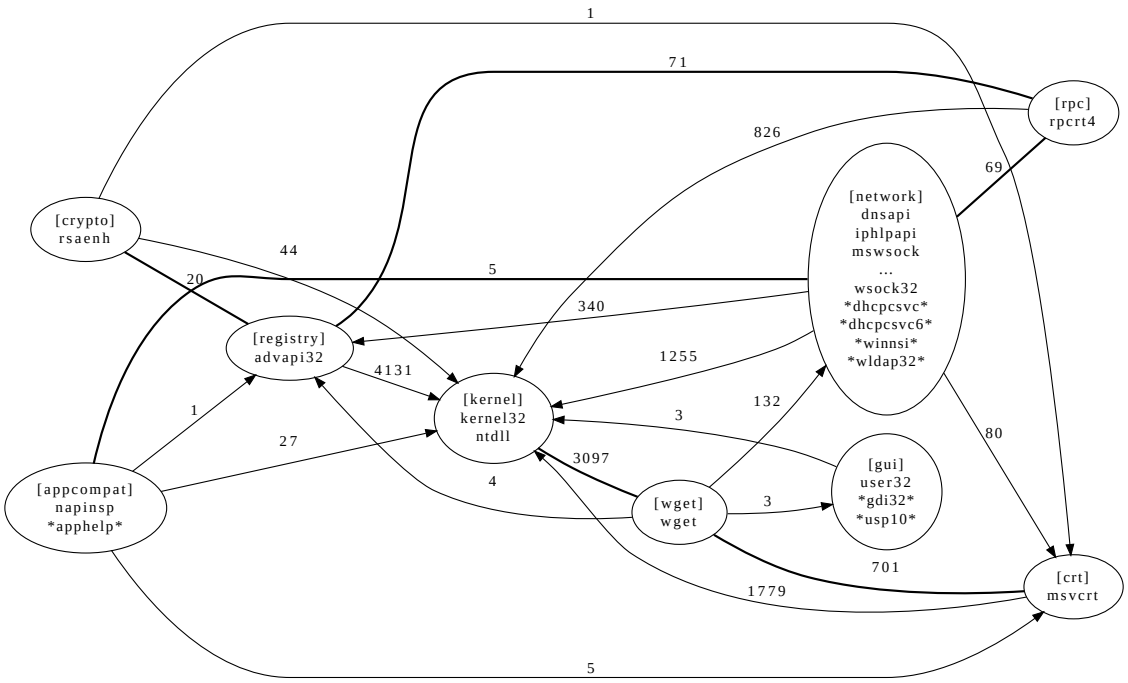


Figure 5.6: DLL dependency graph of `wget` grouped by functionality

Figure 5.3 shows the DLL dependency graph for wget without grouping; while Figure 5.6 shows it grouped by functionality. The edge weights are the number of control

transfers between DLLs or EXEs. As we can see, Figure 5.6 is much more compact because the nodes have been grouped according to the function of the modules.

We manage the assignment of files to groups with a database which is used by the visualization program. A table in the database has three fields: binary, grouping method and group name. Each binary file is assigned to a group under each grouping method. For example, the binary `user32.dll` is assigned to the `gui` functionality group. As there is no single source of information for all the binaries, we mainly searched on the Internet for information using the binary's name. We also used the DLL's exported functions and file version metadata as heuristics.

For vendor grouping, the assignment of most binaries was done automatically using the "Company" information embedded in the file version metadata. In some cases, some binaries do not have metadata, so a manual process was used. One can also have other and multiple mappings beyond the two which we used. In this work, we only used a common mapping for functionality and vendor mappings in all the dependency graphs. The grouping process as a whole is relatively static and can mostly be done once so while some manual effort is needed to create an initial database, the database may only need to be updated infrequently.

Figure 5.3 shows the interactions between the various DLLs used by `wget`. It illustrates that a small program already gives a complex dependency graph. As such, it may be more useful to check some specific details. Figure 5.6 gives a "big picture" of the dependencies in `wget` in terms of how the various functional components interact. We see that a number of DLLs are loaded but not used in our https download (files of the form **x**).

5.1.4 Implementation

Implementation of the visualizations entails collecting traces when binaries are loaded and tracking of function call and return. Binary loading information is obtained using a modified version of WinResMon [76], which makes use of a kernel driver to obtain what binaries are loaded by a process as it runs and what child processes are created. The EXE dependency graph is created from the WinResMon trace file into a dependency graph in Graphviz [35] format and rendered by `dot` in Graphviz.

The system trace from WinResMon is very robust and allows for continuous full system tracing if so desired. It can also trace a large portion of the system boot (Sec 5.1.5.1). It also works with software which employs self-modifying code and other malware techniques to defend against virtual machines, debuggers and code instrumentation such as Skype. Thus, it can handle many more cases than with a more detailed instruction instrumentation or interpretation approach, as is done with PIN below.

The DLL dependency graph is built from a runtime control flow trace from function calls and returns during execution. We employ PIN [54] which instruments the binary

to allow monitoring of the control flow during execution of call and return instructions when a process runs saving these events into a trace file. The trace is then analyzed to generate a DLL dependency graph in GraphViz format.

The DLL dependency graph only needs instrumentation of the call instructions which is straightforward. DLL projection, however, requires tracking of the control flow of both call and return instructions. The full call/return tracking is more complex than one would expect under Windows because one has to handle: non-local goto's from `setjmp()` and `longjmp()`, exception handling, kernel callbacks and multithreading. For example, a naive implementation which matches counts of call and return instructions will fail when a `longjmp()` is used since that means that some function does not return. We make use of stack and return address matching to do this tracking.

In Windows, execution can be interrupted to switch to a new context. When execution in the new context completes, execution of the original context resumes. There are a number of reasons why the context changes such as: asynchronous procedure call (APC), kernel callbacks (kernel calls a user function) and structured exception handling. Thus, context switching has to be tracked to avoid matching call and return from different contexts. We remark that the start of a context switch can be obtained in PIN but not when the context terminates. Instead, termination is observed by watching the `NtCallbackReturn` and `NtContinue` system calls. Context switching can be cascaded, thus, we maintain the unfinished contexts in a stack.

We generate a trace file to record the call/return control flow. The trace records events such as function calls/returns, thread creation/termination, context switching, binary loading and system calls. This allows us to analyze the execution trace in many ways to generate different graphs.

The DLL initialization in Windows is identified by the special DLL entry function `DllMain` which is called when a DLL is loaded. To remove the control flow effect due to DLL initialization, we remove all calls which are due to execution of `DllMain`.

5.1.5 Comprehending Module Dependencies in Real Software

We now give some usage scenarios which demonstrate how our visualizations make it easier to understand the web of dependencies of software in Windows. The dependencies shown are from programs runs in each example and show how programs and binaries used in those runs relate to one another. The figures in this chapter are all scalable and thus can be magnified in the PDF to see more detail and we encourage readers to do so. In this section, the files in some nodes in the graphs are abbreviated to (...) to make it easier to fit the graphs in the thesis. We use red circled numbers with arrows to point to certain parts in the graphs so that we can refer and discuss them in the thesis. These annotations are not part of the visualization. All the DLL dependency graphs remove dependencies

due to DLL initialization. One could choose also to see DLL initialization dependencies, as removal would not make sense for DLLs which do significant work in the initialization.

5.1.5.1 System Boot

Figure 5.7 shows the EXE dependency graph for a boot-to-shutdown cycle for a fresh Windows XP install. The machine was booted and automatically logged in, then logged off, and shutdown. One might expect that the processes form a process tree but that is not the case because some of the processes (e.g svchost, winlogon, services) start before our monitoring tool (WinResMon) does. Looking at the big picture of the startup and shutdown process, it can be seen that some DLLs are shared by many programs. For example, `kernel32` and `gdi32` (highlighted by ⑥) are used by almost all programs. This is expected since some system calls would use `kernel32` while GUI programs would use `gdi32`. On the other hand, some DLLs are used exclusively by one program (e.g. `odbc32` ⑧ is used only by `explorer` ⑦). This is somewhat surprising leading to the question why `odbc32`, the database query interface, is used by `explorer`.

5.1.5.2 Comparing Web Browsers

Figure 5.2 shows the EXE dependency graph for three browsers: Internet Explorer, Firefox and Opera. Each browser displays a YouTube page to play a video till completion. The scenario is intended to show the dependencies of the binaries used by the three programs with similar function but different source code base. The visualization shows the commonalities and differences between the software modules in the three browsers. It also shows dependencies on other software which is not part of the browser code base and which comes in through the use of browser extensions or plugins, e.g. Adobe Flash.

This example shows the compressed EXE dependency graph with grouping. We see that grouping makes the graph easier to visualize. Firstly, the grouped version is much smaller in size. Secondly, it is easier to identify which binaries are shared by the same programs. Some observations are discussed below:

- ①, ② and ③ show the DLLs used exclusively by Firefox, Opera and Internet Explorer respectively. Opera does not have many of its own DLLs. On the other hand, Firefox uses its own DLLs (in `/program files/mozilla firefox`) for many features. In the case of Internet Explorer, the ownership of the DLLs are less clear since most of the DLLs used come from the `system32` directory. This may be due to the deeper integration of Internet Explorer with Windows.

- ④ shows the common binaries shared by all three browsers. It is not surprising that a large number of binaries are shared. This is because most of the major Win32

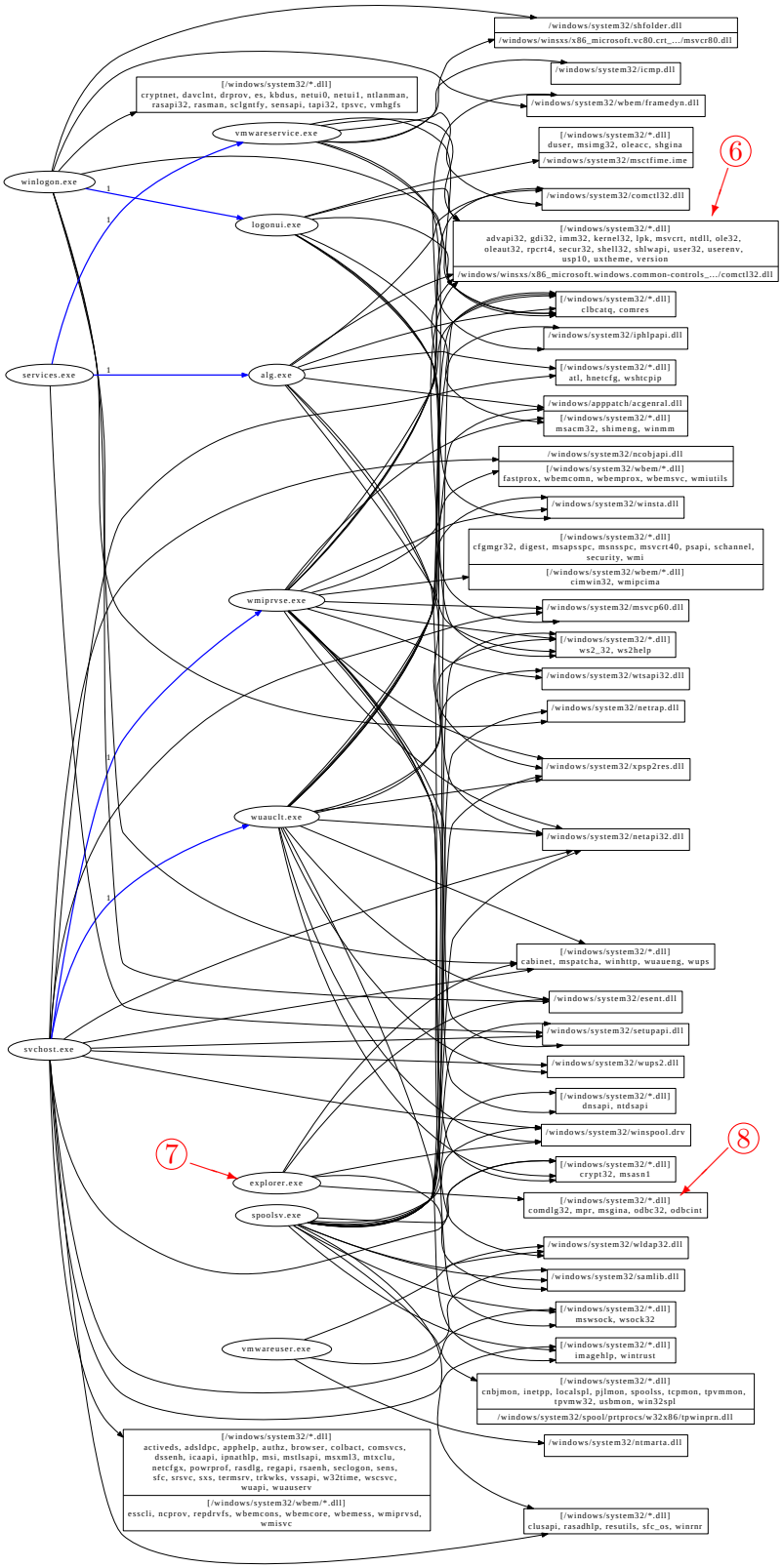


Figure 5.7: EXE dependency graph of the whole system

API calls are implemented in those binaries, e.g. `msacm32.drv` and `wdmaud.drv`, are the audio drivers that all browsers use to play the video.

- (5) shows the different ways that each browser runs Flash. In Internet Explorer, Flash is installed as an ActiveX control `flash10c.ocx`. Firefox and Opera both use `npswf32.dll`, the Flash plugin for Netscape. We discover that the same DLL is duplicated in different directories.

5.1.5.3 Word Processor

Figure 5.8 shows the software dependency graph for Microsoft Word (Office 2003) and OpenOffice Writer (OpenOffice 2.4). In this test, we opened a `.DOC` file, i.e. a Microsoft Word document with both programs. We describe some of the observations from Figure 5.8:

- (9) shows the main OpenOffice program `swriter.exe`. As OpenOffice Writer is composed of 4 individual programs it exhibits more complex dependencies than Word which is only a single program `winword.exe`.
- (10) shows the common binaries between Word and Writer. As expected, most of the common binaries are in the `system32` directory as most of the Win32 APIs are implemented in those common binaries. We can observe that both Word and Writer use `version.dll`. This DLL contains the APIs for checking the version number of a binary. Possibly both programs use this DLL to check the operating system and binary versions and then decide to enable or disable certain features.
- (11) and (12) show all the binaries used by Word and Writer respectively. It can be seen that Writer uses many of its own DLLs.
- (13) shows that there are two versions of `comctl32.dll` (these are with different pathnames). This is possibly related with the use of `version.dll` mentioned above. Windows probably uses a different version of the same file depending on the information provided by `version.dll`.
- (14) shows the Windows application compatibility DLLs being loaded. This is possibly because Windows is checking whether there are any special compatibility flags for Writer. Some examples of such flags are running a program under an older version of Windows and running with a lower display resolution.

5.1.5.4 Gimp

GIMP is an open source image manipulation program. Like many open source software, it uses libraries from various sources. In this experiment, we executed GIMP without

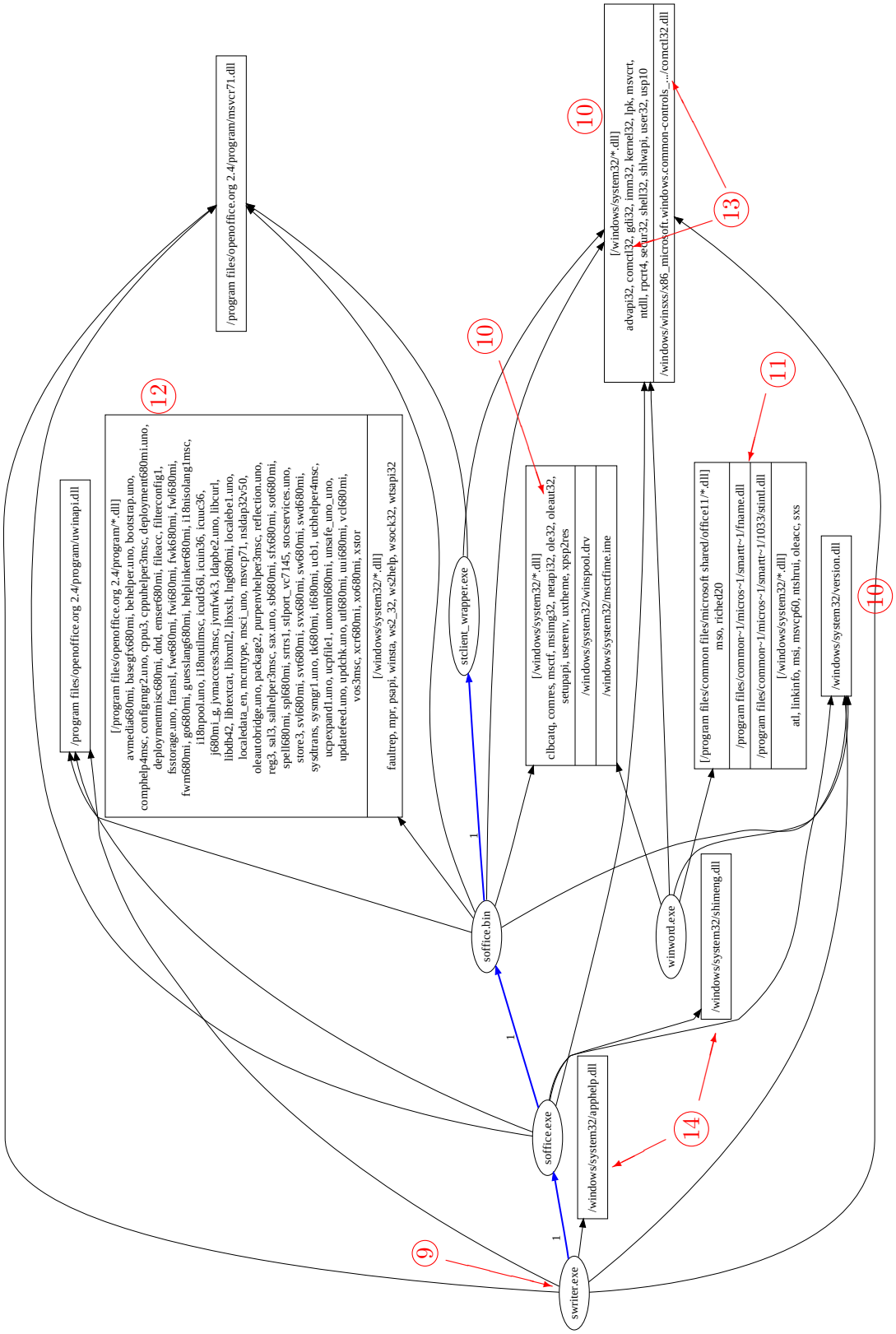


Figure 5.8: Software dependency graph of Microsoft Word and OpenOffice Writer

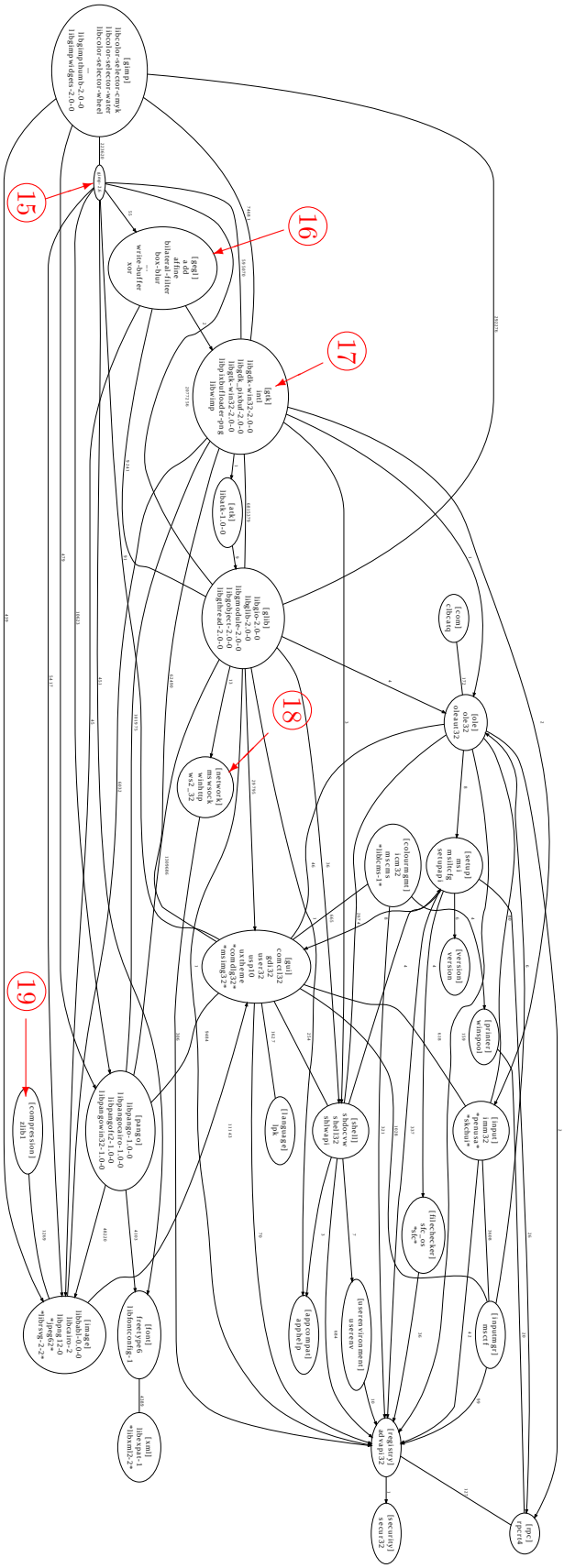


Figure 5.9: DLL dependency graph of Gimp grouped by functionality

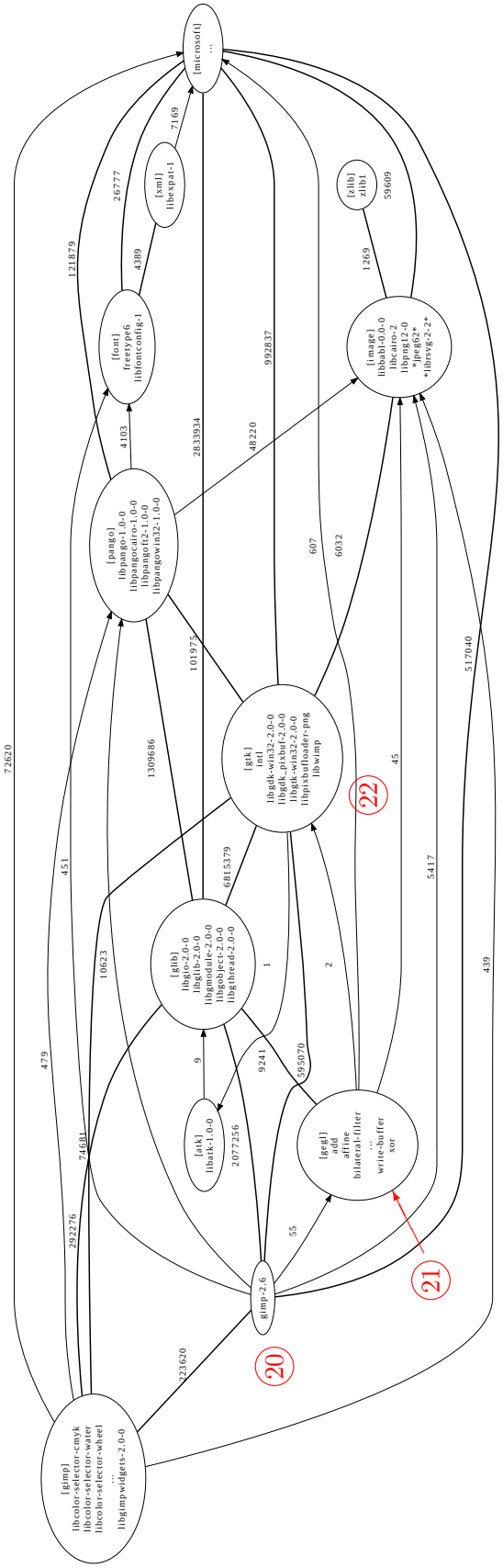


Figure 5.10: DLL dependency graph of Gimp grouped by software vendor

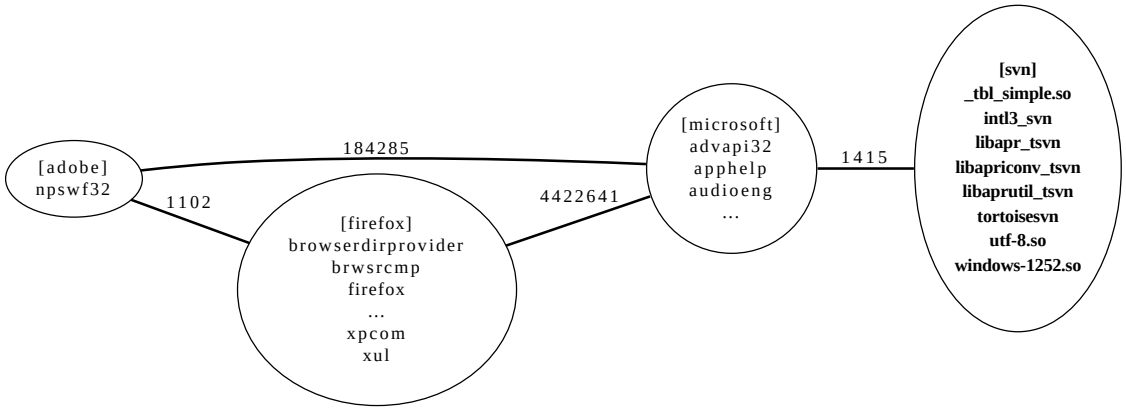


Figure 5.11: DLL dependency graph of Firefox grouped by software vendor

opening any files. We want to see how GIMP uses various libraries and the interactions between them.

Figure 5.9 shows GIMP grouped by functionality while Figure 5.10 shows GIMP grouped by software vendor. GIMP is large but quite modular. The bulk of the DLLs used are from the GEGL, GTK and GLIB frameworks. It also uses the Windows GUI APIs (e.g., gdi32, comctl32) extensively.

Figure 5.9 is larger than Figure 5.10 but is still understandable using the functional grouping, e.g. `gimp` (15 and 20) uses the `[gegl]` (16 and 21) image processing modules, which in turn uses the `[gtk]` (17 and 22) windowing module. The remaining DLLs are not used very much.

Figure 5.10 is interesting because it shows that many different software vendors/providers are used in GIMP. Microsoft is also prominent but is expected since GIMP is running on Windows and uses the Win32 API. The presence of so many different software vendors could also make troubleshooting difficult since it might be difficult to determine which vendor is responsible if GIMP behaves in unexpected ways. Figure 5.10 also shows one of the advantages of open source software, namely, reuse. The binaries belonging to GIMP only turn out to be quite small and GIMP prefers to rely on existing frameworks like `gtk` (GUI) and GEGL (image processing). From Figure 5.9, we can see that the network module (18) is only used by the `[glib]` library directly and the compression module (19) is only used by the image modules directly. This observation may simplify some software debugging and maintenance scenarios, i.e. reducing the set of modules to be examined.

5.1.5.5 Firefox: Plug-ins and a Surprise

Firefox allows extensions through its own plug-in architecture. Figure 5.11 shows the DLL dependency graph of Firefox as grouped by vendor. We find a node for Adobe. This

is because `npswf32` is the Flash plugin for firefox.

We discover a surprising node - the `svn` vendor group (shown in bold font) which is to do with the TortoiseSvn interface to SubVersion (the version control system). TortoiseSvn is unrelated to either Firefox plugins or addons. Thus, one might find its presence strange, when Firefox is being executed. The explanation lies in the way the Tortoise shell extension behaves in Windows. TortoiseSvn injects its DLLs into the `Explorer` shell process when Windows starts. When explorer runs a process, the TortoiseSvn DLLs are also loaded into the new process. The fact that there is interaction with a module which is not part of the Firefox can mean that a bug in that module can also affect Firefox. This can be concern for software developers.

5.1.5.6 Adobe Flash in Internet Explorer

This scenario is to understand the interaction of Adobe Flash in the Internet Explorer browser. The strategy used here is to compare two DLL dependency graphs, Internet Explorer opening a website with flash content and Internet Explorer showing the `about:blank` page since `about:blank` page is the most minimal webpage for Internet Explorer, Figure 5.12 shows the difference (diff) between these two graphs. The DLLs marked with **+x+** are the *extra* DLLs loaded when Flash is played. Most of these extra DLLs fall into the network (23), multimedia (24), DirectX (25) and Flash (26) categories.

To further investigate the interactions resulting from the Flash module, we project the visualization on `flash10c.ocx` which is the Flash ActiveX control. Figure 5.13 shows the projected graph. We can see that Flash (27) is interacting with the browser and multimedia modules. The DLLs marked with **#x#** neither use `flash10c.ocx` nor are they used by `flash10c.ocx`. In the `network` (28) group, we can see that `wsock32` is marked by **#x#** but `ws2_32` is not. Windows has many network modules as shown in the `network` group. `Wsock32` is an older version of the winsock modules whereas `ws2_32` is the newer version. This means the Flash player only uses the latest version of winsock rather than the older version which is closer to the Unix socket API. However, the older version is still used by some other modules because otherwise `wsock32` would have been labelled as ***x***. The EXE dependency graph of the three browsers also shows that both winsock modules are loaded by all three browsers. The difference with projection is that projection is more precise and shows the interaction between the Flash module with the particular network module.

5.1.6 Conclusion

The two visualizations EXE dependency and DLL dependency graphs are just two ways to visualize software module dependency using execution traces. We can have other different visualizations using the traces. For example, instead of using binaries as basic

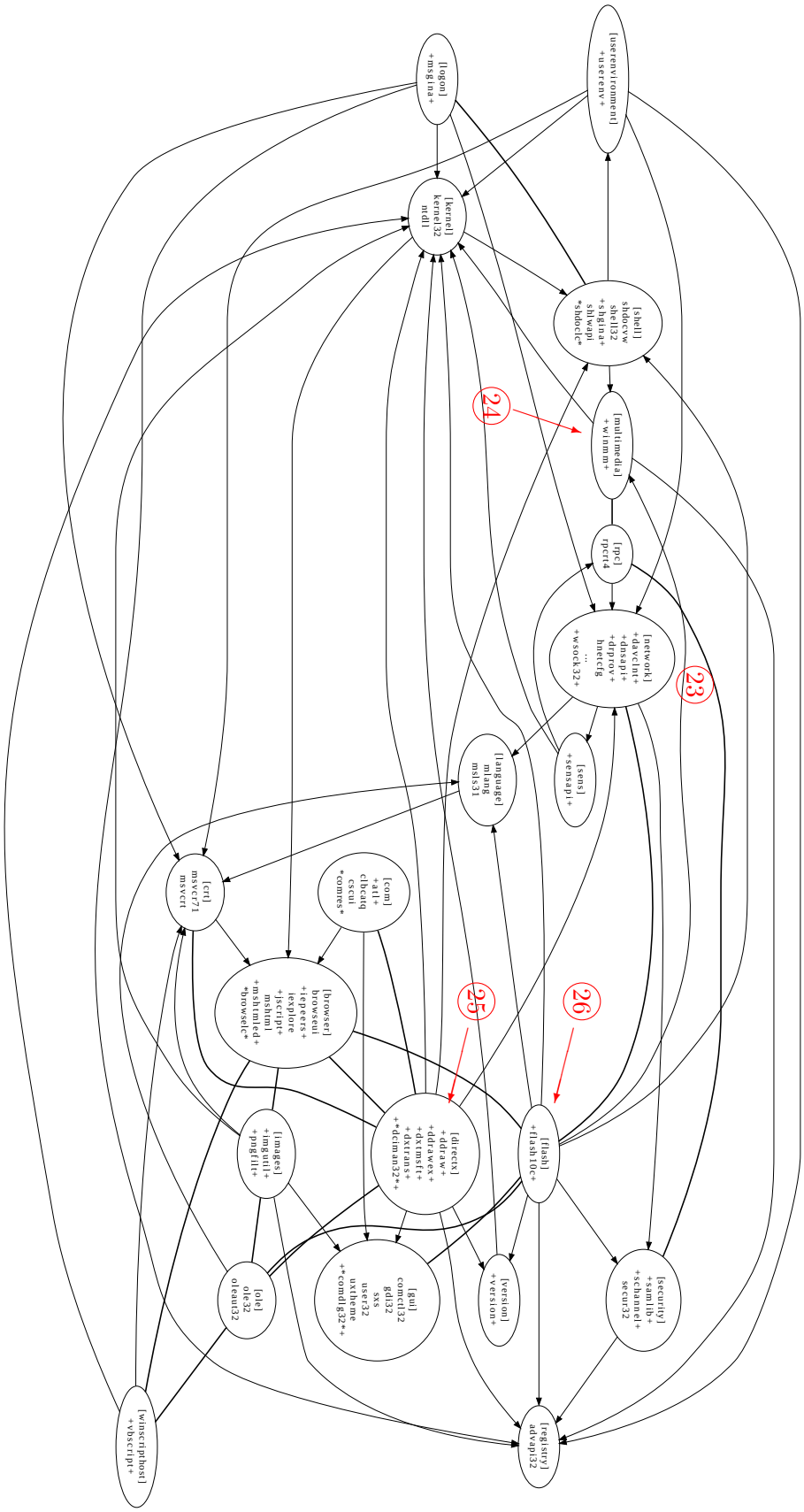


Figure 5.12: Diff of DLL dependency graph of Internet Explorer with Flash and without

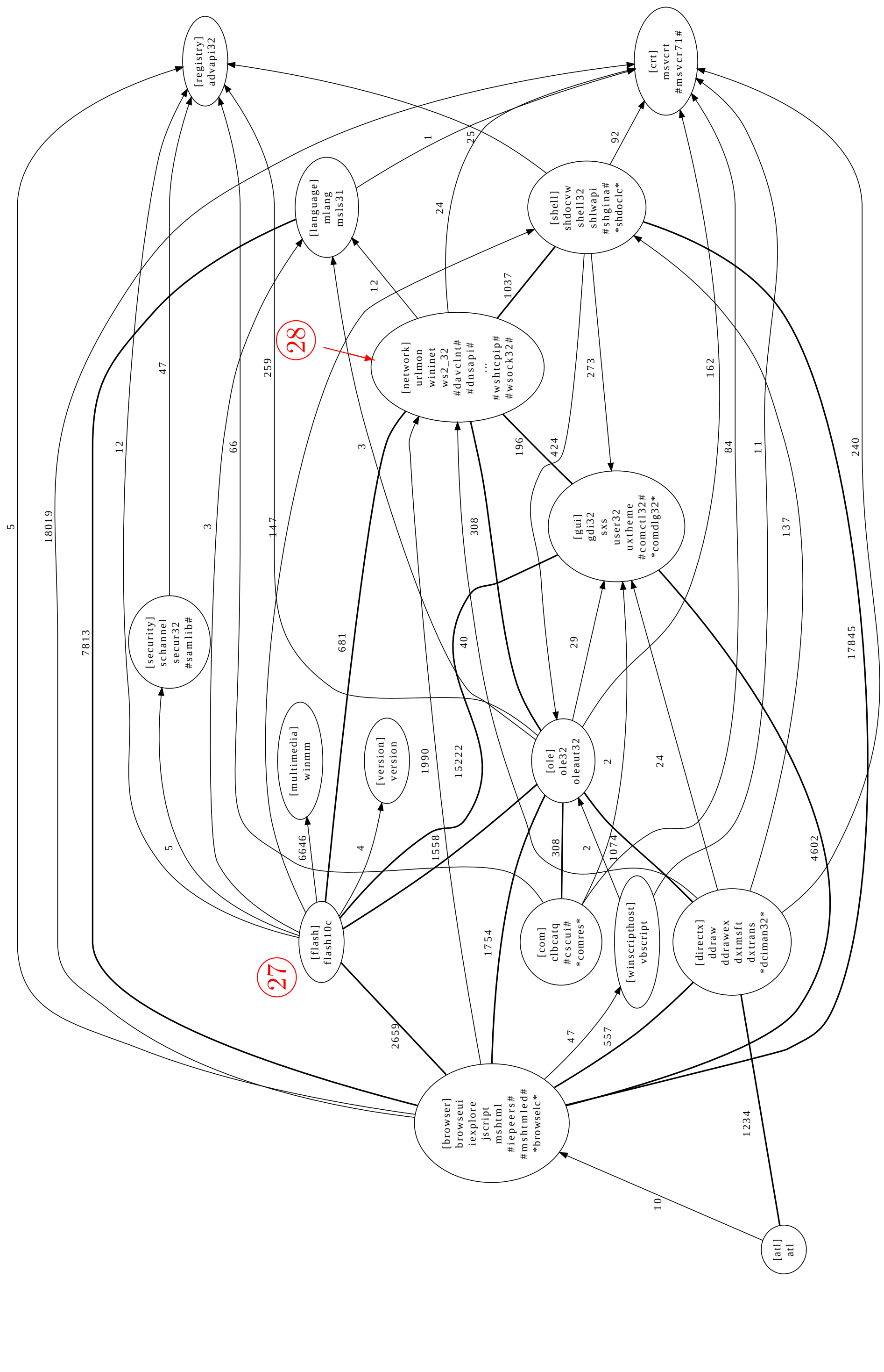


Figure 5.13: Projection of the DLL dependency graph of Internet Explorer on Flash

units, we can use functions to visualize more fine grain behaviour. This makes the graph much more complex because there are many more functions than binaries. We can use similarity grouping, diff and projection to "zoom in" on specific regions. Because we utilize instrumentation of binaries, we can also visualize other runtime properties of the code. For example, visualize the CPU time used by different modules, and scale the nodes in the graph so that the area of each node is proportional to the time (or instructions) spent in the binaries represented by the node.

Although we have focused on visualizations, the traces which are collected contain a wealth of information. A workflow for software comprehension can also make use of the trace data in addition to the visualization. The trace data is at a fine grain level but is hard to understand. Thus, the visualization gives a roadmap for investigating the traces.

We have demonstrated that software dependencies in Windows are quite complex even when looking at the coarse grain level of software components packaged into binaries and executables. Our tools give an effective way of extracting and visualizing the software dependencies and interactions between binaries. We show that even with the complexity of actual Windows software, it is possible to analyse whole system interaction, understand how modules are used and shared, and also discover potential unexpected or unusual interactions/modules. Such an understanding is also useful for software developers, system administrators and also users, to manage the software ecosystem on Windows and to deal with the problems which arise from module updates and potential "DLL hell" repercussions.

Module dependency visualization is not limited to Windows, as dynamically linked libraries are common in many operating systems. The idea can also be extended to interpreted languages such as Java to study the dependencies of classes.

5.2 Visualizing Windows System Traces

Software is increasingly complex with many interactions with other software and the operating system. The previous visualization shows the complexity in the aspect of module dependency. Other factors include persistent state interaction, inter-process communication, networking, etc. This complicates understanding software/component behavior and diagnosing problems or performance. As discussed in Section 2.1, the close source nature of Microsoft Windows (or simply, Windows) exemplifies this trend. Suppose our task is to understand/diagnose/debug some software on Windows. Ideally, system/software documentation or source code analysis would give the answer. In practice, however, this may not be workable either due to lack of source or overall system complexity being too high. An orthogonal approach to address this issue is by examining detailed system behavior. Recently, there are a number of monitoring systems which give detailed system traces such as DTrace [26] for Solaris, DProbes [63, 22] for Linux and Flight Data Recorder [91] and WinResMon (Section 3.2) for Windows. However, as shown earlier in this chapter, the system traces can be very large and also difficult to analyze.

We propose `lviz`, a visualization tool for understanding such large and complex system traces. `lviz` has a number of visualizations. In the thesis, we focus on the visualization which we call VDP. It consists of a number of inter-connected sub-visualizations: an extended DotPlot; two Axis Histograms; and one or more barcodes. The VDP can be flexibly configured for different purpose. In Section 5.2.2.2, we show the design and configurations of VDP. In Section 5.2.3, we show that the prototype of `lviz` is efficient and can handle large system traces with real-time response. For example, a trace of size 120MB with 558K events is loaded in under 3 seconds and after that it can be interactively displayed/zoomed in around 0.5 seconds. The VDP can be used on different traces and for different purpose by using different configurations. In Section 5.2.4, we use some case studies to show that VDP can be used for solving performance problems, program failure diagnosis and finding execution patterns/anomalies.

5.2.1 Related Work

SeeSoft [34] visualizes general log files by zooming out lines of text logs and coloring them by message types. SeeLog [33] visualizes Unix accounting trace files by plotting events by time and process. SeeLog can be viewed as a restricted variant of the VDP visualization. Magpie [20] is a performance analysis tool designed for web servers. It uses collected system events including file I/O, computation, thread scheduling and network to analyze the resources usage of HTTP requests. BootVis [1] is a visualization tool to tune Windows booting plotting resource use versus time.

Bodic et al. [24] propose another visualization tool for web servers. It shows frequency of different HTTP requests and highlight anomalies that may indicate site failures. VDP

on the other hand is a general trace visualizer but it can be used for similar diagnosis (see Section 5.2.4.5). TuningFork [19] is a trace analysis tool for real-time systems including visualizing unsatisfied real time constraints in real-time systems.

EVolve [95] is a visualization framework for understanding Java program by visualization execution traces from the JVM and has a DotPlot visualization. However, VDP uses an extended DotPlot (see Section 5.2.2.2) and generates different visualizations (see Figure 5.23 on page 134) depending on its configuration. `lviz` is designed to scale for large traces and to be efficient to meet user interaction needs. Voigt et al. [92] propose a visualization method which correlates class method invocation, object invocation and time. To correlate methods and objects, it places objects and methods on each axis and draw dots to represent the method-access-object relationship. Cornelissen et al. [30] propose two visualizations, massive sequence and circular bundle views, to examine execution traces. The massive sequence view visualizes a trace by placing each event as rectangles according to its invoking software module (X-axis) and time (Y-axis). The circular bundle view visualizes software module interactions by arranging modules into a tree and draw two-module interaction as a curve along the path in the tree. These visualizations are different as they do not compare events nor work with closed source software.

5.2.2 System and Visualization Design

We describe the design of `lviz`, a system for visualizing Windows system traces. We emphasize that the visualization proposed is not specific to Windows as traces from other operating systems can be used as well. `lviz` can be easily adapted to read traces in other format, which make it friendly to other monitoring systems such as DTrace and SystemTap.

5.2.2.1 System Traces

Our objective is to visualize detailed system traces. In the rest of the section, we will shorten system trace to simply trace. Our traces are obtained with WinResMon (Section 3.2) which captures all operations performed by the Windows kernel from all processes.

A WinResMon trace consists of four classes of events due to operations on processes, files, network and the registry. An event contains the following properties: (the example events in Figure 5.14 use the ordering below)

- **PID/TID** is the ID of the process/thread which performs the operation.
- **Program name** is the pathname of the program binary.
- **Group** and **User name** of the process.
- **Start and end times** of the event.
- **Operation type** is specific to the event. Some examples are given below.

```
1268 1732 "C:\WINDOWS\System32\svchost.exe" "NT AUTHORITY\SYSTEM"
1098307213586 1098307319878 file_create "\Device\HarddiskVolume1\WINDOWS\
Prefetch\NET1.EXE-029B9DB4.pf" access=0x120089, share=0x0, attr=0x0, cd=0x1,
co=0x60, si=0x1 STATUS_SUCCESS
---
204 960 "C:\WINDOWS\system32\cmd.exe" "GROUP\USER" 1101551207684
1101551216008 reg_queryvalue "\Registry\Machine\Software\Policies\Microsoft
\Windows\Safer\CodeIdentifiers" handle=0x78, name="LogFileName", t=0x0, l=0
STATUS_OBJECT_NAME_NOT_FOUND
```

Figure 5.14: Two examples of events

- **Operation parameters**: The semantics depends on the operation type, e.g. resource pathname, file access flags, etc.
- **Return value** from the operation is either success or an error number explaining why the operation failed.
- **Call stack trace** of the process which generated the event. In this work, we use the deepest function call from the corresponding executable, i.e. excluding functions in libraries, but one could also use the full stack trace. This information is useful to associate a program point in the application code with the event.

Examples of events are file operations, e.g. file_create, file_close and file_read; registry operations, e.g. reg_setvalue; process operations, e.g. thread_create; and network operations, e.g. socket_bind. Pathnames of files and registry keys are recorded as operation parameters. Additional information depending on the specific operation is also recorded. For example, file_create, which represents both opening an existing file and creating a new file, the operation parameters include the pathname of the opened or created file, access flags, file attributes, creation disposition (create new, open existing, etc.) and other Windows operating system specific information. Figure 5.14 shows two examples of events. WinResMon can optionally capture the data associated with an event. (see Section 5.2.4.5 which uses network packet data)

5.2.2.2 Trace Visualization

Visualizations for traces need to be able to compress a lot of detail and yet be usable for illustrating many different kinds of system behavior ranging from understanding system administration to debugging to performance evaluation. In order to meet such a wide range of needs, we will see that the visualization should be adaptable and configurable to show specific aspects for a given purpose. It must also be able to scale at different detail levels.

Our visualizations use color for greater visual bandwidth as well as customization flexibility. We support both RGB and CMY color modes. The RGB color mode displays events on a black background and is appropriate if the visualization is displayed on monitors to make it easy to see events. For printed output, a CMY mode, which uses

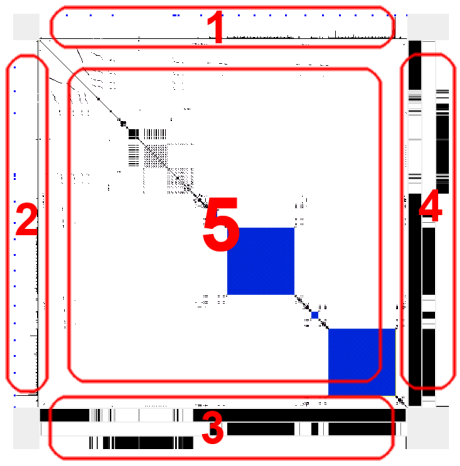


Figure 5.15: Elements of VDP: axis histograms (Region 1,2); barcodes (3,4); and extended DotPlot (5). This figure is same as Figure 5.16 with the added annotation.

subtractive color, tends to be easier to see. In the thesis, being in print format, we use the CMY mode and try to minimize the use of yellow due to its closeness to white.

A *binary DotPlot* is a black & white image where each axis denotes a sequence of events. The color of a dot (or pixel) at position (i,j) is black when the two events at position i and j from the corresponding sequence match, and white otherwise. The rationale for a binary DotPlot is to visualize event similarity between two sequences. It has been used for comparing DNA sequences [55], visualizing the structure of music [39] and investigating data formats and binaries [46]. DotPlots can be used to do self-comparison by comparing a sequence with itself or to compare two different sequences.

Here, we develop a novel DotPlot based visualization which is extensible and can be configured for many uses. Our visualization, which we call *VDP*, consists of a number of sub-visualizations: an *extended DotPlot* (or simply, DP in the rest of the section), two Axis Histograms, and some barcodes. Figure 5.15 gives an overview of all the elements: (i) the DP (region 5); (ii) the x-axis histogram (region 1) and y-axis histogram (region 2); and (iii) the x-axis barcodes (region 3) and y-axis barcodes (region 4). A larger diagram of the same visualization highlighting various details is in Figure 5.16.

An important aspect of the VDP is that it is intended to be extensively customizable. Thus, it gives rise to a family of visualizations. In Section 5.2.4, we show how to apply and customize the VDP for different purposes. The elements of a VDP as follows.

DP: Our DP visualization extends a binary DotPlot in a number of ways. Since a trace can be long, several events are aggregated together into a single pixel in a window. DotPlots which contains aggregated events are drawn as grayscale images which are then colored using the DP coloring rules for customizing the appearance. Normalization or histogram equalization (see Section 5.2.3) is applied on the image so that the color in-

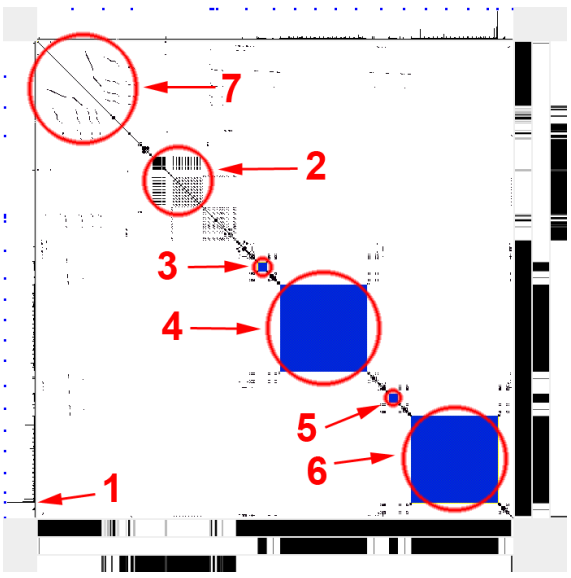


Figure 5.16: Self-comparison event-ordered VDP of `xcopy` copying 8 files of different sizes with the following configuration rules:

DP match :	operation + parameter (pathname)
DP color :	magenta → source; cyan → destination; black → other
Bar1 color :	black → file operation
Bar2 color :	black → source/destination files
Bar3 color :	black → registry operation

tensity fits in a displayable range. Our extended DotPlots (see region 5 in the center of Figure 5.15) can be drawn with event-ordering, time-ordering or time-duration-ordering. In an *event-ordered* DP, the events are plotted in sequence as they occur in the trace. In a *time-ordered* DP, the events are plotted in time order according to the start time of the event. Furthermore, each pixel in the visualization can contain several events as there may be many events occurring close to each other. Time-ordering differs from event-ordering in that it can be sparse and allows empty gap if there are no events in a particular time interval. Furthermore, in event-ordering, the number of events in a pixel is constant, whereas in time-ordering, some pixels do not contain any event while other pixels may have many events. In a time-duration-ordered DotPlot, the events are plotted in a similar way to time-ordering except that the events will be plotted as a rectangular region with dimensions corresponding to its duration obtained from the corresponding events on each axis. Time-ordering, on the other hand, plots an event as a single dot.

DP Matching Rules: Unlike a binary DP, our extended DP can look quite different depending on how matching and coloring rules are specified. The DP matching rule specifies whether two events match. The specification allows for matching on the properties of an event, e.g. PID/TID, program name, operation type, etc. (see Section 5.2.2.1)

It essentially allows any combination of information about the event in the trace. For example, matching based on *program name + operation type + resource name* will give a different visualization from just matching on resource name alone.

DP Coloring Rules: The DP matching rule is used for defining how a DP matches an event. Given an event which matches the DP matching rule, the DP color rule specifies what color to display for that event. There can be multiple DP color rules, we use the first matching rule and there is also a default color if no earlier rules match. A binary DotPlot, on the other hand, is only in monochrome.

The visualization can be further customized by specifying the colors so as to combine multiple colors taking into account either RGB or CMY color modes. This gives flexibility to emphasize or hide any kind of information in the trace. For example, in the RGB mode, the DP color rules can specify that events which are file operations are colored red, registry operations to be blue, and network operations to be green. Note that the DP color rule matches independently of the DP matching rule, e.g. the DP match can be on operation type while the DP color rule could be on program name.

When combining colors, it is easier to use a RGB mode which is more intuitive. However, in the thesis, CMY is used on all figures. One way is to use CMY as 3 orthogonal dimensions which would allow 3 different properties to be visualized. For example, a cyan pixel in a DP could mean the events all have property A and an adjacent magenta pixel would be a different property B. Intersection or union of properties, e.g. property A and B, allows cyan and magenta to combine giving blue. Another way is to simply color properties using custom colors. Our examples mainly use the first method.

Barcodes and **Barcode Coloring Rules**: The barcode sub-visualization acts as a one-dimensional ruler for the trace on the associated axis, hence its name. It visualizes characteristics of events in a single trace. Region 3 and 4 in Figure 5.15 show X and Y axis barcodes. The purpose of the barcodes is to visualize multiple aspects of the trace and allow that to be correlated to the associated event in the DP.

Barcodes are colored using barcode coloring rules similar to DP coloring rules. For example Figure 5.16 (see Section 5.2.4.1) colors the barcode by resource type. Multiple barcode coloring rules can be specified. Since barcodes are useful to show properties of the events, there can be more than one barcode. Each barcode has its own set of coloring rules, (see Figure 5.16) which has 3 different barcodes stacked on the X and Y axis. In the thesis, we number the barcode from inside (top or left depending on the axis) to outside, i.e. barcode 1 means the inner most barcode.

Axis Histograms: The axis histogram sub-visualization serves as another kind of ruler for each of the traces on the X and Y axis, see region 1 and 2 in Figure 5.15, providing time-event correlation information to the traces. It consists of ticks (small blue square on the outer edge) representing unit time intervals and an associated histogram. Ticks are evenly distributed in a time-ordered or time-duration-ordered VDP but can be

irregularly distributed in an event-ordered VDP, since there may be many or few events within a unit time interval. Closely spaced ticks in an event-ordered VDP mean low event frequency.

The histogram component (see region 1 in Figure 5.16) is used to visualize the density of events or time of the trace. The histogram in a time-ordered or time-duration VDP represents the number of aggregated events. In an event-ordered VDP, it represents the total time taken (recall that an event has start and end time) on the aggregated events.

5.2.2.3 A VDP Example

Figure 5.16 illustrates the event-ordered VDP of a trace of running `xcopy` to copy two directories with files of different sizes. The VDP uses self-comparison of the `xcopy` trace with the DP matching rule on operation and parameter (resource pathname). The VDP configuration consisting of a DP ordering, DP matching rule, DP coloring rules and barcode coloring rules is given in the caption of the figure. The non-white regions inside the DP show when `xcopy` is using the same resource with the same operation and parameter. Its color represents the resource. Each blue square is the aggregated visualization effect from copying a file (see Section 5.2.4.1). The file sizes are apparent from the sizes of the small (region 3 and 5) and big (region 4 and 6) blue squares. We can see that file operations are performed at the start (barcode 1), but those files are neither source nor destination files (barcode 2). Registry operations are performed relatively quickly compared to file operations, because the blue ticks in the histogram above region 2 are spaced further apart than the blue ticks above region 4 and 6. Further discussion on this example is in Section 5.2.4.1.

The DP user interface allows exploration of the trace by zooming in This allows us to see quite different detail from an aggregated view. We can see the effects of visualizations at different scales, Figure 5.16 shows the aggregate behavior which is a different picture from the magnified view in Figure 5.17 which shows the individual file operations in `xcopy`. Zooming creates a new window as one may want to see many different detail levels at the same time. Clicking inside the DP selects a pixel to give more details of the events aggregated in that pixel. When the mode is event-ordering, the events displayed are those events contained in that pixel. Whereas with time-ordering or time-duration-ordering, the events are those events which intersect in time with the time interval represented by that pixel.

5.2.3 VDP Implementation and Scalability

There are two important challenges in implementing VDP in `lviz`. Firstly, traces are usually much larger than the size of the screen. The challenge is to be able to still have usable visualizations given that there will not be sufficient pixels to show every event.

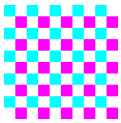


Figure 5.17: The alternate zoomed-in view of a blue region in Figure 5.16 showing reading (magenta) and writing (cyan) operations.

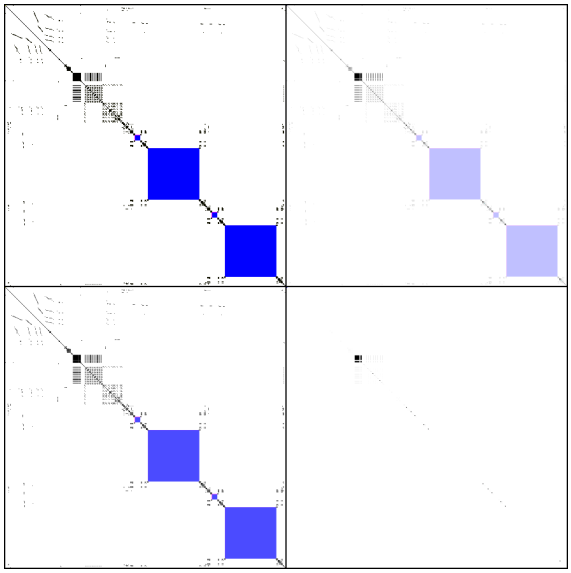


Figure 5.18: Clockwise from top-left: histogram equalization, $\gamma = 1$, $\gamma = 1/4$ and $\gamma = 4$.

Secondly, rendering a VDP on large traces needs to be sufficiently fast to meet interactive UI needs (i.e sub-second response). Finally, our implementation takes advantage of parallelism given the trend in multi-core processors. In this section, we describe an effective and efficient implementation which meets these challenges.

Consider a trace with N events and a display window size with width W and height H. For large traces, e.g. $N = 10^6$, clearly $N \gg W$ and $N \gg H$, and a binary DotPlot with two traces of $N = 10^6$ corresponds to an image with 10^{12} pixels (terrapixel). Our VDP visualization aggregates multiple events into a single pixel, normalizing it to RGB intensities between $0 \ldots 255$. Unlike a digital photo, the intensity of a pixel could be up to N, since aggregating events could end up with all of them in a single pixel which could easily exceed 255. `lviz` provides two options to deal with this issue by borrowing ideas from image processing. The first is to simply normalize the values. We extend normalization to include gamma correction, i.e. given an input value x, the output value y is mapped using the equation $y = x^{1/\gamma}$ (pure normalization corresponds to $\gamma = 1.0$). The second option is histogram equalization which is a well-known technique in image processing.

Figure 5.18 shows our `xcopy` example visualized with various normalization options. In this example with an event-ordered VDP, the intensity of a pixel shows how many matching events there are in the DP. We see that histogram equalization (the default) is usually effective in allowing matching burst events (high intensity) to seen together with spread out events (low intensity). While smaller γ shows details of the high intensity events, larger γ shows low intensity events.

Most of the cost of rendering a VDP has to do with various forms of matching: applying DP matching, DP coloring and barcode rules. We can optimize the rendering to reduce matching costs by pre-processing the traces. `lviz` performs the pre-processing steps when reading the traces. Subsequent visualizations such as zooming in then make use of the pre-computed results of the pre-processing phase.

An event in the raw trace is simply a string. The pre-processing steps involves reducing an event to a digest value as a more compact representation for DP matching to operate on and also pre-computing the results of various coloring rules. Each event is represented as a hash value which we call a *digest*. The digest helps to speed up event comparisons as comparing digest values of events is much faster than the corresponding string comparison, e.g. string matching is slow as the strings of an event property often have a long common prefix such as absolute pathname. The digest of an event is computed based on the DP matching rule that specifies how to tell if events are considered similar (Section 5.2.2.2). For example, if the matching rule is *program name + operation type + resource name*, the digest of the event is a hash on those three properties of the event in that order. We will refer to all the events having the same digest under the set of DP matching rules as a *digest group*. Thus, the number of digest groups in a trace is equal to the number of distinct hash values. Note that an event can only belong to one digest group and the hash function should be good enough to be effectively collision free so that two events which do not match (according to the DP matching rule) are not hashed to the same digest group.

It is common that the pattern in a coloring rule only needs a string match. Since there can be many color rules, we want to avoid applying to the matching events one rule at a time. We instead employ the Aho-Corasick's string matching algorithm [13] to compile all the rules (the string patterns) into a single automaton. This allows us to identify the color in a single pass rather than multiple passes for each rule.

The two pre-processing steps above are performed once and cached in memory for later use when rendering the VDP. The VDP rendering has to be fast as it is used interactively for real-time browsing which entails zooming in on a region within the DP. As the number of events in a trace, N, grows in order of hundreds of thousands, the naive DP Algorithm 1 takes $O(N^2)$ becomes infeasible. To simplify our illustration, we assume both logs have the same number of events. We developed a new DP rendering algorithm which scales well to large traces.

The core of new DP rendering algorithm is based on two observations. The first

Input: $L_X[0 \dots N-1]$ as log X; and $L_Y[0 \dots N-1]$ as log Y
Output: 2D array $S[0 \dots W-1, 0 \dots H-1]$ as pixel intensity

zero fill S;
for $x \leftarrow 0$ **to** $N-1$ **do**
  for $y \leftarrow 0$ **to** $N-1$ **do**
    if $L_X[x]$ *matches* $L_Y[y]$ **then** `// based on DP matching rule`
      $S[xW/N, yH/N]++$
    end
  end
end

Algorithm 1: Naive $O(N^2)$ Algorithm

Input: $L_X[0 \dots N-1]$ as log X; and $L_Y[0 \dots N-1]$ as log Y
Output: Associative array $D_X[d]$ whose key d is the digest of an event and value is a set of integers representing indexes of events in L_X whose digests are d. Similarly for $D_Y[d]$.

for $x \leftarrow 0$ **to** $N-1$ **do**
  $d \leftarrow digest(L_X[x])$; `/* digest() calculates the digest based on the DP matching rule */`
  insert x to the set $D_X[d]$
end
for $y \leftarrow 0$ **to** $N-1$ **do**
  $d \leftarrow digest(L_Y[y])$;
  insert y to the set $D_Y[d]$
end

Algorithm 2: Preprocessing step to get digest group

observation is that for each matching event, a certain intensity value can be added incrementally. Two events are considered a match if they are in the same digest group. Events in different digest groups will never match thus will never be rendered. The incremental algorithm effectively processes only the events that contribute to the visualization and prunes all other events. Consider digest group D_i which has S_i events. It can be processed in time $O(S_i^2)$ and the result for this digest group can be added incrementally to the end result. This significantly reduces the number of matching comparisons needed from $O(N^2)$ down to $\sum_{i=1}^{G} S_i^2$ where G is the number of digest groups in the trace. Algorithm 2 pre-process the two logs L_X and L_Y into digest groups stored in associative arrays D_X and D_Y. This preprocessing step takes $O(N)$ time and is executed only once. Algorithm 3 iterates each digest group and accumulates the number of matches.

However, in the worst case, a digest group S_i can be as large as N. This drawback leads us to the second observation which gives an optimization for the case when the digest group size is large. It exploits the fact that $N > W \times H$ and often also $N \gg W \times H$. Consider a digest group with S_i events. The intensity (color) for each event in that group

Input: $D_X[d]$ and $D_Y[d]$ from Algorithm 2
Output: 2D array $S[0 \dots W-1, 0 \dots H-1]$ as pixel intensity

```
zero fill S;
foreach digest d do
    foreach x in D_X[d] do
        foreach y in D_Y[d] do
            S[xW/N, yH/N]++
        end
    end
end
```

Algorithm 3: New algorithm for large screen size

Input: $D_X[d]$ and $D_Y[d]$ from Algorithm 2
Output: 2D array $S[0 \dots W-1, 0 \dots H-1]$ as pixel intensity

```
zero fill S;
foreach digest d do
    declare temporary integer array S_X[0...W-1] and S_Y[0...H-1];
    zero fill S_X and S_Y;
    foreach x in D_X[d] do S_X[xW/N]++ ;
    foreach y in D_Y[d] do S_Y[yH/N]++ ;
    for w ← 0 to W-1 do
        for h ← 0 to H-1 do
            S[w,h] ← S[w,h] + S_X[w] × S_Y[h]
        end
    end
end
```

Algorithm 4: New algorithm for small screen size

can be recorded as its projection on the X and Y axis of the window. This is easily done by keeping track of the color intensities in two arrays with size W and H respectively in $O(S_i)$ time. Therefore, the resulting DP on a window of size $W \times H$ can be drawn based on the projected intensities in $O(S_i + W \times H)$ time, as shown in Algorithm 4.

Both algorithms can be combined together to get the best of both worlds, which gives a DP rendering time complexity of $\sum_{i=1}^{G} min(S_i^2, S_i + W \times H)$, as shown in Algorithm 5. That is, for each digest group S_i that satisfies $S_i^2 < S_i + W \times H$, the $O(S_i^2)$ algorithm is used; otherwise, the $O(S_i + W \times H)$ algorithm is used to obtain a sub-image. Adding the intensities of all the sub-images of all digest groups gives the final image.

We illustrate the complexity of the algorithm to process $N = 10^6$ events. The worst case is when the number of digest groups G is 10^3 and each digest group has size 10^3. Suppose the window screen size is 1000×1000. Then the first $O(S^2)$ algorithm is used giving $GS^2 = 10^9$ operations. If the window size is resized to 600×400, then the second $O(S+W \times H)$ algorithm is used giving $G(S+WH) = 2.41 \times 10^8$ operations. This is much

Input: $D_X[d]$ and $D_Y[d]$ from Algorithm 2
Output: 2D array $S[0 \dots W-1, 0 \dots H-1]$ as pixel intensity

```
zero fill S;
foreach digest d do
α   if |D_X[d]| × |D_Y[d]| < |D_X[d]| + |D_Y[d]| + W × H then
        foreach x in D_X[d] do
            foreach y in D_Y[d] do
                S[xW/N, yH/N] ++
            end
        end
    else
        declare temporary integer array S_X[0 ... W − 1] and S_Y[0 ... H − 1];
        zero fill S_X and S_Y;
        foreach x in D_X[d] do S_X[xW/N] ++ ;
        foreach y in D_Y[d] do S_Y[yH/N] ++ ;
        for x ← 0 to W − 1 do
            for y ← 0 to H − 1 do
                S[x, y] ← S[x, y] + S_X[x] × S_Y[y]
            end
        end
    end
end
```

Algorithm 5: Combined algorithm from 3 and 4. (Line α can be further tuned with some constant factors)

faster than the naive algorithm which would take $O(10^{12})$ operations. The best case is achieved when each digest group only contains one event. This means $G = 10^6$ and the number of operations is $GS^2 = 10^6$. Another good case is when there is only one digest group, so $S = 10^6$ and the number of operations is $G(S + W \times H) = 1.24 \times 10^6$.

`lviz` is multi-threaded to make use of multi-core processors. The pre-processing steps and rendering algorithms can be easily parallelized. The events in the trace can be split up into several sub-traces and the pre-processing can be applied to each sub-trace independently and in parallel. We implemented and tested the rendering algorithm using OpenJDK 1.6.0_18 (64bit) on an Intel Core i7-960 3.20GHz machine with 12G RAM. An uncompressed 120MB trace file with 558K events took 2.8 seconds to read and finish the two pre-processing steps using 8 threads. To see the effectiveness of parallel pre-processing, it took 6.0 seconds, when the execution is single-threaded. In our rendering algorithm, each digest group can be processed independently and in parallel as there is no dependency between digest groups. We also render all the sub-components of a VDP, the DP, barcodes, and histograms in parallel. It took about 0.5 seconds to render the trace where the DP has 550K events on both axis. Thus, `lviz` can process large traces with sufficient real-time response when the user interactively zooms/resizes the VDP or

changes the histogram.

To speed up subsequent runs, `lviz` creates a cache containing the results of the pre-computations of the digest, DP color, barcodes color of each event since they require the most computation and are the bottleneck of the system. The cache avoids recalculating the digest and the coloring rules every time the program is run. This cache is rebuilt if the corresponding rules change. Loading from the cache skips the pre-computation steps and cuts down the trace loading time to 1.3 seconds. The speedup is more significant when the matching and coloring rules are more complicated.

5.2.4 Case Studies

The following case studies show how `lviz` can be used to understand system behavior and to solve system or application problems. We first use a well understood task, file copying, to familiarise the reader with VDP. We then use a software compilation example to show how VDP can be used for software failure diagnosis. We use two visualizations on whole system traces to understand how the operating system works and examine performance problems. Lastly, we use VDP to examine performance problems in a browser benchmark. These examples also illustrate that the VDP visualization is highly customizable and in practice, one could use many different visualizations on the same trace.

The notation Bar n refers to the nth barcode. The configuration of each visualization is described in the corresponding figure caption.

5.2.4.1 Comparing File Copying Programs

We use an example of a simple program and operation, namely, file copying using `xcopy`, to explain various aspects of the VDP visualization. Figure 5.16 shows a self-comparison event-ordered VDP of `xcopy` copying 8 files with sizes 1MB, 10KB, 10MB, 100KB, 1MB, 10KB, 10MB and 100KB and in that given order. A self-comparison VDP has a 45° diagonal line from top-left to bottom-right corner because events on the diagonal always match. The structure of the files being copied is visible as blue squares in Region *3*, *4*, *5* and *6* which show copying the first 1MB and 10MB, then the second 1MB and 10MB files. It is interesting to see the effect of scale on the visualization. At this scale, which shows everything, the relative file sizes are also visible for the large files. The smaller 10KB and 100KB files are too small to be seen at this scale but are visible when zoomed in. By correlating Bar1 and Bar2, we find that there are some file related operations in the early phase, but those files are neither source nor destination files of `xcopy`. This might seem surprising. To answer that question, we click on Region 7, which shows that the files in question are DLLs. Thus, the top-left region shows `xcopy` loading DLLs.

Bar3 shows that Region *2* has many registry operations which might be surprising for a file copying task. The histogram spike at Region 1 shows that some events at the end of

the trace take a long time. Figure 5.17 is a zoomed-in view to one of the blue squares in Figure 5.16. The checkerboard-like pattern shows alternating read (magenta) and write (cyan) operations which is what we expect for file copy. Since $cyan + magenta = blue$, this explains that the blue squares in Figure 5.16 represents reading from source files and writing to destination files. We see also that visualizations at different scales can be used for different purposes. The extreme magnification in Figure 5.17 shows the actual operations but may be too detailed a view for the overall picture which emerges at the other extreme in Figure 5.16.

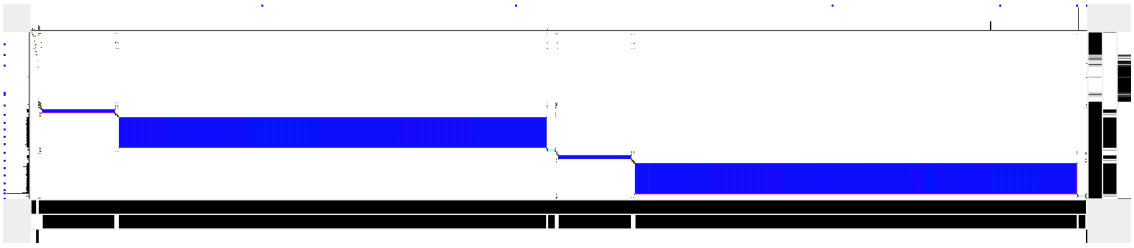


Figure 5.19: Event-ordered VDP comparing `cp` (x-axis) and `xcopy` (y-axis) copying the same files. The configurations are the same as in Figure 5.16.

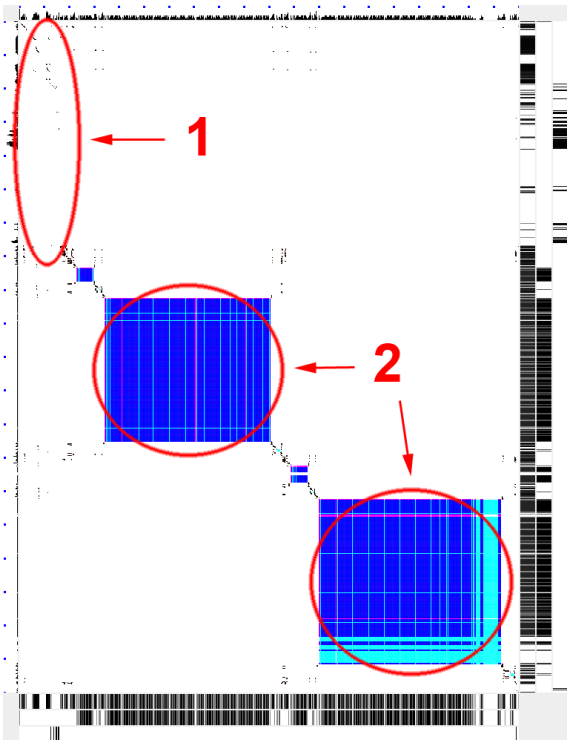


Figure 5.20: Time-ordered VDP comparing `cp-64k` (x-axis) and `xcopy` (y-axis). The configurations are the same as in Figure 5.16.

In general, a VDP is used with two traces rather than a single trace as in a self-VDP. Figure 5.19 compares `cp` (x-axis: a Windows version of GNU `cp`) and `xcopy` (y-axis) copying the same 8 files. The VDP is much wider than its height, giving wide blue rectangles instead of blue squares (Figure 5.19 versus Figure 5.16). Thus, `cp` performs many more operations than `xcopy`. To investigate further, zooming in and examining individual events, we find that each read and write operation uses a buffer size of 4K, while `xcopy` uses 64K. This means that `cp` has 16 times more read and write operations than `xcopy`. This is consistent with the width-height ratio in the visualization.

In order to compare the performance of `cp` and `xcopy`, we can use a time-ordered VDP. Changing the visualization to a time-ordered one (not shown) shows that `cp` runs slower than `xcopy` though less than 16x. We then modified `cp` to obtain another version `cp-64k` which uses 64K buffers. Figure 5.20 shows the time-ordered VDP comparing `cp-64k` (x-axis) and `xcopy` (y-axis). From Region *2*, we can see that copying of the largest file is performed in about the same time in both programs. However, `xcopy` has a slow initialization (as shown in Figure 5.20 Region *1* which includes the registry operations in Figure 5.16 Region *2*), which causes `xcopy` to be slower in total than `cp-64k`.

5.2.4.2 A Software Build Case Study

We now apply visualization to diagnose program failure. In Figure 5.21, we compare a successful software project build with a failed one to explain the failure. The project consists of 4 C files, a header file and a `makefile`. We use `nmake.exe` from Visual Studio to build the project. To deliberately cause failure, one of the C files is changed to be unreadable.

We first explain what the visualization shows. From the Bar2 of x-axis, we can see 4 `cl.exe` processes (4 cyan regions), which tallies with the number of C files. There is one invocation of `link.exe` (magenta), occurring at the end of the trace. By correlating Bar2 and Bar3, we see that `cl.exe` processes first read a C file (tiny blue bar in region *7*) and then read a number of header files (long magenta bar in region *8*). Different `cl.exe` processes read a different number of header files, see sizes of regions *3*, *4*, *5*, *6* differ.

In a self-comparison VDP, there will be a 45° diagonal line from top-left to bottom-right corner. Figure 5.21 shows a 45° diagonal (Label *1*) with some breaks, from the start to region *2*, near the end of the failed build (y-axis). It shows that the traces are basically similar till the failure point when the build terminates. By zooming in (not shown), we see that the failure occurs at the third invocation of `cl.exe`, just before reading the C file. Thus, the VDP shows how file I/O works in the build and we use this to identify a failed build. One might argue that the compiler's error message is a better solution. However, firstly, not all software give detailed error message like the compiler in this case. Secondly, due to software bugs, the error message could be wrong, while the system trace

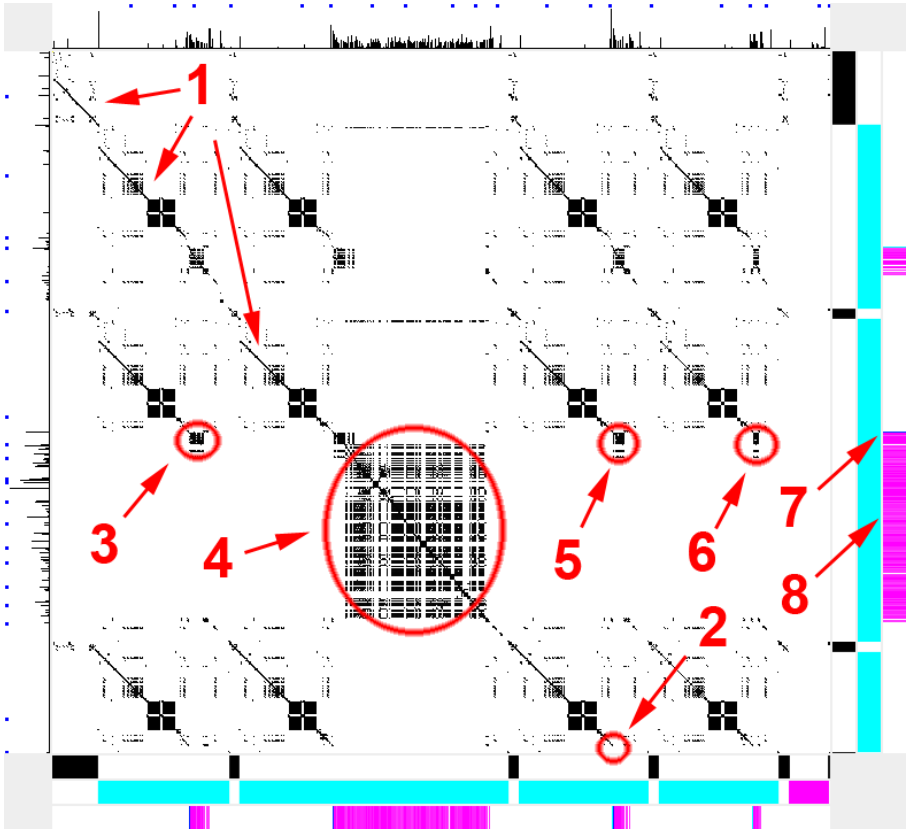


Figure 5.21: Event-ordered VDP comparing a successful (x-axis) software build process and a failed (y-axis) one.

DP match :	program + operation + value (pathname)
DP color :	black → any
Bar1 color :	black → `nmake.exe`
Bar2 color :	cyan → `cl.exe`; magenta → `link.exe`
Bar3 color :	cyan → reading `.c` files;
	magenta → reading `.h` files

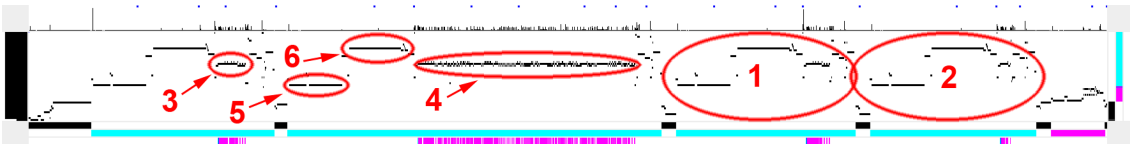


Figure 5.22: Program point event-ordered VDP of project build: pseudo program point trace (y-axis).
DP match: program name + program point; DP color: black → any; Barcode coloring rules are the same as in Figure 5.21; Bar3 in y-axis is undefined as the pseudo trace does not have operation type.

reports the actual underlying issues.

The build example is reused to show a visualization of the code structure of `cl.exe` in Figure 5.22. We use a pseudo trace (y-axis) consisting of the program name concatenated with its program point in the binary (an instruction address) which is sorted by program name and address. The program point comes from the stack walk corresponding to the event and corresponds to the latest return address in the stack trace coming from the executable or earliest DLL. The x-axis trace is as before in Figure 5.21.

Region *1* and *2* show two invocations of `cl.exe` having similar patterns. In fact, all 4 invocations follow the same pattern which can be broken up into three main components: region *5*, *6* and *4*. By zooming in and looking at the events, we infer that region *5* and *6* is in the initialization phase of the compiler, which include loading of DLL files and reading configurations from the registry. Region *6* is the reading of a large DLL `rsaenh.dll`. By looking at Bar3, region *3* and *4* are the reading of C and header files. As different C files have different header includes, region *3* and *4* have different widths. The fine squiggles in region *3* and *4* represent three different file operations: open, read and close. For each header file, there is one file open and close event, with a differing number of file read operations depending on the file size. This makes the squiggles irregular. Since this VDP is based on program points, it shows directly the structure of the function calls. Thus, it visualizes the code while Figure 5.21 compares the operations due to system calls. It is also different from the source code visualization [52] and we do not need to rely on availability of source code.

The configurations play an important role in a VDP. Figure 5.23 shows two VDPs which are visually different from each other and also Figure 5.21 but they are all visualizations of the same underlying trace. The VDPs in Figure 5.23 and Figure 5.21 only differ in the DP matching rule and the rest of the configuration is the same. The left side VDP in Figure 5.23 uses only the event operation as the DP matching rule. It can be used to highlight same or different operations. The right side VDP uses the program name, and can be used to show different programs used in the project build process.

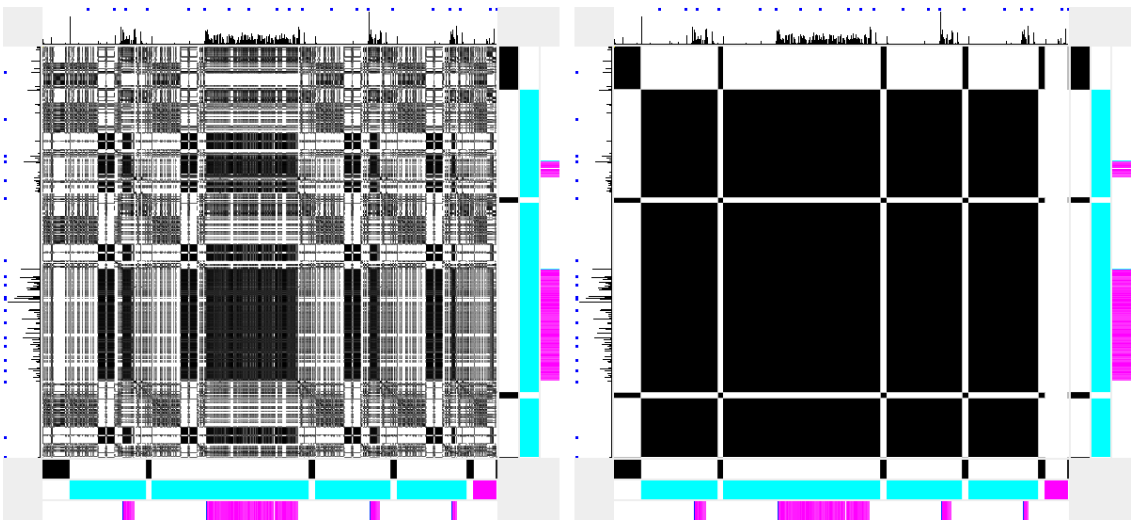


Figure 5.23: Changing the DP matching rule of Figure 5.21. Left side DP matching rule is operation; Right side is program name.

5.2.4.3 Visualizing Idle System Traces

Previous examples compared the trace of particular programs (or to understand how a program behaves using a self-similarity VDP). We now show that visualization can also be useful for system traces as a whole – a system trace is the complete trace of the entire operating system for a period of time.

When Windows is idle (i.e. no user interaction), system services and system processes still run. We want to visualize whether such services and programs have (some expected) periodic behavior. Figure 5.24a (left) shows a time-ordered VDP from two different idle machines for an hour at different times. The size of the x and y-axis traces are 851528 and 652713 events respectively. The configuration is described in the figure caption. Bar1 and Bar2 show which events belong to the four most active programs: `lsass.exe` is the Local Security Authority Subsystem Service which enforces security; `svchost.exe` runs various services; `explorer.exe` is the graphical shell; and `wuauclt.exe` is windows auto-update service. Bar3, Bar4 and Bar5 show the operation types.

Figure 5.24 shows periodic structure which appears fractal-like. Bar1 and Bar2 in Figure 5.24a show that events from `lsass.exe`, `svchost.exe` and `explorer.exe` occur continuously in the traces but `wuauclt.exe` only occurs during a short period of time. Since this is time-ordering and the trace is one hour, we can conclude that `wuauclt.exe` occurs for about 7 minutes. (Bar2's magenta region spans about 2 ticks in the histogram and there are 20 ticks in total.) During this short period, we can see that a lot of events occur, because the histogram at Region *a1* and *a4* have peaks. We select two obvious regions to study the structure: Region *a2* which is zoomed in Figure 5.24b and Region

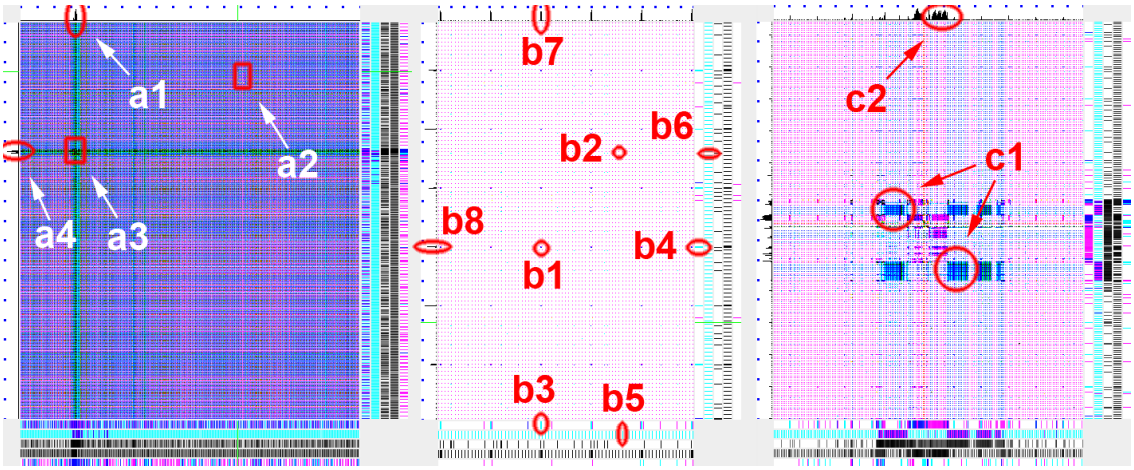


Figure 5.24: Time-ordered VDP comparing two idle systems. **a.** (left) comparing one hour interval between two machines; **b.** (middle) zoom in of Region *a2*; **c.** (right) zoom in of *a3*. The different DP color intensity in the zoomed views is caused by histogram equalization.
DP match: program + operation; *DP color*: cyan → file; magenta → registry; yellow → others; *Bar1 color*: cyan → lsass.exe; magenta → svchost.exe; *Bar2 color*: cyan → explorer.exe; magenta → wuauclt.exe; *Bar3 color*: black → file; *Bar4 color*: black → registry; *Bar5 color*: cyan → network, magenta → process & thread creation & termination.

a3 zoomed in Figure 5.24c.

We turn to the zoom in Figure 5.24b. Bar1 and Bar2 in Figure 5.24b show more clearly that `explorer.exe` (Region *b5* and *b6*) runs more frequently than `lsass.exe` (Region *b4* and *b3*). Thus, we see two kinds of periodic events. The numerous dots such as Region *b2* are the more frequent periodic events from `explorer.exe` while the darker and less frequent dots such as Region *b1* come from `lsass.exe`. Although `explorer.exe` runs more frequently, the event frequency histogram at Region *b7* and *b8* shows that `lsass.exe` has more events each time, hence Region *b1* is darker than *b2*. The white portions and together with the histogram show that there are no other events in these portions of the idle trace.

We have covered the periodic events in Figure 5.24a except for Region *a3*. To explain Region *a3* and its zoom in Figure 5.24c, we notice the cyan in Bar5 (network events) shows more activity. Selecting those network events we find the program is `wuauclt.exe` (Windows update) – both machines run Windows update at different time points explaining the increase in network events. The dark Region *c1* shows that many of the events are the same in both machines in Windows update. These events are file read and write to `C:\windows\softwaredistribution\datastore\datastore.edb` (the Windows update database). Other bursts of events in Region *c2* are generated by `svchost.exe` (it runs various Windows services) which operates on registry keys related to windows installation.

This example shows that different machines not synchronized to each other have com-

mon periodic patterns and behavior. We note that as we used idle traces, the frequency of events will be low and a direct visualization is quite faint. As such, VDP employs equalization on the image to make the low frequency events more visible.

5.2.4.4 System Boot

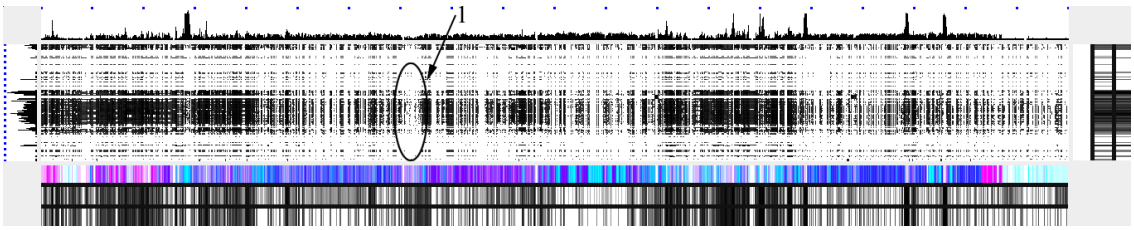


Figure 5.25: Time-ordered VDP comparing boot of a clean (Y axis) and a dirty (X axis) system.
DP match: program + operation + parameter; *DP color*: black → any; *Bar1 color*: cyan → Google Desktop; magenta → AVG; *Bar2 color*: black → any program except Google Desktop and AVG; *Bar3 color*: black → file operation on Windows system directories.

In Figure 5.25, we use `lviz` to investigate system boot and try to identify possible factors causing a system to boot slowly. To do this, we use a time-ordered VDP to compare a clean system which is known to boot quickly with a (dirty) system which has many programs installed and boots slowly.

We see that the clean system boots much faster than the dirty system as the height is much shorter than the width. Whenever Bar2 on the x-axis is not black, the AVG (AVG antivirus) or Google Desktop is responsible for events. We see that most of the events in the dirty boot are due to AVG or Google Desktop. We can also see from comparing Bar2 with Bar3 that the dirty system has other installed programs running which are not in Windows directories.

Even ignoring other programs outside the Windows directories, the slowdown is also due to extra work by programs in Windows directories. We can see that this because some of the white DP gaps match up with black portions in Bar2 (program from Windows directory), e.g. Region 1. This can be caused by the modification of the Windows system by third party programs, e.g. installing new services which are running in svchost.exe, and adding new registry keys or files which will be scanned by windows programs. To find out the details, we can focus on Windows built-in programs which are identified by Bar3. Vertical gaps in the Windows built-in program region identify the additional work done by the Windows built-in programs in the dirty system. By clicking in Region 1, we found out that the gap is caused by `svchost` (the generic processes for services) since the new software installed uses `svchost` to run some services.

5.2.4.5 Web Browser Benchmark

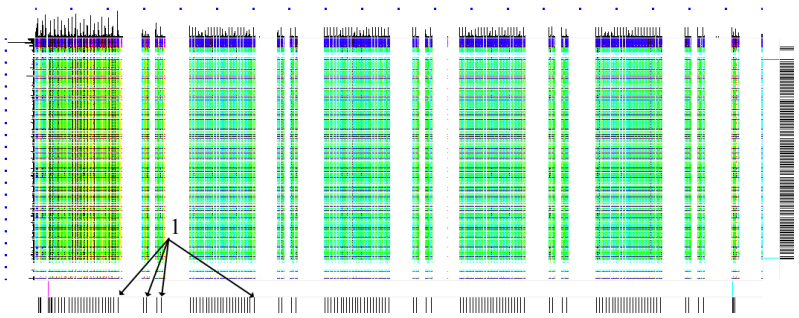


Figure 5.26: Time-ordered VDP comparing IE7 (x-axis) and Chrome (y-axis) performing the SunSpider JavaScript benchmark.
DP match: operation; *DP color*: cyan → file; magenta → registry; yellow → network operation; *Bar1 color*: cyan → benchmark start time; magenta → benchmark end time; *Bar2 color*: black → HTTP GET event.

We use time-ordered VDP (Figure 5.26) to investigate the SunSpider JavaScript benchmarks on two web browsers, Internet Explorer 7 (IE7) and Google Chrome. The benchmark consists of 26 different tests repeated for 5 times. Since the browser fetches a web page for each test, we use the HTTP GET[1] event (shown in Bar2) to mark the beginning of each test. By looking at the time-ordered DP, we know that Chrome is about three times faster than IE7 in total and also in each test. From Bar2, we can also observe that some tests (followed by long white gap, pointed by marker 1) take much longer time than others. By clicking on those events, we found out the slow ones to be all string operation[2] benchmarks. We conclude that one reason why IE7 is slower than Chrome is due to string operations.

5.2.5 Conclusion

The objective of this work is twofold. Firstly, we show that visualization can be beneficial for problems in software understanding and diagnosis. We demonstrate this for traces with a single program as well as with several programs and a complete system trace. Secondly, we show that a DotPlot-based visualization, VDP, is effective in visualizing a range of problems. One feature which distinguishes `lviz` is that as we work at the operating system level with system and stack traces, we do not need source code of the software. We believe that the range of understanding and diagnosis applications

[1]To mark HTTP GET events, we need to turn on network data I/O collection and use a suitable regular expression as the barcode coloring rule. The start and end events are marked by matching the two URLs.

[2]The URLs are `/perf/sunspider-0.9/string-base64.html`, `/perf/sunspider-0.9/string-tagcloud.html`, and `/perf/sunspider-0.9/string-validate-input.html`

shows that visualization of operating system traces is an interesting approach. It also complements other kinds of analysis such as program analysis and data mining.

Our VDP visualization is only one general purpose visualization. There are some other problems where dependency visualizations (Section 5.1) or special variants of VDP would be more appropriate. For example, to understand resource usage, a special kind of VDP with resources and traces overlaid with resource lifetimes would give more information than a regular VDP. We have not taken advantage of source code and this can further enhance the visualization but it would need monitoring infrastructure which can correlate execution with the source. We also remark that although we do not make use of source code, we see that visualizations using program points can be used to understand how the code works (at the native code level).

We have developed the VDP tool for Windows because such visualizations are more beneficial given the closed source nature and system complexity of Windows. However, VDP is not reliant on a particular monitoring infrastructure as the input trace format is very simple. The same visualization ideas can be applied to other systems with a rich source of execution traces, e.g. UNIX variants.

Chapter 6

Binary Integrity

Securing Windows is a challenge because of its large attack surface which can lead to many ways where binaries (Section 2.1.4) can be loaded and subsequently executed. Many of the system security problems such as malware stem from the fact that untrusted binaries are executed. This is further complicated by software in Windows having many interactions on binaries such as automatic online updates, installing third party plugins, hijacking system libraries and creating temporary binaries. These interactions are often not well understood or specified. With the help of our monitoring infrastructure (Section 3.2) on the interactions with binaries, we can tackle the security problems from the *binary integrity* point of view. Binaries are the only type of files that contain native code which can be directly loaded by the operating system. Many security attacks such as USB drive propagated virus and drive-by downloads are directly targeted on binaries. Most other attack methods such as buffer overflows and social engineering are usually performed together with introducing malicious binaries so that the attacks can be persistent to the system. The term binary integrity means binaries in the system should be trusted by the user. In this chapter, we present our security system which ensures binary integrity in Windows.

The security system has two goals. Firstly, the security objective is to ensure the use of trusted binaries and maintain their integrity. This protects against attacks which attempt to modify or remove binaries commonly used in malware or denial of service attacks. We prevent malware attacks which rely on getting a malicious executable to be run or a malicious DLL to load. In addition, we prevent the malware from being persistent.

Secondly, the security mechanism needs to be usable with typical software in binary form. It needs to be compatible with the software life cycle of software in the system, i.e. software may be installed, then used, then updated, and finally uninstalled. Portions of installed software, e.g. DLLs, may be shared with other software. Ideally, the security mechanism should be (mostly) transparent to this software life cycle. While it may not be

possible to be completely transparent/compatible for arbitrary binaries, it should support the mechanisms used by typical software. We remark that as we are concerned with closed source and binaries, approaches which change the process of installation/update, usually requiring (some source code), are not relevant. Clearly, there is a tradeoff between security and usability – we seek a practical middle ground with improved security while being compatible with typical software.

This chapter is organized as follows. In Section 6.1, we first present *BinAuth* which achieves the first goal. BinAuth can be used in a relatively static environment where the binaries rarely change or binary changes are well managed by the system administrator. The aim is to have a reliable and efficient security mechanism to authenticate binaries. We then extend it in Section 6.2 to make it more usable in a general non-static setting. Our new system *BinInt* allows easy installing and updating software while still prevent untrusted software from running.

6.1 BinAuth: Secure Binary Authentication

Malware such as viruses, trojan horses, worms, remote attacks, are a critical security threat today. A successful malware attack usually also modifies the environment (e.g. file system) of the compromised host. Many of the system security problems such as malware stem from the fact that untrusted code is executed on the system. We can mitigate many of these problems by ensuring that code which is executed only comes from trusted software providers/vendors and the code is executed in the correct context. In this section, we show that this can be efficiently achieved even on complex operating systems such as Windows. Our system provides two guarantees: (i) we only allow the execution of binaries (in the rest of the section, we refer to any executable code stored in the file system as a *binary*) whose contents are already known and trusted — we call this *authenticating binary integrity*; and (ii) as binaries are kept in files, the pathname of the file must match its content — we call this *authenticating binary location*. Binary integrity authentication ensures that the binary has not been tampered with, e.g. `cmd.exe` is not a trojan. Binary location authentication ensures that we are executing the correct executable content. A following extreme example illustrates location authentication. Suppose the binary integrity of the shell and the file system format executables are verified. If an attacker swaps their pathnames, then running a shell would cause the file system to be formatted. In this section, we refer to *binary authentication* to mean when both the binary's data integrity and location are verified.

Most operating systems can prevent execution of code on the stack due to buffer overflow, e.g. NX protection. Combining stack protection with binary authentication makes the remaining avenues for attack smaller and more difficult. Binary authentication is also beneficial because it is even more important for the operating system to be protected against malicious drivers and the loading of malware into the kernel.

Most work on binary integrity authentication is on Unix/Linux [16, 101, 90]. However, the problem of malware is more acute in Windows. There are also many types of executable code, e.g. executables (.exe), dynamic linked libraries (.dll), ActiveX controls, control panel applets, and drivers. In this section, we focus on mandatory binary authentication of all forms of executables in Windows. Binaries which fail authentication cannot be loaded, thus, cannot be executed. We argue that binary authentication together with execute protection of memory regions (e.g. Windows data execution prevention) provides protection against most of the malware on Windows.

A binary authentication needs to be flexible to operate under different scenarios. Our prototype signs binaries using a HMAC [49], which is more lightweight than having to rely on PKI infrastructure, although it can also make use of it. The authentication scheme additionally allows for other security benefits. Not only is it important to authenticate software on a system but one also needs to deal with the maintenance of the software over

time. Nowadays, the number of discovered vulnerabilities grows rapidly [27]. This means that binaries on a system (even if they are authenticated) may be vulnerable. This leads to a vulnerability management and patching problems. We propose a simple software ID system leveraging on the binary authentication infrastructure and existing infrastructures such as DNS and certificate authorities to handle this problem.

Windows has the Authenticode mechanism [59]. In Windows XP version and earlier, it alerts the users of the results of signature verification under a few situations. However, it is not mandatory, and can be easily bypassed. The Windows Vista UAC mechanism makes use of signed binaries but it only deals with EXE binaries. It is also limited to privilege escalation situations. One common drawback of existing Windows mechanisms is that they do not authenticate the binary location. Moreover, requiring PKI infrastructure and certificates, we believe, is too heavy for a general purpose mechanism.

The main contribution of this work is that we believe that it provides the first comprehensive infrastructure for trusted binaries for Windows. This is significant given that much of the problems of security on Windows stems from inability to distinguish between trusted and untrusted software. It provides mandatory authentication for the full range of binaries under Windows, and goes beyond authenticated code in XP and Vista. We also protect driver loading which gives increased kernel protection. Our scheme provides mandatory driver authentication which 32-bit Windows does not, and can be integrated with more flexible policies which 64-bit Windows does not support. We also analyze the security of our system. Our benchmarking shows that the overhead of comprehensive binary authentication can be quite low, around 2%, with a caching strategy.

6.1.1 Windows Issues

We showed previously the complexities and challenges in Section 2.1. We now discuss below some special problems of Windows which make it more difficult to implement binary authentication than in other operating systems such as Unix. Windows NT (Server 2000, XP, Server 2003, Vista) is a microkernel-like operating system. Programs are usually written for the Win32 API but these are decomposed into microkernel-like operations. However, Windows is closed source — only the Win32 API is documented and not the microkernel-link API. The mapping from Win32 API to system calls can be complicated in some cases. For example, the `CreateProcess()` function in Win32 API creates a child-process given an executable file. This function calls four[1] system calls. Our prototype makes use of both the documented and undocumented kernel infrastructure. However, it is not possible to make any guarantees on the completeness of the security mechanisms (which would also be a challenge even if Windows was open source). Some of the specific issues in Windows which we deal with are:

[1]It is based on our observation. There can be more system calls invoked in other conditions.

- **Proliferation of Binary Types:** It is not sufficient to ensure the integrity of `EXE` files. In Windows, binaries can have any file name extension, or even no extension. Some of the most common extensions include `EXE` (regular executables), `DLL` (dynamic linked libraries), `OCX` (ActiveX controls), `SYS` (drivers), `DRV` (drivers) and `CPL` (control panel applets). Unlike Unix, binaries cannot be distinguised by an execution flag. Thus, without reading its contents, it is not possible to distinguish a binary from any other file.

- **Complex Process Execution:** A process is created using `CreateProcess()` which is a Win32 library function. However, this is not a system call since Windows is a microkernel, and in reality this is broken up at the native API into: `NTCreateFile()`, `NTCreateSection()`, `NTMapViewOfSection()`, `NTCreateProcess()`, `NTCreateThread()`. Notice that `NTCreateProcess()` at the microkernel level performs only a small part of what is needed to run a process. Due to this, it is more complex to incorporate mandatory authentication in Windows.

- **DLL loading:** To load a DLL, a process usually uses the Win32 API, `LoadLibrary()`. However, this is broken up in a similar way to process execution above.

- **Execute Permissions:** Many code signing systems, particularly those on Linux [16, 90], implement binary loading by examining the execute permission bit in the access mode of file open system-call. The same mechanism, however, does not work in Windows. Windows programs often set their file modes in a more permissive manner. Simply denying a file opening with execute mode set when its authentication fails, will cause many programs to fail which are otherwise correct on Windows. Instead, we need to properly intercept the right API(s) with correctly intended operation semantics to respect Windows behavior.

Compared to other open platforms, Windows potentially also makes the issue of locating vulnerable software components more complicated. A great deal of binaries created by Microsoft contain an internal file version, which is stored as the file's meta-data. The Windows update process does not indicate to the user which files are modified. Moreover, meta data of the modified file might still be kept the same. Thus, it is difficult to keep track of files changes in Windows. More precisely, one cannot ensure whether a version of a program P_i remain vulnerable to an attack A. It is rather difficult for a typical administrator who examines vulnerability information from public advisories to trace through the system and pinpoint the exact affected components. Our software naming scheme, associates binaries with their version and simplifies software vulnerability management.

6.1.2 Related Work

Tripwire [48] is one of the first to do file integrity protection but is limited as it in user-mode program and checks file integrity off-line. It does not provide any mandatory form of integrity checking and there are many known attacks such as: file modification in between authentication times, and attacks on system daemons (e.g. cron and sendmail) and system files that it depends on [18, 85].

There a number of kernel level binary level authentication implementations. These are mainly for Unix such as DigSig [16], Trojanproof [101] and SignedExec [90], which modify the Unix kernel to verify the executable's digital signature before program execution. DigSig and SignedExec embed signatures within the ELF binaries. For efficiency, DigSig employs a caching mechanism to avoid checking binaries which have been verified already. The mechanism is similar to ours here but we need to handle the problems of Windows. It appears that DigSig provides binary integrity authentication but not binary location authentication.[2] In this section, we examine the implementation issues and tradeoffs for Windows which is more complex and difficult than in Unix.

Authenticode [59] is Microsoft infrastructure for digitally signing binaries. In Windows versions prior to Vista, such XP with SP2, it is used as follows:

1. During ActiveX installation: Internet Explorer uses Authenticode to examine the ActiveX plugin and shows a prompt which contains the publisher's information including the result of the signature check.

2. A user downloads a file using Internet Explorer: If this file is executed using the Windows Explorer shell, a prompt is displayed giving the signed the publisher's information. Internet Explorer uses an NTFS feature called Alternate Data Streams to embed the Internet zone information –in this case, the Internet– into the file. The Windows Explorer shell detects the zone information and displays the prompt. This mechanism is not mandatory and relies on the use of zone-aware programs, the browser and GUI shell cooperating with each other. Thus, it can be bypassed.

Since Authenticode runs in user space, it can be bypassed in a number of ways, e.g. from the command shell. It is also limited to files downloaded using Internet Explorer. Only the EXE binary is examined by Authenticode, but DLLs are ignored. One possible attack is then to put malware into a DLL and then execute it, e.g. with `rundll32.exe`. Furthermore, Authenticode relies heavily on digital certificates. Checking revocation such as Certificate Revocation Lists (CRL) may add extra delay including timeouts due to the need to contact CA. In some cases, this causes significant slowdown.

[2] Mechanisms based solely on signatures embedded in the binaries do not have sufficient information for binary location authentication. For example, one could replace `ls` with `rm`, where both binaries are signed.

The latest Windows Vista improves on signed checking because User Account Control (UAC) can be configured for mandatory checking of signed executables. However, this is quite limited since the UAC mechanism only kicks in when a process requests privileged elevation, and for certain operations on protected resources. UAC is not user friendly since there is a need for constant interactive user approval. Vista does not seem to prevent the loading of unsigned DLLs and other non EXE binaries. The 32-bit versions of Windows (including Vista) do not checked whether drivers are signed. However, the 64 bit versions (XP, Server 2003 and Vista) require all drivers to be signed (this may be too strict and restrict hardware choices).

The closest work on binary authentication in Windows is the Emu system in by Schmid et al. [81]. They intercept process creation by intercepting the NtCreateProcess system call. It is unclear whether they are able authenticate all binary code since trapping at NtCreateProcess is not sufficient to deal with DLLs. No implementation or performance benchmarks are given, so it is unclear how if their system is efficient.

6.1.3 BinAuth and Software IDs

We now present our binary authentication system, BinAuth. We want a lightweight binary authentication scheme which can work under many settings without too much reliance on other infrastructure. Furthermore, it should help in the management of binaries, and incur low overhead. Management of binaries includes determining which binaries should be authentic, dealing with issues arising from disclosed vulnerabilities, and software patching.

6.1.3.1 Software ID Scheme

We complement binary authentication with a software ID scheme meant to simplify binary management issues. The idea is that a *software ID* associates a unique string to a particular binary of a software product. The software ID should come either from the software developer or alternatively be assigned by the system administrator. The key to ensuring unique software_ID, even among different software developers, lies on the standardized format of the ID. We can define software_ID as follows:

$$\text{Software\_ID} ::= \langle\ \text{opcode\_tag} \,||\, \text{vendor\_ID} \,||\, \text{product\_ID} \,||\, \text{module\_ID} \,||\, \text{version\_ID}\rangle.\ ^{3}$$

Here, || denotes string concatenation. Opcode_tag distinguishes different naming convention, eg. $Software\_ID$ and $Custom\_ID$ defined below.

Ideally, we want to be able to uniquely assigning vendor_IDs to producers of software which can make the software_ID unique. This problem in practice might not be as difficult

[3] Module_ID suffices to deal with software versioning. Having a separate version_ID, however, is useful to easily track different versions (or patched versions) of the same program.

as it sounds since it is similar to domain name registration or the assignment of Medium Access Control addresses by network card manufacturers. One can leverage on existing trust infrastructures to do this. For example, the responsibility for unique and well known software_IDs can be assigned to a Certificate Authority (CA), which then define the vendor_ID as ⟨ CA_ID || vendor_name ⟩. Alternatively, one might be able to use the domain name of the software developer as a proxy for the vendor_ID.

A software_ID gives a one to one mapping between the binary and its ID string. This is useful for dealing with vulnerability management problems [88]. Suppose a new vulnerability is known for a particular version of a software. This means that certain binaries, providing that they correspond to that software version may be vulnerable. However, there is no simple and standard way of automatically determining this version information. Once we have software_IDs associated with binaries then one can check the software_ID against vulnerability alerts. The advisory may already contain the software_ID. Automatic scanners can then be used to tie-in this checking with the dissemination of vulnerability alerts to automatically monitor/manage/patch the software in an operating system. General management of patches in an operating system can also be done in much the same way.

In the case where no software_ID comes with a software product, one can alternatively derive one. It can be constructed, for instance, using the following (coarse-grained) string naming:

$$Custom\_ID ::= \langle \text{ opcode\_tag } || \ hash(\text{vendor\_URL} + \text{product\_name} + \text{file\_name} + \text{salt}) \ \rangle.$$

The salt expands the name space to reduce the risk of a hash function collision.

6.1.3.2 BinAuth System Architecture

In the following discussion, we assume here that binaries already come tagged with a software_ID. During the BinAuth set-up, preferably done immediately after the targeted binary installation, we generate the Message Authentication Code (MAC) values for each binary. In the case where binaries are digitally signed by its developer, then we verify the signature and then generate the MAC for each binary. Thus, only one public-key operation needs to be done at install time. We choose to use a keyed hash, the HMAC (Hash-based MAC) algorithm [49], so there is a secret key for the administrator. This is mainly to increase the security of the stored hashes. To authenticate binary integrity for any future execution of the code, only the generated HMAC needs to be checked. In what follows, we mostly write MAC which already covers the choice of HMAC.

One way of storing the generated MAC is by embedding it into the binary. However, doing so may interfere with file format of the signed binaries and may also have other complications. We instead use an authentication repository which stores all the MAC values of authenticated binaries with their pathnames. During the boot-up process, the

kernel creates its own in-memory data structures for binary authentication from it. We ensure that the repository is protected from further modification except under the control of the authentication system to add/remove binaries.[4] We can also customize BinAuth on a per user basis rather than system-wide which is a white-list of binaries approved for execution. In the case, when the initial binary does not have a digital signature, then the administrator can still choose to approve the binary and generate a MAC for it.

There are two main components of the system: the **SignatureToMac** and **Verifier**. The SignatureToMac maintains the authentication repository, *Digest_file*, consisting of ⟨path, MAC⟩ tuples. The Verifier is a kernel driver which makes use of *Digest_file* and decides whether an execution is to be allowed.

SignatureToMac Once software is installed on the system, Figure 6.1 shows how SignatureToMac processes the binaries:

1. Checks the validity of the binary's digital signature. If the signature is invalid, then report failure.

2. It consults the user or system administrator whether the software is to be trusted or not (this is similar to the Vista UAC dialog but only happens once). Other policies (possibly mandatory) can also be implemented.

3. It generates the MAC of the binary (including software_ID string) using a secret key, *Hashing_key*, to produce *software_digest*. The *Hashing_key* is only accessible by the authentication system, e.g. obtained on bootup.

4. It adds an entry for the binary as a tuple ⟨*path name*, *software_digest*⟩ into the *Digest_file* repository and informs the Verifier. The repository is protected against modification. Note that because the entries are signed, the repository can be read for other uses, e.g. version control and vulnerability management.

Verifier The Verifier performs mandatory binary authentication — it denies the execution of any kind of Windows binary which fails to match the MAC and pathname. There are two general approaches for the checking. One is *cached MAC* which avoids generating the MAC for previously authenticated binaries. The other is *uncached MAC* which always checks the MAC. As we will see, they have various tradeoffs. The cached MAC implementation needs to ensure that binaries are unmodified. Hence, the Verifier monitors the usage of previously authenticated files on the cache, and removes them from the cache if it can be potentially modified.

[4]Further security can be achieved by integrating binary authentication with a TPM infrastructure. We do not do so in the prototype as that is somewhat orthogonal.

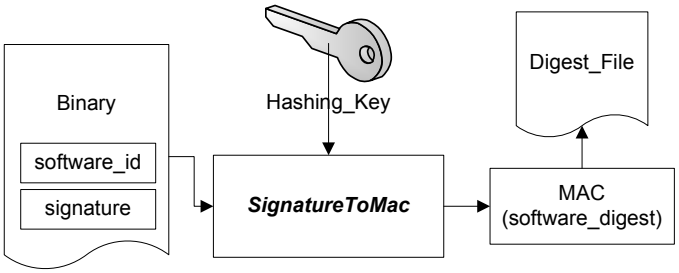


Figure 6.1: SignatureToMac: Deriving the MAC

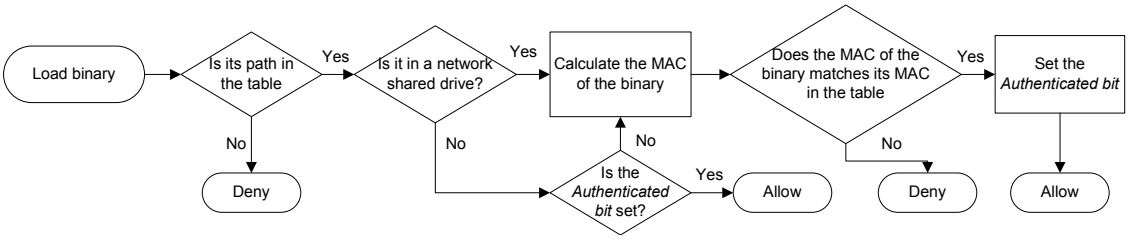


Figure 6.2: Verifier: The Verifier in-kernel authentication process

The core data structure of the Verifier component can be viewed as a table of tuples in the form ⟨*Kernel_path*, *FileID*, *MAC*, *Authenticated_bit*⟩ representing the allowed binaries. It is indexed on *Kernel_path* and *FileID* for fast lookup. The fields are as follows:

- The *Kernel_path* is Windows kernel (internal) pathname representation of a file. In Window's user space, a file can have multiple absolute pathnames, due to: (i) 8.3 file naming format, e.g. "`C:\Program Files\`" and "`C:\progra~1\`" are the same; (ii) symbolic links (reparse points is similar); (iii) hard links; (iv) volume mount points; or (v) the `SUBST` and `APPEND` DOS commands. The *Kernel_path* is a unique representation for all the possible pathnames. When the system loads ⟨*path name*, *software_digest*⟩ from *Digest_file* during the startup, *path name* is converted to *Kernel_path* since all subsequent checks by Verifier in the kernel all use the latter.

- The *FileID* is a pair of ⟨*device_name*, *NTFS_object_ID*⟩. The *device_name* is a Windows internal name to identify a disk or partition volume. For instance, the device name `HarddiskVolume1` usually refers to `C:\`. The *NTFS_object_ID* is a 128-bit length number uniquely identifying a file in the file system volume. This is not the same as UNIX inode number because it is optional. It by default does not exist and is created on demand. The Verifier uses the *FileID* to identify the same file given more than one hard link. This prevents an attacker from creating a hard link for modifying a binary without invalidating the binary cache. The *FileID* values will be queried from the system and filled into the table during system boot.

- The *MAC* is same as a *software_digest* entry in *Digest_file*. Our prototype implements the HMAC-MD5 [49], HMAC-SHA-1 and HMAC-SHA-256 [32] hash algorithms.[5]
- The *Authenticated_bit* remembers whether the binary has been previously authenticated. It is initially set to false, and set to true after successful authentication.

Figure 6.2 shows the authentication process when a binary executes/loads:

1. It checks if the binary's *Kernel_pathname* exists in the table. If not, then deny the execution and optionally log the event. A notification is accordingly sent to the user.
2. It the file is on a network shared drive, goto step 4. The MAC is always recomputed as we cannot keep track of modification to files on network shares.
3. If the *Authenticated_bit* is set go to step 7.
4. It performs MAC algorithm operation on the binary.
5. If the resulting MAC doesn't match with the *MAC* stored in the table, execution is denied.
6. It sets the *Authenticated_bit* of the binary.
7. It passes the control to the kernel to continue the execution.

To control binary execution, we intercept the section creation action (`NtCreateSection()` system call) which is better than:

- Intercepting file opening (`NtCreateFile()` and `NtOpenFile()`). This would need to authenticate any file opened with with execute access mode. As discussed in Section 6.1.1, however, this introduces unnecessary overheads and can cause some correct programs to fail if the files do not pass authentication. There are also technical difficulties to distinguish between process creation and regular file IO operations, which is not always easy given its microkernel nature.
- Intercepting process creation (`NtCreateProcess()`). This method is not effective for our purpose. Firstly, we cannot use it to control `DLL` loading. Secondly, it is more difficult to get the pathname of the binary because process creating is broken down into microkernel operations.

It turns out that since all code from any kind of binary needs to have a memory section to execute, it suffices to intercept `NTCreateSection()`.

[5]Due to recent concerns which show weaknesses and attacks against MD5 [96], we also have stronger hash functions, namely SHA-1 and the stronger SHA-256.

```
Input: FilePath
if FilePath is in NTFS then
    FileID := GetFileID(FilePath) ; /* FileID can be NULL                    */
    if FilePath is in the table then
        Entry := LookupTableByPath(FilePath);
        if FileID = NULL then
            /* This can happen when the file is deleted and created
               again                                                          */
            Entry.FileID := CreateFileID(FilePath) ; /* generate a new FileID
            and update the table                                             */
        else if FileID != Entry.FileID then
            /* This can happen when the drive is unmounted, then id
               changed off-line and re-mounted                              */
            Entry.FileID := FileID
        end
        Entry.Authenticated := false
    else if FileID != Entry.FileID then
        Entry := LookupTableByID(FileID);
        Entry.Authenticated := false
    end
else if (FilePath is in FAT) and (FilePath is in table) then
    Entry := LookupTableByPath(FilePath);
    Entry.Authenticated := false
end
```

Procedure UponModification(FilePath)

The cached MAC verifier needs to ensure that binaries which have been already authenticated are not modified. However, the uncached verifier will not need to perform file monitoring. A binary with pathname P is considered modified, if the following occurs:

- P is created: Hence, we monitor system call `NtCreateFile()` and `NtOpenFile()`.
- P is opened with write access mode: the previous two system calls are also intercepted for this purpose. [6]
- Another file is renamed to P: We monitor file renaming (`NtSetInformationFile(FileRenameInformation)`) system call.
- A drive containing P is mounted: We monitor drive mounting `IRP_MJ_VOLUME_MOUNT`.

Note that we do not need to monitor file deletion since we only care about executing correct files but not missing files. The details of file modification monitoring are given in Procedure UponModification.

[6] An alternative way is to monitor the file (block) writing operation (`NtWriteFile()`). However, it is less efficient because file block writings take place more frequently than file openings as one opened file for modification might be subject to multiple block writings. Furthermore, it cannot capture file-memory mapping.

Upon modification of P, we reset the *Authenticated_bit* of binary P, and update the *FileID* in the table if it is changed. Should FAT file system be used, pathname is used to identify the binary as *FileID* is not supported but neither are hard and soft links. Since *FileID* is optional and can be removed, we monitor *FileID* removal (`NtFsControlFile(FSCTL_DELETE_OBJECT_ID)`) and deny the removal if the *FileID* is in the table. Due to the semantics of NTFS, our use of *FileID* can coexist with other applications using it.

If additional hardware and infrastructure is available to support secure booting, such as the Trusted Platform Module (TPM) initiative, the system can benefit from increased security. Offline attacks would have to first attack the TPM. The Hashing_key can also be stored securely by the TPM.

6.1.3.3 Security Analysis

The security of BinAuth relies on the strength of the chosen hash functions (MD5, SHA-1, SHA-256) as well as the HMAC algorithm. Thus, we assume that any change in a binary can be detected through a changed MAC.

In our authentication on binary with digital signature, the subsequent invocations using MAC verification is sufficient to ensure the authenticity of the binary. In other words, MAC authentications "*preserve*" the previously established properties of BinAuth derived from digital signature. A subtlety comes when the certificate expires or is revoked at some point in time after SignatureToMac. We view that the question of whether one should keep trusting the binary for execution depends on one's level of trust on certificate expiration/revocation. If the certificate expiration or revocation means that the public key must no longer be used, but the fact that *previously established* goodness binary properties still hold, then we can keep trusting the binary for execution (as long as we still believe the issuer).

Here we discuss some possible attacks to the authentication system. All the attacks except the last two target the caching system. More precisely, the attacker attempts to modify an already authenticated binary without causing the *Authenticated_bit* to be set to false.

- **Manipulating symbolic links:** The attacker can use the path S which is a symbolic link of an authenticated P to indirectly modify P and subsequently execute P. However, the modified file will not be executed successfully, because Windows kernel resolves symbolic links to real paths. More precisely, the symbolic link S is resolved to the real path P. As a result, the *Authenticated_bit* of P will be set to false. When P is executed, its MAC will be recalculated and it will not pass the authentication.

- **Manipulating hard links:** The attacker can create a hard link H on an already authenticated file P and then modifies the file using path H. This attack will not succeed because we use *FileID* to identify files. H has the same *FileID* as P, thus the *Authenticated_bit* will be set to false. Note that this attack will not succeed in FAT file system either, even though we cannot use *FileID*. This is because hard link is not supported in FAT.

- **Manipulating FileID:** Recall that *FileID* consists of *device_name* and *NTFS_object_ID*. The latter is optional and thus can be removed. The attacker can remove the *NTFS_object_ID*, and then performs the previous attack. We handle this attack by denying *NTFS_object_ID* removal on authenticated files. This is implemented by monitoring the file system control event `FSCTL_DELETE_OBJECT_ID`

- **Remote File Systems:** Since we cannot keep track of modification on a network shared file system, we do not cache the authentication. More precisely, the MAC of the binary is always calculated upon loading. Same applies to removable media such as floppy in which we cannot keep track of modification of files.

- **Memory Mapped Files:** The modification of memory mapped files cannot be monitored by system calls because system calls are not used. However, to map a file in memory for writing, a process has to firstly open the file for writing. BinAuth clears the *Authenticated_bit* upon this operation, thus after the modification is finished, its MAC will be recalculated upon loading. During the binary is memory mapped, it cannot be loaded because Windows requires modification lock upon loading.

- **TOCTTOU:** TOCTTOU stands for Time-Of-Check-To-Time-Of-Use. It refers to race condition bugs in access control systems where the resource is changed between the time of checking the resource and the time of using the resource. In the BinAuth context, the binary may be modified after the time it is authenticated and before the time it is executed. However, we observed that all binaries are exclusive-write-locked when it is opened. That means binaries cannot be modified from the time it is opened to the time it is closed. Also note that the file is authenticated after it is opened and before it is executed. As a result, binaries cannot be modified during TOCTTOU.

  When the binary is in a network shared volume, i.e. SMB share, and the write-lock is not properly implemented in the SMB server, an attacker is able to modify the binary after authentication. However, we have observed that both Windows and Samba implement write-lock properly. Thus the attack is only possible when the SMB server is compromised. One way to prevent this is to disallow binary loading from SMB share.

- **Driver Loading:** BinAuth authenticates all binaries including kernel drivers. This means all drivers are authenticated thus driver attacks such as kernel rootkits and malware drivers can be prevented.

- **Offline Attack:** An offline attack means modification of the file system when Windows is not in control. For example, boot another operating system or remove the disk drive for modification elsewhere. Such an attack will require physical access to the machine. Offline attack can corrupt data or change programs/files and affect the general functioning and we cannot prevent that. What we can do is to ensure the integrity of executable code and other data loaded in memory for processes.

  We assume that the kernel is still secure, i.e. authentication occurs early in the boot. We also assume that kernel functioning is not impaired, e.g. deleting some system files does not cause the kernel to have an exploitable vulnerability.

  Since the hashing_key is not stored in the machine, it is not available to the attacker. The attacker can still change the digest file and the binaries, however, MACs of modified binaries cannot be produced without the hashing_key. Thus modified binaries cannot be executed when the system is online.

6.1.4 Testing

We test our implementation in terms of the protection it provides and the overhead it incurs. The following test is performed on a machine with Intel Core 2 Duo processor and 2GB of memory, running Windows XP SP2.

6.1.4.1 Protection

We test our prototype on various binary execution mechanisms in Windows. We try different mechanisms to load a binary not in the authentication repository. In the following text we use the term *invalid binary* to denote that. We remark that all the mechanisms succeed when the binary is valid (i.e. matches the authentication repository).

- **`cmd` and Explorer shells**
  Running an invalid binary in the `cmd` and `Explorer` shells fails with an error "`Application` is not a valid Win32 application". This would also cover cases such as the social engineering file extension attack.

- **driver loading**
  It is not possible to load invalid drivers. Using the `net start` command to load an invalid driver fails with the error "The specified driver is invalid".

- **service loading**
  Invalid services cannot be started. Using the `Services` tab in the Microsoft management console to start the service fails with the error "The dependency service or group failed to start".

- **shell extensions**
  We tested the `TortoiseSVN`, `AudioShell` and `FLV` extensions. `TortoiseSVN` enhances `Explorer` to interface to the Subversion revision control system with a context menu usable by right clicking any directory or file in Explorer. `AudioShell` is a MP3 tag previewer that is activated by mouse-over on audio files in Explorer. `FLV` is a codec which displays Flash video.

  If the DLLs for the shell extensions are invalid, they do not load. Thus, the `TortoiseSVN` context menu was missing and the `AudioShell` mouse-over is no longer shown. We tested the use of `Explorer` to open a folder containing `FLV` (Flash video) files. If the FLV codec, `flvsplitter.ax`, is valid, `Explorer` scans the directory, displaying a thumbnail for each FLV file. When the codec is invalid, `Explorer` does not display any thumbnails and displays the same icon as files with unrecognized extensions. We remark that in many cases, users are not aware that shell extensions are running and may think that the features are part of `Explorer`.

- **Browser Helper Objects (BHO)**
  As web browsers are one of the main targets of malware, it is important to control what binaries are loaded in the browser. We tested the Flash player control, `Flash10a.ocx`, for Internet Explorer. If the control is invalid, webpages with flash content do not work as IE is blocked from loading the Flash control. However, the rest of the page content is displayed.

- **PATH manipulation**
  An attacker may change the `PATH` environment variable or DLL search order to substitute system executable or DLL with a malicious one. This is prevented no matter what the search order is, as long as the malicious executable or DLL is invalid.

6.1.4.2 Performance Overhead

We examine the three factors which impact on system performance: (i) Verifier checking upon binary loading (execution); (ii) file modification monitoring; and (iii) binary set-up during the SignatureToMac process. The first two above are the most important as they directly affect user's waiting time for process execution and affect overall system operation. The tests here are meant to determine the worst case overhead as well as average overheads.

We use micro and macro benchmarks to test the upper bound and real-life performance overhead respectively. Each benchmark is run five times. As we want to investigate the effect of the cached Verifier, each benchmark is run with caching and without caching. When caching is enabled, we ignore the result of the first run because the overhead of authentication overhead is already shown in the uncached case. Even if we count the first run, its impact will be very small because some of the microbenchmarks run for 10K times, so the authentication overhead becomes negligible.

To see the difference of using different hashing algorithms, we implement and benchmark three algorithms: MD5, SHA-1 and SHA-256. Only MD5 and SHA-256 are shown in Table 6.1 as the results of SHA-1 are always between these. When caching is enabled, results of different hashing algorithms are not distinguished (shown as Cached-MAC in the table), because binaries are not require MAC checking during the benchmark. The reason is that the first run is ignored, and the binaries are not modified during the benchmark.

To see the difference with digital signature based authentication system, we also compare the performances of our scheme against the Microsoft official Authenticode utility called `Sign Tool` [61], and another Sysinternals (now acquired by Microsoft) Authenticode utility `Sigcheck` [60]. Note that the two tools are user-mode programs. They are there to illustrate the difference between non-mandatory strategies used with Authenticode with our in-kernel mandatory authentication.

The first two benchmarks investigate system performance under two scenarios:

1. **Micro-benchmark:** The micro-benchmark aims to measure the worst case performance overhead incurred by the scheme. Note that this is primarily intended to measure the authentication cost but not other overhead, which is done by the last file modification microbenchmark. Here, we have two micro-benchmark scenarios.

   (a) **EXE Loading:** This executes the `noop.exe` program, a dummy program that immediately exits, for 10K times. This scenario measures the overhead for authenticating the `EXE` file. The benchmark program first calls `CreateProcess()`, and waits for the child process' termination using the `WaitForSingleObject()` function. We use different binary sizes (40KB, 400KB, 4MB and 40MB, only the 40K and 40MB results are displayed) for noop.exe to see how executable size impacts performance.

   (b) **DLL Loading:** The second scenario executes the `load-dll.exe` program for 100 times. This scenario is used to find out how the number of loaded DLLs impacts the performance. Program `load-dll.exe` loads 278 standard Microsoft DLLs with a total file size of ∼75MB. The size of the `load-dll.exe` itself is 60KB. Note that in Windows, the bulk of code is often in DLLs which is why the `EXE` file may be small, e.g. Open Office has over 300 DLLs.

2. **Macro-benchmark:** The macro-benchmark measures overhead under a typical usage scenario. Our benchmark is to create the Windows DDK sample projects using the `build` command. In each test run, 482 C/C++ source files in 43 projects are built. This benchmark is chosen as it is deterministic, non-interactive, creates many processes and uses many files.

We benchmark `Sign Tool` and `Sigcheck` in the following fashion. We first sign `noop.exe` and `load-dll.exe` using `Sign Tool`'s signing operation. We then measure the execution time of authenticating and executing the two programs. For the macro-benchmark, we replace each development tool in the DDK (i.e. *build.exe*, *nmake.exe*, *cl.exe* and *link.exe*) with a wrapper program which first authenticates the actual development tool and then invokes it. For the micro-benchmark, we consider two settings: (i) `EXE` only; and (ii) all binaries (`EXE` + `DLL`). The macro-benchmark, however, only tests the EXE case. This is because, during the macro-benchmark, many programs are invoked, and each program each invocation may dynamically load a different set of DLLs. Thus, it is hard to keep track of what DLLs are loaded, and it is unfair to simulate with all DLLs used.

The results are given in Table 6.1 but we have not shown "noop 400K" and "noop 4M" because they are bounded by the results of "noop 40K" and "noop 40M". Other results not shown are that the overhead is approximately linear with respect to the file size, e.g. the results of the All-binaries/Uncached/SHA-256 benchmarks are 40K:30.42s, 400K:85.43s, 4M:598.0s, 40M:9302s.

We can see that the overhead of `Signtool` and `Sigcheck` makes it unusable if DLLs are to be checked (352x slower on `load-dll`). If only `EXE` are checked, then at least 40% overhead and based on the `load-dll` benchmark, one could expect about an order of magnitude worse if all DLLS are checked. Of course, using these tools would incur additional overhead from creating a process and the main purpose is just to show the difference between what can be done in user-mode versus in-kernel. We can see that all the uncached-MD5/SHA256 are considerably faster than `Signtool` and `Sigcheck`.

Authenticating only `EXE`, the difference between uncached has overheads around 8% while cached brings this down to very small, around 2%x and almost neglible in the `load-dll` benchmark (0.02%). Note that as uncached overhead is quite small, the results are dominated by non-determinism in timing measurements. Moving to all DLLs (`EXE` + `DLL`), we can see the effect of Windows programs using many DLLs (more code in `DLL` than `EXE`). The overhead incurred by caching is still small while uncached can grow to between 20-40% depending on the hash algorithm. Note that the uncached overhead is applicable for files which cannot be cached.

The final microbenchmark investigates the tradeoffs between cached and uncached verification. Caching means that MAC verification is amortized over executions but has

Authentication System	Micro-Benchmark						Macro-Benchmark	
	`noop` 40K		`noop` 40M		`load-dll`		`build`	
	time	slowdown	time	slowdown	time	slowdown	time	slowdown
Clean	22.76	–	30.07	–	45.32	–	66.26	–
EXE Only:								
`Signtool`	2822	11637%	4850	16033%	73.49	62.16%	97.00	46.39%
`Sigcheck`	1720	7457%	5629	18623%	62.82	38.62%	110.5	66.72%
Uncached-MD5	25.96	14.08%	2150	7052%	45.34	0.05%	70.85	6.93%
Uncached-SHA256	30.29	33.07%	9005	29851%	45.34	0.05%	71.79	8.35%
Cached-MAC	23.20	**1.93**%	30.63	**1.88**%	45.33	**0.02**%	67.62	**2.06**%
All Binaries:								
`Signtool`	11867	52043%	14030	46565%	16018	35244%	–	–
`Sigcheck`	4283	18772%	6186	20478%	12548	27587%	–	–
Uncached-MD5	26.10	14.67%	3881	12811%	128.8	184.1%	79.31	19.69%
Uncached-SHA256	30.42	33.67%	9302	30839%	201.3	344.0%	91.80	38.55%
Cached-MAC	23.25	**2.14**%	30.58	**1.72**%	45.35	**0.07**%	67.88	**2.45**%

Table 6.1: Benchmark results showing times (in seconds) and slowdown factors. The worst slowdown factors for each benchmark scenario are shown with underline, whereas the best are in bold. We define $slowdown_x = (time_x - time_{clean})/time_{clean}$.

added overhead from monitoring file modification, while uncached is the opposite. Our micro-benchmark opens a file for writing 100K times to measure the worst case overhead incurred by file modification monitoring. We have 3 experiments: (i) a clean system without BinAuth; (ii) BinAuth with cache and the modified file is a binary; and (iii) BinAuth with cache and the modified file is not a binary.

The results for the file modification micro-benchmark show that for BinAuth with a cache, it doesn't matter whether the file being written to is a binary or not. Both cases incur about 60% overhead compared to a clean system. Since BinAuth with no-cache has no overheads for file modifications, this means that under some usage scenarios where file modification is very high, the uncached strategy may be preferable over cached even when Verifier overhead is higher.

6.1.5 Conclusion

We have shown BinAuth, a comprehensive system which authenticates both content and pathname for Windows to ensure that only trusted binaries are executed. Unlike other operating systems, Windows poses significant challenges. We show that it is possible to ensure that only trusted binaries can be loaded from files for execution. This can also be combined with a simple software ID scheme which simplifies binary version management, and dealing with vulnerability alerts and patches. BinAuth is lightweight and integrates well with PKI and trust mechanisms without having to rely on them. The overheads of our prototype are quite low when caching is used. In the case of workloads with heavy file modifications, an uncached strategy might be preferable. The overheads are still low in this case, since the system overhead will be dominated by I/O rather than binary authentication, so the overall binary authentication would still be low as a percentage of overall system overhead. In summary, although this is a prototype, it significantly adds to the security of any Windows system but at the same time is sufficiently flexible so that it can be tailored for different usage scenarios.

6.2 BinInt: Usable System for Binary Integrity

The previous section showed BinAuth, a reliable and efficient system to authenticate binaries in Windows. It is designed for environments where the binaries are relatively static, i.e. the binaries rarely change or the changes are well managed by the system administrator. However, in systems such as personal machines, it is common for binaries to change. In particular, users can install or uninstall software; The software can perform automatic updates periodically; and some software creates temporary binaries.

In this section, we first look at the dynamics of binaries and compare it with attacks in Section 6.2.1. We then discuss the related work in Section 6.2.2. After that, we present a new security model for binaries called *BinInt* in Section 6.2.3. It provides a number of modes for operation: default, install and temporary trusted modes. BinInt provides protection (in default mode) against a broad range of binary-based attacks. In Section 6.2.4, we show the implementation and in Section 6.2.5, we analyze the security of our model. In Section 6.2.6, we evaluation BinInt in terms of performance and usability. Our system is efficient and has negligible overhead in default and temporary trusted modes. In install mode, our benchmarks show ∼12% overhead which is reasonable since the use of install mode is infrequent.

6.2.1 Normal Usage versus Malicious Attacks

A security model must be able to distinguish malicious attacks from normal software usage. We firstly look at typical software usage scenarios in the software life cycle, namely running, installing, updating and developing a software. We then look at some attacks which exploit binaries whose behavior is similar to the normal usage scenarios.

The most common scenario is that of *running* a software. Normal usage typically involves loading binaries, reading/writing data files and possibly the creation of child processes. It is, however, not common for binaries to be created or modified. We highlight that binaries can come from various (3rd party) sources. For example, Windows Explorer (Windows GUI shell) loads 3rd party DLLs to provide shell extensions. The shell extensions are used to preview images or video, to add a control panel component, to customize file icons and context menus, etc. Another example is anti-virus software which forces its DLLs to be loaded by *all* processes, e.g. `avgrsstx.dll` from the AVG anti-virus software.

Installing software involves creating binaries and data files. Installers downloaded from the Internet are typically packed as self-extracting archives, which unpack themselves into a temporary directory and launch from there. This unpack-and-run behaviour is also observed in the normal running of some software, e.g. many `Sysinternals` tools unpack a kernel driver into a temporary directory and then load it.

There are a number of software *update* mechanisms. Microsoft Windows Update

employs a dedicated service (daemon process) to check, download and apply updates. In this case, the updater is completely separate from the software being updated. Other software, such as `Mozilla Firefox` and `Sun JDK`, employ a self-update mechanism. We use Firefox to illustrate the mechanism. Firefox checks online if there is a new version available and downloads the update as a compressed archive file. After downloading, Firefox invokes the updater as a helper process and terminates itself. The updater process reads the downloaded archive and updates `firefox.exe` and other files if necessary. When updating is finished, the updater re-launches Firefox (`firefox.exe`) and terminates itself. The helper process technique is usually used because in Windows, a binary cannot be written to when it is loaded by another process.

Software *development* means that binaries have to be created. For example, to debug a software using an IDE, the IDE firstly builds the program and then runs it. Thus the IDE creates or modifies binaries then executes them.

We now turn to some common attacks that involve binary loading or modification. Furthermore, these attacks use binary vulnerabilities in trusted programs. We highlight how the attacks behave in a similar fashion to the normal usage scenarios described earlier and the security mechanism needs to distinguish between normal usage versus an attack.

6.2.1.1 Running Unintended Executables

A common attack is to employ social engineering techniques to get the user to run a malware executable. However, the attack can be more subtle and exploit features of various applications to hide the executable. For example, by default, Windows Explorer hides file extensions. Consider a malware `postcard.jpg.exe` where the icon of the malware also looks like a photo. The user is then fooled into thinking it is a JPEG image and click it for viewing in Windows Explorer. The malware could execute its payload and then display the photo. Thus, the user would probably think this is just the normal display of a JPEG image. Notice that the behaviour of both a direct or file extension social engineering attack is similar to the regular software running scenario.

6.2.1.2 Application Data with Embedded Binaries

Malware binaries can be embedded within documents. A recent vulnerability in PDF readers [87] illustrates this attack where a legal PDF file containing an embedded executable. When the PDF file is viewed, the PDF viewer (e.g. `Acrobat`) also automatically executes the embedded executable. Notice that the behavior in the attack resembles the unpack-and-run in the normal installation scenarios.

6.2.1.3 DLL Attacks

DLL attacks typically exploit some feature of an application to inadvertently load a malicious DLL. The "carpet bomb" [10] attack involves bugs of two different browsers. Safari had a feature[7] which downloaded files onto the user's desktop automatically. Internet Explorer had a bug where certain DLLs are specified by the file name rather than a full pathname. By combining the behavior of Safari and Internet Explorer, a malicious website can execute arbitrary code on a machine that visits the site. It is surprising to see that this vulnerability where applications do not use a sufficiently qualified path when loading DLLs has recently re-surfaced (in Aug 2010) as "binary planting" attacks [62].

In some cases, an application has features to execute binaries. This can be exploited. A vulnerability [11] in handling Windows Help files in Internet Explorer was recently discovered. A malicious web page can cause Internet Explorer to load a malicious WinHelp (`.hlp`) file from an arbitrary path. WinHelp also has a feature to allow loading of arbitrary DLLs. The bug and the feature together enable a malicious website to execute arbitrary code on the client.

The recent Stuxnet worm [57] also uses a DLL attack. Interestingly, Windows 7 does not even give a UAC warning with this attack. Stuxnet exploits a vulnerability in Windows Explorer which displays an icon for shortcut files by using a DLL specified in the shortcut.

All above attacks load existing malicious DLLs which could be accidentally downloaded/copied or exist in a network drive. Once the DLL is in place, the behaviour of the attacks is similar to the regular software running scenario.

6.2.2 Related Work

6.2.2.1 Isolation Models

Isolation can be used to prevent malware from changing other parts of the system. A straight forward form of isolation is to have multiple independent systems or virtual machines (VM). This is useful for protecting one application against another though it might not protect against the PDF and WinHelp attacks discussed in Section 6.2.1 which are against the applications themselves. There is also the question of whether it is effectively feasible to run each software within its own VM. On the one hand, this by definition also prevents sharing which is incompatible with the use of implicit sharing by software in Windows and may be incompatible with system services and utilities. It also means a loss of usability since data is not sharable between applications. The isolation also leads to high maintenance cost of updating all VMs against vulnerabilities as there are multiple copies of the same software or library. On the other hand, using isolation

[7] Apple did not consider it as a bug.

separately on each software can also have significant system overheads since large numbers of virtual machines need to be run.

One-way isolation is a form of isolation which lets untrusted programs run in a sandbox where they can read from the trusted base system but modifications are confined to the sandbox. In one-way isolation, processes can be executed in isolation domains, other processes are executed normally in the base system. The semantics of file access is modified such that an isolated process can read files in the base system but when it tries to modify them, a private copy is duplicated from the base system and subsequent modification is applied on the private copy. Special care must be taken to handle file deletion so that files in the base system are not accessed.

Kato et al. proposed an one-way isolation system named SoftwarePot [47], whose main purpose is to allow software to securely circulate among different hosts. Secure circulation means that software in the sandbox should not interfere with the base system. A security policy specifies files visible to the sandbox, path mapping from base system to sandbox, system calls allowed and network addresses allowed to interact. Liang et al. proposed another one-way isolation system named Alcatraz [53]. The main difference between SoftwarePot and Alcatraz is that the latter enables modification from the sandbox to be *committed* to the base system.

The problems of one-way isolation are similar to that of virtual machines, namely, that isolation is the opposite of sharing. Other than that, one-way isolation protects the integrity of non-isolated system from attackers in the isolated system, but it does not protect the integrity of the isolated system. There are many forms of interactions between the sandbox and the base system, such as file system access, networking and IPC (including local sockets, pipes, process synchronization mechanisms, signals, window messages, etc.). The first two forms are easier to sandbox because their semantics are known. However, IPC is application dependent and can be complicated and undocumented. Even if only considering documented ones, it would take too much effort to implement filters for different IPC protocols (e.g. shared memory, pipe, CORBA, D-Bus, SunRPC etc.). SoftwarePot does not discuss IPC and Alcatraz simply disallows some forms of IPC. However, IPC is important and cannot be ignored. For example, the X window system, is implemented as local sockets; the Windows GUI requires (undocumented) IPC to the `smss.exe` and `csrss.exe` processes. Some services are implemented as IPC such as service management and domain name resolution in Windows, PulseAudio (the sound system in Linux), and syslog (the UNIX logging service). Simply allowing and denying them will lead to either insecure sandboxing or unusable software.

6.2.2.2 Signed Binaries

There are systems [43, 16, 101, 90, 66] that only allow signed binaries to be loaded or executed. In these systems, signature of binaries are either embedded in the files themselves or signed by a third party, such as the administrator and stored in a secure database. Executing or loading are only allowed if the signature can be verified.

There are three basic problems with signed binaries. Firstly, the security relies on the trust of the signing keys. If key is leaked, the trust is broken. For example, the Stuxnet worm [57] uses signed drivers with real certificates which are trusted by Windows. Secondly, revocation checking is expensive as it cannot be done locally and furthermore may not be timely. Lastly, we cannot assume that all software vendors sign their binaries. A more important difference is that most signed binary schemes are not intended to deal with the software lifecycle issues which we are concerned with. Another difference is that signed binaries only ensures that the binaries are not modified but may not prevent deletion.

Self-signed executables Self-signed executables [109] is a different approach to signed binaries which allows for software updates. In their proposal, all binaries are embedded with their signatures signed by the software vendor. Unlike the previous signed binary systems, the assumption is that any binary which has been signed can be trusted. Using the observation that binaries are usually updated by a newer version from the same vendor as the original, binary updating is allowed if the signature can be verified with the same public key.

The self-signed executables system binds paths with keys. Once bound, the key cannot be changed, even when the file is removed. An attacker can also sign malware as long as it is the first software installed with that path. Alternatively, this can constitute a denial of service attack since an attacker binding the path first would disallow future correct software to be installed at that path. As deleted files leave behind a stub, the number of stubs will monotonically increase over time. This can lead to yet another denial of service attack to exhaust storage.

In order to compare the signatures between the old and new copy, semantics of file writing has to be changed such that writing is applied on a shadow copy and later updated atomically. This can break software that assumes normal POSIX semantics.

6.2.3 The BinInt Security Model

In Section 6.2.2, we discussed a number of existing security models. While those models can be used to provide security for binaries, the main drawback is that they do not cover the usage spectrum across the software lifecycle nor do they handle all the typical attack scenarios mentioned. We now present a new security model for binaries which we call

BinInt. This model combines file integrity together with aspects of signed binaries and isolation to give a binary security mechanism which addresses usability in the software lifecycle. We remark that while we focus on Windows, this model has general applicability to other operating systems.

We are cognizant that a security mechanism should provide the desired form of protection in practice for the intended system usage. Full two-way isolation protects the host against malware in the VM but may not protect against other attacks. More importantly, it does not protect the software running in the VM from malware. This is why our model combines a number of features together to provide binary integrity. We focus solely on integrity and security of binaries as want to preserve compatibility with existing software (in Windows) while allowing implicit sharing. Our model should not be considered as a sole security mechanism and is meant to be used with other security mechanisms, e.g. ACLs or MAC, IDS, etc.

Our goals are three fold. Firstly, to prevent attacks exploiting vulnerabilities which load untrusted binaries. Secondly, to ensure integrity of binaries. This both prevents malware from being installed on the system and prevents denial of service attacks which delete binaries. Thirdly, we want to achieve a middle ground between usability and security. Any changes to system usage should not be onerous and existing software should (mostly) be able to run, including closed source binaries. While it is impossible to guarantee that all unmodified software can still function normally together with a security mechanism, our goal is that it should work in practice for typical software. Thus, frequent tasks, such as running and updating benign software, should be (mostly) transparent. Other infrequent tasks, such as software installation and removal, can be a little different but should not cause much inconvenience. As the mechanism should be applied to all processes, it should only have small overheads especially for frequent tasks.

We now describe our BinInt model. All processes are in one of the following execution modes: *d-mode* (default mode), *i-mode* (install mode) and *t-mode* (temporary trusted mode). Intuitively, d-mode corresponds to running already installed software, while i-mode is for installing/updating software. Special cases for running software which needs to dynamically create and load binaries but is not a software install are handled by t-mode, e.g. building and running binaries in an IDE or dynamic temporary binaries as in Sysinternals tools.

Each process and binary is associated with a *software domain*, which relates the process or binary to a particular installed software. The idea is that we may use the name of a particular software or the software vendor as the software domain. A special software domain $\perp$ denotes binaries which do not have a valid software domain. Binaries whose software domain is not $\perp$ are called *b-valid*.

Now we describe the BinInt model using the following rules. Rules 1–2 enforce loading of binaries; rules 3–6 enforce creation, modification, and deletion of binaries; and rules

7–8 define the execution modes for processes.

R1 A d-mode process cannot load b-invalid binaries, i.e. can load only b-valid binaries. Intuitively, this means that untrusted binaries cannot be loaded by normal processes.

R2 An i-mode or t-mode process can load all binaries.

R3 A d-mode or t-mode process cannot modify or delete b-valid binaries, i.e. can modify or delete only b-invalid binaries. Intuitively, this means that trusted binaries cannot be modified by normal processes.

R4 An i-mode process can modify or delete a b-valid binary if their software domains are the same. It can also modify or delete b-invalid binaries. Intuitively, this means that an installer or updater should update only its *own* binaries.

R5 A new binary created by d-mode or t-mode process is b-invalid.

R6 A new binary created by i-mode process is b-valid and the binary's software domain is the same as the process. This rule together with rule R5 means that only an installer should create binaries. Instead of completely preventing normal process from creating binaries, we choose to label those binaries b-invalid. This is because it is not possible to determine during file operations whether a file is a binary or not.

R7 When the system is booted, the initial process(es) run in d-mode. When a new process is created, the execution mode and software domain inherits from its parent process. There are two mechanisms to change the mode and domain. The first mechanism is *execution mode policy*, which maps the path of an EXE to a corresponding execution mode and software domain. When a new process is created, the path of the EXE is checked against the execution mode policy to find the execution mode and software domain. We require that the EXE should be b-valid. If no such mapping is found in the execution mode policy, it inherits the execution mode and software domain of its parent process. The execution mode policy is defined by a protected configuration file.

R8 The other mechanism changes the mode and domain of a running process by performing a special operation similar to a system call. In order to ensure that only the user with appropriate privileges can change the execution mode and software domain, the operation is authenticated with a password sent to the kernel. We have implemented a `sudo`-like utility called `modetrans` using this operation. `modetrans` takes the password from the user and executes a user-specified program in a user-specified execution mode and software domain. Although `modetrans` is a command-line program, a more user friendly GUI version can be implemented to integrate with the Explorer shell, so that the user can right-click an EXE and choose "run in install mode".

Both execution mode policy and `modetrans` are used to run an EXE in a specified execution mode and software domain. The execution mode policy is preferred if the EXE should always run in that execution mode and software domain. For example, since the Windows auto-updater `wuauclt.exe` should always run in i-mode and with "Microsoft" software domain, the execution mode policy should have an entry which maps `C:\windows\system32\wuauclt.exe` to i-mode and "Microsoft" software domain. `Modetrans` has an option to require that the EXE is b-valid before execution. This is useful when we want to guarantee that the installer is trusted before it is allowed to create new trusted binaries.

6.2.4 Implementation

We implemented BinInt in Windows XP. We have a kernel driver that intercepts binary loading, file modification and some other operations.

In Sec 6.2.3, we assume all file modifications are under the control of the operating system kernel. This assumption can be invalid in some cases. When the system mounts a network shared file system, (e.g. through SMB) an attacker can change the binaries outside the system. We call these unmonitorable binaries. An attacker can change local binaries when the system is offline, or change binaries in removable media when it is offline. To prevent these attacks, we use file signatures to detect modification. We keep signatures of all b-valid binaries in the *binary signature database*, which is similar to BinAuth. Information in the database includes path, signature, software group, modification history and other meta data. We also generate logs of how binaries were used. Log maintenance is done outside the kernel. The database can be used to by tools to analyze binary dependencies, trace the origin of a binary, etc.

For unmonitorable b-valid binaries, their signatures are updated immediately after they are updated and verified every time when their labels are accessed. For binaries that just come online, we verify the signatures once for each binary and cache the result. We adopt a lazy way of updating their signatures in order not to affect performance.

The kernel driver also maintains the signatures of binaries and labels of processes and binaries. When a binary f is loaded by a process p running in d-mode, f's needs to be checked whether it is b-valid. We intercept the native call `NtCreateSection`, which is a necessary step in binary loading. Checking whether f is b-valid is achieved by calculating the signature of the contents of f and match against the signature database, if f is loaded for the first time since boot or f is in an unmonitorable media. Otherwise, the software domain of f can be looked up using its path. Paths are normalized into the internal kernel path to disambiguate 8.3 filename, long file name and symbolic links. For NTFS, we use object ID to disambiguate hard links. If binary f is b-invalid, the loading will be denied, thus `NtCreateSection` fails.

The rest of the implementation deals other modes and maintaining integrity of binaries. If the process is in i-mode or t-mode, the checking will be bypassed and the loading will always be allowed. When a process p tries to open a b-valid binary f with write permission, (i.e. try to modify) the open operation is only allowed if p is in i-mode and the software domain of p and f are the same. As there is no distinguishing feature of a binary, other than its format, we need to immediately test whether or not the file is a binary by reading the file header which makes i-mode more costly than other modes. We also modify the semantics of Windows slightly so that files opened for writing in i-mode are in exclusive mode to simplify signature creation. We do not expect this to be a major restriction as the installer is likely to be creating files sequentially. When p closes the file handle of f, the file contents is now complete and f's signature will be re-computed and the signature database will be updated accordingly. The signature re-computation is done immediately if f is in un-monitorable media. Otherwise, it is lazily performed at any time before system shutdown since its only needed on next boot. Note that, to remove or rename a file in Windows, a process has to open the file with write permission first.

We do not prevent creation of new binaries, but new binaries created by processes in d-mode or t-mode will not be b-valid and thus cannot be loaded in d-mode.

Process creation is intercepted using the kernel API `PsSetCreateProcessNotifyRoutine` to inherit e-mode and s-domain to child processes. The systemcall-like `modetrans` operation is implemented as a IOCTL (I/O control).

A software installer may launch several helper programs from the main one. This is why child processes inherit their parent process' mode and domain. However, there are exceptions where installer processes are not child (or descendant) processes of the first installer process. MSI (Windows Installer) is a generic installation engine for installing and updating software on Windows. It is commonly used for Microsoft and non-Microsoft software. MSI makes use of a service (daemon) process to perform installation. The service process is always running and is not part of the process hierarchy of the original installer. We handle this by monitoring the communication channel, a named pipe `\Pipe\Net\NtControlPipeX`, between the installation process and the MSI service. When an i-mode process triggers the service to start installation, the service is switched to i-mode with the same domain as the triggering process. When the installation terminates, the service is switched back to d-mode. The MSI service is used exclusively so that there is no interference between concurrent requests.

6.2.5 Security Analysis

We analyse the security provided by BinInt model. We assume[8] the operating system kernel together with our BinAuth kernel driver is trusted.

Security of **d-mode**:
Most processes run in d-mode, thus its security is very important. The primary properties in d-mode are: b-invalid binaries cannot be loaded; b-valid binary cannot be modified; and binaries created are b-invalid. The consequence is that a d-mode process cannot introduce new binaries to d-mode processes including itself. This prevents download-and-run or download-and-load attacks. It is also not possible to delete d-valid binaries.

Security of **i-mode**:
A privilege escalation is required to transition from d-mode to i-mode using `modetrans`. Thus, malware cannot directly do this, i.e. our implementation requires user authentication. However, users might make mistakes or social engineering attacks can be used.

There are two cases to consider. If the malware is installed in a new software domain, the malware cannot modify existing b-valid binary, thus their integrity is assured. However, the malware can install new binaries which may be loaded into existing software as in the DLL hijacking attacks described in Section 6.2.1. The partial order extension can prevent this if the trust level of the malware is lower than the existing software. Similarly, the signing extension can also prevent this since the malware may not be able to sign the files or it might be denied by revocation or whitelist. We remark that as the binary signature database and logs describe binary usage and loading relationships, malicious behavior can be detected and also be removed more easily.

If the malware is installed in an existing domain, the same arguments apply. However, unlike the UAC privilege escalation mechanism in Windows, the binary database can be used to explain whether an executable is connected to the software in the existing domain.

Security of **t-mode**:
Similar to i-mode, t-mode is requires authentication for the privilege escalation. In terms of side effects, it behaves like d-mode, a t-mode process cannot modify b-valid binaries, thus a malicious t-mode process cannot introduce new binaries to d-mode processes. However, t-mode processes can load b-invalid binary. This enables them to introduce new binaries to themselves. However, if they want to persist in the system (survive after reboot), they have to lure the user to authenticate and run them in t-mode or i-mode, as they would not be able to execute in d-mode.

[8] The external monitoring shown in Chapter 4 can be used to detect some attacks when the assumption is false.

6.2.6 Evaluation

We now evaluate the system based on its usability in the software usage life cycle and its performance overhead.

6.2.6.1 Usability Evaluation

We tested our system on the main elements of the software usage life cycle – installation, running and uninstallation of the following software: Internet Explorer 8, Winamp (music player), Yahoo Messenger (instant messaging client), Firefox, Google Chrome, Adobe Acrobat Reader and Java Development Kit. We describe the observations from the binary database and logs for the software tested.

Internet Explorer 8 (IE8) tests Microsoft software installation. We choose the "Microsoft" software domain to install IE8 to on a Windows XP SP2 machine. No problem was observed during installing and running IE8. The Windows auto-updater handles the update of all Windows related software including IE8. We defined the updater, `wuauclt.exe`, in the execution mode policy to run in i-mode. Thus, it updates transparently.

The Winamp installer uses its own Nullsoft installer. No problem was observed during running and uninstalling Winamp. Yahoo Messenger uses a network-based install where the installer is a small initial installer which downloads a much larger installer. The installer tries to upgrade the Flash ActiveX plugin `flash.ocx` if it is not the latest version. This action is blocked because the software groups do not match. However, this is not really a problem as the Flash plugin can be updated separately. We noticed that a `YahooAUService.exe` service is created for auto-update. In order for the auto-update to work, we should add `YahooAUService.exe` to the execution mode policy to run in i-mode with the "Yahoo" s-domain.

No problem was observed during Firefox installation. An `updater.exe` in Firefox handles updates and also needs to be added to the execution mode policy. No problem was observed for Google Chrome. Adobe Acrobat Reader and Java Development Kit use the MSI engine which is handled transparently without any user interaction.

We have chosen software which cover a range of mechanisms for installation, uninstall, and update. We found that all the software life cycle aspects are usable with little effort needed and in some cases, completely transparently. Auto-updates such as Yahoo Messenger and Firefox have to be listed in the execution mode policy for them to work. This can be done manually immediately after installation if the user knows which program does the update. It can also be done at the first time the updater performs the updates. In this case, the user will be notified about the attempt to modify binaries and the information from the binary database is sufficient for understanding how to set the execution mode policy.

workload	environment	running time	overhead
build	base	59.0 ± 2.0s	
	d-mode	58.7 ± 1.8s	-0.5%
	t-mode	59.8 ± 1.3s	1.4%
	i-mode	60.3 ± 1.9s	2.2%
archive	base	41.4 ± 1.4s	
	d-mode	40.0 ± 2.2s	3.4%
	t-mode	42.2 ± 0.8s	2.0%
	i-mode	46.1 ± 1.1s	11.4%
javascript	base	1257.7 ± 21.1ms	
	d-mode	1257.3 ± 25.2ms	-0.0%
	t-mode	1249.5 ± 26.1ms	-0.7%
	i-mode	1247.9 ± 15.8ms	-0.8%

Table 6.2: Performance overhead

6.2.6.2 Performance Evaluation

The performance overhead of our system is caused by system call interception as described in Section 6.2.4. The objective of the performance evaluation benchmark is to test the overhead of the modes with the base system, a vanilla unmodified Windows XP SP3. The machine is a VMWare virtual machine running Intel Core 2 Duo 2.33GHz CPU (with one core allocated to VMWare) and 512MB memory. We use the following test cases: software build (building `putty`, a SSH client, with 86 C files) using Visual Studio command line interface, WinRAR archive extraction (extracting the Linux kernel source code), and the SunSpider JavaScript Benchmark in Firefox. The three test cases are chosen to test process creation, I/O operations and a CPU workload respectively as these are the main sources of overheads in our prototype. Each test case is performed in the base system (without our system), d-mode, t-mode and in i-mode. Each test case in each environment is performed 5 times to calculate the mean and standard deviation.

Table 6.2 shows the result of the tests. Except for the archive extraction test in i-mode, all the average performance overheads are small. In most cases, the overhead is smaller than the standard deviation[9]. While individual runs have some small differences, we conclude that with the exception of the archive test, the performance can be considered to be similar to the base system.

The 11.4% overhead of the i-mode archive case shows the additional costs of i-mode when the workload is mostly file writing. Although the extracted files are not binaries, the system has to check whether or not they are binaries. Since software installation usually happens infrequently, we are less concerned with efficiency of i-mode as long as it is still reasonable.

[9]This is caused by environmental factors such as caches, etc., and in some others it is even negative (slightly faster)

The performance overhead of BinInt is in general higher than BinAuth, because BinInt pays additional cost of maintaining and updating the binary signature database, whereas BinAuth has a static database. In addition, BinInt needs to track the execution modes and software domains of processes and files.

6.2.7 Conclusion

We have presented a binary security model which caters to the dynamic use of binaries within the software life cycle while protecting against attacks in default mode and giving isolation between software domains in install mode. Our prototype is efficient and usable while protecting a broad range of binary loading/execution mechanisms in Windows. We found our system to be mostly transparent in usage on typical Windows software throughout its software lifecycle. Thus, BinInt is a practical solution which gives a good tradeoff between usability and security to protect binaries on Windows. Comparing to UAC, we believe that our model provides better protection especially for user-owned binaries, while having better user experience in terms of interference (security prompt). Our model can also be combined with other security mechanisms such as file system access control and user privilege separation.

Chapter 7

Conclusion

This chapter concludes this thesis. We summarize the contributions of this thesis in Section 7.1, and discuss some future directions in Section 7.2.

7.1 Summary of the Thesis

In this thesis, we propose our monitoring frameworks: LBox and WinResMon. The information collected by them can be used to solve many problems including software comprehension, fault diagnosis and system security. Based on the frameworks, we developed two visualizations: module dependency visualization and `lviz`. As the frameworks are resource-based, we can easily adapt them to control the access of resources in system and apply them on system security. Our binary integrity system prevents loading of untrusted binary by monitoring file access in the operating system. To detect malware when the operating system kernel is possibly compromised, we correlate information gathered from external sensors and network routers.

LBox We show our user-level monitoring framework LBox. User-level monitoring differs from traditional super user operated monitoring in that user-level monitoring can be safely used by all users without worrying about confidentiality and denial-of-service problems. LBox also allows cascading of monitors. Comparing state of the art monitoring frameworks such as Solaris DTrace and SystemTap, LBox does not allow user supplied script to run in kernel space. This reduces the complexity and more importantly reduces the chances of introducing security bugs of compromising the kernel. To compensate the flexibility of in-kernel script, LBox provides a fine-grained event filtering API. Our experiments are very encouraging showing that the overhead is comparable to in-kernel mechanisms such as DTrace.

WinResMon We presented the motivation, design, implementation and usage of WinResMon. Its main use is to inspect resource access and software dependency issues in Microsoft Windows environments. As WinResMon is extensible, system administrators can also build tools using WinResMon for custom queries and system analysis. Benchmarking shows that WinResMon is reliable and is comparable to other popular tools.

External Monitoring When the operating system kernel is compromised, information collected from the host cannot be trusted. We propose a framework that incorporates external information from sensors in securing host computers. Since the sensors are external, they are difficult to be accessed and tampered by a compromised host. The framework can be applied to detect malware and also mitigate the impact of a compromised host. Our experiments show that our framework can successfully detect three types of malware: email spammer, DDoS zombie and password cracking zombie. The framework also takes advantage of the growing popularity of pervasive computing and sensor networks which make it feasible for cost-effective deployment in the near future.

Module Dependency Visualization We have demonstrated that software dependencies in Windows are quite complex even when looking at the coarse grain level of software components packaged into binaries and executables. Our module dependency visualization gives an effective way of extracting and visualizing the software dependencies and interactions between binaries. We show that even with the complexity of actual Windows software, it is possible to analyse whole system interaction, understand how modules are used and shared, and also discover potential unexpected or unusual interactions/modules. Such an understanding is also useful for software developers, system administrators and also users, to manage the software ecosystem on Windows and to deal with the problems which arise from module updates and potential "DLL hell" repercussions.

LViz We present a DotPlot-based visualization for studying execution traces. We use examples to show that LViz is flexible to look at a wide range of problems. As the traces can be very large, LViz is implemented to be efficient and responsive for interactive use. We have developed the VDP tool for Windows because such visualizations are more beneficial given the closed source nature and system complexity of Windows. However, VDP is not reliant on a particular monitoring infrastructure and the same visualization ideas can be applied to other systems with a rich source of execution traces, e.g. Unix.

BinAuth We performed a study on the feasibility of mandatory software component authentication. Our approach uses a lightweight authentication technique using message authentication code. According to our benchmarks, we found that using a cache reduces the overhead as it reduces the number of repeated authentication requests. Based on

these benchmarks, we can conclude that our mandatory authentication can be added to Windows without significant overhead. We have also introduced the concept of "software naming". The idea behind this is to uniquely identify a file. We have mentioned a few applications for this idea. Given the security benefits of binary authentication and software naming, and only small overhead incurred even on complex OS like Windows, we envisage that the presented results can better convince the OS and user community alike to start deploying them more universally in realizing secure software distribution and execution.

BinInt Based on BinAuth, we propose BinInt, a security model which caters to the dynamic use of binaries within the software life cycle while protecting against attacks in default mode and giving isolation between software domains in install mode. Our prototype is efficient and usable while protecting a broad range of binary loading/execution mechanisms in Windows. We found our system to be mostly transparent in usage on typical Windows software throughout its software lifecycle. Thus, BinInt is a practical solution which gives a good tradeoff between usability and security to protect binaries on Windows. Our model can also be combined with other security mechanisms.

7.2 Future Work

In future work of monitoring infrastructures include functional extensibility and performance optimization. More types of events such as internal state changes and low level hardware operations (hard drive, keyboard and mouse) can be included. As multi-core CPUs becoming more common, we can optimize the implementation to make use of them. We can apply our framework in the cluster or cloud computing setting, and have query API to aggregate information from multiple hosts. Log compression can benefit from the high redundancy of the logs. We can have an operating system independent abstraction which make it easier to port for future operating systems.

Our binary integrity model, BinInt, focuses on binaries only. However, securing only binaries may not be sufficient. For example, the attacker can modify Java class files to change the behaviour of java programs. The same applies to shell scripts and configuration files. We can generalize BinInt to protect the integrity of any kind of file. The difficulty is that data files are much more dynamic than binaries. Unlike binaries which are only changed during install, update and removal of the software, data files are usually changed much more frequently. This makes direct use of d-mode for data files unusable. More fine grained policies can be applied for data files.

Bibliography

[1] Bootvis. http://www.techrepublic.com/article/step-by-step-use-bootvis-to-improve-xp-boot-performance/5034622.

[2] Easysen SBT80 product page. http://www.easysen.com/SBT80.htm.

[3] FBI investigates allegations webcam used to monitor student. http://articles.cnn.com/2010-02-20/justice/laptop.suit_1_webcam-district-court-laptop.

[4] Filemon for Windows. http://technet.microsoft.com/en-us/sysinternals/bb896642.

[5] The myth of the four-minute windows survival time. http://www.edbott.com/weblog/?p=2071.

[6] Process and thread manager routines. http://msdn.microsoft.com/en-us/library/ff559917.aspx.

[7] Regmon for Windows. http://technet.microsoft.com/en-us/sysinternals/bb896652.

[8] Unpatched PC "survival time" just 16 minutes. http://www.informationweek.com/news/showArticle.jhtml?articleID=29106061.

[9] Windows NT system-call hooking. http://www.ddj.com/184410109.

[10] Safari carpet bomb. http://www.oreillynet.com/onlamp/blog/2008/05/safari_carpet_bomb.html, 2008.

[11] CVE-2010-0483. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0483, 2010.

[12] L.V. Ahn, M. Blum, N.J. Hopper, and J. Langford. CAPTCHA: Using hard AI problems for security. In *Proceedings of the 22nd International Conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT'03)*, pages 294–311. Springer-Verlag, 2003.

[13] A.V. Aho and M.J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[14] K. Alkio. Mbr rootkit, a new breed of malware. `http://www.f-secure.com/weblog/archives/00001393.html`, 2008.

[15] R. Anderson. The end of DLL hell. *MSDN Magazine*, 2000.

[16] A. Apvrille, D. Gordon, S. Hallyn, M. Pourzandi, and V. Roy. DigSig: Runtime authentication of binaries at kernel level. In *Proceedings of the 18th USENIX Conference on Large Installation System Administration (LISA'04)*, pages 59–66. USENIX Association, 2004.

[17] C.A. Ardagna, M. Cremonini, E. Damiani, S.D.C. di Vimercati, and P. Samarati. Supporting location-based conditions in access control policies. In *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security (ASIACCS'06)*, pages 212–222. ACM, 2006.

[18] E. Arnold. The trouble with tripwire: Making a valuable security tool more efficient. `http://www.securityfocus.com/infocus/1398`, 2001.

[19] D.F. Bacon, P. Cheng, and D. Grove. Tuningfork: a platform for visualization and analysis of complex real-time systems. In *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming, Systems and Applications Companion (OOPSLA'07)*, pages 854–855. ACM, 2007.

[20] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, pages 18–18. USENIX Association, 2004.

[21] M. Basseville and I.V. Nikiforov. *Detection of abrupt changes: theory and application*. Prentice Hall, 1993.

[22] S. Bhattacharya. Dynamic probes — debugging by stealth. In *Proceedings of Linux.Conf.Au*, 2003.

[23] K.J. Biba. Integrity considerations for secure computer systems. Technical report, MITRE CORP BEDFORD MA, 1977.

[24] P. Bodik, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A. Fox, M.I. Jordan, et al. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *Proceedings of the Second International Conference on Autonomic Computing (ICAC'05)*, pages 89–100. IEEE Computer Society, 2005.

[25] D.L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[26] B.M. Cantrill, M.W. Shapiro, and A.H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'04))*, pages 2–2. USENIX Association, 2004.

[27] CERT. Cert statistics. `http://www.cert.org/stats/`.

[28] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 41(3):1–58, 2009.

[29] E.C. Chang, L. Lu, Y. Wu, R.H.C. Yap, and J. Yu. Enhancing host security using external environment sensors. In *Proceedings of the 6th International ICST Conference on Security and Privacy in Communication Networks (SecureComm'10)*, volume 50, pages 362–379. Springer, 2010.

[30] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J.J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC'07)*, pages 49–58. IEEE Computer Society, 2007.

[31] W. Cui, R.H. Katz, and W. Tan. Design and implementation of an extrusion-based break-in detector for personal computers. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 361–370. IEEE Computer Society, 2005.

[32] D. Eastlake and T. Hansen. US secure hash algorithms (SHA and HMAC-SHA). Technical report, RFC 4634, July, 2006.

[33] S.G. Eick and P.J. Lucas. Displaying trace files. *Software Practice and Experience*, 26(4):399–409, 1996.

[34] S.G. Eick, M.C. Nelson, and J.D. Schmidt. Graphical analysis of computer log files. *Communications of the ACM*, 37(12):50–56, 1994.

[35] J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull. Graphviz — open source graph drawing tools. In *Graph Drawing*, pages 594–597. Springer, 2002.

[36] M. Ewing and E. Troan. The RPM packaging system. In *Proceedings of the First Conference on Freely Redistributable Software*, 1996.

[37] R. Faulkner and R. Gomes. The process file system and process model in UNIX System V. In *Proceedings of the 1991 Winter USENIX Conference*, 1991.

[38] J. Finke. Process monitor: Detecting events that didn't happen. In *Proceedings of The 16th Large Installation Systems Administration Conference (LISA'02)*, pages 145–153, 2002.

[39] J. Foote. Visualizing music and audio using self-similarity. In *Proceedings of the seventh ACM International Conference on Multimedia*, pages 77–80. ACM, 1999.

[40] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS'03)*, 2003.

[41] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is not transparency: VMM detection myths and realities. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems (HotOS'07)*, page 6. USENIX Association, 2007.

[42] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. Bothunter: Detecting malware infection through ids-driven dialog correlation. In *Proceedings of 16th USENIX Security Symposium*, page 12. USENIX Association, 2007.

[43] F. Halim, R. Ramnath, Y. Wu, and R. Yap. A lightweight binary authentication system for windows. *Trust Management II*, pages 295–310, 2008.

[44] K. Ingham and S. Forrest. A history and survey of network firewalls. *University of New Mexico, Tech. Rep*, 2002.

[45] J.P. John, A. Moshchuk, S.D. Gribble, and A. Krishnamurthy. Studying spamming botnets using botlab. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NDSI'09)*, pages 291–306. USENIX Association, 2009.

[46] D. Kaminsky. Black ops 2006. *Blackhat USA*, 2006.

[47] K. Kato and Y. Oyama. Softwarepot: An encapsulated transferable file system for secure software circulation. *Software Security – Theories and Systems*, pages 217–224, 2003.

[48] G.H. Kim and E.H. Spafford. The design and implementation of tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security (CCS'94)*, pages 18–29. ACM, 1994.

[49] H. Krawczyk, M. Bellare, and R. Canetti. RFC2104: HMAC: Keyed-hashing for message authentication. *Internet RFCs*, 1997.

[50] S. Kumar, T. Sim, R. Janakiraman, and S. Zhang. Using continuous biometric verification to protect interactive login sessions. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 441–450. IEEE Computer Society, 2005.

[51] G. Kwang, R. Yap, T. Sim, and R. Ramnath. An usability study of continuous biometrics authentication. *Advances in Biometrics*, pages 828–837, 2009.

[52] G. Lefebvre, B. Cully, M.J. Feeley, N.C. Hutchinson, and A. Warfield. Tralfamadore: unifying source code and execution experience. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys'09)*, pages 199–204. ACM, 2009.

[53] Z. Liang, W. Sun, VN Venkatakrishnan, and R. Sekar. Alcatraz: An isolated environment for experimenting with untrusted software. *ACM Transactions on Information and System Security (TISSEC)*, 12(3):1–37, 2009.

[54] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.

[55] J.V. Maizel and R.P. Lenk. Enhanced graphic matrix analysis of nucleic acid and protein sequences. *Proceedings of the National Academy of Sciences of the United States of America*, 78(12):7665, 1981.

[56] P. Manadhata and J.M. Wing. An attack surface metric. In *Proceedings of the USENIX Security Workshop on Security Metrics (MetriCon'06)*. USENIX Association, 2006.

[57] A. Matrosov, E. Rodionov, D. Harley, and J. Malcho. Stuxnet under the microscope. `http://ece.wpi.edu/~dchasaki/papers/Stuxnet_Under_the_Microscope.pdf`, 2010.

[58] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy, and M. Hiramatsu. Probing the guts of KProbes. In *Proceedings of the 2006 Linux Symposium*, 2006.

[59] Microsoft. Authenticode. `http://technet.microsoft.com/en-us/library/cc750035.aspx`.

[60] Microsoft. Sigcheck download page. `http://technet.microsoft.com/en-us/sysinternals/bb897441`.

[61] Microsoft. Signtool. `http://msdn.microsoft.com/en-us/library/aa387764.aspx`.

[62] Microsoft. Binary planting attacks. `http://www.microsoft.com/technet/security/advisory/2269637.mspx`, 2010.

[63] R.J. Moore. A universal dynamic trace for Linux and other operating systems. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 297–308. USENIX Association, 2001.

[64] S. Motiee, K. Hawkey, and K. Beznosov. Do windows users follow the principle of least privilege?: investigating user account control practices. In *Proceedings of the Sixth Symposium on Usable Privacy and Security (SOUPS'10)*. ACM, 2010.

[65] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement (ESEM'07)*, pages 364–373. IEEE Computer Society, 2007.

[66] S. Nanda, W. Li, L.C. Lam, and T. Chiueh. Foreign code detection on the windows/x86 platform. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 279–288. IEEE Computer Society, 2006.

[67] G. Nebbett. *Windows NT/2000 native API reference*. Sams, 2000.

[68] N. Nethercote and J. Seward. Valgrind:: A program supervision framework. *Electronic notes in theoretical computer science*, 89(2):44–66, 2003.

[69] Oracle. *System Administration Guide: Security Services. Part IV: Auditing and Device Management*.

[70] E.S. Page. Continuous inspection schemes. *Biometrika*, 41(1/2):100–115, 1954.

[71] G.J. Popek and R.P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[72] G. Post and A. Kagan. The use and effectiveness of anti-virus software. *Computers & Security*, 17(7):589–599, 1998.

[73] V. Prasad, W. Cohen, F.C. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. In *Proceedings of the 2005 Ottawa Linux Symposium*, volume 2, pages 49–64, 2005.

[74] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 18–18. USENIX Association, 2003.

[75] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting system emulators. In *Proceedings of the 10th International Conference on Information Security (ISC'07)*, Lecture Notes in Computer Science, pages 1–18. Springer, 2007.

[76] R. Ramnath, S. Sufatrio, R.H.C. Yap, and W. Yongzheng. WinResMon: a tool for discovering software dependencies, configuration and requirements in Microsoft Windows. In *Proceedings of the 20th Conference on Large Installation System Administration (LISA'06)*, pages 175–186. USENIX Association, 2006.

[77] M. Salah and S. Mancoridis. A hierarchy of dynamic software views: From object-interactions to feature-interactions. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 72–81. IEEE Computer Society, 2004.

[78] M. Salah, S. Mancoridis, G. Antoniol, and M. Di Penta. Scenario-driven dynamic analysis for comprehending large software systems. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 71–80. IEEE Computer Society, 2006.

[79] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[80] C. Schiller, S. Fogie, C. DeRodeff, and M. Gregg. *Infosecurity 2008 threat analysis.* Syngress Publishing, 2007.

[81] M. Schmid, F. Hill, AK Ghosh, and JT Bloch. Preventing the execution of unauthorized win32 applications. *DARPA Information Survivability Conference and Exposition*, 2:1175, 2001.

[82] Panda Security. Six percent of computers scanned by Panda Security are infected by the Conficker worm. `http://press.pandasecurity.com/news/six-percent-of-computers-scanned-by-panda-security-are-infected-by-the-conficker-worm/`, 2009.

[83] A. Siraj, R.B. Vaughn, and S.M. Bridges. Intrusion sensor data fusion in an intelligent intrusion detection system architecture. In *Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04)*, volume 9. IEEE Computer Society, 2004.

[84] Skywing. Anti-virus software gone wrong. `http://uninformed.org/index.cgi?v=4&a=4`, 2006.

[85] M. Slaviero, J. Kroon, and M.S. Olivier. Attacking signed binaries. In *Proceedings of the 5th Annual Information Security South Africa Conference (ISSA'05)*, 2005.

[86] A. Srivastava, J. Thiagarajan, and C. Schertz. Efficient integration testing using dependency analysis. *Microsoft Research, TechReport MSR-TR-2005-94*, 2005.

[87] D. Stevens. Escape from PDF. `http://blog.didierstevens.com/2010/03/29/escape-from-pdf`, 2010.

[88] Sufatrio, R.H.C. Yap, and L. Zhong. A machine-oriented integrated vulnerability database for automated vulnerability detection and processing. In *Proceedings of the 18th USENIX Large Installation Systems Administration Conference (LISA'04)*, pages 47–58, 2004.

[89] C. Thomas and N. Balakrishnan. Improvement in intrusion detection with advances in sensor fusion. *IEEE Transactions on Information Forensics and Security*, 4(3):542–551, 2009.

[90] L. van Doorn, G. Ballintijn, and W.A. Arbaugh. Signed executables for Linux. Technical Report CS-TR-4259, University of Maryland, 2001.

[91] C. Verbowski, E. Kiciman, A. Kumar, B. Daniels, S. Lu, J. Lee, Y.M. Wang, and R. Roussev. Flight data recorder: monitoring persistent-state interactions to improve systems management. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, pages 117–130. USENIX Association, 2006.

[92] S. Voigt, J. Bohnet, and J. Dollner. Object aware execution trace exploration. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'09)*, pages 201–210. IEEE Computer Society, 2009.

[93] D.A. Wagner. *Janus: an approach for confinement of untrusted applications.* PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California, 1999.

[94] H. Wang, D. Zhang, and K.G. Shin. Detecting SYN flooding attacks. In *Proceedings of the 21st Annual IEEE International Conference on Computer Communications (INFOCOM'02)*, 2002.

[95] Q. Wang, W. Wang, R. Brown, K. Driesen, B. Dufour, L. Hendren, and C. Verbrugge. Evolve: an open extensible software visualization framework. In *Proceedings of the 2003 ACM Symposium on Software Visualization (SoftVis'03)*, pages 37–ff. ACM, 2003.

[96] X. Wang and H. Yu. How to break MD5 and other hash functions. *Proceedings of the 24nd International Conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT'05)*, pages 19–35, 2005.

[97] 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 *Proceedings of the 18th USENIX Conference on Large Installation System Administration (LISA'04)*, pages 33–46. USENIX Association, 2004.

[98] C. Wee. LAFS: A logging and auditing file system. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'95)*, pages 231–240, 1995.

[99] N. Wilde, R. Huitt, and S. Huitt. Dependency analysis tools: reusable components for software maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM'89)*, pages 126–131. IEEE Computer Society, 1989.

[100] M. Wilhelm and S. Diehl. Dependency viewer — a tool for visualizing package design quality metrics. In *Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'05)*, page 34. IEEE Computer Society, 2005.

[101] M.A. Williams. Anti-trojan and trojan detection with in-kernel digital signature testing of executables. `http://www.net-security.org/dl/articles/sigexec.pdf`, 2002.

[102] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the Linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, pages 17–31. USENIX Association, 2002.

[103] Y. Wu, Sufatrio, R.H.C. Yap, R. Ramnath, and F. Halim. Establishing software integrity trust: A survey and lightweight authentication system for windows. In Z. Yan, editor, *Trust Modeling and Management in Digital Environments: from Social Concept to System Development*, chapter 3, pages 78–100. IGI Global, 2009.

[104] Y. Wu and R.H.C. Yap. A user-level framework for auditing and monitoring. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 95–105. IEEE Computer Society, 2005.

[105] Y. Wu and R.H.C. Yap. The problem of usable binary authentication. In *Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement Companion (SSIRI'10)*, pages 34–35. IEEE Computer Society, 2010.

[106] Y. Wu and R.H.C. Yap. Towards a binary integrity system for Windows. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS'11)*, pages 503–507. ACM, 2011.

[107] Y. Wu, R.H.C. Yap, and F. Halim. Visualizing Windows system traces. In *Proceedings of the 5th International Symposium on Software Visualization (SoftVis'10)*, pages 123–132. ACM, 2010.

[108] Y. Wu, R.H.C. Yap, and R. Ramnath. Comprehending module dependencies and sharing. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, volume 2, pages 89–98. ACM, 2010.

[109] G. Wurster and PC van Oorschot. Self-signed executables: Restricting replacement of program binaries by malware. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Security (HotSec'07)*, page 8. USENIX Association, 2007.

[110] R.H.C. Yap, T. Sim, G.X.Y. Kwang, and R. Ramnath. Physical access protection using continuous authentication. In *Proceedings of the IEEE Conference on Technologies for Homeland Security (HST'08)*, pages 510–512. IEEE Computer Society, 2008.

[111] T.F. Yen and M. Reiter. Traffic aggregation for malware detection. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 207–227, 2008.

[112] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 531–540. ACM, 2008.