Design and Implementation of a Hypervisor-Based Platform for Dynamic Information Flow Tracking in a Distributed Environment

Andrey Ermolinskiy
Scott Shenker

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-50
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-50.html

May 12, 2011

Design and Implementation of a Hypervisor-Based Platform for Dynamic Information Flow Tracking in a Distributed Environment

by

Andrey Ermolinskiy

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Scott Shenker, Chair
Professor Ion Stoica
Professor Deirdre Mulligan

Spring 2011

Design and Implementation of a Hypervisor-Based Platform for Dynamic Information Flow Tracking in a Distributed Environment

by

Andrey Ermolinskiy

Abstract

Design and Implementation of a Hypervisor-Based Platform for Dynamic Information Flow Tracking in a Distributed Environment

by

Andrey Ermolinskiy

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Scott Shenker, Chair

One of the central security concerns in managing an organization is protecting the flow of sensitive information, by which we mean either maintaining an audit trail or ensuring that sensitive documents are disseminated only to the authorized parties.

A promising approach to securing sensitive data involves designing mechanisms that interpose at the software-hardware boundary and track the flow of information with high precision — at the level of bytes and machine instructions. Fine-grained information flow tracking (IFT) is conceptually simple: memory and registers containing sensitive data are tagged with taint labels and these labels are propagated in accordance with the computation. However, previous efforts have demonstrated that full-system IFT faces two major practical limitations — enormous performance overhead and taint explosion. These challenges render existing IFT implementations impractical for deployment outside of a laboratory setting.

This dissertation describes our progress in addressing these challenges. We present the design and implementation of PIFT (for Practical Information Flow Tracking) — a hypervisor-based IFT platform that achieves substantial performance improvements over previous systems and largely eliminates the problem of kernel taint explosion. PIFT takes advantage of spare CPU cores to track the flow of information asynchronously and in parallel with the primary instruction stream.

To the best of our knowledge, PIFT is the most efficient full-system IFT platform available at the time of writing and is the only implementation that supports real-time tracking of information flow in graphical desktop environments.

Dedicated to my parents Ludmila and Alexandr, to my brother Pavel, and to my loving wife Olga.

Contents

List of Figures

List of Tables

Acknowledgements

My time as a graduate student at UC Berkeley was an amazing, once-in-a-lifetime journey and I would like to express my sincere gratitude to those who have made this journey possible.

First and foremost, I am greatly indebted to my academic advisor and thesis committee chair, Scott Shenker, for without him this dissertation would have been a distant dream. His mentorship and unique insights were an immense asset to my research and I am thankful for his unwavering support, guidance, and patience. For six years, Scott has guided me towards both questions and answers and his influence on me will last for many years to come. I am truly fortunate to have had the opportunity to work with and learn from him.

I am grateful to my dissertation committee members, Ion Stoica and Deirdre Mulligan, for their insightful comments and invaluable guidance on my research. I am sincerely thankful to Dawn Song, whose feedback and critical comments during my qualifying exam, as well as numerous prior discussions, have strengthened the technical content of this dissertation.

I am also sincerely grateful to Sachin Katti, whose ideas and perspectives have ignited in me the initial sparks of interest in information flow tracking. Sachin was instrumental to formulating the core research direction of PIFT and defining the initial set of technical objectives. Most of the ideas and mechanisms presented in this dissertation were nurtured through in-depth technical conversations with Sachin and his efforts to improve the technical quality and presentation of the numerous paper drafts were simply invaluable.

Several other individuals have contributed their time, efforts, and insights to this project at various stages. I am particularly thankful to Lisa Fowler and Murphy McCauley for their work on the analysis of taint explosion and the assessment of PIFT's performance properties in interactive graphical environments. The usability study presented in Section 5.3 was conducted by Lisa using a set of measurement tools that she has developed in collaboration with Murphy. Lisa has also contributed towards the understanding of taint explosion and some of her initial results are presented in Section 6.1.

During my graduate studies, I also participated in several collaborative research projects that are beyond the scope of this dissertation. I would like to express my sincerest thanks to Daekyeong Moon, Byung-Gon Chun, Teemu Koponen, Mohit Chawla, Kye Hyun Kim, Ion Stoica, and Sylvia Ratnasamy for these wonderful collaboration opportunities. I greatly enjoyed working with John Kubiatowicz as a teaching assistant for CS162. Kubi was an amazing mentor and I was always inspired by his extraordinary dedication, attention to detail, and enthusiasm for teaching.

I would also be remiss to leave out my RADLab colleagues and friends, as I cannot imagine what my time at Berkeley would have been like without them. In addition to those already listed, I would like to thank Igor Ganichev, Brighten Godfrey, Dilip Joseph, Junda Lai, Arsalan Tavakoli, and Matei Zaharia. Many of them have provided valuable feedback on my ideas and their comments have been useful in refining my research and publications.

I had an opportunity to work with several brilliant mentors during my internships with Intel Research (Berkeley), IBM Research (Almaden), and Qualcomm (Bay Area R&D). I would like to express my sincere gratitude to Sylvia Ratnasamy, Petros Maniatis, Renu Tewari, Frank Schmuck, Roger Haskin, Pablo Montesinos, Calin Cascaval, and Chris Vick for making my internships enjoyable, fun, and rewarding research experiences. I would also like to thank Patrick Kavanagh and his colleagues at Banyan for helping me expand my horizons, teaching me about the world, and encouraging me to explore diverse career paths.

I would like to thank Lola Zyscovich, Garin Lucy Ekmekjian, and Masha Shantie, whose support, companionship, and laughter have encouraged me to move forward and kept me sane.

I am blessed to have a loving and supportive family that has taught me the importance of being true to myself while pursuing excellence. I am eternally grateful to my parents, Ludmila Ermolinskaya and Alexandr Ermolinski, who taught me the value of education and instilled in me the desire to strive for the best. They, along with the rest of my family, have been the pillars of support throughout my Ph.D journey and I am deeply indebted to them for their love and unwavering faith in me. They were happy even at my small successes and supported me during difficult times. I am deeply grateful to my brother, Pavel Ermolinski, who has been a steady source of companionship, assistance, patience, and support at each step of the journey, throughout the peaks and troughs. Finally, I am grateful from the bottom of my heart to my beautiful and loving wife, Olga Avramenko, who has stood by me and supported me in countless ways. She has graced my life with her gentle presence, her patience, and her acceptance of my countless flaws. Her love, care, and unwavering support were instrumental to the success of this dissertation. Olya, you are the wind beneath my wings and this flight would not have been possible or meaningful without you. And your lemon cake is delicious!

Curriculum Vitæ

Andrey Ermolinskiy

Education

University of California at Berkeley (September 2005 - May 2011, *Berkeley, CA*)

Ph.D. in Computer Science

Research advisor: Professor Scott Shenker

Dissertation committee members:

Professor Scott Shenker (Chair)

Professor Ion Stoica

Professor Deirdre Mulligan

Major field of study: Distributed Systems and Networking

Minor field of study: Database Management Systems

University of California at Berkeley (September 2007 - June 2009, *Berkeley, CA*)

Certificate in Management of Technology

Haas School of Business

University of California at Berkeley (September 2005 - May 2007, *Berkeley, CA*)

M.S. in Computer Science

Research advisor: Professor Scott Shenker

Princeton University (September 1998 - May 2002, *Princeton, NJ*)

B.S.E. in Computer Science

High Honors academic recognition

Elected to the Society of Sigma XI for excellence in research

Research Focus

Broadly, my research interests lie in the areas of hypervisor-based security, enterprise data management, high-performance distributed storage systems, and Internet architecture.

My dissertation investigates Practical Information Flow Tracking (PIFT) — a novel hypervisor-driven information security architecture for enterprise environments. PIFT enables organizations to track the movement of confidential information, enforce dissemination restrictions, and protect sensitive documents from accidental disclosure to unauthorized parties. PIFT relies on fine-grained (instruction-level) information flow tracking techniques and achieves efficiency through asynchronous parallel execution. Unlike numerous prior efforts in this area, PIFT requires no changes to applications or the operating system.

In [31], I investigated the problem of concurrent access coordination in shared-disk parallel applications. I conceived and led the development of a novel synchronization primitive that lifts the fundamental safety and liveness limitations associated with traditional approaches based on conservative distributed locking.

In [33], I proposed and implemented a cooperative peer-to-peer data caching scheme for parallel and widely-distributed computational environments.

My earlier work focused on developing and evaluating robust protocols for inter-domain routing [32, 30, 75] and exploring clean-slate approaches to Internet architecture [46].

Selected Publications

"Towards Practical Information Flow Tracking" A. Ermolinskiy, S. Katti, S. Shenker, L. Fowler, M. McCauley. In submission (*December 2010*).

"Minuet: Rethinking Concurrency Control in Storage Area Networks" A. Ermolinskiy, D. Moon, B. G. Chun, S. Shenker. Proceedings of FAST'09 (*February 2009*).

"c2cfs – a Collective Caching Architecture for Distributed File Access" A. Ermolinskiy, R. Tewari. Proceedings of NSDM'09 (*June 2009*).

"S3 – Securing Sensitive Stuff"' S. Katti, A. Ermolinskiy, M. Casado, S. Shenker, H. Balakrishnan. OSDI'08 Work-in-Progress (WiP) report (*December 2008*).

"Reducing Transient Disconnectivity using Anomaly-Cognizant Forwarding" A. Ermolinskiy, S. Shenker. Proceedings of ACM HotNets-VII (*October 2008*).

"The Design and Implementation of Free Riding Multicast" A. Ermolinskiy. Master's Report (*May 2007*).

"A Data-Oriented (and Beyond) Network Architecture" T. Koponen, M. Chawla, B. G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, I. Stoica. Proceedings of ACM SIGCOMM'07 (*August 2007*).

"Revisiting IP Multicast" S. Ratnasamy, A. Ermolinskiy, S. Shenker. Proceedings of ACM SIGCOMM'06 (*September 2006*).

"Disaster Recovery with General Parallel File System" A. Ermolinskiy. IBM technical whitepaper (*August 2004*).

"Pitch Histograms in Audio and Symbolic Music Information Retrieval" G. Tzanetakis, A. Ermolinskiy, P. Cook. Proceedings of ISMIR 2002 (*October 2002*).

"Beyond the Query-by-Example Paradigm: New Query Interfaces for Music Information Retrieval" G. Tzanetakis, A. Ermolinskiy, P. Cook. Proceedings of ICMC 2002 (*September 2002*).

Employment History

Qualcomm Research Silicon Valley Intern (June 2010 - August 2010, Santa Clara, CA)

- Design and implementation of parallel scheduling algorithms and runtime environments for future generations of mobile devices.

IBM Research Almaden Intern (June 2007 - August 2007, San Jose, CA)

Storage systems group

- Research on algorithms and techniques for lazy replica synchronization and cooperative caching in widely-distributed GPFS/p-NFS environments.

Intel Research Intern (June 2006 - August 2006, Berkeley, CA)

- Research on algorithms and tools for use in forensic analysis of distributed denial-of-service (DDoS) attacks against network servers.

IBM Corporation Software Engineer (September 2002 - August 2005, Poughkeepsie, NY)

General Parallel File System (GPFS) development group

- Designed and implemented support for disaster-resilient GPFS clusters to address high-availability requirements for several key customers. The design is based on logged synchronous mirroring of data and metadata across a pair of geographically separated sites, using a third-site node as a tiebreaker for consensus protocols.
- Architected and implemented a number of other features and improvements in the areas of I/O performance, distributed token management, transaction recovery, and cluster configuration management.

Sun Microsystems Intern (June 2001 - August 2001, Menlo Park, CA)

Solaris kernel group

- Designed and implemented several kernel extensions for the Solaris platform including: (1) A resident set monitor for memory-intensive workloads. (2) A page pre-faulting mechanism to reduce start-up times of large software applications.

IBM Corporation Intern (June 2000 - August 2000, Austin, TX)

General Parallel File System (GPFS) development group

- Designed and implemented an optimized kernel memory allocation mechanism for GPFS on Linux.
- Implemented error logging support for GPFS on Linux.
- Helped on porting other parts of the project from AIX to Linux.

HB International Intern (June 1999 - August 1999, Reykjavik, Iceland)

Centara development group

- Designed and implemented a protocol for EFT (Electronic Fund Transfer) on top of NetBIOS.

Professional Service

Outside reviewer for the following conferences: SRDS'08, PD-CAT'09.

Skills Summary

Programming Languages: C/C++, x86 assembly, Java, ML, MS Visual Basic, HTML, SQL.

Application-level development: Linux, AIX, Solaris, and Windows.

Kernel-level development: Linux and AIX; Significant experience with the internals of the Linux kernel (in particular its storage and networking stacks).

Hypervisor and emulator development: Strong familiarity with the internals of Xen and QEMU. Design and implementation of mechanisms for instruction-level information flow tracking, dynamic code generation, and security policy enforcement.

Concepts, algorithms, and protocols: Strong familiarity with the theoretical and practical aspects of developing robust, scalable, and fault tolerant distributed systems; Expert understanding of distributed protocols (consensus, leader election, mutual exclusion, state machine replication, atomic commit, replica synchronization, transaction recovery, etc); Expert understanding of parallel and distributed filesystem architecture. Expert understanding of Internet transport and routing protocols (TCP/IP, BGP, OSPF, RIP, DVMRP, PIM-SM, PIM-DM); Expert understanding of file- and block-level remote storage protocols (NFS, iSCSi).

Personal

Born: January 5, 1981. Cherepovets, Russian Federation (former USSR)

Countries of residence: Russian Federation (1981 - 1987)
Ukraine (1987 - 1989)
Republic of Hungary (1989 - 1992)
Republic of Iceland (1992 - 1998)
U.S.A. (1998 - present)

Nationality: Icelandic

Languages: Fluent in Icelandic, English, Russian

Chapter 1

Introduction

> *"Information wants to be free because it has become so cheap to distribute, copy, and recombine — too cheap to meter. It wants to be expensive because it can be immeasurably valuable to the recipient. That tension will not go away."*
>
> Stewart Brand, 1987.

One of the central security concerns in managing an organization is controlling the flow of sensitive internal information, by which we mean ensuring that data and documents can be accessed only by the authorized parties. The recent history of major leakages of sensitive information [85, 54] has demonstrated that many organizations, including those in government, education, and the commercial world, are severely deficient in this regard. This is hardly surprising — as our reliance on computing infrastructure continues to grow, it becomes increasingly difficult to track the dissemination of sensitive data and enforce confidentiality policies. Considering the sheer number and diversity of information transfer channels that are available to users in a typical IT environment (e-mail, instant messaging, wikis, blogs, databases, distributed filesystems, and portable storage devices, to name just a few examples), tracking the flow of information across all these channels may seem like a daunting, if not altogether impossible, task.

Unauthorized disclosure of private or classified information can cause catastrophic damages to the business interests of an organization and threaten the well-being of the broader society. As an illustrative example, consider the highly-publicized incident [85] from March 2009, when one of the Transportation Security Administration employees inadvertently posted a 93-page internal TSA document to a public website on the Internet. This federal document, marked "sensitive information", serves as a manual for airport security screening personnel and contains detailed descriptions of passenger screening procedures. This leak of information revealed the criteria for exemption from certain screening measures and could offer insight into how to sidestep airport security.

After all the money and effort devoted to developing new security technology, why is it so hard to prevent leaks of sensitive information even in well-managed organizations with well-intentioned employees? Although it would be imprudent to place the blame on any one factor, it is difficult to overestimate the significance of basic human error. Today, protecting sensitive material requires users to remember, understand, and always obey the appropriate dissemination restrictions. Yet we know that in reality, users tend to be careless and impatient; they occasionally email documents to the wrong parties, transfer sensitive data to insecure machines, or otherwise inadvertently allow data to leak. Stricter security regulations are not necessarily the answer because, as Don Norman notes, *"...when security gets in the way, sensible, well-meaning, dedicated people develop hacks and workarounds that defeat the security"*[64]. According a recent survey by ISACA, 35% of corporate employees admitted that they have *knowingly* violated corporate data dissemination restrictions at least once. 42% have e-mailed confidential material to their home system and 22% have transferred sensitive corporate data using a portable USB storage device, acting in direct violation of corporate rules [2].

Can technology assist in identifying such manifestations of carelessness and mitigating their consequences? We believe that the answer is a clear "yes", but devising an effective and practical technological solution to this problem will require addressing a number of complex issues. One of the most significant challenges lies in tracking the vast array of information transfer channels available to users and accounting for the myriad ways, in which data can be manipulated, transformed, and transferred.

Most of the widely-used operating systems and user applications offer very little assistance in this regard. Simply put, current OSes and applications provide mechanisms for mediating access to data objects, such as files or database tables, but are not well-positioned to track subsequent manipulations on these objects and the flow of information between them. Consider a user, who opens a confidential text document in her word processor for editing. In a rare moment of carelessness, she copies a paragraph of this text into a public document, on which no dissemination restrictions are imposed. This elementary action produces a second copy of the confidential paragraph, but this new copy lacks any association with the original document, its confidentiality status, and restrictions on its dissemination — a situation that is precariously close to information leakage.

We believe that preventing incidents of this nature, without a complete bottom-up redesign of the software stack, requires a comprehensive and transparent platform for tracking the flow of user data, intercepting the channels of information exchange, and enforcing security policies. Our hypothesis is that (1) a comprehensive information flow tracking platform that works with unmodified applications and operating systems is within the realm of being practical on today's commodity hardware; and (2) a specialized *hypervisor* provides the most effective and natural architectural foundation for such a platform.

To explore this hypothesis, we present PIFT (Practical Information Flow Tracking) — a novel security architecture and a set of associated mechanisms for fine-grained information flow tracking in enterprise environments. Our high-level goal is to develop a robust information management platform that will enable organizations to specify and enforce end-to-end policies concerning the dissemination and usage of sensitive information.

1.1 Design Goals for a Comprehensive IFT Platform

To provide context and delineate the scope of this dissertation, we begin by describing the central goals of PIFT and the overall principles that guided its design. We begin by noting that none of the incidents cited above were caused by malicious activity on the part of insiders; the cited leaks were all caused by human error and carelessness. Although malicious entities (such as rogue employees, hackers, and malware) undoubtedly pose a threat to security, a large fraction of data breaches occurring in the U.S. (77% percent according to a recent survey [49]) can be attributed to negligence and lack of discipline among employees. Hence, our goal in PIFT is to design a practical system of safeguards — one that recognizes the limitations of human users and focuses on the dominant issue of internal carelessness, rather than the far less common problem of data theft. PIFT aims to provide a **comprehensive**, **deployable**, and **usable** information flow tracking (IFT) platform. More specifically:

1. To be **comprehensive**, PIFT should track the flow of information across all computational elements, data storage devices, and communication channels.

2. To be **easily deployable**, our platform should not require significant reconfiguration in existing IT environments and should be fully compatible with widely-deployed operating systems and applications. While there are other techniques that can produce simpler and perhaps more efficient mechanisms by modifying operating systems, application runtimes, or applications themselves, these techniques face a very significant barrier to adoption given the large investment already made in legacy software stacks.

3. To be **adopted for everyday use**, an IFT platform has to be efficient and not demonstrably impair user-perceived application performance. Furthermore, it has to be correct and parsimonious in how it propagates the sensitivity status.

To avoid confusion, it is important to state explicitly our assumptions regarding the environment, in which PIFT intends to operate:

1. Users are benign, in that they do not intentionally exfiltrate data, but are intolerant of inconvenience and liable to forget dissemination restrictions.

2. The software used by these users is non-malicious, meaning that it does not attempt to circumvent our information flow tracking techniques.

This dissertation describes how to devise a comprehensive IFT platform for sensitive user data that functions in an environment, where these assumptions hold. Viewed from a different angle, these assumptions reveal our explicit non-goals — what we do not hope to accomplish in the context of this dissertation. In broad terms, we do not hope to track information flow and enforce policies in the presence of sophisticated malicious activity. In particular, we do not try to protect sensitive data against theft by rogue employees or against

exfiltration by malicious or compromised software. Malevolent code intent on stealing has myriad channels (including *implicit data flows* [23] and *covert information channels* [48]), through which it can siphon off sensitive data. Malicious employees can exploit the analogue gap (for instance by printing sensitive documents on paper [76] or taking a picture of the screen showing sensitive data [83]) for the same purpose. Tracking all these channels in a comprehensive manner is technically hard for all existing information flow control systems and we do not attempt to close these gaps with PIFT.

1.2 The Design Philosophy of PIFT

Practical Information Flow Tracking (PIFT) is a novel information security architecture that focuses on preventing sensitive information leaks resulting from the actions of well-intentioned, but forgetful and careless users. In general terms, this requires mechanisms for tracking the propagation of user data as it is manipulated or computed upon, intercepting access to exit points, and enforcing the appropriate security restrictions. The high-level goals and assumptions outlined in the previous subsection have led us to adopt a specific set of design principles, which we now describe:

1. **Granularity of information flow tracking**: In order to provide a comprehensive solution and reduce the risk of false positives, we track the flow of information at *byte-level* granularity. Furthermore, since PIFT must retain compatibility with unmodified applications and can make no assumptions regarding their internal structure and data manipulation behavior, the most natural design option is to interpose at the software-hardware boundary and track the computation at the level of *machine instructions*.

   An alternative strategy would be to maintain a coarser-grained view of data sensitivity (e.g., page-level labels for the information stored in volatile memory and sector-level labels for data residing on disk). While this approach would alleviate the storage and computational burden incurred by our current implementation, it would sacrifice the precision of tracking and make the system more susceptible to false positives. For instance, copying a block of text from a world-readable file F_1 into a sensitive file F_2 and then copying the same text from F_2 to F_3 could cause F_3 to wrongly acquire the sensitivity tag with this strategy.

2. **Point of interception and enforcement**: Since PIFT focuses mainly on restricting the flow of information between users in an organization, security checks need to be performed *only* when the data is externalized in some fashion and no limitations need to be imposed on how the data is handled locally on a user's machine. Hence, PIFT intercepts device I/O requests that externalize data in order to enforce policies, but does not try to prevent application code from touching or performing computation on sensitive data. As we explain below, this method of policy enforcement enables several novel and important performance optimizations.

Given the above design choices, an architecture based on a hypervisor with byte- and instruction-level information flow tracking appears to be the most natural and practical solution. Modern hypervisors are fully compatible with legacy software stacks, yet have sufficient privileges to monitor and track the computation in the guest machine. Furthermore, hypervisors routinely virtualize and mediate access to exit points (i.e., output devices such as network interfaces and block storage devices). PIFT can reuse this functionality to intercept the data at these exit points and impose security checks. Finally, although hypervisors add management complexity, we believe that with the recent rapid increase in hypervisor deployment, it is not unreasonable for a security solution to require their use on every endhost that handles sensitive data.

PIFT is based upon the theory that a hypervisor-based information flow tracking platform strikes the most reasonable balance between security, usability, deployability, and performance. In the broader perspective, this view appears to be gaining general acceptance in the research community and a similar position has been articulated in a recent paper by another research team [98].

Instruction-level information flow tracking is conceptually simple: memory containing sensitive data is tagged with *taint labels* and the label values are propagated in a manner that mirrors the computation. The typical implementation approach for full-system IFT involves running the guest system in a hardware emulator (such as QEMU), which has been augmented with machine instruction analysis and taint tracking capabilities. Both operations incur very significant computational costs — prior work on full-system IFT reports slowdown factors in the range 50-200x relative to unmonitored native code execution. Although such levels of overhead are acceptable for the purposes of offline security analysis, which has been the dominant focus of previous IFT efforts, our target usage scenario requires the ability to track the movement of sensitive data and enforce security policies in real-time, minimizing the user-perceivable slowdown.

The runtime performance costs associated with previous instruction-level IFT systems render them unsuitable for the purpose of real-time information flow tracking. This dissertation presents our progress on addressing the performance challenges and proposes several novel optimizations that enable us to reduce the runtime overhead to a much more manageable level. The key insight behind our performance optimizations is that emulation and information flow tracking can be viewed as two separate and, for the most part, independent computations. As we demonstrate in the evaluation, decoupling these operations can be extremely beneficial and enables us to achieve dramatic performance improvements. Furthermore, unlike previous implementations, which track the propagation of taint labels by analyzing emulator-specific microinstructions, PIFT tracks these labels at the level of abstraction that directly matches the semantics of the native instruction set. This enables a range of additional optimizations that are difficult or altogether impossible to apply at the microinstruction level.

1.3 Evaluating PIFT

PIFT investigates a novel architecture for tracking the flow of sensitive information, based on the claim that interposition at the software-hardware boundary using a hypervisor can produce a practical and usable solution. To evaluate this claim, we must examine PIFT in the context of the original design goals and requirements, as outlined in Section 1.1. A practical system for sensitive information tracking must achieve reasonable performance on common application workloads and propagate taint labels correctly, avoiding over- and under-tainting.

PIFT's runtime performance overhead is easily quantifiable and Chapter 5 presents our detailed performance evaluation based on microbenchmarks and application-level measurements. Encouragingly, tracking the flow of information at a higher level of abstraction (native machine instructions) using multiple processor cores enables PIFT to reduce the computational overhead by a significant margin. While it was difficult for us to provide direct side-by-side comparisons with previous systems [98], in the two specific cases where we could do so, PIFT achieved a slowdown of roughly 1.5× (compared to native code), as opposed to prior efforts in which the two comparison cases suffered slowdowns of roughly one and two orders of magnitude, respectively. The results of our application-level experiments indicate that although PIFT's code analysis and information flow tracking primitives impose a nontrivial performance overhead for CPU-intensive workloads, our design succeeds in mitigating these sources of overhead through asynchrony and parallelism. Further, our user studies suggest that PIFT does not significantly impair user experience and productivity in interactive graphical application environments.

Turning to the question of label propagation correctness, we were able to directly evaluate the natural flow of user data in several popular consumer applications and we report the results from this application study in Chapter 6. These initial results are encouraging, but we also observed limitations in applying the technique of taint tracking to environments that use commodity off-the-shelf software products.

First, we observed that in some scenarios, the guest environment suffers accidental *taint poisoning*, which then amplifies into full-scale *taint explosion* and causes significant portions of the guest system state to become tainted. These observations are fully consistent with the results of earlier studies [80], but present a major challenge for platforms such as PIFT. Left unchecked, this phenomenon significantly impairs the performance of our information flow tracking substrate in two important respects. First, it unnecessarily forces our system to spend more time emulating the guest environment and adds perceptible overhead. Second, it confuses users and applications, since they have no way of telling whether a piece of sensitive data has been tainted due to the right reasons or accidentally.

One of the most alarming cases of taint explosion is accidental propagation of taint into the internal data structures of the OS kernel and our early experimentation with PIFT revealed that the Linux kernel is highly susceptible to this phenomenon. In this scenario, applications that do not operate on sensitive user data would become tainted when they interact with the kernel via system calls, causing taint labels to spread between applications and eventually rendering the whole system unusable. Our in-depth analysis of this

phenomenon reveals that kernel-level tainting is accidental and does not reflect explicit information transfer. PIFT leverages this insight to effectively eliminate kernel taint poisoning by interposing and scrubbing taint labels at a small number of kernel entry points.

We also examine the dynamics of *user-level* taint propagation within the address space of an application. For some applications, we were able to confirm that PIFT propagates taint labels correctly, in a manner that directly reflects the user's actions and intentions. Other applications appear to be susceptible to various degrees of taint poisoning and explosion, whereby non-protected data items (such as configuration files and non-sensitive user documents) acquire the taint status of sensitive documents that have been previously manipulated in the same application instance. It appears that in order to fully take advantage of hypervisor-based information flow tracking, applications must follow the natural channels for information exchange and several do not. We present a preliminary analysis of these phenomena and discuss their overall implications on fine-grained information flow tracking systems such as PIFT.

1.4 Summary of Contributions

The main contribution of this thesis is the architecture and implementation of PIFT — a robust information management platform that is designed to track the flow of sensitive data and enforce confidentiality policies, while satisfying the design constraints outlined in Section 1.1. Although our prototype implementation is based upon completely standard building blocks (the Xen hypervisor and QEMU), its architecture is novel. PIFT provides a taint-aware filesystem that enables users to store sensitive data (annotated with the appropriate taint labels) persistently on disk. Further, in order to ensure that taint labels are preserved across network transfers, PIFT transparently augments the networking stack in the guest environment to intercept all outbound network packets and annotate the payload with the corresponding taint label(s).

The performance overhead of fine-grained information flow analysis constitutes one of the most significant obstacles that must be overcome before real-time IFT systems can become fully practical. PIFT makes several important contributions in this area. Our implementation employs *on-demand taint tracking* — a technique whereby the guest system is dynamically transferred between emulated execution using QEMU and native execution within a Xen guest domain. Enabling emulation and IFT computation only for those regions of guest code that directly interact with tainted data allows PIFT to substantially reduce the runtime performance costs. Just as crucially, the information flow analysis computation is performed at a higher level of abstraction that directly matches the semantics of the native machine instruction set (x86 in our implementation). This is a significant departure from existing systems, which track information flow on the basis of QEMU microinstructions. Furthermore, while previous approaches tend to conflate emulation with information flow tracking, PIFT explicitly decouples these two operations. As we demonstrate later on, this strategy enables asynchronous parallelized taint tracking and leads to a further reduc-

tion of runtime performance costs. Operating together, these techniques enable our PIFT prototype to achieve a $60\times$ performance improvement over the best previously-published results.

For kernel taint explosion, we show that a major cause of this phenomenon on Linux is accidental tainting of kernel control data structures. We undertake a detailed analysis and track its origin to a small number of kernel entry functions. By interposing at these specific entry points and securely scrubbing taint, we prevent accidental tainting of these data structures and effectively eliminate Linux kernel taint explosion for all practical purposes.

The cumulative effect of our techniques is making transparent real-time information flow tracking significantly more practical. In fact, our PIFT prototype is now fully usable and supports real interactive user activities — we edited portions of this dissertation in a Linux guest environment running on top of PIFT. To the best of our knowledge, PIFT is the first real-time instruction-level IFT platform that has been demonstrated to be usable with an interactive graphical guest environment. While there is more work to be done before real-time taint-tracking can be widely used in practice, we wanted to report on our progress-to-date so that the community can help in overcoming the remaining barriers.

1.5 Thesis Roadmap

This dissertation is divided into seven chapters. The next chapter places PIFT in perspective with other academic research, introduces the necessary terminology, and provides a brief survey of existing literature in several relevant areas. Chapter 3 presents the overall system architecture of PIFT ands explain the partitioning of functionality between the hypervisor and the augmented emulator. This chapter also details PIFT's overall security model and discusses the semantics of fine-grained (byte- and instruction-level) information flow tracking. Chapter 4 provides an in-depth description of our prototype implementation, including its hypervisor-level components, the augmented emulator, the label-aware storage and networking stacks, and the mechanism for enforcing policies. For readers interested in understanding the low-level technical aspects of our work, this chapter details the implementation of PIFT's information flow analysis algorithms within QEMU, discusses parallelization techniques for taint tracking, and presents the on-disk layout of our label-aware filesystem. We present our detailed performance evaluation of PIFT across a range of experimental environments, which include microbenchmarks, application-level measurements, and usability studies in Chapter 5. Then, in Chapter 6 we turn our attention to the dynamics of taint label propagation and examine the problem of taint explosion. Finally, we conclude in Chapter 7 by summarizing our work, discussing the remaining barriers, and outlining what we believe to be promising directions for further research.

Chapter 2

Background and Related Work

This dissertation builds on a large body of prior work in the areas of information flow control and dynamic taint analysis. This chapter intends to provide a brief survey of previous research and review some of the most relevant and influential efforts in these related areas.

Information Flow Control (IFC) is concerned with restricting the flow of user data and protecting it against theft and misuse by untrusted applications. Section 2.1 discusses the state of the art in IFC, examining both static and dynamic techniques, and reviews some of the existing systems that influenced the design of PIFT.

Section 2.2 discusses prior work on dynamic taint analysis — a complementary set of techniques, which provide a means of tracking information flow within an application with high precision. Previous applications of taint tracking have, by and large, focused on detecting security attacks and analyzing the behavior of malware in a laboratory setting; PIFT is among the first systems to apply taint tracking to the problem of confining sensitive information flow.

It would be disingenuous to suggest that the work presented in this dissertation does not borrow from these previous studies and, indeed, many of the systems described in this chapter provided inspiration for our design, while others served as direct building blocks. Yet, all of these previous efforts face limitations, which prevent them from being directly applicable to the central problem we seek to address in the context of this dissertation; namely, controlling the flow of sensitive user data in a distributed environment without modifying the software stack or augmenting the hardware platform.

2.1 Information Flow Control

Most modern operating systems provide mechanisms for discretionary access control (DAC); examples of mechanisms in this category include file permissions, ACLs, and capabilities. In broad terms, the DAC security model imposes restrictions on which principals (users, processes, or machines) can access a particular data object at a specific point in time, but does not track the propagation and usage of this data after its initial release. A DAC-based security architecture delegates the responsibility for enforcing data flow policies to the application and offers little protection against the actions of careless users or malicious code.

Information flow control (IFC) provides a more powerful and secure alternative to DAC by allowing users to specify high-level system-wide restrictions on the use and dissemination of sensitive data. As an informal example, the following high-level policy may arise in an enterprise environment: *"The contents of a sensitive file F and all data derived from F can be disseminated only to the members of a project group G"*. Discretionary access control schemes offer mechanisms for restricting the initial release of the information contained in F, but are not sufficiently powerful to enforce restrictions on the derived data.

An information security framework based on IFC abstractions offers tighter controls over the use and dissemination of sensitive information compared to the traditional DAC-based model, but providing such controls requires additional mechanisms. Most significantly, the system must have the capability to track the propagation of sensitive data through computation and enforce policies at a set of well-defined information flow boundaries.

Prior work on information flow control can be broadly categorized into *static* language-based techniques, which seek to detect and prevent information leakage at compile time, and *dynamic* runtime enforcement.

2.1.1 Static Language-Level Analysis

Static checking of information flow policies has a long history; the initial research in this area was done in the 1970s and driven by the needs of the defense industry [24]. Denning's pioneering work in this area [22] introduced the concept of a *security lattice* — an abstract model of access control, whereby each data object and principal is assigned a *security class* (also called its *security label*). Information flow is controlled by imposing restrictions on the transfer of information between these entities.

In 1997, Myers and Liskov [56] introduced a *decentralized* model for information flow control (DIFC), which defines a set of rules that programs must follow in order to prevent leaks of sensitive information. The DIFC model provides security by allowing users to associate integrity and secrecy labels with data resources and constraining the flow of information according to these labels. JFlow [55] and its successor Jif [58] apply the DIFC model to the Java programming language, enabling information flow control within a program at the granularity of individual language-level variables. In Jif, all variables and

expressions are labeled with security policies which, together with ordinary Java type declarations, form an extended type system. The Jif compiler performs static type checking and rejects programs that might violate information flow restrictions. Informally, a value v can be assigned to a variable x only if the policy associated with x is at least as restrictive as the policy for v, in which case the assignment does not leak information and is considered legitimate. To prevent "label creep", Jif provides language-level features for selective declassification that enable a trusted code module to relax policies. Fabric [50] is a follow-on effort that extends the Jif programming language to a distributed environment and provides support for transactional semantics.

Static IFC analysis is a powerful technique, which provides the ability to track the propagation of sensitive data within a program (and between programs) with little or no runtime overhead. During compilation, the static checker constructs a proof that no possible execution path in a program contains data flows that are disallowed by the IFC policies. As a result, this technique can detect implicit information flows, as well as some classes of covert channels, both of which can be extremely difficult to detect with dynamic runtime tracking mechanisms.

However, these research efforts have demonstrated that adding static information flow control to a powerful general-purpose programming language is difficult. Jif is based on Java — the most expressive language for which static IFC has been attempted — but does not currently support several of its essential features, such as multithreading. Further, static IFC analysis imposes a new programming model and requires developers to rewrite applications. Finally and perhaps most importantly, many realistic usage scenarios involve dynamic policies that fundamentally cannot be evaluated at the time of program analysis and necessitate some form of runtime tracking and enforcement. The above scenario involving a sensitive corporate document and a project group is an example of such a policy.

PIFT focuses on supporting this broader class of policies through dynamic runtime enforcement and offers full compatibility with legacy application code. Another important distinction concerns the granularity and the level of abstraction, at which these mechanisms operate: PIFT tracks the flow of sensitive data across CPU registers, physical memory addresses, and disk sectors, as opposed to language-level primitives. Our architecture does not attempt to track implicit data flows and enforces policies on the movement of information between principals in a distributed environment, interposing at the software-hardware boundary.

2.1.2 Dynamic Enforcement of Information Flow Rules

OS- and runtime-level information tracking: There has been significant work on incorporating DIFC mechanisms into operating systems and runtime environments with the goal of tracking the dissemination of user data and enforcing dynamic end-to-end information flow constraints. Asbestos [25] and HiStar [93] are new operating systems, which track information flow dynamically and guarantee strong isolation of application code using a relatively small, trusted kernel. In Asbestos, each process carries a *secrecy* label that provides a

conservative estimate of all sensitive inputs observed by the process. The operating system intercepts all inter-process communication and verifies compliance with security policies. HiStar defines several low-level objects types (threads, containers, quotas, address spaces, gates) and controls information flow to and from each object instance. Collectively, these object classes provide building blocks for traditional OS primitives, such as file systems and processes, which are implemented in an untrusted user-level library. Both systems provide mechanisms for controlled declassification that allow privileged application-level components to externalize sensitive data through a set of legitimate and well-defined information channels. Assuming trustworthy declassifiers and an uncompromised kernel, these systems can enforce IFC policies even in the presence of misbehaving or malicious application-level code. DStar [94] extends the OS-level information flow control architecture to a distributed environment with the goal of mitigating the effects of untrustworthy distributed applications and compromised machines. Flume [47], one of the more recent efforts, demonstrates that runtime DIFC does not require a clean-slate redesign of the software stack and can be retrofitted into an existing UNIX-based operating system. Loki/LoStar [97] is a follow-on effort to the HiStar system and demonstrates that the amount of trusted code can be further reduced by extending the hardware architecture with support for byte-level memory labeling. The resulting system implements dynamic IFC in a minimal *security monitor* that resides underneath the OS and can enforce information flow policies despite kernel compromises.

Compared to static analysis, OS-level dynamic IFC techniques can support a much broader range of policies, but invariably impose some performance overhead. Furthermore, DIFC-enabled operating systems can only track information flow on the basis of coarse-grained OS-level primitives (e.g., processes, files, sockets) and are oblivious to fine-grained information transfers between variables or data structures within a process. Laminar [77], one of the more recent proposals, investigates a hybrid design that integrates language-level and dynamic OS-level DIFC abstractions in an effort to combine their strengths. Laminar designs a new operating system, which mediates access to system resources, and a specialized VM, which enforces fine-grained DIFC rules within the address space of an application. This system provides DIFC guarantees at the granularity of lexically scoped code blocks (called *security regions*), which makes it relatively easy to retrofit existing applications with security policies.

Another recent effort, RESIN [92], proposes a new application runtime that associates policies with application-level data objects and filters information transfer at system I/O boundaries. RESIN helps application developers detect and correct errors in the security logic by enforcing application-specific information flow assertions. Unlike the OS-level solutions discussed above, RESIN operates in an interpreted programming environment (such as Python or PHP) and tracks the propagation of sensitive data at the level of program variables. Program assertions offer a more flexible way to specify data flow restrictions compared to OS-level labels and can be used to express high-level application-specific policies (e.g., *"the password of a user U can leave the system only as part of an e-mail message sent to U's e-mail address"*).

Systems that rely on OS- and runtime-level IFC mechanisms require substantial

changes in the structure of the OS kernel and thus face a significant barrier to deployment. At the application level, programmers have to expose data flows to the OS by explicitly restructuring their applications into multiple modules in accordance with data flow restrictions and writing trusted declassifiers. Even systems such as Flume, which seek to provide compatibility with current operating system interfaces, require developers to partition the application into unprivileged and trusted privileged components.

PIFT aims to achieve similar goals, namely data containment via dynamic information flow control, but makes very different tradeoffs. A key requirement for our system is deployability, which implies compatibility with legacy software stacks and rules out approaches that require kernel redesign or application restructuring. Our system does not change the programming model and can be deployed as an incremental extension to current IT environments with little or no modification to existing software stacks. PIFT interposes a hypervisor at the software-hardware boundary and tracks the flow of information at the granularity of bytes. While the coarser process-level tracking exemplified by HiStar suffices to intercept all potentially sensitive output, this scheme offers only a binary "hammer" when it comes to policy enforcement: once a process P has observed a sensitive input S, all subsequent external output produced by P is conservatively assumed to be tainted with S. In practice, this means that declassification can be overly restrictive or excessively permissive, reducing the application's usability and adoptability. In contrast, PIFT tracks sensitive data at the granularity of bytes and monitors the computation at the level of machine instructions. This allows PIFT to maintain a more accurate view of sensitive data movement and make more informed enforcement decisions by checking if the specific data being externalized is sensitive or not.

At a conceptual level, the design of PIFT bears some similarity to Loki, which pushes the data labeling functionality into the hardware platform. Our approach extends the (virtual) machine architecture with mechanisms for both labeling sensitive data and tracking its propagation. A hypervisor-based design enables us to emulate these features in software and thus maintain full compatibility with current hardware platforms.

VM-level isolation: Yet another influential and related research direction is centered around the *Red/Green* isolation paradigm articulated by Butler Lampson and his colleagues [28]. In this scheme, users interact with applications and sensitive data using two independent and mutually isolated environments. The green environment confines important data and does not permit the use of untrusted applications. Conversely, the red environment allows users to execute potentially untrustworthy code and access external networks, but prohibits access to sensitive documents. Lampson further argues that a virtual machine monitor (another term for a hypervisor) provides the right mechanisms for enforcing such isolation and that the two environments should thus correspond to virtual machines. This paradigm erects an effective first-class isolation boundary between valuable data and the threats posed by malicious code. It also provides a cost-effective alternative to the practice of physical air-gapping that has been the traditional method of separating user activities in high-security environments. Other systems that use virtual machine technology for the purposes of data containment and isolation include LiveWire [36](a VM-based intrusion

detection platform) and the NetTop [60] project. The architecture of PIFT is influenced by these proposals, but focuses on the broader goal of end-to-end policy enforcement. This requires the ability to track the propagation of confidential information among users and intercept external output.

2.2 Dynamic Taint Analysis

A complementary body of work applies machine code analysis and recompilation techniques to track information flow at the level of machine instructions — a technique known as *dynamic taint analysis* or *taint tracking*. Broadly, taint analysis mechanisms operate by annotating the low-level machine state (CPU registers, memory addresses, and disk regions) with byte-level *taint labels*, examining the machine instruction stream to determine the information flow effects of each instruction, and propagating the taint labels accordingly. In order to enable such fine-grained analysis facilities, taint tracking systems must instrument the target application at the binary code level or run it in a specialized emulator.

2.2.1 Applications of Taint Tracking Techniques

Byte-level taint analysis is one of the central functional primitives in PIFT, but its applications extend far beyond monitoring the computation on sensitive user data. In its general form, taint tracking is a powerful and widely-used technique that has been applied to a broad range of problems in the areas of security and information management.

TaintCheck [61], one of the foundational efforts in this area, aims to provide an effective defense mechanism against fast-spreading Internet worms through automated exploit detection and signature generation. TaintCheck marks any data that originates from an untrusted external source (e.g., network sockets) as tainted and uses binary rewriting mechanisms to track the subsequent propagation of such data. To detect attacks, TaintCheck looks for dangerous and potentially illegitimate operations on tainted data, such as the use of a tainted value as the destination for a jump instruction, which would be suggestive of an attempt to redirect control flow. The implementation is based on Valgrind [59] and can track the propagation of tainted inputs within the virtual address space of a single user-level process.

Sweeper [86] user dynamic taint analysis as a component in a comprehensive worm defense system that offers low overhead and efficient post-attack recovery. During normal execution, the system takes periodic process-level checkpoints and performs lightweight monitoring to detect suspicious activity. After an attack is detected, Sweeper performs a rollback and re-executes the process from an earlier checkpoint in a controlled environment. At that stage, heavyweight analysis tools such as TaintCheck are employed to identify the exact nature of the attack and produce an *antibody* that protects the process against further attacks.

Analogously to these systems, PIFT uses emulation and instruction-level information flow tracking to monitor the propagation of tainted data within the system and enforce policies. We extend on these efforts with a broader goal of ensuring data confidentiality in a distributed enterprise setting. While in TaintCheck and Sweeper policies impose restrictions on fine-grained data movement within a process (e.g., "the instruction pointer cannot be loaded with a tainted value") with the overall goal of enforcing *system integrity*, PIFT policies seek to ensure *confidentiality* by confining the flow of sensitive user data between principals.

Turning to confidentiality, there has been significant previous work in the use of taint tracking to protect sensitive information against malware. Panorama [91] uses full-system emulation and dynamic taint analysis to detect malicious access to sensitive user data and identify privacy-breaching malware. A code sample is loaded into an emulated system environment and subjected to a series of automated tests, whereby sensitive user inputs are introduced into the guest system. The taint tracking engine monitors how sensitive (tainted) information propagates within the system and flags any suspicious interaction between the unknown code sample and the tainted data.

Another recent proposal [26] combines system-level taint tracking with static analysis to identify malicious behavior in BHOs (browser helper objects). The BHO framework allows third-party code to execute within the address space of Internet Explorer and is a common deployment vehicle for spyware. The proposed system uses taint analysis to accurately track the flow of potentially sensitive data (e.g., visited URLs) as it is processed by the browser and any loaded BHOs. When tainted data leaks due to activity on behalf of a BHO, the resulting information flow is flagged as malicious. A lightweight form of static analysis is used to identify direct control dependencies and this information helps mitigate some instances of evasion via implicit channels.

PIFT is analogously concerned with protecting sensitive information against misuse, but extends on these efforts with a broader goal of enforcing end-to-end information flow restrictions that are both more general and more macroscopic in nature. Our system focuses on regulating the flow of sensitive data between benign principals in a distributed setting and uses taint analysis techniques to track its dissemination and usage.

Polyglot [11] applies dynamic taint analysis to the problem of automated protocol reverse engineering. This system takes as input the binary image of a program along with some tainted input data (such as messages received from the network), monitors how the program processes the data, and produces a fine-grained trace of taint propagation. This information is used in conjunction with the instruction-level execution trace to reconstruct the protocol message format. For example, if the program performs an indirect memory access that touches tainted data at a memory address d, and d has itself been derived from a tainted value at another address l, then one can reasonably assume that l stores the length of a variable-length message field that terminates at d.

Dispatcher [10] extends this work and presents a novel technique for inferring the full protocol format and semantics, which requires analyzing both inbound and outbound application messages. Dispatcher proposes a technique called *buffer deconstruction*, which

extracts the structure of outbound messages and infers field semantics by tainting the inputs and outputs of specific system APIs with well-known semantics and observing their propagation to/from the message buffers. Collectively, these techniques provide a powerful tool for black-box protocol analysis and have been used to reverse-engineer a complex command-and-control protocol for botnets.

Renovo [44] uses dynamic binary analysis to undo the effects of code packing and recover malicious executable code that has been obfuscated through compression or encryption. Given an executable code sample containing hidden code, Renovo loads it into an emulated environment and carefully tracks its execution. Whenever the program performs a memory write operation, the emulator marks the corresponding destination address as tainted. If, at some point during program execution, the instruction pointer jumps to a tainted memory address, Renovo infers that the corresponding memory location stores executable code that has been generated at runtime. This straightforward technique can be used to identify the completion of the code unpacking routine and the point of entry into hidden code.

HookFinder [90] proposes a framework for systematic analysis of malware hooking behavior. This system uses binary analysis and fine-grained taint tracking to identify *points of impact*, which emerge when the contents of OS-level data structures are modified by untrusted external code. Analyzing the effects of these impact points on the control flow can assist in identifying the placement of hooks.

TaintDroid [27] is a recently-proposed system for monitoring the flow of sensitive user data through third-party applications on smart mobile devices. This system extends Android [4] (a popular Linux-based open-source platform for smartphones) with fine-grained taint tracking and analysis capabilities. Sensitive user information is first identified and labeled as such at a set of well-defined taint sources, which include local sensors (such as a microphone, camera, and GPS receiver), shared information databases (such as address books and SMS message repositories), and unique device identifiers. The Dalvik VM interpreter is modified to track the computation on tainted data within an application and several additional OS-level extensions provide mechanisms for tracking its propagation between applications, as well as transfers to/from persistent storage. In contrast to most other systems discussed in this section, which track information flow on the basis of native machine instructions and low-level hardware registers, TaintDroid analyzes Dalvik byte-code and tracks taint propagation at the level of architecture-independent byte-code variables. A detailed application study conducted with the aid of TaintDroid revealed numerous instances of potential misuse, suggesting that popular Android applications routinely externalize users' private information without their explicit consent.

Neon [98] explores system support for derived data management and proposes a set of mechanisms that enable organizations to enforce end-to-end data containment policies. Analogously to PIFT, this system seeks to achieve binary-level compatibility with existing operating systems and applications using a hypervisor-based design. Neon associates a 32-bit *tint*, which represents a policy, with each byte in the guest virtual machine and uses a modified version of QEMU to track the propagation of tints between memory and CPU registers at the level of machine instructions. Similarly to its predecessor [41] and our

system, Neon implements on-demand emulation and uses the paging hardware to trap the initial access to sensitive data during native execution.

PIFT and Neon have very similar goals and share some aspects of the design, but differ in several crucial respects. First, Neon assumes that all sensitive files reside persistently on a central server and supports only coarse-grained file-level policy labels. Centralized file access may present a usability hindrance, while maintaining only coarse file-level labels can lead to overtainting and unwarranted denial of dissemination privileges. In contrast, PIFT implements a specialized taint-aware filesystem that permits users to store sensitive data persistently on the local disk, without a central server, and supports both file- and byte-level labeling. Second, while both systems track information flow by analyzing machine code, PIFT implements a number of advanced code analysis and translation techniques that substantially reduce the computational burden of dynamic information flow tracking. Specifically, our system analyzes information flow at the level of native x86 machine instructions, without first decomposing them into sequences of primitive microinstructions. PIFT also tracks information flow asynchronously and in parallel with the main instruction stream using a separate CPU core. Section 3.3.2 compares these two systems in greater detail and further describes our novel optimization techniques. Operating in concert, these techniques allow PIFT to achieve an order-of-magnitude improvement over Neon in a direct performance comparison.

2.2.2 Improving the Performance of Dynamic Taint Analysis

Dynamic instruction-level taint tracking is a computationally expensive task and reducing its runtime performance costs is one of the major directions of our work. A number of optimizations to speed up taint analysis have been explored in earlier work and many of them are directly applicable in our context. Ho *et al.* [41] proposes *demand emulation* as a practical technique for online full-system taint tracking, whereby a hypervisor and a taint-tracking emulator cooperate to dynamically switch the guest system between *native virtualized* and *emulated* modes of execution. Our system uses a similar technique, but focuses on the broader goal of confining sensitive data and enforcing high-level restrictions on the movement of information between principals. PIFT tracks information flow at a higher level of abstraction (native x86 instructions as opposed to QEMU microinstructions) and attains further performance gains through asynchronous parallelized execution of the taint tracking instruction stream.

Another method of exploiting asynchrony for taint analysis acceleration was proposed by Nightingale *et al.* [63]. The Speck framework allows taint tracking (and other forms of runtime security analysis) to execute in parallel on a separate core, while providing the safety guarantees of pure synchronous taint tracking. Speck relies on OS-level support for speculative execution, rollback, and deterministic replay [62] to prevent malicious code from permanently damaging the system, thereby permitting postponed asynchronous execution of security checks. Analogously to our system, a memory-based log is used to achieve coordination between the main application thread and the taint analysis threads.

Providing support for speculative execution requires significant changes to the operating system kernel, which we hope to avoid in PIFT. The current implementation of Speck supports taint tracking only within a single-threaded user-level process, whereas PIFT tracks the flow of sensitive data across process and machine boundaries using a hardware emulator, aiming to achieve a comprehensive view of information flow.

LIFT [73] investigates a low-overhead IFT system for detecting security attacks and proposes several algorithmic optimizations to improve the performance of taint tracking. The Fast-Path optimization extends the concept of demand emulation with the goal of further reducing the amount of unnecessary taint tracking work. Before executing a basic code block, LIFT checks the taint status of its live-in and live-out locations (registers and memory) and decides whether it is necessary to run the fully-instrumented version that tracks information flow. If all live-in and live-out locations carry empty taint labels, it can be deduced that the code block performs only zero-to-zero taint propagation and thus, the fast non-instrumented version can be safely executed. The Merged Check optimization exploits the temporal locality and spatial locality properties to reduce the number of taint transfer and taint checking operations. LIFT performs memory reference analysis and clusters nearby references into a group, which allows it to coalesce the corresponding taint transfers.

Some of the optimization techniques proposed in this work are general-purpose in nature and are directly applicable to our system, while others are more implementation-specific and take advantage of specialized features of the IA-64 architecture — the hardware platform, on which LIFT intends to operate. The two systems also differ in the scope of information flow tracking: LIFT focuses on tracking taint labels within the *virtual* address space of an application and is currently limited to supporting single-threaded user-level programs, while our system aims to track information flow across the entire machine, including the OS kernel, and maintains taint labels at the physical address level.

2.2.3 Hardware Extensions for Dynamic Taint Analysis

Yet another thread of related work investigates hardware-based architectural extensions for dynamic information flow analysis. Compared to PIFT, these efforts take a more forward-looking view and agree to forfeit compatibility with existing hardware architectures in hopes of producing more robust and efficient hardware-driven IFT platforms.

One of the key concerns in designing a hardware-assisted scheme is deciding how to partition the information flow analysis and enforcement functionality among the various components of the architecture: pushing a set of functions into the hardware platform provides a way to reduce the runtime performance costs, while a software-based implementation is inherently more flexible and configurable.

Minos [20] is one of the first systems to investigate hardware-assisted IFT. This effort proposes a complete microarchitecture that tracks the integrity of all data in the system and aims to protect against control diversion attacks that overwrite return addresses or function

pointers with untrusted values. Minos augments every memory word and machine register with an integrity bit and propagates this additional value across all stages of the processor pipeline. The system uses this additional bit to implement Biba's low-water-mark integrity policy [8], which is hard-wired into the Minos architecture.

Raksha [21], a more recent effort, investigates a hybrid DIFT architecture that tries to combine the strengths of hardware- and software-based techniques in order to provide a flexible low-overhead solution. This work proposes an augmented processor architecture that annotates all storage locations, including registers, caches, and main memory, with taint tags. Analogously, all machine instructions are extended with additional functionality to propagate these tags from input to output registers and the exact rules for taint transfer are specified by a set of dedicated control registers. Another group of registers specifies which operands or elements of the processor state should be monitored for the acquisition of taint tags and a security exception is raised when a non-empty tag propagates into one of these locations. When an exception occurs, Raksha redirects the execution to a user-level exception handler, which can implement arbitrary security checks and policies. By partitioning the IFT functionality in this manner and pushing the low-level mechanical functions of taint tracking into the hardware platform, Raksha can provide highly flexible and programmable security policies with low performance overhead.

Integrating information flow tracking functionality into an existing CPU design requires significant modifications to the processor core, which increases design complexity and may affect the footprint and clock frequency. In a follow-on effort to Raksha, Kannan *et al.* [45] proposed an alternative hardware-driven IFT architecture that decouples taint tracking functionality onto a separate coprocessor, thereby eliminating the need to modify the design or layout of the main processor. The coprocessor encapsulates all state and functionality associated with taint tag propagation and the main CPU core operates only on data without any concern for the presence of tags. The key observation that enables this approach is that the state of taint tags need not be maintained with instruction-level accuracy and that synchronized instruction-by-instruction propagation of tags is an overkill for most practical applications of taint tracking. In many security scenarios, it suffices to maintain a loosely-synchronized view and provide up-to-date taint information only at a set of coarse-grained and well-defined synchronization points, such as system call entry or generation of externally-observable output. Hence, the coprocessor can execute the stream of IFT operations *asynchronously* and the main CPU core must only communicate the necessary information about the sequence of committed instructions and their effects on the state of taint tags. In the current implementation, a shared queue is used to communicate the instruction encodings and the addresses of memory operands between the two processors.

The Log-Based Architectures (LBA) project [14, 15] proposes an alternative and even less disruptive design, which executes the information flow tracking computation using a general-purpose processor on a multi-core CPU, instead of delegating it to a specialized coprocessor. In this design, the hardware platform needs to be extended only with a low-level logging facility that captures an instruction-level trace of execution on the main processor core and relays it to a software-based trace analysis engine, running on a separate core. While LBA seeks to provide a low-overhead general-purpose framework for detailed

monitoring and tracing of application-level code, dynamic instruction-level taint tracking is one of the natural applications of this technology. Follow-on work [78] proposes several novel techniques for parallelizing the IFT computation that leverage hardware support for instruction-level tracing and achieve significant speedups on realistic workloads. While explicit computation of taint values is not naturally amenable to parallelization due to serial dependencies between instructions, the taint status can be computed *symbolically* by tracking its inheritance and then producing a concise summary of the net propagation effects for each code segment. The latter technique is easily parallelizable and explicit taint values can then be computed by performing a single sequential pass over the intermediate symbolic state. Another significant challenge to efficient parallel execution is posed by binary taint operators, which aggregate taint values from two or more sources. While several techniques for mitigating the performance impact of binary operations can be considered, it has been pointed out that for some applications, the IFT rules can be relaxed in a manner that avoids tracking the propagation of tags through binary operators [78]. This approach is controversial, since it weakens security guarantees, but one that enables significant parallel speedups.

Another recent proposal, SHIFT [13], investigates a hardware-based IFT scheme that operates on *unmodified* commodity processors by leveraging some of their existing architectural features and reusing them for the purposes of taint tracking. SHIFT's starting point is a novel and somewhat unusual observation — the architectural machinery needed for associating taint bits with low-level processor components and propagating them along the program execution path bears a strong resemblance to the mechanisms for tracking *speculative* and *deferred* exceptions, which are already present in many modern processors. Hence, treating tags that describe tainted data as deferred exceptions allows reusing these existing mechanisms to compute the propagation of taint tags between processor registers. Based on this observation, SHIFT implements a single-bit taint tracking substrate that functions on unmodified Itanium processors with minimal performance overhead. Although this approach does not easily generalize to multi-bit taint labels, it demonstrates the importance of studying the full capabilities of modern processors and applying their architectural features to novel problems.

The design proposed in this dissertation is similar in spirit to all these previous systems in that it provides fine-grained information flow tracking capabilities by augmenting the hardware platform. As we explain in Chapter 4, PIFT implements a specialized *taint processor* with a custom ISA, which operates in tandem with the main CPU and tracks information flow along its execution path using opaque 32-bit taint label values. Some of the previous efforts [45, 14, 15] argue in favor of decoupling program execution from information flow analysis and letting these computations proceed in a loosely-synchronized manner. PIFT espouses a similar point of view and leverages *asynchrony* and *parallelism* to attain dramatic overhead reductions. In our design, the taint processor is implemented within a software-based emulator that operates on a separate CPU core and explicit synchronization is required only when the guest environment issues a request to externalize data via device I/O.

The key distinction from these earlier efforts is that in PIFT, the hardware extension

takes on a *virtual* form and we *emulate* its functionality on a general-purpose CPU in order to preserve full compatibility with existing commodity hardware platforms. Such emulation is inherently expensive and PIFT must devise several novel performance optimization techniques in order to bring the overhead down to a manageable level. Further, while most of the previous systems focus on tracking information flow within a single user-level process and apply IFT techniques to the problem of detecting and preventing low-level security attacks (such as hijacking control flow by overwriting the return address), PIFT assumes a benign environment and focuses on the broader goal of tracking the flow of user data between applications, as well as between machines in a distributed environment. Thus, existing mechanisms for monitoring data flow within an isolated virtual address space are insufficient for our purposes; PIFT must also track the propagation of user data between applications and the OS kernel (and hence monitor the execution of privileged instructions), as well as intercept all network data transfers.

Chapter 3

The PIFT Architecture and Information Flow Tracking Model

This chapter describes the overall system architecture of PIFT and outlines its central functional components. We introduce the notion of data labels in Section 3.1 and describe our model of information flow tracking in Section 3.2. Finally, Section 3.3 describes the core functional components of PIFT, including the hypervisor, the augmented emulator, and our label-aware filesystem.

3.1 Data Labels and the Policy Model

The cornerstone of our policy model is the notion of a *principal*, which represents a recipient of information and serves as a basic unit of granularity for the purposes of access control. A principal can represent an individual user in an organization or a group of users that share the same access privileges (*e.g.*, employees in the accounting department).

PIFT's mechanisms for policy specification and enforcement are based on the *decentralized label model* [56, 57] — a simple, but powerful model of access control that enables multiple principals to protect their private information and share it in a controlled manner.

In the decentralized label model, each data value is assigned a *label*, which expresses a certain set of restrictions on its dissemination. Conceptually, a label represents an unordered set of *confidentiality policies*. Each policy has an *owner* and defines a set of *authorized readers*. The owner of a policy associated with a data item d is a principal, whose information was observed to create its value and who wishes to restrict its exposure by defining a policy. The reader set specified by the policy denotes principals that are autho-

rized by the owner to observe and compute on d. A single principal may appear in multiple reader sets and may own multiple policies. Furthermore, a principal can modify (relax or strengthen) its own policy on a specific data item by altering the reader set accordingly.

By default, all newly-created data items are assigned an *empty label* (denoted $L_\emptyset \equiv \{\}$), which contains no policies and represents completely public data. We say that a data item is *tainted* if it carries a non-empty label. In the case of labels with multiple policies, data may be observed by a principal p if and only if all of the policies specify p as an authorized reader. The intersection of all reader sets in a label forms is *effective reader set*.

To illustrate these definitions, consider a data item d labeled with $L_d = \{\{o_1 : r_1, r_2\}, \{o_2 : r_2, r_3\}, \{o_3 : r_2, r_4\}\}$. This label has three policies, owned by principals o_1, o_2, and o_3. The policy of principal o_1 permits r_1 and r_2 to observe the value of d; the policy of o_2 allows r_2 and r_3 to observe d, and the policy of o_3 grants permissions to r_2 and r_4. In this example, the effective reader set contains the common element r_2 and thus, only this principal has the authority to access and manipulate d.

As a more concrete example, consider two employees, Alice and Bob, who are collaborating on an internal project that involves confidential information. Suppose that Alice has a confidential file f_1 on her machine that she wishes to share securely with Bob. She can do this by defining a new confidentiality policy $p_A = \{Alice : Alice, Bob\}$ and labeling f_1 with p_A before releasing it to Bob. This policy allows Bob to access, store, and compute on f_1, while preventing him from disclosing its contents to other parties. Suppose that Bob owns another file f_2 that is labeled with $p_B = \{Bob : Bob, Charles\}$ and, at a certain point, decides to combine the information in these two files, for example by cross-referencing f_1 against the contents of f_2. The output of this computation is a new file f_3 labeled with the union of their policies: $\{p_A, p_B\}$. Forgetting that the output contains data derived from Alice's private file, Bob inadvertently tries to release f_3 to Charles. However, since Alice's policy does not specify Charles as an authorized reader for her data, PIFT must block this action in order to prevent information leakage.

Suppose that at a later stage, another employee, David, joins the confidential project and asks for permission to access the associated files (f_1 and f_3, but not f_2). In order to grant him access to f_1, Alice adds David to the set of authorized readers in p_A. In order to make f_3 available, Bob establishes a new policy $p_{B'} = \{Bob : Bob, David\}$ and relabels f_3, replacing his previous policy p_B with $p_{B'}$. Note that as an alternative, Bob could make f_3 available to David by adding him to the reader set of p_B, but this action would have an unwanted side-effect of exposing f_2 to David.

3.2 The Model of Information Flow Tracking

PIFT tracks computation on sensitive data values at the granularity of machine instructions and propagates the labels accordingly. Some instructions involve combining the values of multiple (typically two) distinct operands and PIFT handles such operations by *merg-*

```
int compute_sum(int a, int b) {
  return a + b;
}
```

```
<compute_sum>:
 +0x0 push %ebp
 +0x1 mov %esp,%ebp
 +0x3 mov 0xc(%ebp),%edx
 +0x6 mov 0x8(%ebp),%eax
 +0x9 add %edx,%eax
 +0xb pop %ebp
 +0xc ret
```

Figure 3.1. An example of instruction-level IFT: the `compute_sum` function.

ing the labels of the input values. Label merging is a foundational functional primitive that produces a new data label by aggregating the policies specified by the input labels. More formally, given a pair of labels $L_1 = \{p_1\}$ and $L_2 = \{p_2\}$, where p_1 and p_2 represent arbitrary policy sets, the *merge operator* (denoted $\oplus$) produces a new label that corresponds to the union of the input policy sets: $L_1 \oplus L_2 \equiv \{p_1 \cup p_2\}$. Defined in this manner, label merging precludes any possibility of information leakage through computation expressed via binary operators. The resulting label defines the least restrictive confidentiality policy that also enforces all the restrictions on the input operands used in the computation.

In our current design, PIFT tracks all explicit data flows resulting from variable assignments and arithmetic operations. We also track indirect flows that occur as result of pointer dereferencing, whereby a sensitive data value is used as a base pointer or an offset to access another value in memory. However, PIFT does not currently track implicit channels that arise from control flow dependencies, such as when a labeled value influences a conditional branch. It is exceedingly difficult to identify such dependencies correctly at runtime without the benefit of prior static analysis at the source code level.

Next, we provide several basic examples to illustrate the notion of instruction-level label tracking and clarify the differentiation between explicit and implicit information channels. As a first example, consider a simple function `compute_sum`, whose C-language implementation and the corresponding assembly code are shown in Figure 3.1. As the name suggests, this function accepts as input two integer arguments and returns their sum. Suppose that the input variables (`a` and `b`) are tainted with data labels $L_{\mathtt{a}}$ and $L_{\mathtt{b}}$, respectively. When this function is executed in a PIFT-managed environment, PIFT tracks the computation at the instruction level and propagates the labels in the following manner: The first instruction pushes the old value of the stack base pointer (register `ebp`) onto the stack. In the context of information flow tracking, we consider `ebp` to be a *control register* that does not normally carry sensitive user data. For this reason, we do not track the propagation of labels into this register and assume that its contents are non-sensitive (*i.e.*, labeled with $L_\emptyset$) at all times. Hence, this instruction has the effect of transferring a non-sensitive four-byte value to the top of the stack and, as a result, PIFT clears the sensitivity label associated with the respective memory address: $L_{mem[\mathtt{esp}+(0...3)]} \leftarrow L_\emptyset$. The instruction at offset 0x3 loads `b` from the stack into `edx` and to track its effects, PIFT assigns the label $L_{\mathtt{b}}$ to `edx`: $L_{\mathtt{edx}} \leftarrow L_{\mathtt{b}}$. Analogously, the next instruction propagates the label $L_{\mathtt{a}}$ into `eax`: $L_{\mathtt{eax}} \leftarrow L_{\mathtt{a}}$. Finally, the instruction at offset 0x9 computes the sum by adding the value in `edx` to the contents of `eax` and PIFT updates the register labels accordingly, by merging the labels of

```
int table[] = {...};

int table_lookup(int index) {
  return table[index];
}
```

```
<table_lookup>:
 +0x0 push %ebp
 +0x1 mov %esp,%ebp
 +0x3 mov 0x8(%ebp),%eax
 +0x6 mov 0x8049700(,%eax,4),%eax
 +0xd pop %ebp
 +0xe ret
```

Figure 3.2. An example of instruction-level IFT: the `table_lookup` function.

the two input operands: $L_{\texttt{eax}} \leftarrow L_{\texttt{eax}} \oplus L_{\texttt{edx}}$. The last two instructions return control to the caller by restoring the values of `ebp` and `eip` from the stack and since PIFT does not track the flow of information through these registers, no action needs to be taken.

The example shown in Figure 3.2 implements a basic table lookup operation and illustrates the propagation of data labels through pointer dereferencing. Suppose that the table stores sensitive values marked with $L_{\texttt{t}}$ and that the input argument (table index) is tainted with $L_{\texttt{i}}$. We observe from the assembly code that the lookup operation is implemented via a sequence of two instructions: loading `index` from the stack into `eax` (offset 0x3) and computing a pointer to the respective table entry and dereferencing it into `eax` (offset 0x6). In this scenario, the instruction at 0x3 taints the `eax` register with the label of the input argument: $L_{\texttt{eax}} \leftarrow L_{\texttt{i}}$. The instruction at 0x6 performs an indirect memory reference via a tainted pointer and PIFT handles it by merging the pointer label with the label of the memory location(s) being accessed: $L_{\texttt{eax}} \leftarrow L_{\texttt{eax}} \oplus L_{\texttt{t}}$.

While several previous studies [80, 91] have questioned the viability and usefulness of tracking indirect pointer-based channels, suggesting that this method tends to generate explosive and unwanted propagation of taint, we believe that in the context of our design, tracking labels across pointer access is beneficial and, in fact, essential for correctness. Table lookups are an extremely common operation and occur in a variety of scenarios that involve manipulation of sensitive user data, character set conversion being one specific example. Failure to track indirect flows arising from table access can easily lead to undesirable loss of sensitivity status in many common scenarios.

While the issue of taint explosion undoubtedly merits an in-depth study, our analysis and experience with the PIFT prototype suggest that the highly pessimistic conclusions regarding the utility of pointer tracking presented in these previous studies are unwarranted. As we demonstrate in Chapter 6, several simple preventive steps can be taken to eliminate kernel-level taint explosion, allowing us to proceed with comprehensive tracking of all direct and indirect information channels.

Finally, the example in Figure 3.3 illustrates an *implicit* information channel that does not get tracked by PIFT. Note that the value of the input argument `v` influences the conditional branch instruction at offset 0xa: if `v` is not equal to 0, the execution jumps to 0x15 and otherwise, it proceeds to the next instruction (0xc). In both cases, this function loads an immediate (constant) value (0 or 1, depending on which branch is taken) into a tem-

```
int is_nonzero(int v) {
  if (v == 0)
    return 0;
  else
    return 1;
}
```

```
<is_nonzero>:
 +0x0 push %ebp
 +0x1 mov %esp,%ebp
 +0x3 sub $0x4,%esp
 +0x6 cmpl $0x0,0x8(%ebp)
 +0xa jne <is_nonzero+0x15>
 +0xc movl $0x0,-0x4(%ebp)
 +0x13 jmp <is_nonzero+0x1c>
 +0x15 movl $0x1,-0x4(%ebp)
 +0x1c mov -0x4(%ebp),%eax
 +0x1f leave
 +0x20 ret
```

Figure 3.3. An example of instruction-level IFT: the `is_nonzero` function.

porary memory location and subsequently transfers it into `eax`. An immediate value is, by definition, non-sensitive and hence, this function will always return a value tainted with $L_\emptyset$, regardless of how the input value in tainted. Putting it differently, this function leaks one bit of information about the input value `v`.

Prior work [58, 26] suggests that implicit channels are extremely difficult to track through runtime dynamic analysis and most previous approaches that attempt to track such channels rely on some form of static analysis at the source code level. The inability to track implicit channels is problematic in the presence of malicious code, since they provide a relatively easy way to "launder" sensitive data for exfiltration. However, since our current focus is on securing the flow of information in a benign environment, implicit flows do not present a major problem for PIFT. In the course of our initial experimentation with the system, we confirmed that non-malicious applications rarely, if ever, leak information through implicit channels and that our current tracking mechanisms comprehensively capture all explicit data manipulation activity in several widely-used applications.

3.3 System Architecture

Figure 3.4 sketches the high-level architecture of PIFT. The focal component of our design is an augmented hypervisor — a thin software layer that exposes the underlying hardware platform in virtualized form and allows several virtual machines to execute concurrently. Our current system prototype is based on Xen [89, 6] — an open-source hypervisor platform that achieves high performance on x86 by implementing the *paravirtualization* model of virtualized execution. In our architecture, the hypervisor conceptually extends the capabilities of the underlying physical machine, giving it the ability to track the flow of information with high precision — at the granularity of individual bytes and machine instructions.

All user-facing applications run inside a *protected virtual machine (VM)* on top of the

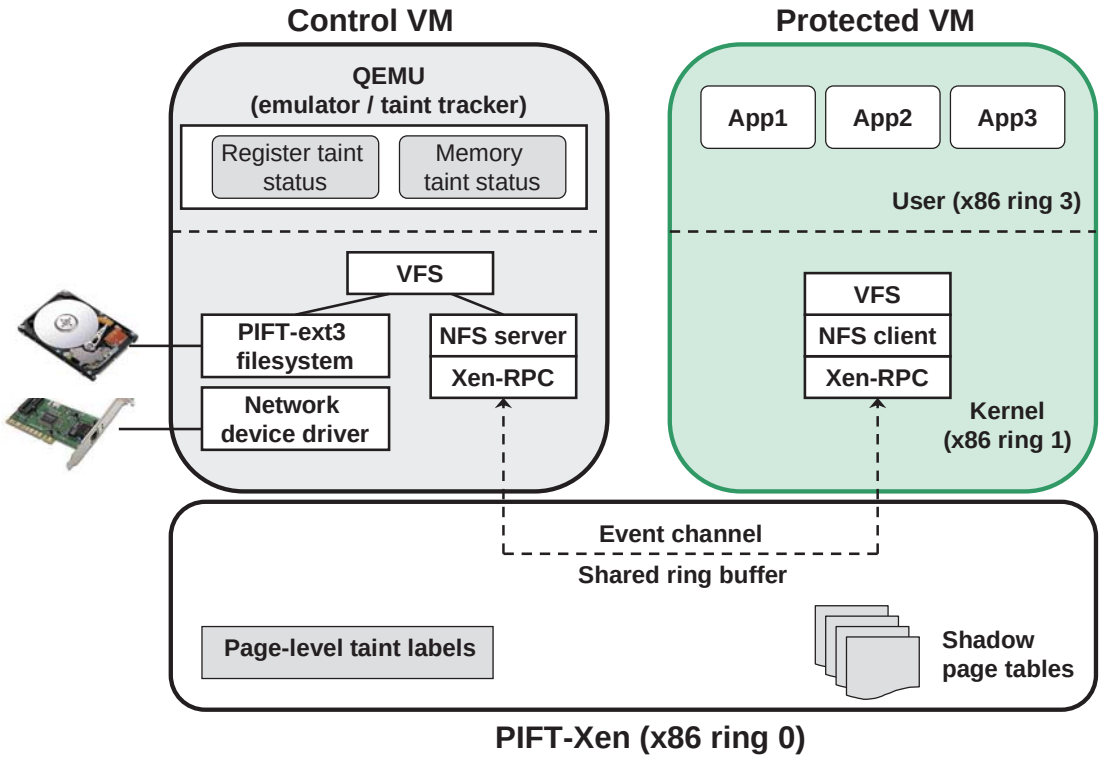


Figure 3.4. The high-level architecture of PIFT.

hypervisor. In addition, a specialized *control VM* operating in background provides a number of supporting modules: a robust full-system emulator, a label-aware filesystem, and drivers for virtualized I/O devices. PIFT-Xen tracks the propagation of labels between the virtual CPU registers, memory, and disk belonging to the protected VM. The hypervisor also intercepts all externally observable output actions (e.g., network communication, writing data to a mobile storage device, sending data to a printer) and enforces security policies, allowing or denying specific application requests to externalize sensitive data.

We illustrate the overall machinery of PIFT by walking through a typical usage scenario — enforcing confidentiality policy on the contents of a sensitive file f stored on a user's local machine. For simplicity of exposition, we assume that the user taints the entire file with a single label containing one policy p and that this file initially resides on the local disk.

Initially, all application- and OS-level code in the protected VM executes at native speed directly on the host CPU, but the hypervisor instruments the hardware page tables in a manner that allows us to intercept instructions that access tainted memory pages. When an application first opens the sensitive file, the call is routed via the hypervisor to the label-aware filesystem running in the control VM. Before returning the file handle, the filesystem makes a call to the hypervisor, informing it of the file's label. In response, the hypervisor marks the memory pages holding the file contents as tainted and updates the page tables accordingly.

When an application running in the protected VM tries to access the file contents from

a tainted memory page, the hardware memory management unit generates a page fault and immediately transfers control to PIFT-Xen. The hypervisor suspends the native execution context of the protected VM, takes a snapshot of its CPU register state, and transfers control to our augmented emulator, which resumes the execution of the protected VM in emulated mode. Our current implementation handles emulation using a heavily-modified version of QEMU, which runs as a user-level process in the control VM. The emulator is instrumented to track the movement of labels in accordance with the IFT model described in Section 3.2.

On a conceptual level, PIFT associates a data label with each individually-addressable byte in the protected VM, including:

- **Volatile memory**: PIFT maintains a label for each byte of physical memory allocated to the protected VM. Since a straightforward linear mapping would incur prohibitive storage costs, PIFT stores memory labels in a page table-like data structure that exploits spatial locality and achieves a reasonable trade-off between the storage overhead and the latency of label lookups.

- **User CPU registers**: PIFT maintains a label for each byte of every data register accessible from application-level (non-privileged) code. On the x86 platform these include:

  – The general-purpose integer registers, excluding the stack pointer (`esp`) and the stack base pointer (`ebp`).

  – The FPU register stack (`ST0` through `ST7`).

  – Registers associated with the various vendor-specific extensions to the core instruction set, such as SSE2 and MMX.

- **Network**: Our design seeks to provide end-to-end information tracking guarantees in a distributed setting. To this end, PIFT annotates the payload of outgoing packets with the associated labels. Upon receiving a packet from a remote PIFT-enabled endpoint, the hypervisor analogously transfers the labels associated with the payload to memory.

While emulating the protected virtual machine, QEMU executes additional logic to update data structures that keep track of labels for machine registers and memory addresses. When the protected VM ceases to manipulate tainted data, QEMU suspends the emulated machine context and notifies the hypervisor, which reverts the protected VM back to native virtualized execution.

When the protected VM tries to externalize tainted data through an I/O device, PIFT intercepts the device request and invokes the appropriate security checks. These operations are handled by a group of device-specific *interception modules* that are implemented as extensions to the backend device drivers operating in the control VM. As an illustrative example, if an application tries to externalize the data derived from f through a virtualized network interface, the backend driver for the network card would intercept the outgoing network packets and invoke a security check based on the label and the policies attached

to the payload. As an example, the policy may specify that the contents of f may not be forwarded to network endpoints situated outside the organizational boundaries. In this case, the backend driver would inspect the destination network address and, if this address specifies an external destination, it would drop the offending packets and signal an error condition to the frontend driver. The error handler in the frontend component provides a convenient insertion point for custom policy filters, which can either propagate the error up the stack, log it for audit purposes, or filter the content being externalized. To ensure end-to-end tracking between PIFT-enabled endhosts inside the enterprise, the network enforcement module also prepends all outgoing packets with a shim header carrying the associated labels.

One must keep in mind that a decentralized label can hold an arbitrarily large set of policies and each policy can define an arbitrary number of reader principals. Of course, it would be impractical to annotate each byte of memory and disk space in the protected VM with a complete and self-contained representation of the corresponding label, since the latter can be arbitrarily large. Instead, PIFT introduces a level of indirection and maps decentralized data labels onto a space of opaque 32-bit values called *taint labels*. These fixed-length surrogates carry no inherent meaning and their sole purpose is to provide a concise, easily-manipulatable, and globally-recognizable name for a decentralized data label. The mappings between these 32-bit names and the respective data labels are maintained using an external infrastructure, whose design and implementation are beyond the scope of this dissertation. PIFT does not specify how these mappings are to be stored and distributed, nor does it impose any restrictions on the meaning of the term *principal*. Our system is also oblivious to the specifics of authentication procedures that establish user credentials and bind network endpoints to principals.

Given these explicit non-goals, the overall purpose of PIFT is to provide a robust and comprehensive information flow tracking substrate that operates at the level of abstract taint labels. Our system monitors the computation inside the protected VM, tracks the propagation of taint labels at the instruction level, and invokes external user-supplied security checks when it detects an attempt to disclose a piece of sensitive data to another principal through device I/O. It is up to these external security modules to resolve the 32-bit taint value into the corresponding data label and evaluate the associated policies.

3.3.1 PIFT-ext3: A Label-Aware Filesystem

A comprehensive IFT solution requires the ability to track the flow of tainted data to and from persistent storage. Our system enables this capability by providing a specialized label-aware filesystem.

When it comes to designing a persistent storage layer for tainted data, one of the central design considerations involves choosing the level of abstraction, at which to maintain these labels. As an alternative to designing a specialized filesystem, PIFT could instead maintain the taint metadata at the level of the underlying physical block device, for instance by associating taint labels with individual disk sectors. While a sector-based taint

storage scheme would lead to a simpler (and arguably more efficient) design, we believe that a filesystem-based solution is a more desirable option, as it allows PIFT to deliver the expected behavior in a variety of common usage scenarios. For example, consider the action of appending new data to an existing file. If the file has been tainted with a specific confidentiality policy p, the owner of p probably expects that any new data subsequently appended to this file will automatically become associated with this policy, without requiring her to re-apply the label to the entire file. Such semantics would be difficult to provide in a design, which maintains taint labels only at the level of physical disk sectors and, as the above example suggests, the notion of *file-level* labels appears to be useful in practice.

Our label-aware filesystem is based on an augmented version of ext3 [87] — one of the most robust and mature Linux filesystem implementations currently in use. PIFT-ext3 maintains additional on-disk metadata that concisely represents the taint status of each individual byte offset within a file, as well as a file-level taint label that logically covers the entire length of the file. The file-level label is maintained in a new dedicated i-node field, while for byte-level taint values we extend the format of the leaf indirect block to carry pointers to *block taint descriptors* alongside pointers to the data blocks themselves. A block taint descriptor is a new on-disk data structure that compactly stores byte-level taint labels for the corresponding data block.

PIFT-ext3 is a large and complex software module that, like most other full-strength Linux filesystems, integrates directly with the kernel. Deploying a PIFT-ext3 partition directly within the protected VM would be highly disruptive and require significant changes to its software stack (at the very least — recompiling its kernel or inserting a new custom-built kernel module). Since we seek a solution that requires little or no change to the software running in the protected VM, we look for alternative and less disruptive methods of making this functionality available in the user-facing environment.

In the current design, we create and deploy a PIFT-ext3 filesystem inside the control VM and export it to the user-facing VM through a standard remote file access protocol (NFSv3). While this split-up configuration is not as efficient as running PIFT-ext3 directly inside the protected VM and incurs additional latency overhead, it allows us to maintain full compatibility with unmodified OS binaries. Furthermore, while our current prototype implementation focuses on supporting paravirtualized Linux guests, this scheme enables us to reuse our current filesystem implementation for other guest operating systems we may support in the future. In principle, any guest OS that can run an NFS client can connect to a PIFT-ext3 partition and take advantage of its taint storage functionality.

As a performance optimization, we designed and implemented a custom RPC transport mechanism that optimizes the transfer of file data between the front-end (NFS client in the protected VM) and the backend (NFS server in the control VM) components. By default, NFS uses TCP as the underlying transport for its client-server communication and the two sides submit RPC messages directly to the Linux networking stack. NFS commands that carry file data (such as a `WRITE` request or a response to a `READ` request) must execute several expensive memory transfer operations that impose a heavy load on the memory subsystem and increase latency. Typically, the sender first copies the data from the filesystem cache into its network socket buffers and then transfers the data to the destination machine via a

memory copy to its network buffers. When the destination machine receives these packets, it performs yet another memory transfer to move the data from the network buffers into its local filesystem cache. These memory transfers are unnecessary in our configuration and represent pure overhead, since the client and the server operate on the same physical host and share its physical memory address space. Our new RPC transport mechanism (Xen-RPC) takes advantage of this property and provides a way to *efficiently* transfer file data between a pair of VM instances by setting up temporary shared memory page references. Xen-RPC eliminates the unnecessary memory transfers to and from network-level buffers and enables direct transfers between filesystem caches. Section 4.3 describes Xen-RPC and the other components of our label-aware filesystem in further detail.

3.3.2 Comparing PIFT to Existing Hypervisor-Based IFT Systems

At a high level, the overall machinery described thus far is analogous to Neon [98] (another recent proposal) and our system makes use of similar building blocks; namely, a hypervisor and an emulator augmented with instruction-level IFT. However, Neon fails to meet the important requirements of high performance and correct yet parsimonious label propagation for the following reasons:

1. Taint tracking by plain instrumented emulation is extremely expensive. The results reported in the Neon study indicate that even a simple computation on tainted data can incur a slowdown on the order of $95\times$ when only 1/64th of the input file is tainted and no data is provided on how the system behaves with a more stressful amount of taint. Such a slowdown is unacceptable in practice and significantly hinders the adoption of dynamic real-time IFT systems for everyday use.

2. To be comprehensive, IFT systems have to track indirect information flows resulting from pointer-based memory references. However, prior work [80] has shown that this leads to accidental tainting of kernel data structures. Any other application that interacts with the kernel also acquires taint and eventually, the taint status propagates to all data in the system. Such taint explosion renders the whole system ineffective (since the taint ceases to have the correct significance) and substantially impairs the performance of the system, as running in emulated mode incurs a heavy performance penalty.

PIFT proposes and implements several novel techniques that help us address the above challenges and thus bring real-time information flow tracking significantly closer to the realm of practicality. Specifically:

1. PIFT performs taint tracking in the emulator at a higher abstraction level than Neon and other previous systems. Emulators such as QEMU break down each emulated guest instruction into a series of micro-instructions. Prior work performs taint tracking by instrumenting each micro-instruction to propagate taint, which incurs a significant, but non-essential overhead. In contrast, PIFT tracks the flow of information

directly at the level of native machine instructions (x86 in our implementation). As we explain in Section 4.2, tracking at a semantic level that matches the physical architecture of the emulated machine enables a range of optimizations that are difficult or altogether impossible to apply at the micro-instruction level.

2. PIFT performs information flow tracking asynchronously and in parallel with the main emulation codepath. The key insight is that up-to-date label information is needed only at the point where policies are invoked. Hence, instead of tracking the propagation of data labels synchronously and in lockstep with emulation, PIFT generates a separate stream of taint tracking instructions and executes them asynchronously on another processor core.

   In Chapter 5, we demonstrate that asynchronous tracking performed at a level of abstraction that directly matches the architecture of the emulated machine can produce a $60\times$ performance improvement over the best previous results.

3. Finally, we identify via empirical evaluation that accidental tainting of kernel data structures happens through a very narrow interface — a few specific functions in the kernel. We design techniques to intercept such channels of taint explosion and securely control taint flow, such that kernel data structures do not unnecessarily get tainted. We propose several minor modifications to the Linux kernel that eliminate accidental tainting and solve the kernel taint explosion problem for all practical purposes.

Chapter 4

Prototype Implementation

We have implemented a proof-of-concept prototype of PIFT and this chapter presents a detailed description of our implementation. Before we proceed to this description, we note that PIFT is a large and relatively complex system that unifies a hypervisor-based virtualization environment with a fully-featured emulator and augments the resulting platform with policy enforcement and IFT capabilities. While the traditional practices of software engineering encourage layering, modularization, and well-defined interfaces and these principles are, without a doubt, useful in a variety of contexts, it is quite common for low-level systems projects to deviate from these principles and espouse a non-modular monolithic design in an effort to improve performance.

The Linux kernel, the Xen hypervisor, and QEMU (the core building blocks, on which our implementation is based) are best viewed as monolithic designs and each of these systems implements a range of non-trivial optimizations that improve performance, but violate modularity. Unsurprisingly, PIFT is also an unambiguous example of a monolithic system, and one that readily forfeits architectural elegance in favor of runtime performance.

This property makes it slightly more challenging to present a clear and structured description of our implementation, as its components cannot be easily broken apart and presented in isolation. Still, we make an effort to modularize our discussion as much as possible and divide our description of the implementation into five self-contained sections, which correspond to the high-level architectural components of PIFT. Section 4.1 describes the hypervisor-level component and our extensions to Xen. Section 4.2 describes the internals of the emulation component, which encapsulates most of the IFT functionality. We present the implementation of our taint-aware filesystem in Section 4.3 and discuss policy enforcement in Section 4.4. Finally, Section 4.5 explains how to extend the single-node PIFT implementation to a distributed environment and provide the ability to track information flow across network transfers.

While a comprehensive full-system performance evaluation of our prototype is the subject of Chapter 5, we are also interested in understanding the performance characteristics

of the individual components. Hence, in the following sections we also report the results of our low-level performance measurements based on component-specific microbenchmarks.

4.1 The Hypervisor-Level Component of PIFT

The focal component of the PIFT architecture is an augmented hypervisor, which monitors the protected VM and selectively enables emulation to execute the regions of code that manipulate tainted data values. Our prototype implementation is based on Xen (version 3.3.0) — a robust open-source hypervisor platform that achieves high performance on commodity processors through paravirtualization. In this section, we describe the hypervisor-level component of PIFT; we start off with a general overview of Xen (Section 4.1.1) and then present our extensions (Section 4.1.2).

4.1.1 Overview of Xen

Xen [89, 6] is a widely-used hypervisor-driven virtualization system that originated as a research project at the University of Cambridge. It operates on commodity hardware platforms and enables multiple strongly-isolated OS images to run concurrently on a single host machine.

Although the widely-deployed x86 architecture does not easily lend itself to full virtualization, requiring the use of complex and computationally expensive binary rewriting techniques to virtualize certain privileged instructions, Xen sidesteps these issues by presenting a somewhat simplified VM abstraction that is similar, but not completely identical, to the underlying physical machine. This approach, termed *paravirtualization* [88], allows Xen to run multiple isolated OS instances on a single physical x86 processor with high performance, but requires the guest OS to be explicitly ported to the paravirtualized interface.

The hypervisor interposes itself between the hardware platform and the set of virtual machines, mediating all access to the physical resources, as well as all inter-VM communication. All controlled interactions between the hypervisor and an overlying guest VM are implemented upon the foundation of two generic control mechanisms: synchronous *hypercalls* and asynchronous *event notifications*. Hypercalls are analogous to the *system call* facility provided by conventional operating systems and allow guest VMs to perform privileged operations by trapping into the hypervisor. As a specific example of hypercall usage, Xen exposes hardware page tables to the guest environment in read-only mode in order to ensure proper isolation of memory resources and the guest kernel must issue a hypercall if it wishes to update an entry in its page table. Recent versions of Xen implement hypercalls using the software interrupt instruction (`int 0x82`) and pass hypercall arguments in general-purpose integer registers.

Lightweight event notifications are a form of virtual interrupts; they replace the

usual delivery mechanisms for hardware device interrupts and allow Xen to communicate other low-level system events to a guest VM asynchronously. Xen events also serve as the primary means of inter-VM communication, allowing a pair of virtual machines to signal each other in a controlled and lightweight manner. Pending notifications are maintained using a per-VM bitmap of event types, which resides in a shared memory page. To deliver an event to a guest system, Xen updates this bitmap, interrupts the guest VM, and redirect its execution to an *event callback* — a specialized routine in the guest kernel, whose address (`arch.guest_context.event_callback_cs` and `arch.guest_context.event_callback_eip`) is specified by the guest system during startup. The guest also specifies the address of a kernel-level stack (`arch.guest_context.kernel_ss` and `arch.guest_context.kernel_sp`) to be used for executing the event callback and the hypervisor ensures that the stack pointer registers (`ss` and `esp`) are loaded with these values prior to invoking the callback routine. Analogously to non-virtualized "bare-metal" platforms, which typically allow the OS kernel to temporarily suspend the delivery of hardware interrupts, Xen can disable the invocation of event callbacks at the discretion of the guest kernel.

In addition to mediating the reception of asynchronous device interrupts, Xen interposes on the delivery of all synchronous processor exceptions, such as page faults and software interrupts. On the x86 platform, this is accomplished by modifying the contents of the interrupt descriptor table (IDT) and replacing the guest's descriptor entries with alternate descriptors that reference a hypervisor-level handler. As a result, every hardware exception triggers a trap into the hypervisor, where the event is examined and, in typical cases, relayed to the appropriate guest machine for processing. One notable exception in this scheme, introduced in recent versions of Xen as a performance optimization, pertains to the use of the `int 0x80` instruction. This instruction provides the standard mechanism for invoking system calls on Linux, but trapping into the hypervisor upon each system call entry can become a source of significant overhead. Thus, recent versions of Xen allow paravirtualized Linux guests to register a guest-level interrupt descriptor for this particular interrupt type. Xen loads this descriptor (`arch.int80_desc`) directly into the hardware IDT, thereby allowing guest applications to invoke system calls without the hypervisor's involvement.

MMU Virtualization and Shadow Paging

Xen's approach to virtualizing the functions of the memory management unit (MMU) deserves special attention in the context of our discussion. In the paravirtualized model, the guest OS kernel is exposed to real physical memory addresses and assumes partial responsibility for managing its own page tables. During normal operation, the hypervisor exposes the guest's hardware page tables (i.e., those that are loaded into the physical MMU) directly to the guest OS without any form of translation or virtualization, but restricts the guest's access privileges to read-only. Thus, the guest kernel can read its virtual-to-physical mappings, but does not have the privileges to modify them or to switch page tables. To

update an entry in its page table, the guest invokes a hypercall to Xen, which performs the appropriate security checks and applies the update.

Recent versions of Xen support a variation of this strategy, termed *shadow paging*, which was initially introduced to facilitate the tracking of writable working sets for live VM migration [16]. When this mode of operation is enabled, the hypervisor maintains a private (shadow) copy of each guest page table and exposes these internal copies to the paging hardware. The shadow tables are completely invisible to the guest environment and are populated on-demand by translating the corresponding sections of the guest page tables. When the guest kernel issues a hypercall to update a page table entry (PTE), Xen validates the request and updates the corresponding PTE in the shadow table.

This approach to virtualizing the MMU incurs additional management overhead, requiring the system to maintain and update two distinct sets of page tables, but makes it easy to deploy a range of advanced and novel features in a manner that is entirely transparent to the guest environment. For example, the hypervisor can be configured to track the writable working set in the guest system by initializing the shadow PTEs with read-only mappings that are otherwise identical to the original guest mappings. Then, if the guest VM tries to write to a page of memory, the hypervisor can trap the resulting page fault and update the working set information [16]. As we explain below, PIFT implements similar mechanisms and leverages the shadow paging infrastructure to detect and intercept the initial access to tainted memory areas.

4.1.2 Transforming Xen into a Comprehensive IFT Platform

Xen provides a suitable and attractive foundation for a comprehensive information flow tracking platform such as PIFT. While some of the previous efforts [61] propose running the guest environment inside a full-system emulator (such as QEMU) augmented with taint tracking, PIFT explores a more intricate design that combines an emulation environment with a hypervisor-based virtualization platform. In PIFT, these modules operate in concert to enable a novel feature called *on-demand emulation* — the ability to seamlessly move the guest system between *virtualized* execution within a Xen VM and *emulated* execution within QEMU. In the first mode, the guest system runs at native speed directly on the physical CPU with minimal supervision and with little or no additional overhead. In the second mode, the system runs on an emulated processor and suffers the overhead of emulation, but benefits from the ability to track information flow at the level of machine instructions. On-demand emulation allows PIFT to improve the performance of full-system information flow tracking by dynamically switching between these modes and enabling the IFT computation only when needed. In principle, any piece of OS- or application-level code that does not interact with tainted data (and thus does not modify the state of taint labels) can be executed at native speed in the virtualized mode and heavyweight information flow analysis must be enabled only for those regions of code that directly manipulate sensitive data.

On-demand emulation is a powerful technique that can help PIFT fulfill its promises of

comprehensive tracking and high performance. From a practical standpoint, enabling this feature requires addressing two non-trivial technical challenges, specifically:

1. We must design a mechanism that enables the hypervisor to detect and securely intercept the initial access to tainted data during virtualized execution.

2. We must design a mechanism that enables seamless migration of processor state between a paravirtualized Xen VM and an emulated machine. The transition must be performed in a manner that is fully transparent to the protected environment and does not overburden the system with context-switching overhead.

Fortunately, both mechanisms can be realized with a modest number of extensions to the standard implementation of Xen and we describe these extensions on the following pages.

Intercepting the Initial Access to Sensitive Data

The key challenge in trapping tainted data access is efficiency. A naïve implementation would trap to the hypervisor upon *every* memory access from the guest VM to determine whether sensitive data is being accessed, but this strategy would incur unacceptable overhead. Instead, we leverage the capabilities of the hardware paging unit and configure it to generate a trap upon every access to a memory page that is known to contain sensitive data. To accomplish this, PIFT-Xen creates a set of shadow page tables for the guest environment, clearing the `PAGE_PRESENT` (`P`) flag in the shadow PTEs that hold mappings to tainted memory pages. Thus, when the guest VM tries to access a tainted page (either for reading or writing), the memory management unit generates a page fault and transfers control to a hypervisor-level fault handler.

The set of tainted memory pages is stored using a page-level bitmap — a simple data structure managed by the augmented emulator and mapped for shared access from the hypervisor context. This data structure maintains one bit for each page of physical memory assigned to the protected VM and the emulator is responsible for synchronizing its contents with the fine-grained byte-level memory taint data structures.

The PTE modification logic is implemented via a simple extension to the `_sh_propagate` function (defined in `xen/arch/x86/mm/shadow/multi.c`) — the "heart" of the shadow paging code, which constructs the shadow PTEs from the corresponding guest entries. In this function, we clear the `PAGE_PRESENT` bit in the shadow PTE if the physical memory page referenced from the PTE is marked as containing sensitive data according to the bitmap. Figure 4.1 illustrates the format of a page table entry on a 32-bit machine, highlighting the position of the `PAGE_PRESENT` bit.

Naturally, we must also extend Xen's page fault handing mechanisms to differentiate a genuine page-not-present condition from the side-effects of shadow paging. Our current implementation modifies the `sh_page_fault` routine (also defined in `xen/arch/x86/mm/shadow/multi.c`), which is invoked to handle a hardware page fault

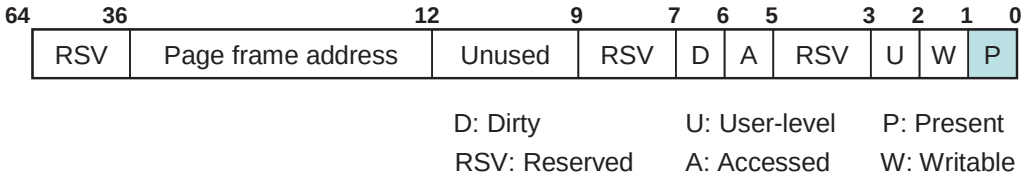


Figure 4.1. The format of a leaf page table entry on a 32-bit PAE-enabled x86 machine.

in the shadow paging mode. In this function, we walk the guest page table to obtain the guest PTE and then examine its contents. If the guest PTE references a valid physical page, the fault must have been triggered by an attempt to access a tainted memory area and in this case, PIFT-Xen initiates a transition to emulation by setting the `enable_emulation` flag in the `vcpu` structure representing the faulting virtual processor. Otherwise, if the page is marked as "not present" in the guest PTE, the hypervisor invokes the non-interception codepath, which propagates the fault to the guest OS using existing mechanisms.

Note that while the guest system accesses and manipulates tainted data at the granularity of machine words, PIFT-Xen's faulting mechanism maintains a more coarse-grained view, which allows us to intercept access only on the basis of page-level taint bits. This mismatch can be seen as an inherent limitation of our design, which hurts performance in certain cases by incurring unnecessary context switches and transitions to the emulated mode. On a conceptual level, this issue bears a strong resemblance to the problem of false sharing that affects memory coherence protocols on modern multiprocessors [9]. Unfortunately, commodity x86 processors do not offer a mechanism for generating faults upon access to specific byte-level memory addresses. However, prior work [72] has demonstrated that ECC-enabled memory controllers can be used in novel ways to implement finer-grained memory fault mechanisms and we believe that applying similar techniques to PIFT can help ameliorate this mismatch.

Switching between Virtualized and Emulated Execution

Migrating the protected system from the virtualized mode of execution to the emulated mode involves suspending the native VM, producing a comprehensive snapshot of its virtual CPU state, and initializing the emulated processor from this snapshot. In PIFT, the hypervisor and the emulator coordinate their activities and exchange state using a common data structure (`struct shared_info_xen_qemu`). This data structure resides on a shared memory page and Figure 4.2 illustrates its format.

The starting point for a virtual-to-emulated (V2E) transition is the page fault handler (`_sh_page_fault`), which, as we explain in the preceding subsection, consults the page-level taint status bitmap and sets the `enable_emulation` flag if the page fault was a side-effect of accessing a tainted memory page. This flag instructs Xen to suspend the native guest VM and switch to emulation immediately upon return from the hypervisor context.

```
struct shared_info_xen_qemu {
   struct guest_cpu_context ctxt;         /* Snapshot of the virtual CPU      */
   int evtchn_upcall_pending;                    /* 'Event pending' flag      */
   int status;        /* PIFT_EMULATION_REQUESTED or PIFT_EMULATION_COMPLETED */
};
```

Figure 4.2. The format of the `shared_info_xen_qemu` structure.

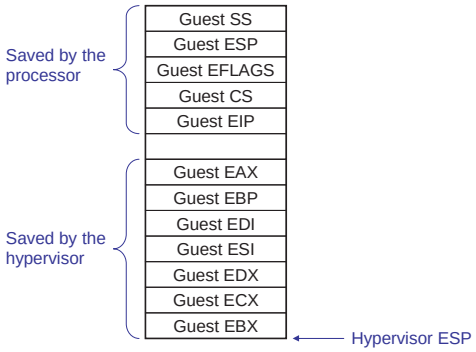


Figure 4.3. The contents of the hypervisor-level stack upon entry to `restore_all_guest`.

The real work begins in `restore_all_guest` (`xen/arch/x86/x86_32/entry.S`) — the final stage of the hypervisor exit codepath, which restores the guest's CPU state and returns control to the VM by executing the `iret` instruction. Restoring the processor state involves loading the hardware CPU registers with the corresponding guest register values. For the purposes of this discussion, it is important to note that Xen maintains the guest registers on the hypervisor's stack, as Figure 4.3 illustrates. We modify the `restore_all_guest` code block to check the value of the `enable_emulation` flag. If this flag is set, the hypervisor invokes the central `pift_emulate_guest` function, whose implementation is illustrated with pseudocode in Figure 4.4. This function can be broken down into three distinct phases, which correspond to the V2E transition, the period of emulated execution within QEMU, and the reverse emulated-to-virtual (E2V) transition.

In the first phase, the hypervisor initializes the `guest_cpu_context` structure, which encapsulates a comprehensive snapshot of the virtual CPU. Table 4.1 details the individual fields of this data structure and describes how their values are obtained during the transition. The snapshot of user-level registers (`eax`, `ebx`, `esp`, `eip`, and others) is initialized from the guest CPU context residing on the hypervisor's stack, while control register snapshots (`cr0`, `cr3`, `cr4`) are initialized by reading the live values in the corresponding physical registers. The FPU context is recorded by executing the `fxsave` instruction, which marshals the complete state of the FPU into a 512-byte memory buffer. Note that we must also record and transfer the values of several additional variables that do not represent any specific elements of the *physical* CPU, but are essential to attaining a comprehensive representation of the *virtual* machine. These variables can be viewed as artefacts of paravirtualization

```
extern struct shared_info_xen_qemu *shared_info;

asmlinkage void pift_emulate_guest() {
 /*** Phase 1: virtual-to-emulated (V2E) transition ***/
 save_shared_cpu_snapshot() {
  save user registers
  save segment registers
  save FPU state
  save control registers
  save clock cycle counter
  save local and global descriptor table registers
  save artefacts of paravirtualiation
 }

 local_irq_enable();                        /* Enable hardware interrupts */

 shared_info->status = PIFT_EMULATION_REQUESTED;  /* Notify the emulator */
 send_guest_vcpu_virq(dom0->vcpu[0], VIRQ_PIFT_EMULATE);

 /*** Phase 2: emulated execution ***/
 while(shared_info->status != PIFT_EMULATION_COMPLETED) {
  process_pending_timers();
  if (current->vcpu_info->evtchn_upcall_pending & 0xff) {
   shared_info->evtchn_upcall_pending = 1;
   mb();                                              /* Memory barrier */
  }
 }

 /*** Phase 3: emulated-to-virtual (E2V) transition ***/
 local_irq_disable();                      /* Disable hardware interrupts */

 restore_from_shared_cpu_snapshot() {
  restore user registers
  restore segment registers
  restore FPU state
 }

 invalidate_dirty_shadow_ptes() {
  for (each page p in the dirty page list)
   sh_remove_all_mappings(current, _mfn(p));
 }

 current->enable_emulation = 0;
 return;                                   /* Return to restore_all_guest */
}
```

Figure 4.4. The implementation of the `pift_emulate_guest` function.

and include the kernel stack pointer (`kernel_ss` and `kernel_sp`), the address of the event callback (`callback_cs` and `callback_eip`), and the interrupt descriptor to be used for servicing guest system calls (`int80_desc`). Once a comprehensive snapshot has been obtained, the hypervisor signals QEMU (operating as a user-level process in the control VM)

Fields	Initialized from

`/* User registers */`
`unsigned int eax, ebx, ecx,`
`edx, esi, edi, ebp,`
`esp, eip, eflags;`
	Values saved on the hypervisor's stack

`/* Segment registers */`
`unsigned int cs, ss, ds,`
`es, fs, gs;`
	Values saved on the hypervisor's stack

`/* FPU state */` `char fpu_state[512];`	Hardware FPU context fetched using `fxsave`

`/* Control registers */`
`unsigned int cr0;`
`unsigned int cr3;`
`unsigned int cr4;`
	Physical register value fetched using `read_cr0`
	`arch.guest_table.pfn`
	Physical register value fetched using `read_cr4`

`/* Clock cycle counter */` `unsigned long long tsc;`	Physical register value fetched using `rdtsc`
`/* Global descriptor table */` `unsigned int gdt_base, gdt_limit;`	Physical register value fetched using `sgdt`
`/* Local descriptor table */` `unsigned int ldt_base, ldt_limit;`	Physical register value fetched using `sldt`

`/* Artefacts of`
`paravirtualization */`
`unsigned int callback_cs;`
`unsigned int callback_eip;`
`unsigned int kernel_ss;`
`unsigned int kernel_sp;`
`char int80_desc[8];`
	
	`arch.guest_context.event_callback_cs`
	`arch.guest_context.event_callback_eip`
	`arch.guest_context.kernel_ss`
	`arch.guest_context.kernel_sp`
	`arch.int80_desc`

Table 4.1. The components of the `guest_cpu_context` structure (left) and the sources, from which they are initialized (right).

through a virtual IRQ and instructs it to initiate emulation. When QEMU receives this signal, it initializes the state of the emulated CPU based on the information contained in the snapshot and launches the main emulation loop.

In the meantime, the native virtual machine enters the second stage of `pift_emulate_guest`, during which it waits for the completion of emulated execution. The current version of Xen does not allow blocking the guest context at an arbitrary point within the hypervisor, which is why our prototype currently implements a form of busy waiting. During the waiting period, the native version of the protected VM has the appearance of being "stuck" in the page fault handler, while in reality the VM continues executing on the emulated processor managed by QEMU. Further, as we explain in Section 4.2.7, all event signals (such as virtual timer interrupts and notifications from paravirtualized I/O devices) sent to the native VM in this phase must be intercepted by the hypervisor

and relayed to the emulated context in a proper manner. Currently, this is accomplished by polling the `current->vcpu_info->evtchn_upcall_pending` variable and propagating its value to `shared_info->evtchn_upcall_pending`. The emulator periodically polls the latter location and, when an event notification arrives, interrupts the emulated CPU and redirects execution to the guest event callback, imitating the actions of the hypervisor.

When QEMU decides to stop emulation and resume native execution, it signals this decision to the hypervisor by setting the `shared_info->status` variable to `PIFT_EMULATION_COMPLETED`. This action initiates the third and final phase of processing, in which the emulator and the hypervisor cooperate on performing the reverse (E2V) transition. QEMU writes the most recent (post-emulation) CPU register values to the shared snapshot and Xen transfers them back to the hypervisor-level stack of the native VM.

Note that prior to relinquishing control, the hypervisor must update the native VM's shadow page tables and synchronize them with the most recent state of the page-level taint bitmap in order to ensure that all subsequent accesses to tainted memory locations are properly trapped. The most straightforward way to accomplish this is by destroying all existing shadow tables and letting the hypervisor reconstruct them on-demand, one entry at a time, in response to subsequent page faults. However, this technique is not very practical, since it would incur an enormous performance penalty, triggering page faults upon *every* subsequent memory access irrespective of the page taint status.

PIFT implements a different and somewhat less heavy-handed strategy based on the observation that the taint status of a memory page (and hence its shadow PTE) can change *only* if the page has been modified during emulated execution. Hence, we instrument QEMU to maintain a list of dirty memory pages and provide this list to the hypervisor during the E2V transition. The hypervisor walks through the list of shadow page tables, examines their entries, and reconstructs those that reference a dirty memory page. This technique minimizes the number of subsequent page faults, but requires scanning all shadow PTEs to locate all mappings of a specific physical page — still a relatively expensive operation. Further improvements to our prototype may include adding an auxiliary data structure that will enable us to identify such mappings more efficiently.

In the final step, the `pift_emulate_guest` function returns, transferring control back to the hypervisor exit codepath (`restore_all_guest`). The hypervisor restores the native VM's processor registers from its stack and returns control to the VM by executing the `iret` instruction.

4.2 Information Flow Tracking with QEMU

Our system uses QEMU as a foundational building block for emulation and extends it with fine-grained information flow tracking capabilities. While the standard implementation of QEMU offers a self-contained and robust emulation environment, significant modifications and extensions were needed to transform it into a comprehensive and efficient

IFT platform. In this section, we describe the design and implementation of our extended emulator that tracks the flow of taint labels in the guest system and explain how it integrates with the other major components of PIFT.

Section 4.2.1 starts our discussion by reviewing the design and implementation of QEMU, with a special focus on its code translation mechanisms. Our approach to taint tracking is based on augmenting the emulated machine with a virtual hardware extension in the form of a *taint processor*. We discuss our general approach and highlight the key distinctions from earlier taint tracking systems in Section 4.2.2. Section 4.2.3 describes the instruction set architecture, upon which the taint processor is based and Section 4.2.4 illustrates its usage with several examples. We discuss the internals of the taint processor and introduce the key data structures for managing register and memory taint labels in Section 4.2.5. Section 4.2.6 discusses how our system exploits asynchrony and parallelism to improve the runtime performance of taint tracking. Finally, Section 4.2.7 discusses how we integrate the augmented version of QEMU with the other core components of the PIFT architecture, such as the hypervisor and the taint-aware filesystem. We defer the detailed performance evaluation of our taint tracking substrate to Chapter 5.

4.2.1 Overview of QEMU

QEMU [7, 70] is a robust open-source processor emulator, originally developed by Fabrice Bellard. It provides a full-system emulation environment for several popular hardware platforms, including x86, x86-64, ARM, Alpha, ETRAX CRIS, MIPS, MicroBlaze, PowerPC, and SPARC. The emulated (guest) machine executes in the context of a single user-space emulator process in the host OS and all elements of the guest machine state, including its CPU registers, physical memory, and various peripherals, are represented with corresponding data structures in the address space of this process. To emulate memory accesses from the guest machine, QEMU implements a software-based memory management unit that provides guest-virtual to guest-physical address translation, mimicking the behavior of a hardware MMU for the guest architecture. In addition, a software-based TLB maintains mappings between guest virtual addresses and the corresponding addresses in the emulator's own virtual address space.

Unlike Xen and other paravirtualized environments, QEMU does not require patching or otherwise modifying the guest operating system and does not alter the software-hardware interface. Conversely, QEMU aims to provide an accurate software-based representation of the guest hardware platform, whose behavior is indistinguishable from that of a real "physical" machine from the vantage point of the guest OS.

Like other powerful emulators, QEMU achieves good performance by implementing just-in-time dynamic code translation mechanisms. The basic unit of granularity for the purposes of code recompilation is a *translation block*, defined as a block of guest machine instructions terminated by a jump or by a CPU state change which cannot be deduced statically by the compiler [71]. When the emulator first encounters a new translation block (B_t), it analyzes the constituent instructions and generates a corresponding block of code

in the host instruction set (B_h), which emulates B_t and updates the logical representation of the guest machine according to its effects. The resulting code block B_h is executed natively on the host CPU and when its execution completes, control is transferred back to the main emulation dispatch loop. At that point, QEMU examines the emulated instruction pointer (`eip`) to locate the next translation block and then executes the corresponding native code block. In cases where the new instruction pointer value can be determined in advance (i.e., at the time of code translation), QEMU can patch the resulting code so that it jumps directly to the next native block without returning control to the central dispatch loop — an important performance optimization referred to as *translation block chaining*. For the purposes of PIFT, the emulated processor is simply a replica of the physical host CPU and thus, the dynamic code compiler is configured to perform x86-to-x86 translation.

Just-in-time code recompilation is a well-known technique for improving the runtime performance of emulated systems and has been successfully applied in a number of contexts. This approach is more complex, but at the same time vastly more efficient, than its main alternative — straightforward binary interpretation. Early on in the development process, we experimented with several other x86 emulation environments, which were all based on binary interpretation. We have found that they tend to impose a severe slowdown (up to 3 orders of magnitude relative to native code execution) and such costs are clearly unacceptable for our purposes.

On a more concrete level, code recompilation in QEMU is a two-stage process. In the first stage, the "frontend" component of the compiler (implemented in `target-i386/translate.c`) disassembles a basic block of guest code one instruction at a time and transforms it into a machine-independent intermediate representation. In this intermediate form, the code is expressed as a sequence of RISC-like microinstructions based on the TCG (Tiny Code Generator) notation. In the second phase, the "backend" component (implemented in `tcg/i386/tcg-target.c`) translates the TCG representation into a block of native instructions for the host machine. Several important optimizations, including liveness analysis and constant expression evaluation [5], are attempted during this stage. The results of code translation are cached in a pre-allocated memory buffer, which allows QEMU to amortize the computational costs of recompilation.

Next, we elucidate the inner workings of the QEMU code translator with a concrete example. Consider the `push %ebx` instruction, which pushes one of the general-purpose registers onto the stack. Figure 4.5 demonstrates the output from both stages of code translation for this particular instruction. In the first stage, QEMU decomposes it into a series of six TCG microinstructions. The first microinstruction loads a 32-bit word located at offset `0xc` from the memory address pointed to by `env` into a temporary internal variable `tmp0`. `env` is a pointer to a global data structure that maintains the complete state of the guest CPU and offset `0xc` represents the location of the emulated `ebx` register within this data structure. Hence, this microinstruction has the effect of transferring the current value of the guest `ebx` register into a temporary variable `tmp0`. Using similar machinery, the second microinstruction loads the emulated stack pointer into a temporary variable `tmp2`. The next two microinstructions decrement the value in `tmp2`, thereby updating the stack pointer in a manner that reflects the effects of a push operation with a downward-growing stack. Mi-

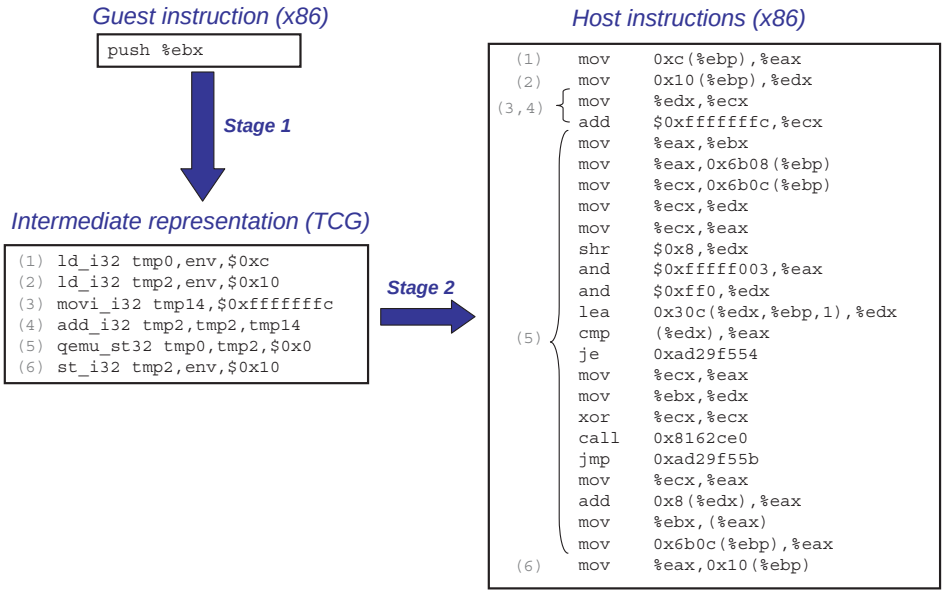


Figure 4.5. An example of dynamic code translation in QEMU.

croinstruction 5 performs an emulated memory store, which writes `tmp0` (holding the value of the `ebx` register) into the memory location that corresponds to the new top of the stack. The last microinstruction writes the updated stack pointer to its permanent location within the global `env` structure.

In the second phase of code translation, QEMU transforms the above sequence of microinstructions into native code for the host processor. In this phase, the compiler maps the abstract temporary variables onto the host CPU registers and attempts several optimizations. The block of machine instructions that emerges from this phase represents the final results of the translation process. As Figure 4.5 shows, it contains 25 instructions and combines two distinct execution paths segregated by a conditional branch. The "fast" path handles the case where the address translation entry for the current stack page is present in the software TLB, while the "slow" path, invoked in the event of a TLB miss, calls a precompiled helper routine (`call 0x8162ce0`) to resolve the translation entry using the guest page table. The fast execution path contains a total of 20 machine instructions, which include 9 memory accesses and 1 conditional branch. Viewed collectively, these numbers can be taken as a rough approximation of the fundamental cost of emulating a `push` instruction with the current implementation of QEMU.

In a number of special cases, QEMU refrains from dynamic code generation in the second stage of translation and instead redirects the host processor to a statically-compiled native routine, which updates the state of the emulated machine in the desired manner. This method of emulation is used to handle instructions with complex semantics and side-effects (often involving conditionals on the runtime state of the machine), for which writing a dynamic code translator would be a cumbersome and error-prone undertaking.

Figure 4.6 illustrates emulation using statically-generated helper routines on the ex-

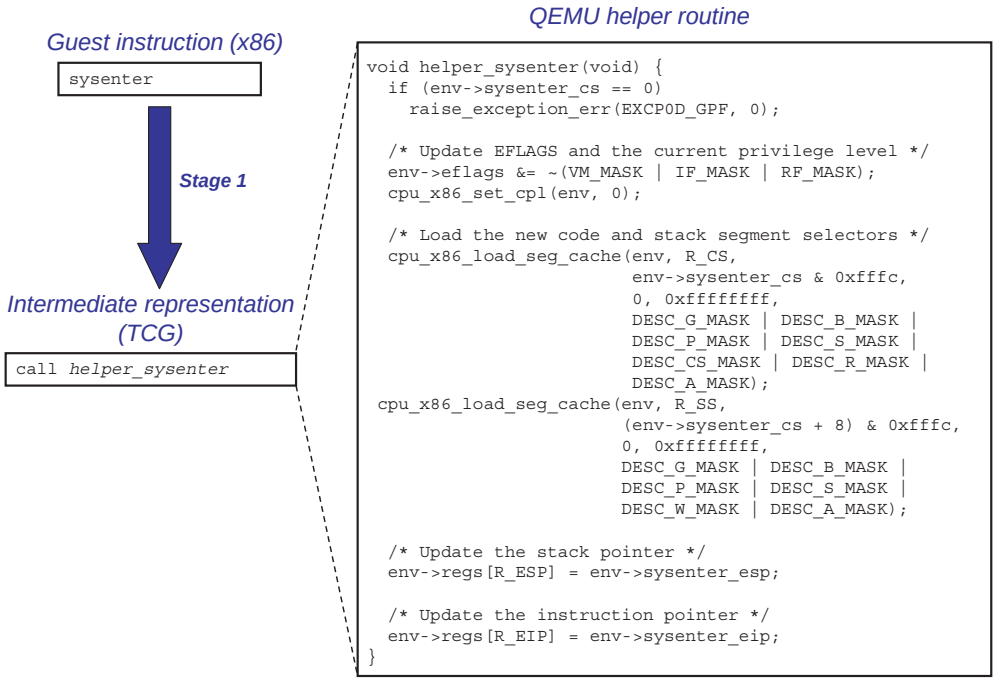


Figure 4.6. An example of code translation with static instruction handler routines.

ample of the `sysenter` instruction. This x86 instruction provides an efficient mechanism for invoking system calls and transferring control to the OS kernel without incurring the full overhead of a software interrupt. This instruction has relatively complex semantics, which involve changing the processor's privilege level, modifying several bits in the `eflags` status register, loading the new code and stack segment descriptors from the global descriptor table, and finally setting the instruction and stack pointers to preconfigured values maintained in a pair of designated model-specific registers (`sysenter_eip_msr` and `sysenter_esp_msr`). Under certain abnormal conditions, such as when `sysenter_cs_msr` contains an invalid kernel code segment selector, this instruction raises a processor exception and transfers control to a kernel-level exception handler routine instead of the system call handler. To emulate this complex sequence of operations, QEMU invokes a pre-compiled helper function `helper_sysenter`, whose source code is shown on the right side of Figure 4.6.

This emulation strategy results in a more readable and less error-prone implementation; after all, a C-language function is much easier to debug and maintain than a piece of compiler code that dynamically synthesizes a functionally equivalent block of machine instructions. However, as is often the case with systems software, readability and maintainability come at the cost of performance. Invoking a statically-compiled helper routine from the emulator context incurs the full costs of a function call on the x86 platform. These include the overhead of adjusting the stack frame and the costs of saving and restoring the host machine registers used as temporaries by the helper routine. Further, a statically-

compiled instruction handler does not easily integrate with the dynamic code generator and cannot benefit from its liveness analysis and register allocation optimizations.

4.2.2 Extending QEMU with Taint Tracking: Approach Overview

Having provided a brief introduction to QEMU and its code translation mechanisms, we can now discuss our approach to designing a comprehensive information flow tracking substrate on top of this emulation technology. Doing so requires addressing several important design questions. Perhaps most crucially, we must consider and decide at what level of abstraction we should track the flow of sensitive information (represented by taint labels) in the guest environment.

One possible approach, and one that has been extensively explored in prior work, is to track the propagation of taint labels at the level of TGC primitives. This can be accomplished by instrumenting each microinstruction handler with additional logic that updates the taint data structures in the corresponding manner. As an illustrative example, consider the microinstruction `add_i32 tmp2,tmp2,tmp14` from the code fragment shown in Figure 4.5. When the backend compiler reaches this instruction, it generates a block of host machine code that adds the value stored in `tmp14` to the contents of `tmp2`. With this approach, the compiler extends the output with an additional sequence of instructions, which merges the taint label of `tmp2` with the label of `tmp14` and taints `tmp2` (the destination operand) with the resulting label. Other microinstructions are handled in an analogous manner.

This approach to taint tracking is straightforward, convenient, and relatively easy to implement, since TGC microinstructions have very simple semantics. However, as we demonstrate below, this technique inevitably imposes a drastic performance penalty that would render the resulting implementation unusable in our target setting, i.e., real-time IFT for interactive user-facing applications. However, despite this inherent performance hit, all previous systems [61, 41, 98] that have attempted to extend QEMU with instruction-level taint tracking employ this technique.

PIFT starts with the same foundational building block — the QEMU emulator, but takes a very different approach and, accordingly, makes different trade-offs. While previous techniques tend to intertwine emulation with taint tracking, our approach views them as two separate and, for the most part, independent computations. In conceptual terms, our system offloads the taint tracking workload to a dedicated *taint processor* — a specialized hardware extension to the x86 architecture similar to a SIMD module for vector algebra or an FPU. We devise a new instruction set architecture (ISA) for expressing taint propagation actions in a concise and efficient form and this new ISA provides the primary means of programming the taint processor.

While our long-term objective is to implement the taint processor in real hardware, this extension takes on a *virtual* form in our current design and we emulate its functionality in software via a set of functional extensions to QEMU. As a result, PIFT provides a

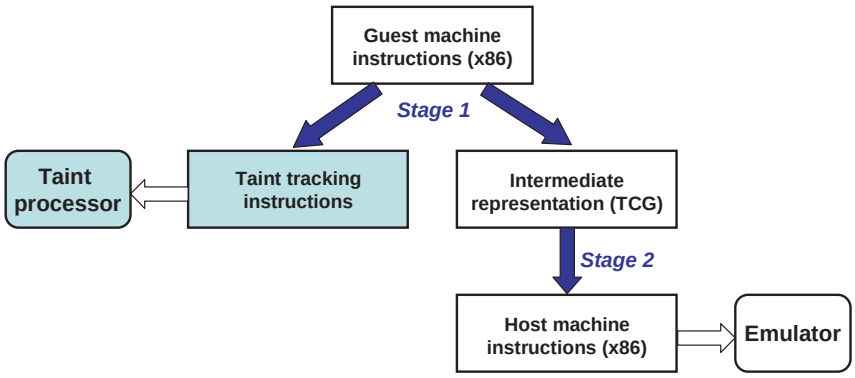


Figure 4.7. Our extensions to QEMU's dynamic code translator.

software-only solution that is fully compatible with existing and widely-deployed hardware platforms, while retaining the potential for incremental migration to hardware-assisted platforms at a later stage.

Our design makes several modifications to QEMU's dynamic code compiler, as illustrated in Figure 4.7. For each input block of guest machine code, the code translator in PIFT generates two distinct sequences of instructions: the emulated version of the original guest code and the corresponding block of *taint tracking instructions* for the taint processor. To produce the latter, we interpose at the first stage of the code translation process, where the initial stream of guest x86 instructions is disassembled and converted into intermediate code blocks. We examine each instruction in the input block, determine its effects on the state of taint labels in the system, and synthesize some number of taint tracking instructions that capture these effects. Crucially, the taint tracking code is generated directly from guest x86 instructions, prior to their decomposition into the TCG notation. As we explain below, preserving the semantics of the original guest instruction set can be highly beneficial to performance, as it enables the taint processor to apply a number of important optimizations. During emulation, QEMU submits these auxiliary blocks of instructions to the taint processor in the order that matches the execution sequence of actual code blocks in the emulated machine. The taint processor executes taint tracking instructions in a sequential manner and updates the state of labels in the system according to their specifications.

The main complication with the basic scheme described above is that in certain scenarios, the information needed to fully specify a taint tracking action may not be available at the time of code translation. In other words, given a guest machine instruction, the code translator must determine its precise effects on the state of taint labels in the emulated system and, in certain cases, this requires knowing something about the system's runtime state. To illustrate this problem, we return to the example involving the `push %ebx` instruction, which pushes one of the guest machine registers onto the stack. In order to account for the flow of information produced by this instruction, we must propagate the taint label from the source register to the memory location that represents the top of the stack. While the taint source operand (register `ebx`) is encoded into the instruction and is thus readily available to

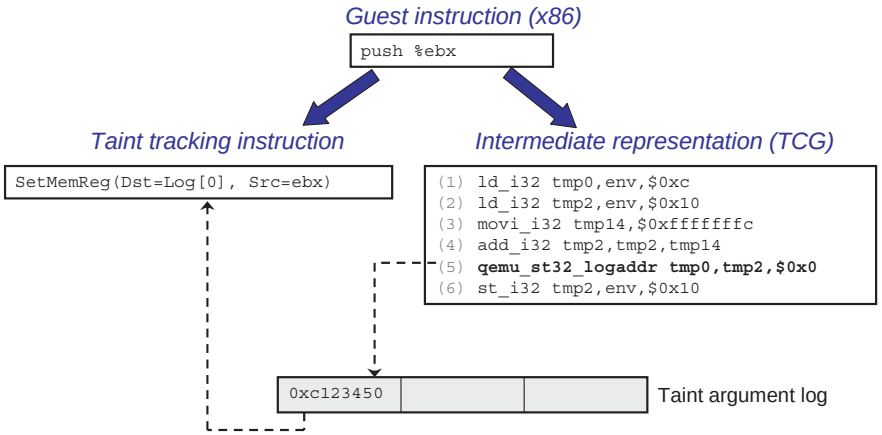


Figure 4.8. An illustrative example of dynamic code translation in PIFT.

the compiler, the destination memory address is, of course, not known at the time of code analysis.

To handle such cases, the compiler instruments the emulated version of the problematic instruction with a small amount of additional logic that resolves these unknown values at runtime and communicates them to the taint processor. In our current implementation, these dynamic temporary values are communicated via a shared circular memory-based log. In operational terms, the emulator and the taint tracker are in a producer-consumer relationship and use the log to coordinate their activities.

Most of these problematic cases involve instructions that manipulate (read or update) the contents of memory, as tracking the resulting flow of information requires knowing the exact physical address of the memory operand. To handle this particularly common scenario, we define new variants of the emulated load and store microinstructions: `qemu_ld_logaddr` and `qemu_st_logaddr` (replacing the original versions `qemu_ld` and `qemu_st`, respectively). In addition to reading or updating a memory location in the emulated machine, these microinstructions also resolve the physical address of the memory operand and record it into the shared log for subsequent consumption by the taint processor.

Figure 4.8 illustrates how our system handles the `push` instruction from one of the previous examples. Upon reaching this instruction, PIFT's dynamic compiler synthesizes the corresponding taint tracking instruction, which has the following form: `Set(Dst=MEM_LONG, Src=ebx, ArgLogPos=0)`. This instruction sets the taint label for the destination memory address to the label associated with the source register operand. Since the destination address cannot be resolved at the time of code translation, the compiler specifies a placeholder that references slot 0 in the taint argument log. To generate this address, the compiler modifies the intermediate notation, replacing the emulated store microinstruction with `qemu_st_logaddr`. At runtime, this microinstruction causes the emulator to compute the physical address of the destination memory operand and write it into the argument log.

TCG microinstruction	Taint tracking action
(1) ld_i32 tmp0,env,$0xc	$L_{tmp0} \leftarrow L_{ebx}$
(2) ld_i32 tmp2,env,$0x10	$L_{tmp2} \leftarrow L_{\emptyset}(= L_{esp})$
(3) movi_i32 tmp14,$0xfffffffc	$L_{tmp14} \leftarrow L_{\emptyset}(= L_{Constant})$
(4) add_i32 tmp2,tmp2,tmp14	$L_{tmp2} \leftarrow L_{tmp2} \oplus L_{tmp14}$
(5) qemu_st32 tmp0,tmp2,$0x0	$L_{mem[tmp2+(0..3)]} \leftarrow L_{tmp0}$
(6) st_i32 tmp2,env,$0x10	*No-op*

Table 4.2. The sequence of taint tracking actions required to handle the `push %ebx` instruction with previous approaches.

To summarize, PIFT proposes a novel design for information flow tracking using QEMU that differs from the previous techniques in two crucial respects:

1. PIFT explicitly decouples information flow tracking from emulation, treating them as two separate and largely independent computations.

2. PIFT tracks the flow of information at a higher level of abstraction that captures the specifics of the guest ISA.

We believe that the above points warrant a more detailed analysis and we now proceed to examining these design choices with the goal of articulating their main advantages and implications.

One of the central contentions of this dissertation is that tracking the flow of information at a level that directly matches the semantics of the guest instruction set is inherently more efficient than tracking at the microinstruction level. While we defer the detailed description of the taint processor and its instruction set to the next subsection, it is quite easy to show informally that our approach can be expected to provide significant performance gains.

First and foremost, mapping the stream of guest machine instructions *directly* onto taint-tracking instructions, without first decomposing them into TCG, allows us to avoid tracking the propagation of taint through the intermediate internal variables defined by the TCG language. Looking again at the decomposition of `push` in Figures 4.5 and 4.8, we note that if we instrument each microinstruction with taint tracking, as previous systems do, we must track the propagation of labels through the intermediate variables `tmp0`, `tmp2`, and `tmp14`. This results in five distinct taint propagation actions, as illustrated in Table 4.2. In contrast, handling this example in PIFT requires only one taint transfer action — one that is specified by the taint tracking instruction `Set(Dst=MEM_LONG, Src=ebx, ArgLogPos=0)`.

Another noteworthy point is that the presence of higher-level semantics enables a range of novel and highly effective optimizations that are difficult or altogether impossible to apply at the microinstruction level. As an illustrative example, consider the widely-used `repz movsd` [19] mnemonic on x86, which provides an efficient way to copy an arbitrarily-sized region of memory between a pair of virtually-contiguous memory buffers. This instruction is commonly used to implement the `memcpy` C library routine and the Linux kernel uses

`repz movsd` to transfer file data between an application-level buffer and the kernel-level page cache when servicing `sys_read` and `sys_write` system calls. To emulate this seemingly simple instruction, QEMU converts it into a looped sequence of microinstructions. Each iteration of the loop decrements a counter and transfers one word of data between the two buffers using one emulated load (`qemu_ld`) and one emulated store (`qemu_st`) microinstruction. Instrumenting these microinstructions with IFT logic (the approach taken by earlier systems) implies that memory taint labels are also updated one word at a time. The costs of traversing the memory taint data structures in each iteration of the loop quickly add up and can lead to a tremendous slowdown — up to 3 orders of magnitude relative to the data transfer itself.

PIFT handles this situation quite differently. In the code translation stage, our compiler examines the `repz movsd` guest mnemonic and emits its taint tracking equivalent: `RepSet(Dst=MEM_LONG, Src=MEM_LONG)`. When this instruction executes, the taint processor carefully examines the buffer size and alignment properties and optimizes the transfer accordingly. In a common scenario, `repz movsd` is invoked with page-aligned memory buffers and the source page(s) carry uniformly-tainted data. In this case, it suffices to transfer page-level taint values, instead of copying the constituent fine-grained taints one word at a time. Assuming 4KB-sized memory pages, 32-bit words, and a 32-bit taint label space, this technique reduces the computational burden of taint tracking by a factor of 1024 in this common case. It is essential to note that this optimization is enabled by the presence of higher-level semantics. In this example, they allow the taint processor to recognize an important special case — a contiguous transfer between page-aligned regions of memory — and such information would be difficult to recover once `repz movsd` is decomposed into a loop of microinstructions.

Finally, by explicitly separating information flow tracking from emulation and treating them as two loosely-coupled computations, PIFT gains additional flexibility in scheduling these tasks. In particular, we can let the taint tracking computation proceed *asynchronously* with respect to the main emulation context. Moving the overhead of taint tracking out of the critical execution path in this manner allows us to improve application response time and interactivity, as we demonstrate in our evaluation. On multicore architectures, performance can be further improved by executing the emulation and taint tracking contexts concurrently on two distinct processor cores. The detailed design of asynchronous parallel taint tracking is provided in Section 4.2.6.

Granted, our approach has costs. First, our implementation is significantly more complex than previous systems. While annotating TCG microinstruction handlers with taint tracking logic is a relatively straightforward task, crafting a specialized machine code translator for the x86 instruction set is a more daunting undertaking. Synthesizing taint tracking instructions requires understanding and correctly handling the rich semantics and intricacies of the guest instruction set. However, notwithstanding the complexity and semantic richness of the x86 ISA, we have found that its taint propagation effects can be efficiently mapped onto a modest number of well-chosen taint tracking instructions, which we present in the next subsection.

The second concern pertains to the portability of our implementation. Our specialized

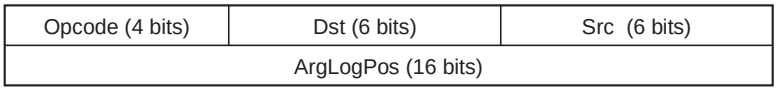


Figure 4.9. The general format of a PIFT taint tracking instruction.

code translator and taint tracking processor extensions were designed for the x86 machine architecture and are not directly usable on other platforms. To perform taint tracking on a different processor architecture (such as ARM or PowerPC), we will have to design a new virtual instruction set and implement a new code translator. However, given the dominant deployment of x86-based hardware, we believe that this loss of generality is a prudent cost to pay for the performance improvements.

4.2.3 The PIFT Instruction Set

PIFT constructs a virtual processor extension and a new instruction set for manipulating taint labels. This instruction set tries to capture the precise semantics of the guest machine ISA and avoids decomposing multi-stage taint propagation actions associated with complex x86 instructions into groups of simpler actions, since the latter are necessarily suboptimal from the performance standpoint. Each taint tracking instruction specifies a certain transformation on the state of taint labels in the emulated system, including the taint status of its CPU registers and physical memory.

General instruction format: A PIFT taint tracking instruction comprises a fixed-length 32-bit static component and a variable number of dynamically-generated instruction arguments. The static portion of an instruction is synthesized by the PIFT compiler during the first stage of code translation. The dynamic component contains a set of instruction-specific arguments, whose values depend on the runtime state of the guest system and thus cannot be determined at compile time. As described in the previous section, these values are resolved at runtime from the main emulation context and communicated to the taint processor through a circular memory-based log.

Figure 4.9 illustrates the high-level format of the static instruction component. In its general form, this component specifies a source and a destination operand, an instruction opcode, and an `ArgLogPos` value, which indicates the position of the first dynamic argument for this instruction in the taint tracking log. This position is specified as an offset relative to the starting point of the argument array for the current translation block. For instructions that require more than one dynamic argument, additional arguments are written to the log at consecutive positions and thus, the taint processor can easily locate them based on the position of the first argument.

Fixed
0x0 NULL

Integer registers					
0x01 eax	0x02 ecx	0x03 edx	0x04 ebx	0x05 esi	0x06 edi

FPU registers							
0x07 st0	0x08 st1	0x09 st2	0x0A st3	0x0B st4	0x0C st5	0x0D st6	0x0E st7

Memory				
0x0F MEM_BYTE	0x10 MEM_WORD	0x11 MEM_LONG	0x12 MEM_QUAD	0x13 MEM_10BYTES

Other		
0x14 TMP1	0x15 TMP2	0x16 NONE

Table 4.3. Taint tracking instruction operands.

Instruction operands: The `Src` and `Dst` fields specify the source and destination operands for a taint transfer action. Table 4.3 lists the legitimate operand values and we describe them in further detail below.

0x00: This value represents a fixed "null" taint label. It can only be specified as a source operand and provides an easy and efficient way to clear the destination's taint label (i.e., replace it with $L_\emptyset$).

0x01-0x0E: Values in this range provide a means of manipulating the taint status of guest machine registers. As in previous approaches [98, 41], PIFT tracks the current taint label for each of the data registers, which include the general-purpose integer registers and the FPU register stack. Our current prototype does not track the propagation of labels through SIMD vector data registers associated with the various vendor-specific extensions to the core x86 architecture, but we plan to implement support for at least one such extension in future work. Also in line with previous efforts, PIFT does not maintain taint labels for system registers that manage the processor's control plane and system resources. These include the stack pointer (`esp`), stack base pointer (`ebp`), instruction pointer (`eip`), segment registers, control registers, descriptor table registers, status flags, MSRs, hardware counters, and others. Maintaining taint labels for these low-level components would incur unnecessary overhead, since it would be unusual for sensitive application-level data to propagate into these registers.

0x0F-0x13: Values in this range are used to specify memory operands. Memory access instructions on the x86 platform come in different forms and support several different units of transfer, including *bytes*, *words* (16 bits), *long words* (32 bits), and *quad-words* (64 bits). PIFT supports all of these options by providing a matching set of operands: `MEM_BYTE`, `MEM_WORD`, `MEM_LONG`, and `MEM_QUAD`. The last memory operand type (`MEM_10BYTES`) was added to handle an important special case — moving numbers in the 80-bit extended precision format between memory and the FPU

Opcode	Mnemonic	Accepts destination operand	Accepts source operand
0x0	`Set`	yes	yes
0x1	`Merge`	yes	yes
0x2	`CondSet`	yes	yes
0x3	`CondMerge`	yes	yes
0x4	`RepSet`	yes	yes
0x5	`RepMerge`	yes	yes
0x6	`FPUPush`	no	yes
0x7	`FPUPop`	yes	no
0x8	`ExtendedOp`	n/a	n/a

Table 4.4. Taint tracking instruction opcodes.

register bank using instructions such as `fld` and `fstp`. Note that in most cases, the actual physical address of a memory operand cannot be resolved at compile time and must be supplied to the taint processor in the form of a dynamic argument.

0x14-0x15: We define two internal temporary variables (`TMP1` and `TMP2`) for storing intermediate results during multi-stage taint label computations. These variables assist us in a small number of special cases that involve a complex guest instruction and a specific pattern of taint propagation. More concretely, these scenarios require merging taint labels from multiple sources and propagating the resultant label to multiple destinations. Saving the result of the merging step in a temporary variable allows us to avoid recomputing it for each destination.

0x16: This value indicates that the operand field is unused.

Instruction opcodes: PIFT defines 9 distinct instruction opcodes which, in combination with the abovementioned set of operands, allow us to express a wide range of taint manipulation actions. We summarize these opcodes in Table 4.4 and describe them more fully in the following paragraphs.

[0x0] `Set`: This opcode sets the destination operand's taint label to the label of the source operand: $L_{Dst} \leftarrow L_{Src}$.

[0x1] `Merge`: This opcode merges the destination operand's taint label with the label of the source operand: $L_{Dst} \leftarrow L_{Dst} \oplus L_{Src}$.

[0x2] `CondSet`: This opcode implements the *conditional set* action, which sets the destination operand's taint label to the label of the source operand if the value of a condition flag is non-zero. This opcode was added to provide support for conditional transfer instructions, such as `cmov`, defined by the x86 architecture. The emulator evaluates the conditional parameter at runtime and communicates it to the taint processor in the form of a dynamic argument.

[0x3] `CondMerge`: This opcode implements the *conditional merge* action predicated on the value of a dynamic argument.

[0x4] `RepSet`: This opcode implements the *repeat set* action, which propagates the taint label from the source operand to multiple contiguous instances of the destination operand. This opcode was added to provide support for repeated transfer instructions defined by the x86 architecture. One particularly important example is the `repz movs` instruction, which allows applications to set up an arbitrarily long data transfer between a pair of contiguous memory buffers. In our experience, such repeated transfer actions have proved to be among the most difficult to handle and the main complication arises from the fact that the *number of* dynamic arguments for these instructions cannot be determined (or reliably estimated) at compile time. To see why, we must consider the fact that while `repz movs` defines a virtually contiguous memory copy operation, the resulting data transfer may not be contiguous at the level of *physical* memory pages. Hence, in order to describe the effects of this instruction to the taint processor, it is not enough to simply specify the starting memory addresses and the length of the transfer; we must examine the side-effects at the level of physical pages and record the resulting physical addresses into the taint argument log. The problem is that the number of such addresses (and hence the number of log slots we must reserve prior to emulating the instruction) cannot be determined at the time of code translation — it depends on the length of the transfer which is, in turn, specified by the runtime value of the `ecx` register.

PIFT handles this nontrivial scenario by allocating a set of *secondary argument logs*, which can grow and shrink dynamically. When handling a repeated transfer guest instruction, the emulated version of the instruction reserves some number of such logs and fills them with the necessary address information. Note that since memory addresses *within* a page are physically contiguous, it suffices to record one physical address for each page in the source and destination buffers. These arrays of page-level addresses, together with the starting page offset in both buffers and the transfer length, provide sufficient information to fully describe the effects of the instruction for taint tracking purposes. The addresses of these secondary logs are then communicated to the taint processor via the main argument log.

[0x5] `RepMerge`: This opcode implements the *repeat merge* action. It is semantically and functionally equivalent to the `RepSet` instruction described above, except that the labels in the destination buffer are merged with (rather than replaced by) the source labels.

[0x6] `FPUPush`: This opcode pushes the source operand's taint label onto the label stack representing the FPU register bank:
$L_{st7} \leftarrow L_{st6}$
...
$L_{st1} \leftarrow L_{st0}$
$L_{st0} \leftarrow L_{Src}$.

[0x7] `FPUPop`: This opcode pops the topmost value from the label stack representing the

FPU register bank and taints the destination operand with this value:
$L_{Dst} \leftarrow L_{st0}$
$L_{st0} \leftarrow L_{st1}$
...
$L_{st6} \leftarrow L_{st7}$
$L_{st7} \leftarrow L_{\emptyset}$.

[0x8] `ExtendedOp`: While the set of opcodes and operands described above offers a powerful language for expressing taint propagation actions, some elements of the x86 instruction set do not naturally fit into this framework. These non-conforming instructions have relatively complex taint tracking side-effects that are not easily expressible via the source-destination operand notation. In some cases, the instruction's exact semantics (and hence its information flow effects) are influenced by a set of conditionals on the state of the emulated machine, which must be evaluated at runtime and communicated to the taint processor.

We consider such problematic instructions on a case-by-case basis and define a set of custom taint tracking routines to handle them. In order to invoke one of such custom handlers at runtime, the compiler synthesizes an `ExtendedOp` instruction and specifies one of the *extended opcode* values in the 12 least-significant bits, replacing the `Dst` and `Src` fields. Next, we enumerate these extended opcodes and briefly comment on their usage:

[0x0] `ExtFSave`

Dynamic arguments:

`DstMemStartAddr`	`DstMemEndAddr`	`RegisterAddrMask`

This extended opcode handles the `fsave` guest instruction, which saves the entire state of the guest FPU (marshalled into a 108-byte data structure) to the memory location specified by the destination operand. For the purposes of information flow tracking, we must propagate taint values from the individual components of the emulated FPU to the corresponding regions of memory and hence, the emulator must provide the destination memory address as a dynamic argument to the taint processor. If the destination area spans two virtually consecutive memory pages, we record both the starting and the ending physical address in order to obtain the mappings for both pages. Finally, if any of the general-purpose registers were used to compute the destination memory address, we must capture the resulting indirect flow by tainting the destination memory area with the labels of the corresponding register(s). We specify the set of registers that participated in the computation of the destination address via a bitmask in the third dynamic argument.

[0x1] `ExtFRstor`

Dynamic arguments:

`SrcMemStartAddr`	`SrcMemEndAddr`	`RegisterAddrMask`

This extended opcode handles taint propagation for the `frstor` guest instruction, which restores the state of the FPU previously saved to memory using the `fsave` instruction. The taint processor transfers the taint labels from the source memory area to the corresponding elements of the FPU state. The source memory address and the set of registers used to compute it are communicated to the taint processor as dynamic arguments.

[0x2] `ExtFXSave`

Dynamic arguments:

`BooleanFlags` `[bit 0] SaveXMMState`	`DstMemStartAddr`	`DstMemEndAddr`
`RegisterAddrMask`		

This extended opcode handles taint propagation for the `fxsave` guest instruction, which writes the current state of the FPU, MMX, SSE, and MXCSR processing elements to a 512-byte memory area specified by the destination operand. Since our current implementation does not track the taint status of the MMX, SSE, and MXCSR components, we only propagate taint labels from the emulated FPU and clear the destination taint status for these other components.

[0x3] `ExtFXRstor`

Dynamic arguments:

`SrcMemStartAddr`	`SrcMemEndAddr`	`RegisterAddrMask`

This extended opcode handles taint propagation for the `fxrstor` guest instruction, which restores the contents of the FPU, MMX, SSE, and MXCSR processing elements from the memory area specified by the source operand.

[0x4] `ExtFStenv`

Dynamic arguments:

`DstMemStartAddr`	`DstMemEndAddr`	`RegisterAddrMask`

This extended opcode handles taint propagation for the `fnstenv` guest instruction, which saves the current FPU operating environment at the memory location specified by the destination operand. Since we do not track the taint status of FPU control data structures, we clear the taint labels for the destination memory area.

[0x5] `ExtFPURotateUp`

Dynamic arguments: none

This extended opcode handles taint propagation for the `fincstp` guest instruction, which "rotates the barrel" of the FPU register bank, moving `st1` into `st0`, `st2` into `st1`, and so forth.

[0x6] `ExtFPURotateDown`

Dynamic arguments: none

This extended opcode handles taint propagation for the `fdecstp` guest instruction, which "rotates the barrel" of the FPU register bank in the opposite direction.

[0x7] `ExtInterrupt`

Dynamic arguments:

`BooleanFlags`
`[bit 0] HasErrorCode`
`[bit 1] IsTaskGate`
`[bit 2] PerformStackSwitch`
`[bit 3] VM86ModeEnabled`
	`DstMemStartAddr`	`DstMemEndAddr`

This extended opcode handles taint propagation for the software interrupt (`int`) instruction — the standard mechanism for implementing system calls on the Linux platform. In typical cases, this instruction redirects the processor to a pre-defined interrupt service routine, switches the stack pointer to an alternate kernel-level stack area, and pushes a 20-byte control data structure onto this new stack. This data structure records the old values of `esp`, `ss`, `eip`, `cs`, and `eflags`. Our system does not maintain taint labels for any of these control registers and thus, the taint processor handles this instruction by clearing the taint status in the destination memory area.

[0x8] `ExtLCall`

Dynamic arguments:

`BooleanFlags`
`[bit 0] IsTaskGate`
`[bit 1] PerformStackSwitch`
	`DstMemStartAddr`	`DstMemEndAddr`

`SrcMemStartAddr`	`SrcMemEndAddr`	`NumParameters`

This extended opcode handles taint propagation for the `lcall` guest instruction, which performs a far (inter-segment or inter-privilege-level) procedure call. In typical cases, the processor switches to the stack for the privilege level of the called procedure and pushes the caller's `esp`, `ss`, `eip`, and `cs` values onto the new stack. Our system does not maintain taint labels for any of these control registers and thus, the taint processor clears the taint status in the destination stack area. This instruction may additionally copy an optional set of 32-bit function parameters from the calling procedure's stack to the new stack and we transfer their labels between the two stack regions accordingly.

4.2.4 Taint Tracking Code Generation

The focal point of our taint tracking extensions to the QEMU compiler is the `disas_insn` function in `target-i386/translate.c`, which decomposes and translates a single instruction from the guest code stream. This function implements a state machine, analyzing the input instruction one byte at a time, decomposing it into TCG notation, and (with our extensions) generating the corresponding taint tracking code. Although a comprehensive review of the x86 instruction set is beyond the scope of this document, as is an exhaustive description of PIFT's code translation rules, we can elucidate the general logic of taint tracking code generation by providing several representative examples.

Example 1: `movb %eax, (%edx)`
This x86 mnemonic dereferences the byte pointer stored in `edx` and loads the resulting 8-bit value into `eax`. In our model of information flow tracking, this instruction performs two information transfers: a direct transfer from the source memory location to the destination register `eax` and an indirect transfer from `edx` to `eax`. The latter arises from the fact that the value of `edx` is used to compute the address of the source memory operand. To capture these effects, our compiler generates the following pair of taint tracking instructions:

```
Set(Dst=eax, Src=MEM_BYTE, ArgLogPos=0)
Merge(Dst=eax, Src=edx)
```

Example 2: `fstp st3`
This mnemonic modifies the state of the FPU register stack by copying the contents of `st0` to `st3` and then popping the stack, discarding the topmost value. Its information flow effects can be captured via the following sequence of taint tracking instructions:

```
Set(Dst=st3, Src=st0)
FPUPop(Dst=NONE)
```

Example 3: `idiv (%ebx)`
This mnemonic implements the signed integer division operation. It divides the 64-bit integer in `edx`:`eax` (constructed by viewing `edx` as the most significant four bytes and `eax` as the least significant portion) by the 32-bit value stored at a memory location referenced by `ebx`. The quotient and the remainder results of the division are placed in `eax` and `edx`, respectively. We capture the resulting information flow by conservatively tainting the entire result with the labels of all source operands (`eax`, `edx`, and the memory location). We additionally taint the destination with `ebx` in order to capture the indirect flow. Our compiler handles this scenario by generating the following sequence of taint tracking instructions:

```
Merge(Dst=eax, Src=edx)
Merge(Dst=eax, Src=MEM_LONG, ArgLogPos=0)
Merge(Dst=eax, Src=ebx)
Set(Dst=edx, Src=eax)
```

4.2.5 Taint Processor Internals

The taint processor is the central architectural module of our information flow tracking substrate. This module is tasked with consuming and executing blocks of taint tracking instructions produced by the code translator. In Section 4.2.2, we alluded to the possibility of implementing the taint processor as a real hardware extension, but since our current design constraints call for compatibility with legacy hardware platforms, our implementation *emulates* the taint processor's functionality in software. In the simple and purely synchronous mode of information flow tracking, the emulated taint processor and the guest emulator itself execute in the same QEMU thread and operate in lockstep: QEMU executes a translated block of guest code and then switches to the emulated taint processor, which consumes the corresponding block of taint tracking instructions and updates the state of taint labels in the guest environment.

We now present the core data structures used by the taint processor to manage the taint status of guest machine registers and memory.

Maintaining register taint status: As we mentioned in the preceding section, our current implementation maintains taint labels for all six of the general-purpose integer registers (`eax`, `ebx`, `ecx`, `edx`, `esi`, and `edi`), as well as the FPU data register stack (`st0` through `st7`). We manage the taint status of these processor registers using a very straightforward array-based scheme. For general-purpose integer registers, we maintain a simple linear array of 32-bit label values, one label per register. For the FPU data register stack, we maintain an array of 8 labels representing the individual stack slots, along with the position of the topmost stack element in this array. Organizing the stack in this manner allows us to handle push and pop operations very efficiently — simply by updating the topmost element pointer, without having to shift the actual label values between array slots. However, accessing an arbitrary stack element is slightly more expensive in this scheme, requiring two memory accesses instead of one.

Maintaining memory taint status: Tracking the byte-level taint status of guest memory requires a somewhat more sophisticated scheme. A naïve implementation that maintains a label for every byte of physical memory would incur a prohibitive storage overhead and is clearly impractical. Our memory taint management module seeks to achieve computational efficiency, while being parsimonious in its use of memory resources. We devise a set of data structures that exploit spatial locality and allow us to balance the memory overhead against the latency of label lookups.

Our current prototype supports environments with up to 4GB of physical memory and is optimized for the standard page size of 4KB. In this configuration, a physical memory address occupies 32 bits (which matches the size of a single slot in the taint argument log) and can be decomposed into a 20-bit *physical page number* and a 12-bit *page offset*. For each page of physical memory addressable by the guest system, PIFT maintains a data structure called a *page taint descriptor* (PTD), which encapsulates a fine-grained byte-level view of taint labels within the respective page. This view is represented using one of three

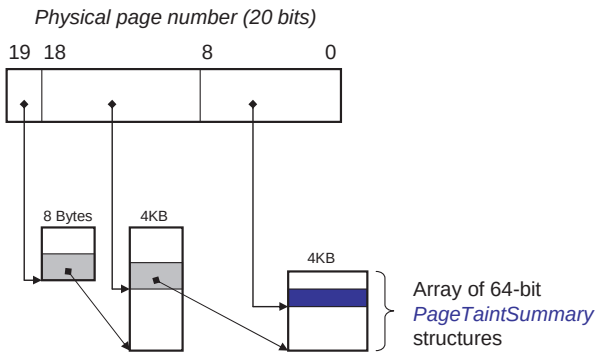


Figure 4.10. The `PageTaintSummary` lookup procedure.

different formats, which allow us to trade off lookup latency and storage overhead. These formats include:

Uniform: Used in situations, where all byte offsets within a page are tainted with the same value. The PTD carries a single 32-bit page-level taint label.

Run-Length Encoding: The PTD carries a sequence of $\langle length, label \rangle$ tuples, as in the standard RLE compression scheme. This format offers a space-efficient way to represent pages carrying more than one label, but lookups require a linear scan.

Taintmap: The PTD carries a flat linear array of byte-level taint labels within the page. This representation provides constant-time lookups on highly-fragmented pages, but incurs significant storage costs (16KB per page of guest memory).

We use a three-level tree data structure analogous to a page table to resolve 20-bit physical page numbers into the corresponding page taint descriptors and Figure 4.10 illustrates the resolution process schematically. A non-leaf (index) node stores an array of 32-bit pointers to child nodes. A leaf node maintains an array of `PageTaintSummary` structures, which concisely summarize the taint status of each physical memory page. Each `PageTaintSummary` instance comprises two 32-bit words and holds the following fields:

`ptdFormat` (bits 0-1):
Stores the format of the PTD associated with the respective memory page. One of {*Uniform*, *RLE*, *Taintmap*}.

`ptdValue` (bits 32-63):
For *Uniform* format: stores the actual page-level taint label.
For *RLE* and *Taintmap* formats: stores the address of the PTD in the virtual address space of the emulator.

Bits 2-31 are currently unused and are reserved for future extensions.

The computational and memory bandwidth costs incurred by a full PTD lookup are nontrivial: we have to walk the tree data structure and perform at least three memory accesses. In order to reduce the recurring costs of tree traversals, our implementation also maintains a TLB-like cache of mappings between the physical page numbers and the corresponding `PageTaintSummary` structures. This cache is organized as a hash table (indexed by the physical page number) and implements a random replacement policy. We have also experimented with LRU replacement, but found that its performance gains were overshadowed by the costs of age tracking.

Executing taint tracking instructions: Having defined the taint label storage primitives for guest machine registers and memory, the next step is to implement the taint tracking instruction handlers that manipulate and update the state of labels. Most of these handlers can be implemented in a fairly straightforward manner, but the key concern is computational efficiency. The overall performance of a guest system running on top of PIFT is influenced to a large extent by the overhead of information flow tracking and instruction handlers can be viewed as the "inner loop" of the IFT computation. Thus, any conceivable optimization that reduces the number of clock cycles spent in this inner loop can have a significant payoff and can be worth exploring.

Our overall philosophy in designing the virtual taint processor and implementing the instruction handlers was to perform as much preprocessing as possible. Some fragments of the computation can be moved out of the inner loop and performed at the compilation stage — either the dynamic compilation of emulation code or compilation of the QEMU executable itself. As a specific example of this strategy, we pre-generate a separate handler function for every valid combination of $\langle$`Opcode`, `Dst`, `Src`$\rangle$ and compile them statically into the QEMU executable. The taint processor maintains an array of pointers to these handlers (`handler_array`), indexed by the numeric value $((\texttt{Opcode} << 12) + (\texttt{Dst} << 6) + \texttt{Src})$. As shown in Figure 4.9, this value always matches the 16 most-significant bits of the instruction's binary value. Hence, given a taint tracking instruction with value `b`, the taint processor can locate the handler function for this instruction simply by evaluating `handler_array[b » 16]`. Although pre-generating handler functions in this manner significantly increases the memory footprint of the QEMU code segment (from 5.2MB in the unmodified implementation to 27.1MB in PIFT), this optimization allows us to avoid spending precious CPU cycles on decoding the operand fields and performing pointer arithmetic to locate these operands.

Next, we illustrate the internals of several commonly-used taint instruction handlers, returning to the instruction sequence from Example 1 in the previous subsection. Figure 4.11 shows the pre-constructed handler for `Set(Dst=eax, Src=MEM_BYTE)` in the source code form and in the final form compiled for the host machine platform. Note that the taint destination address (0x9442184), which represents the location of the `eax` label in the register taint array, has been hard-coded into the instruction stream and does not need to be computed in the inner loop. Figure 4.12 shows the implementation of the pre-constructed handler for `Merge(Dst=eax, Src=edx)`.

```
typedef uint32_t taint_label;

/* Register taint labels */
extern taint_label *reg_labels;

/* Pointer to the argument buffer for the current
   code block */
extern uint32_t *taintarg_blk;

/* ArgLogPos for the current instruction */
extern uint32_t arglogpos;

void taintop_handler_SET_EAX_MEMBYTE() {
  uint32_t src_addr = *(taintarg_blk + arglogpos);
  taint_label *src_taint_p =
    get_mem_label_byte(src_addr);
  reg_labels[0] = *src_taint_p;
}
```

```
push %ebp
mov  %esp,%ebp
sub  $0x8,%esp
mov  0x93e8128,%eax
shl  $0x2,%eax
add  0x9442124,%eax
mov  (%eax),%eax
mov  %eax,(%esp)
call
  <get_mem_label_byte>
mov  (%eax),%eax
mov  %eax,0x9442184
leave
ret
```

Figure 4.11. The implementation of the pre-generated handler for `Set(Dst=eax, Src=MEM_BYTE)` in the source code form (left) and in the final form compiled for the x86 platform (right).

```
void taintop_handler_MERGE_EAX_EDX() {
  taint_label src_taint = reg_labels[3];
  taint_label *dst_taint_p = &reg_labels[0];

  if (!IS_NULL_LABEL(src_taint) &&
     !LABELS_EQUAL(src_taint, *dst_taint_p)) {
    if (IS_NULL_LABEL(*dst_taint_p)) {
      *dst_taint_p = src_taint;
    } else {
      *dst_taint_p =
        merge_labels(*dst_taint_p, src_taint);
    }
  }
}
```

```
      push %ebp
      mov  %esp,%ebp
      sub  $0x18,%esp
      mov  0x94421c4,%edx
      test %edx,%edx
      je   <.L1>
      mov  0x9442184,%ecx
      cmp  %ecx,%edx
      je   <.L1>
      test %ecx,%ecx
      jne  <.L2>
      mov  %edx,0x9442184
.L1:  leave
      ret
.L2:  lea -0x4(%ebp),%eax
      mov  %eax,(%esp)
      mov  %edx,0x8(%esp)
      mov  %ecx,0x4(%esp)
      call <merge_labels>
      mov  -0x4(%ebp),%eax
      mov  %eax,0x9442184
      sub  $0x4,%esp
      leave
      ret
```

Figure 4.12. The implementation of the pre-generated handler for `Merge(Dst=eax, Src=edx)` in the source code form (left) and in the final form compiled for the x86 platform (right).

Merging taint labels: Label merging, as defined in Section 3.2, is one of the foundational operations in the decentralized label model. This operation outputs a new data label that aggregates the confidentiality policies defined by the input labels. The resulting label defines the least restrictive policy that also enforces all the restrictions associated with the input labels.

Recall that while decentralized data labels are at the foundation of our security model, our augmented emulator tracks information flow on the basis of 32-bit taint labels, which serve as concise fixed-length surrogates for the full decentralized data labels. Our architecture treats these 32-bit taint values as opaque bitstrings and relies on external infrastructure to translate them into decentralized data labels and the associated policies. Since PIFT does not prescribe a specific format for decentralized labels and is not concerned with the details of their representation and storage, we also delegate the label merging function to an external user-defined module. When the taint processor needs to merge a pair of taint values, it invokes the user-supplied `merge_labels` routine, as shown in Figure 4.12. This function is declared as follows:

```
taint_label merge_labels(taint_label a, taint_label b);
```

This routine is expected to resolve the supplied taint values (`a` and `b`) into the corresponding data labels (L_a and L_b), merge them (i.e., compute $L_a \oplus L_b$), assign a new 32-bit taint value to the resulting label, and return this value to the taint processor. The implementation of `merge_labels` depends on the specifics of the mapping between taint values and the corresponding data labels and is beyond the scope of our architecture. Our taint tracking substrate imposes no restrictions on the implementation of this user-supplied routine, but assumes this operation to be idempotent and commutative. (It is worth noting that any function that implements the merge operator according to its standard definition, as given in Section 3.2, possesses these properties). PIFT reserves the numeric taint value 0 to represent the null label ($L_\emptyset$) and further assumes that merging with $L_\emptyset$ has no effect, that is:

$$\forall a : L_a \oplus L_\emptyset = L_\emptyset \oplus L_a = L_a.$$

4.2.6 Asynchronous Parallel Taint Tracking

We now turn to a discussion of parallelized taint tracking — an important optimization that substantially reduces the runtime performance penalty for certain types of workloads. The key insight that enables this feature is that emulation and information flow tracking can be viewed as two separate and, for the most part, independent computations. As described above, the PIFT compiler generates two isolated streams of instructions: the emulated version of the original guest instruction stream and the corresponding block of taint tracking code. The latter is handled by the taint processor; the emulator only needs to log the correct execution sequence of the taint tracking blocks and supply the values of dynamic arguments that could not be resolved at compile time.

While the most straightforward implementation would handle both tasks (emulation and taint processing) in a single thread and execute them in lockstep, it is easy to extend

this scheme in a manner that allows the taint tracking instruction stream to be processed asynchronously and in parallel with emulation. This can be accomplished quite easily in our design by moving the emulated taint processor into a separate thread and assigning this thread to another CPU core on the host machine.

In this configuration, we subdivide the taint argument log into a number of smaller *log regions*, which serve as basic units of synchronization between the producer and the consumer. We assign one of the regions to the guest emulation thread, which produces dynamic information for subsequent consumption by the taint processor. As the emulator proceeds with the execution of guest instructions, it writes pointers to the corresponding taint tracking code blocks, along with the dynamically-generated arguments, into its current log region. When the emulator exhausts all available space in its current region, it signals the taint processor thread, submits its current log region for consumption, and then grabs the next available region. The taint processor consumes these log regions as they arrive; it updates the taint-related data structures in accordance with the instructions specified in the log and their arguments, but does not concern itself with the actual state of the emulated machine. In operational terms, the emulator and the taint processor are in a standard producer-consumer relationship with a bounded buffer and they synchronize their activities using a mutex and a pair of condition variables (`cond_empty_region_available` and `cond_full_region_available`).

While the ability to offload the IFT computation to another processor core is clearly advantageous, the size of the taint argument log is a crucial parameter that largely determines the performance gains. Byte-level taint tracking is significantly more expensive than pure emulation and thus, the computational bottleneck is usually at the consumer side. If the emulator (producer) runs out of free log regions, it must block and wait for the taint tracker to make progress and release a region. When this happens, we effectively return to the synchronous mode of taint tracking, where the producer and the consumer operate in lockstep.

We expect, however, that this scenario will rarely arise with interactive I/O-driven applications (and the results of our evaluation support this intuition). To see why, consider the fact that most forms of interactive computing involve human users, who make decisions, act, and submit commands at human timescales. Hence, a typical computational workload on a user-facing machine can be characterized by the prevalence of relatively short bursts of computation (triggered by device interrupts) with large gaps between them (e.g., the user pausing before entering more text). At the end of each burst, modern operating systems usually relinquish the processor with a `hlt` instruction or the corresponding hypercall and wait for the arrival of the next interrupt. During these time intervals, the producer is inactive and does not generate any new work for the consumer. These gaps in computation can be gainfully exploited by the taint processor to advance its position and drain the log. As a result, it becomes less likely that the emulator will need to suspend itself and wait for additional log space while processing a burst of computation.

Of course, this reasoning does not apply equally well to CPU-bound server workloads or interactive applications that routinely launch computationally-intensive tasks. For these types of workloads, asynchrony can be viewed as providing a finite-length buffer that can

absorb a certain amount of taint tracking work, minimizing its impact on the overall performance of the guest system. The size of the taint argument log determines the amount of computation that can be absorbed in this manner.

Finally, we note that in certain scenarios, PIFT must explicitly synchronize the state of taint labels by suspending the emulator thread and waiting for the taint processor to consume the remaining items in its log. Typically, this situation arises when the protected machine makes a request to externalize data (e.g., by sending a network packet or writing to a block storage device) and PIFT intercepts this request to perform security checks. In this situation, the device driver backend makes an upcall to the QEMU process and requests the up-to-date memory taint labels associated with the outbound data buffers. In order to obtain the correct labels, we must wait for the taint processor thread to drain the log and synchronize its state before responding to the driver.

4.2.7 Integration with the Overall PIFT Architecture

Our discussion so far has focused on extending the QEMU emulation platform with fine-grained information flow tracking capabilities. While the taint tracking substrate is the chief component of our architecture and perhaps represents our most significant research contribution, there remains one more crucial step — integrating this substrate with the rest of the PIFT platform. Our extended version of QEMU can be configured to function in isolation, so as to provide a self-contained system emulation environment with taint tracking. However, in order to realize the full benefits of PIFT, including its taint-aware storage and on-demand emulation capabilities, we must establish proper interactions between the emulator and the other central components of the system, such as the hypervisor and the taint-aware filesystem. This necessitates several additional changes and extensions to QEMU, which we briefly summarize in this section.

Bootstrapping the emulator: Upon startup, the default implementation of QEMU bootstraps the guest machine and immediately proceeds to the main emulation loop. However, in our system, the extended implementation of QEMU initiates emulation only upon explicit request from the hypervisor. During startup, the emulator allocates a page-length memory buffer for communication with the hypervisor and creates a new virtual IRQ. The hypervisor signals this VIRQ to request emulation and uses the shared buffer to communicate the processor state snapshot. In the main processing loop, QEMU waits for the hypervisor's signal and, when it arrives, initiates emulation based on the state provided in the snapshot.

Emulating memory accesses: As a full-system emulator, QEMU is responsible for providing the abstraction of a contiguous physical memory address space and managing the associated memory resources. In the standard implementation, QEMU acquires pages for its emulated physical memory by allocating them dynamically from its local heap area and, clearly, this approach is inapplicable in our context. For the purposes of on-demand emulation, memory accesses performed in the emulated mode must operate directly on the regions of physical memory that have been allocated by Xen and granted to the protected

VM. We attain the desired behavior by modifying the address translation logic in QEMU and taking advantage of Xen's *foreign page mapping* feature. When a TLB miss occurs in the emulated environment, QEMU walks the current guest page table for the protected VM to obtain the mapping between the guest virtual page address ($vaddr_g$) and the corresponding physical address ($paddr$). Then, instead of allocating a page from its local heap, the emulator makes a hypercall to Xen and maps the guest page into its own virtual address space at some pre-determined location ($vaddr_e$). The mapping $\langle vaddr_g \rightarrow vaddr_e \rangle$ is then recorded into the software TLB. During code translation, all memory access instructions are converted into reads and writes within QEMU's virtual address space but in fact, these instructions operate directly on the foreign memory pages belonging to the guest VM. The mappings $\langle vaddr_g \rightarrow paddr \rangle$ are needed by the taint tracking infrastructure and we maintain them in a separate data structure.

Relaying Xen event notifications: The paravirtualized model supported by Xen differs from the "bare metal" emulated environment provided by the standard implementation of QEMU in several important respects, most notably in how it handles asynchronous notifications from devices. In the bare metal configuration, these notifications come in the form of interrupts. Xen, on the other hand, shields paravirtualized guests from physical device interrupts and replaces them with *asynchronous event notifications* — a form of abstract device-independent interrupts. The augmented emulator must account for these differences and properly relay the stream of asynchronous events from Xen to the protected VM during periods of emulated execution. To this end, we have modified QEMU to periodically check whether the guest has any pending event notifications (the hypervisor signals this condition via a flag in the shared memory page). When a notification arrives, QEMU interrupts the emulation loop and relays the event to the guest kernel by setting up a bounce frame, effectively emulating the `create_bounce_frame` functionality in `xen/arch/x86/x86_32/entry.S`. The emulator switches to the kernel-level stack (given by `kernel_ss:kernel_sp`) and redirects execution to the event callback handler (given by `callback_cs:callback_eip`). The guest kernel specifies all these parameters to the hypervisor during initialization and Xen, in turn, relays them to the emulator via the shared memory page.

Communicating with kernel-level components: Our extended QEMU-based emulator operates in the control VM as a standard user-level process. In certain scenarios, the emulator must communicate and exchange information with the components of the PIFT architecture that operate in kernel-space of the control VM. These include the device driver backends for paravirtualized I/O devices exposed to the protected VM and the taint-aware filesystem. These components must occasionally contact the augmented emulator and ask it to update the state of memory taint labels or, conversely, provide the up-to-date taint status for specific memory pages. The filesystem can request an update to memory taint data structures when handling a file `READ` request, which transfers tainted user data from disk to a memory buffer. Conversely, handling a `WRITE` system call requires fetching the memory taint status from QEMU and propagating it to filesystem-specific data structures on disk. Device drivers need to communicate with QEMU in order to access the memory taint data structures when performing security checks. To facilitate efficient bidirectional transfer of memory taint information, we implement a client-server model of communication, using

Netlink [79] as the foundational transport mechanism. Netlink is a standard component of the Linux kernel and provides a socket-like mechanism for efficient IPC between kernel- and user-space contexts.

Returning from emulated execution: Another significant challenge is determining in what situations it is safe and beneficial to exit from emulated execution within QEMU and resume native execution within a Xen VM. In general terms, it is safe to terminate emulation as soon as we detect that no tainted data is contained in the registers of the emulated machine (i.e., each emulated register carries $L_\emptyset$). However, this might not be prudent, since the very next instruction might again access sensitive data and immediately trigger a transition back to the emulated mode. Frequent context switches between emulated and native execution can incur a significant overhead and, in degenerate cases, lead to thrashing. PIFT tackles this issue by introducing some delay before returning from emulation. More specifically, QEMU counts the number of consecutive guest instructions that did not access tainted data in memory. If this counter reaches a certain threshold and if all CPU registers are free of taint, QEMU terminates the emulation loop and instructs the hypervisor to resume virtualized execution. In our current prototype, this threshold is set to 50 instructions, as in previous work [98, 41]. This simple strategy is admittedly suboptimal and does not preclude the possibility of thrashing. We believe that investigating more robust and well-tuned heuristics for transitioning to native execution can be a meaningful direction for future work.

Transient transfers to native execution: Finally, certain scenarios make it *necessary* to jump out of emulation and temporarily enable native execution. Typically, these scenarios involve handling the execution of hypercalls, software interrupts, faults, and other types of synchronous exceptions. In the paravirtualized model, these exceptions must go through Xen and always trigger a transition to hypervisor-level code operating at the highest privilege level. The current architecture of Xen makes it very difficult to emulate the execution of hypervisor-level code in a user-space process. Instead, QEMU temporarily suspends its emulation loop, transfers the guest CPU context back to the hypervisor, and instructs it to perform a *transient switch* to native mode at the precise instruction that triggers the exception. For instance, if the guest system issues a hypercall via the `int 0x82` mnemonic, Xen positions the guest `eip` at this instruction and resumes the suspended native VM. The software interrupt immediately triggers a transition to ring 0, causing the hypervisor to regain control and invoke the hypercall handler routine. Transient native execution terminates upon the return from hypervisor-level code.

4.3 A Taint-Aware Storage Layer

A comprehensive information flow tracking platform that also aims to be practical must have the ability to track the flow of tainted data to and from persistent storage devices, such as magnetic disk drives. While it would be easy to augment the abstraction of a virtual disk in a manner that would allow us to taint individual disk blocks, we believe that a fully-

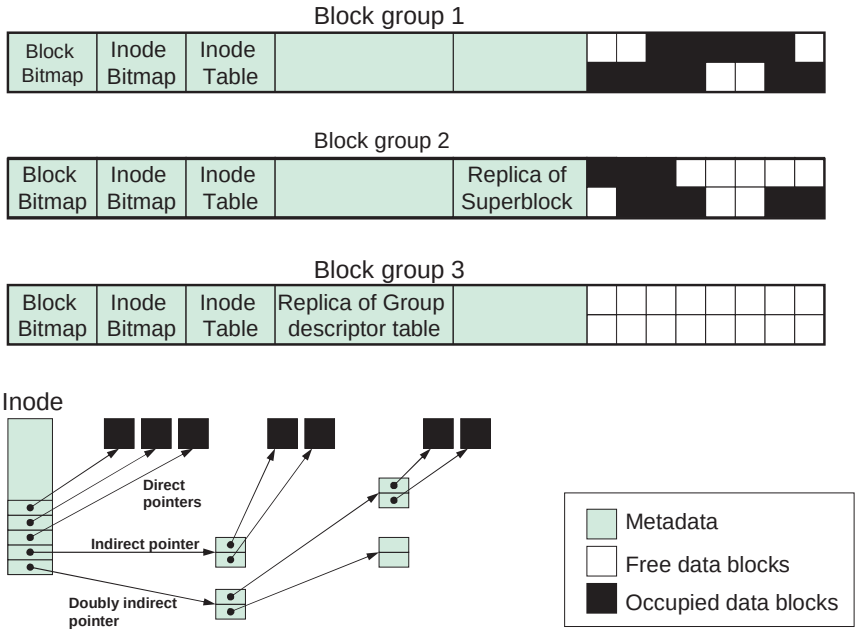


Figure 4.13. The on-disk layout of an ext2/ext3 filesystem.

featured taint-aware filesystem is a more usable alternative (and we outline our reasoning in Section 3.3.1).

In this section, we propose one such filesystem design. The proposed scheme uses ext3 (a popular and widely-available Linux filesystem implementation) as a foundational building block and extends it with additional metadata that allows us to associate taint labels with entire files, as well as individual byte offsets within a file. In order to minimize changes to the protected VM, we adopt a client-server architecture and use NFS (a standard remote file access protocol) to connect the two sides.

This section presents the detailed design of our taint-aware filesystem layer. We begin by reviewing the relevant aspects of ext3 in Section 4.3.1 and then introduce our extensions in Section 4.3.2. Section 4.3.3 discusses our modifications to the NFS layer and Section 4.3.4 presents the design of our new shared-memory RPC transport that seeks to reduce the overhead of inter-VM communication. We evaluate the performance of our filesystem using an array of microbenchmarks in Section 4.3.5.

4.3.1 ext3: Design Overview

The *third extended filesystem (ext3)* [87] is a widely-used journaling filesystem that has become a standard component of the Linux kernel. ext3 offers a relatively straightforward, but robust filesystem implementation and, at the time of writing, serves as the default filesystem choice for many popular Linux distributions.

Core data structures: ext3 was designed as an extension to the ext2 filesystem and retains full compatibility with its on-disk data structure layout. Tracing the ancestral path one step further, the overall format of ext2 is derived from the design of the original Fast File System for UNIX [53].

Figure 4.13 illustrates the high-level layout of a disk partition formatted with ext2 and lists the most important data structures. The physical disk address space is divided up into an array of fixed-length *blocks* (typically of size 4KB) and organized into a series of *block groups* similar to FFS cylinder groups. This is done to reduce external fragmentation and minimize the number of disk seeks incurred during sequential file access. In addition to the actual data blocks, each block group contains an inode table, as well as bitmaps that track the allocation of disk blocks and inodes within the group. Each block group is identified by a *block group descriptor*, which records the location of the block bitmap, the inode bitmap, and the start of the inode table for the respective group. These descriptors are, in turn, stored in a top-level data structure, termed a *group descriptor table*. The *superblock*, another top-level data structure, maintains vital information about the current filesystem state, as well as various configuration parameters. The primary copy of the superblock resides at a fixed disk location (typically offset 0x400) and several additional backup replicas are maintained at other disk locations for reliability purposes.

A file in ext2 (and its descendant ext3) is represented by a fixed-length *inode* data structure that maintains its basic properties such as filesize, access timestamps, and an array of disk pointers to the data blocks. Using this array, the system can locate the data at any given file offset by following a chain of pointers, starting from the inode. Each inode holds a total of 15 disk pointer slots. The first 12 slots store direct pointers to the respective data blocks; slot number 13 points to an indirect block, number 14 to a doubly-indirect block, and number 15 to a triply-indirect block.

Journaling: ext3 reuses the basic filesystem structure of ext2, but adds support for journaling to achieve fast recovery after crash failures. Following a standard and widely-used approach, ext3 implements a basic form of write-ahead logging with redo-only recovery [37]. During normal operation, ext3 records all updates to its data structures into a fully-ordered circular *journal*, grouping sets of related updates into *transactions*. By forcing journal updates to disk before modifying the corresponding data structures, the filesystem guarantees that its on-disk image can always be recovered to a consistent state after a crash failure.

The journal is implemented as a special hidden file positioned in the first block group and ext3 defines a number of additional metadata structures to manage its contents. The *journal superblock* tracks summary information about the journal, such as the block size and its head and tail pointers. A *journal descriptor block* marks the beginning of a transaction and, following the standard write-ahead logging protocol, a *commit block* is appended to the journal at the end of a transaction. Once the commit block is written and flushed to disk, the journaled updates can be recovered without loss in the event of a subsequent failure. During recovery, ext3 simply scans the journal and replays each committed transaction, while discarding the incomplete ones.

4.3.2 Our Extensions to ext3

Taint label metadata: In order to provide support for file- and byte-level taint labeling, PIFT-ext3 makes two extensions to the basic format of ext3 data structures. First, the ext3 inode is extended with a new 32-bit field, which stores the file-level taint label (`fileTaintLabel`). Another option would be to store this label in an extended file attribute, but since ext3 maintains such attributes in a separate file block, accessing this value would likely incur the cost of an additional disk seek.

Second, for each regular file in the filesystem, PIFT-ext3 maintains some additional metadata that records its fine-grained byte-level taint labels. More specifically, each file data block in PIFT-ext3 is associated with a *block taint descriptor (BTD)* that stores byte-level taint labels within the respective block. This data structure is a direct analogue of the page taint descriptor described in Section 4.2.5, which our system uses to manage fine-grained labels for memory-resident data. The byte-level taint information is encoded within a BTD using the same choice of formats, namely *Uniform*, *Run-Length Encoding (RLE)*, and *Taintmap*, which allow us to trade off storage overhead and lookup latency at different levels of label fragmentation.

The BTDs are physically maintained in a dedicated region of disk space referred to as the *taint descriptor store*. This region typically resides on a separate partition or even on a separate physical disk, as specified by the user at the time of filesystem creation. (Maintaining the descriptor store on a dedicated device is advantageous from the performance standpoint, since this allows file data requests to be serviced concurrently with disk requests for the associated taint descriptors). Logically, the taint descriptor store comprises a flat array of fixed-length BTD blocks preceded by an allocation bitmap that keeps track of unused disk space. The size of a BTD block is chosen to match the amount of space required to store a complete *Taintmap* encoding for a single data block. Hence, in the default filesystem configuration with 4KB-sized data blocks and 32-bit taint labels, a single BTD occupies 16KB of disk space. The taint descriptor store is accessed by PIFT-ext3 through the standard low-level block device interface in the Linux kernel, i.e., the `ll_rw_block` function.

Achieving efficiency on file `READ` and `WRITE` operations requires the ability to quickly locate the BTD for any given data block and to this end, we augment the format of leaf indirect block entries. In addition to storing a data block pointer, each leaf indirect entry in PIFT-ext3 holds a concise *embedded taint locator* (`embTaintLoc`). This data structure comprises two 32-bit words and contains the following fields:

`btdFormat` (bits 0-1):
: Stores the format of the BTD associated with the respective data block. One of {*Uniform*, *RLE*, *Taintmap*}.

`btdValue` (bits 32-63):
: For *Uniform* format: stores the actual block-level taint label.

For *RLE* and *Taintmap* formats: stores the disk location of the BTD in the taint descriptor store.

Bits 2-31 are currently unused and are reserved for future extensions.

While this is one of several possible schemes for associating file data blocks with the corresponding BTDs, we believe that this particular method minimizes the overhead of additional disk activity in typical usage scenarios. For instance, when servicing a `READ` request for a given data block, the filesystem must locate the corresponding BTD and then transfer its contents into memory. In our scheme, the first step requires accessing the leaf indirect block to retrieve the `embTaintLoc` structure and this can be done with minimal overhead, since the entire chain of indirect blocks needs to be accessed anyway to locate the data block. While the second step (fetching the BTD) may require one additional disk operation, this overhead is incurred only for those file blocks that carry fragmented taint labels. For data blocks tainted with a *Uniform* label (likely a common case), label lookup comes essentially "for free", since the label is stored directly in the `btdValue` field of the embedded taint locator.

The principal downside of this approach is the reduction of the effective capacity of leaf indirect blocks. Rather than storing a plain 32-bit disk pointer, each leaf indirect entry in PIFT-ext3 holds a 96-bit data structure that combines a pointer with an embedded taint locator. Only 341 such structures can be stored in a single indirect block of size 4KB and as a result, the maximum filesize is reduced to $(12 + 341(1 + 1024 + 1024^2)) \times$ 4KB $=$ 1.3TB. By comparison, the pointer structure in unmodified ext3 allows addressing up to 4TB of data. Furthermore, our design increases the length of the indirect block chain for large files. A simple calculation shows that PIFT-ext3 requires two levels of indirection for files larger than 1.3MB and three levels for file sizes above 1.3GB (compared to 4MB and 4GB, respectively, for unmodified ext3). Increasing the length of the indirect block chain tends to increase the latency of file I/O operations due to additional disk seeks, but as the above calculation suggests, this overhead becomes noticeable only for relatively large files.

Figure 4.14 illustrates how PIFT-ext3 organizes taint label metadata with a simple schematic example. This figure depicts the on-disk structure of a hypothetical file that comprises four data blocks (denoted B_1 through B_4). The first three of these blocks are tainted with fragmented labels, which are maintained in the taint descriptor partition, while B_4 is tainted uniformly with label L_4. In this example, the disk pointers to B_1 and B_2 are maintained directly in the inode, while B_3 and B_4 are referenced through an indirect block. The pointers to BTDs for B_1 and B_2 are also maintained in the inode next to the respective data block pointers, whereas the pointer to the block taint descriptor for B_3 is stored in the indirect block alongside the pointer to B_3 itself. Finally, since B_4 carries a *Uniform* taint label, we avoid allocating space for its BTD in the taint descriptor partition and instead store its taint label (L_4) in the indirect block together with the pointer to B_4.

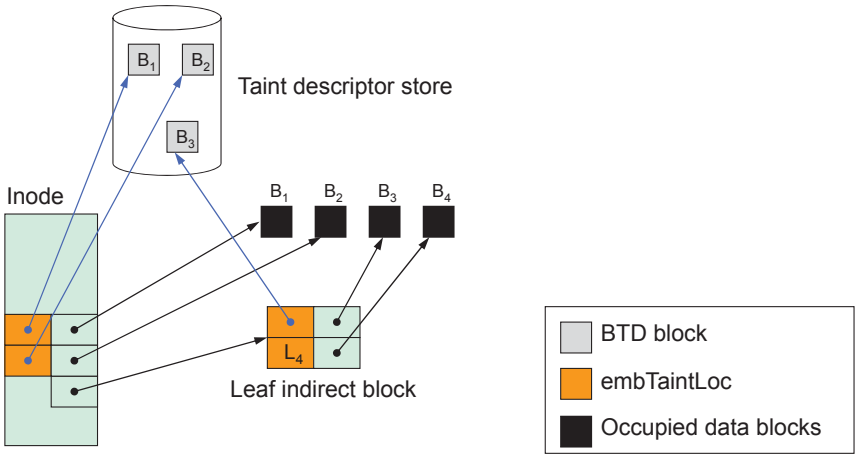


Figure 4.14. The on-disk layout of taint label metadata in PIFT-ext3.

Caching: In an effort to minimize the performance overhead incurred by access to the taint descriptor store, PIFT-ext3 maintains an in-memory cache of recently-used BTDs. This cache resides in heap-allocated kernel memory and is structured as a hash table, indexed by $\langle inodeNumber, blockNumber \rangle$. It has a fixed capacity (set to 1000 BTD entries by default) and implements a basic LRU replacement policy to manage space.

Synchronization: Like most other kernel-level subsystems, PIFT-ext3 must implement proper synchronization mechanisms to protect its shared state and ensure correct behavior in the face of concurrent access. The need for synchronization in a Linux filesystem arises from the fact that multiple kernel- or user-level threads may issue concurrent filesystem requests that manipulate shared data structures. In our case, such requests typically originate from a pool of kernel-level `nfsd` worker threads which, in turn, receive requests from the NFS client module executing in the protected VM. The following data structures maintained by PIFT-ext3 require explicit synchronization due to concurrent access from multiple contexts:

- *The block taint descriptor (BTD) cache.* As mentioned above, we maintain a cache of recently-used BTDs in kernel-level memory. This cache is structured as a hash table and represents a potential point of contention during parallel access. Our current implementation utilizes a simple spinlock primitive (`spinlock_t`, defined in `<linux/spinlock.h>`) to serialize hash table operations. In addition, every cached BTD entry carries a blocking mutex (`struct mutex`, defined in `<linux/mutex.h>`) that serializes conflicting operations on its contents. A thread servicing a file `READ` request must acquire this mutex before reading the contents of the buffer that contains the cached BTD. Analogously, a `WRITE` request handler must acquire this mutex prior to updating the cached BTD with data from a memory taint descriptor. All disk operations (revalidating cached entries and flushing modified entries to the backing store) are also performed under the protection of the corresponding block-level mu-

tex. Finally, each cache entry carries a reference counter, which tracks the number of active users and defends against race conditions, whereby one thread decides to evict a BTD entry from the cache while this entry is being accessed from another thread. The use of reference counters to coordinate the eviction of cache entries is a well-known technique from the domain of database management systems, where it has traditionally been used to protect the contents of the buffer cache against similar race conditions [74].

- *The block allocation bitmap for the taint descriptor store.* This shared on-disk data structure tracks the allocation of disk space from the taint descriptor store. For efficiency, PIFT-ext3 loads the entire bitmap from disk into memory during initialization and all subsequent operations are handled using the memory-based version. Ensuring the atomicity of updates to the bitmap is essential for correctness and our current implementation achieves this by dividing the bitmap into an array of fixed-length chunks and serializing access to each chunk via a blocking mutex.

4.3.3 Our Extensions to the NFS Layer

As we explain in Section 3.3, the protected VM accesses our on-disk filesystem remotely through the NFS [12] protocol. The control VM runs an augmented NFS server, which relays the client's requests to the on-disk filesystem and updates the taint-related data structures. Thus, when servicing a `READ` request for a specific file region, the NFS server first dispatches a regular `READ` operation through VFS (by invoking `vfs_readv`) and then asks the filesystem to provide an array of BTDs that cover the specified file region. This fine-grained taint information is relayed to the user-level taint tracker (described in Section 4.2), along with the physical memory address of the destination buffer for the `READ` operation. Using this information, the taint tracker updates the memory taint data structures in a manner that reflects the propagation of tainted file data into the destination buffer. Analogously, when servicing a `WRITE` operation, the NFS daemon must fetch an array of byte-level labels attached to the source memory buffer from the taint tracker and transfer it into the filesystem.

PIFT defines a standard interface for communicating fine-grained taint descriptors between the NFS layer and the underlying on-disk filesystem. More specifically, we add two new function pointer fields to the `inode_operations` structure and their definitions are shown in Figure 4.15. While not strictly necessary, requiring all interactions with PIFT-ext3 to go through a generic VFS-level interface provides flexibility and allows us to decouple the NFS server from the implementation of the underlying filesystem. From the practical standpoint, such separation is useful because we plan to experiment with alternative taint-aware filesystem designs in future work. Figure 4.16 illustrates how the new `get_filerange_taint` callback is implemented in PIFT-ext3.

```
int (*get_filerange_taint)(struct inode *inode,          // (In) VFS inode
                           loff_t file_offset,           // (In) Starting offset
                           size_t len,                   // (In) Range length
                           taintdescr_fmt_t *descr_fmt,  // (Out) BTD format
                           size_t *descr_len,            // (Out) BTD length
                           void **descr_buf);            // (Out) BTD buffer

int (*set_filerange_taint)(struct inode *inode,          // (In) VFS inode
                           loff_t file_offset,           // (In) Starting offset
                           size_t len,                   // (In) Range length
                           taintdescr_fmt_t descr_fmt,   // (In) BTD format
                           size_t descr_len,             // (In) BTD length
                           void *descr_buf);             // (In) BTD buffer
```

Figure 4.15. New VFS callbacks in the `inode_operations` structure.

4.3.4 Xen-RPC: An Efficient RPC Transport for Inter-VM Communication

The client-server design of our taint-aware filesystem enables us to minimize changes to the protected VM, but at the same time incurs additional performance costs due to inter-VM communication and signaling. In particular, transfer of file data is significantly less efficient in our two-sided design, since this data must be moved between the page cache and the network-level buffers on both sides of the connection.

In the standard implementation, NFS uses TCP as the underlying transport for its client-server communication and the two sides submit RPC messages directly to the kernel networking stack. NFS commands that carry file data (such as a `WRITE` request) must execute several expensive memory transfers that increase latency and impose a significant load on the memory subsystem. In typical cases, the sender first copies the data from the page cache into its network socket buffers and then transfers the data to the destination VM in a sequence of packets. When the destination VM receives these packets, it performs yet another memory copy to transfer the data from the network buffers to its local page cache. These memory transfers represent unnecessary overhead in our environment, since the client and the server operate on the same physical host and share its physical memory address space.

To address these inefficiencies, we have designed and implemented a special-purpose RPC transport layer (Xen-RPC) that allows the system to *efficiently* transfer RPC messages and the associated file data between a pair of VMs by setting up temporary shared memory mappings. Xen-RPC replaces the conventional socket-based transport, thereby eliminating unnecessary transfers to and from network-level buffers and allowing the server-side filesystem to access file data directly from the client-side page cache.

In operational terms, Xen-RPC mimics the traditional design of paravirtualized I/O devices by implementing the split-driver model. Xen-RPC's frontend component, operating

```
int pift_ext3_get_filerange_taint(struct inode *inode, loff_t rangeFileOffset,
                                  size_t rangeLen, taintdescr_fmt_t *descrFmt,
                                  size_t *descrLen, void **descrBuf) {
   int blockNum = rangeFileOffset / FILE_BLOCK_SIZE;
   int blockOffset = rangeFileOffset % FILE_BLOCK_SIZE;

   cachedBTD *btd = btdcache_lookup(inode->inodeNum, blockNum);
   if (btd == NULL) {                    /* Not found in the cache */
      embTaintLoc *emb = read_embedded(inode, blockNum) {
         /* Traverse the chain of indirect block entries and return
             the embedded taint locator in the leaf entry. */
      }

      btd = allocate_btd(inode, blockNum);
      btd->fmt = emb->descrFmt;
      if (emb->descrFmt == Uniform)
         initialize_uniform_btd(btd, emb->btdValue);
      else                                 /* RLE or Taintmap format */
         read_taint_descriptor_store(btd, emb->btdValue);

      if (btdcache_full())
         btdcache_evict_lru();
      btdcache_insert(btd);
   }

   /* Compute the taint descriptor for the specified file range */
   taintDescr *descr = compute_btd_subrange(btd, blockOffset, rangeLen);

   /* Merge the resulting descriptor with the file-level taint label */
   merge_taint_labels(descr, inode->fileTaintLabel);

   *descrFmt = descr->fmt;  /* Return the results */
   *descrLen = descr->len;
   *descrBuf = descr->buf;
}
```

Figure 4.16. The implementation of the `get_filerange_taint` callback in PIFT-ext3.

in the protected VM, registers itself with the `sunrpc` client layer by defining a new instance of `struct rpc_xprt_ops`. It exposes an interface for submitting RPC requests to the server (the `send_request` callback), as well as functions for connection establishment and teardown (`connect`, `destroy`, and `close`). The backend component, operating in the control VM, relays the client's requests to a pool of NFS server threads and communicates their completion status back to the client.

Following an established approach, Xen-RPC uses a *shared ring buffer* to communicate requests and the associated responses asynchronously between the two VMs. The ring buffer holds a window of outstanding requests and resides in a dedicated memory page (donated by the protected VM and mapped for shared access from both VMs). Xen event channels provide a foundational signaling mechanism, allowing the two sides to notify each other when a new message is posted to the shared ring.

```
struct xdr_buf {
  struct kvec head[1],        // RPC header + non-page data
              tail[1];        // Appended after page data

  struct page **pages;        // Array of contiguous pages
  unsigned int page_base,     // Start of page data
               page_len;      // Length of page data

  unsigned int buflen,        // Total length of storage buffer
               len;           // Length of XDR-encoded message
};
```

Figure 4.17. The format of an XDR buffer structure.

In traditional client/server NFS implementations, the two sides communicate by marshalling their requests and responses using the XDR (External Data Representation) format [81] and transmitting the resulting messages via the transport layer. The standard Linux implementation of NFS defines a basic structure (`struct xdr_buf`) to manage the contents of an XDR message. As illustrated in Figure 4.17, this data structure features a pair of linear buffers (`head` and `tail`) and an array of pointers to data pages. The `head` buffer stores the marshalled RPC header and the data payload for short messages. For messages that involve large transfers of contiguous file data, such as `WRITE` requests and responses to `READ` requests, the `pages` array holds references to the actual data pages that serve as sources or destinations for the transfer. Finally, the `tail` buffer allows the sender to append additional payload after the data in the page array. This field is used primarily for supplying padding bytes in order to satisfy the 32-bit alignment requirement in the XDR protocol. In typical scenarios, the `head` and `tail` buffers both reference temporary heap-allocated memory, whereas the `page` array holds pointers to file data buffers in the local page cache.

When the sender submits an `xdr_buf` instance for transmission, the transport layer must transfer the data referenced by its `head`, `tail`, and `pages` fields to the remote endpoint using transport-specific mechanisms. The traditional socket-based transport implementation simply relays the `xdr_buf` structure to the TCP networking layer, which copies the entire message into a local network-level buffer and prepares it for transmission. Notably, the contents of the data buffer must be transferred from the kernel-level page cache into a network socket buffer. On the receiving end, the transport module fetches the incoming RPC message and loads its contents into a local `xdr_buf` instance, which it then passes over to the NFS server layer. With a traditional socket-based transport implementation, this involves copying the file data in the reverse direction — from a local network-level buffer into a page cache buffer.

In contrast, Xen-RPC bypasses the networking layer and instead leverages Xen's inter-VM memory sharing facility to communicate the data payload more efficiently. Having set up a pair of `xdr_buf` structures for the RPC request and the associated response, the client (initiator of the request) temporarily shares the data pages referenced by their `head`, `tail`, and `pages` elements with the control VM. (This is done by issuing a grant table hypercall from the protected VM). Upon receiving the request, the server-side transport module run-

ning in the control VM maps these buffers into its local memory and then synthesizes a pair of local `xdr_buf` instances, which hold pointers to these foreign mappings. As a result, the NFS server can directly access the client's copy of the RPC request and any associated file data. Analogously, the server communicates its response by writing directly into its foreign mapping of the client-side response buffer, avoiding unnecessary transfers through the network-level buffers.

Next, we illustrate the operational aspects of our client-server filesystem stack with a simple example. In this example, we track the end-to-end execution path of a synchronous `WRITE` system call, which writes one page of application data to a file in our label-aware filesystem.

Client side (protected VM)

1. An application running in the protected VM issues a `WRITE` system call, specifying the file pointer, the address of a user-level source memory buffer (denoted Buf_U^P), and the desired length of the transfer (one memory page).

2. VFS dispatches this request to the kernel-level NFS client by invoking its `nfs_file_write` callback and this function, in turn, propagates the request to the page cache management module.

3. The page cache manager copies the application data from Buf_U^P to a kernel-level cache buffer, which we denote by Buf_K^P. This operation causes PIFT to transfer the associated taint labels from the application's address space to the region of physical memory that holds the kernel-level buffer.

4. Since the application has requested a synchronous transfer, the page cache manager invokes the cache writeback routine, which propagates the dirty cache buffer to the backing filesystem (in our case the NFS client) by calling its `writepage` callback.

5. The `nfs_writepage_sync` function receives control, prepares an RPC message structure carrying an `NFS3PROC_WRITE` command, and forwards it to the RPC communication subsystem by invoking `rpc_call_sync`.

6. The RPC layer allocates memory for the send and receive XDR buffers and marshals the arguments of the RPC request into the send buffer. Upon completion of this operation, `rq_snd_buf->head` holds the marshalled arguments of the `WRITE` command, the `rq_snd_buf->pages` array holds a reference to Buf_K^P, and `rq_snd_buf->tail` points to optional padding data. The newly-created request is then submitted to the underlying transport layer, which forwards it to the Xen-RPC client by invoking its `send_request` callback.

7. Xen-RPC receives control and prepares the send XDR buffer for remote access from the NFS server operating in the control VM. Specifically, it grants access to the physical memory pages that hold the `head` and `tail` buffers, as well as the page containing Buf_K^P. This is done by invoking the `gnttab_grant_foreign_access` routine, which

in turn makes a hypercall to Xen. The hypercall updates the grant table permissions and returns a set of grant references. Following an analogous procedure, the client grants foreign access to the `head`, `tail`, and `pages` components of the receive XDR buffer that will hold the response data. Finally, a Xen-RPC request data structure is initialized with the grant table references and written to the shared ring, after which the client notifies the server-side endpoint by signaling the Xen-RPC event channel.

Server side (control VM)

8. On the server side, the kernel-level handler for the Xen-RPC event channel is invoked in the interrupt context. The handler routine removes the request structure from the ring and passes it over to the NFS server for processing. First, it tries to assign the request to one of the idle server worker threads. If there are no idle workers available and the number of existing worker threads is at the maximum, it posts the request to a shared queue that is periodically checked from the worker thread context.

9. When an `nfsd` worker receives a request, it instructs the transport layer to construct a server-side RPC request structure (`struct svc_rqst *rqstp`).

10. Xen-RPC initializes the `rq_arg` and `res_arg` fields of `rqstp` by setting up foreign page mappings to the send and receive XDR buffers provided by the protected VM. Thus, the RPC request header and the incoming data pages are mapped into `rqstp->rq_arg->head` and `rqstp->rq_arg->pages`, respectively. Analogously, the pre-allocated buffers for the RPC response and any associated data are mapped into `rqstp->rq_res->head` and `rqstp->rq_res->pages`. To set up these mappings, Xen-RPC issues a `GNTTABOP_map_grant_ref` hypercall to Xen, supplying the grant table references specified in the shared-ring request.

11. Next the worker thread invokes the main RPC dispatch routine (`nfsd_dispatch`) to process the RPC request. This function unmarshals the RPC arguments from `rqstp->rq_arg->head` into a local data structure and then invokes a procedure-specific handler (`nfsd_write` in our example).

12. The procedure handler resolves the NFS file handle into a VFS file pointer and dispatches a write request to the VFS layer by calling the `vfs_writev` function. Note that the source buffer pointer, which the NFS server specifies as an argument to this function, references a foreign mapping of Buf_K^P.

13. VFS forwards the write request to our on-disk filesystem (PIFT-ext3), which, in turn, passes it over to the page cache manager.

14. The page cache manager in the control VM transfers the dirty file data from the foreign mapping of Buf_K^P to a local page cache buffer, which we denote by Buf_K^C. Since a synchronous operation was requested, the page cache manager invokes the cache writeback routine, which flushes the data from Buf_K^C to the disk partition managed by PIFT-ext3.

15. When the data transfer is completed, the NFS worker thread transfers the associated taint labels from memory to the filesystem. To accomplish this, it first makes an upcall to the QEMU-based taint tracker to fetch the taint descriptor associated with the physical memory page containing Buf_K^P. Having fetched this descriptor, the NFS server invokes the `set_filerange_taint` callback, which is routed via VFS to our taint-aware filesystem. As we describe in Section 4.3.2, this function transfers the labels from the supplied memory taint descriptor to the corresponding BTD.

16. Finally, the NFS worker thread prepares an RPC response message and communicates it to the client by writing it into `rqstp->rq_res->head`. In the last step, the NFS server signals request completion to Xen-RPC. The server-side component of our transport layer unmaps all foreign pages, prepares a Xen-RPC response structure, pushes it to the shared ring, and signals the client-side component via the event channel.

Client side (protected VM)

17. The client-side Xen-RPC module receives an event notification. The event handler (invoked in the interrupt context) tears down the shared mappings and signals the waiting application thread.

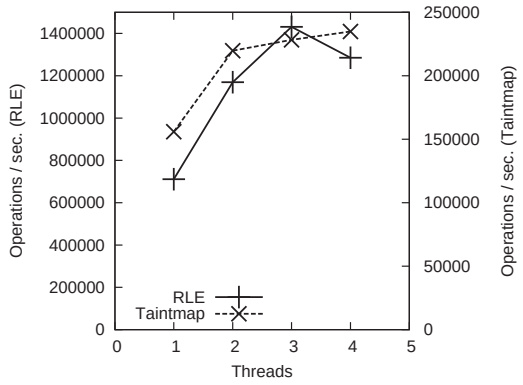
18. When the application thread (blocked on the RPC response) awakens, it processes the response status and returns from `nfs_file_write`, transferring control back to the generic `write` system call handler.

19. The kernel-level system call handler terminates and returns control back to the application.

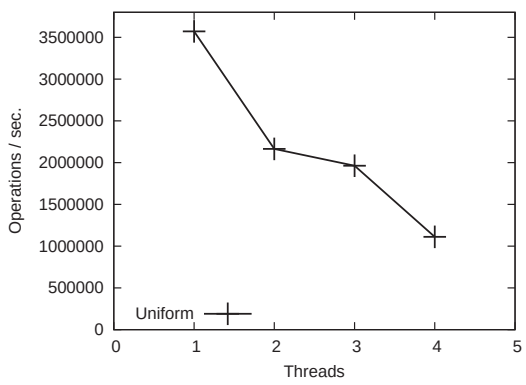
As can be seen from the above discussion, servicing a file-related system call via the PIFT infrastructure involves a carefully-orchestrated sequence of steps, but, crucially, the contents of the data buffer are transferred between memory locations only twice: from Buf_U^P to Buf_K^P and later on from Buf_K^P to Buf_K^C. This process is only slightly less efficient than the corresponding sequence of steps in a pure "bare-metal" system configuration, which requires transferring file data exactly once; namely, between an application-level buffer and a kernel-level page cache buffer.

4.3.5 Evaluation of PIFT-ext3

In this section, we evaluate the core components of our taint-aware storage stack using an ensemble of microbenchmarks. The overall goal of this evaluation is to demonstrate the viability of our design and its ability to achieve reasonable performance under highly stressful workloads. More specifically, our evaluation focuses on addressing the following key questions: (1) How efficient is our taint descriptor cache module and how well does



(a) *RLE* and *Taintmap* descriptor formats (note the different scales on the vertical axis).



(b) *Uniform* descriptor format.

Figure 4.18. Performance of the BTD cache module at varying levels of concurrency.

it scale to support parallel workloads on multi-core processors? (2) How does PIFT-ext3 affect the effective I/O bandwidth of the underlying storage device? (3) How successful is our custom transport layer (Xen-RPC) at reducing the overhead of inter-VM data transfers?

All of the experiments presented in this section were run on a Dell Optiplex 755 machine with a quad-core 1.6GHz Pentium 4 CPU, a 160GB 7200 RPM Seagate hard disk, and 4GB of RAM. We created a PIFT-ext3 filesystem on one of the disk partitions and assigned another partition (on the same physical disk) to serve as its taint descriptor store. These partitions were sized at 10GB and 40GB, respectively. The filesystem partition was configured with 4KB block size, which matched the memory page size in our environment. Both virtual machines (the protected VM and the control VM) ran the Fedora Core [34] distribution of Linux with kernel version 2.6.18-8.

Performance of the BTD Cache

The block taint descriptor (BTD) cache seeks to reduce the number of disk requests to the taint descriptor partition by storing a subset of the recently-used BTDs in memory. Our first experiment measures its maximum sustainable throughput under stressful and highly concurrent workloads. In this experiment, we connect the BTD cache module to a multithreaded user-level request generator. Each thread produces a synthetic stream of requests to the BTD cache, which alternate between reading and updating BTDs for randomly-chosen data blocks.

Recall from Section 4.3.2 that BTDs maintain fine-grained taint information within a data block using one of three different formats: *Uniform*, *RLE*, or *Taintmap*. The *Uniform* representation is highly compact and can be manipulated very efficiently, while the latter two are more expensive to maintain. In this experiment, we quantify this performance difference by measuring the overall request throughput for each of the three descriptor for-

mats at different levels of concurrency. Note that in this experiment, the BTD cache module does not access the disk and hence, the performance its determined solely by its ability to efficiently manipulate in-memory data structures and coordinate concurrent access from multiple threads.

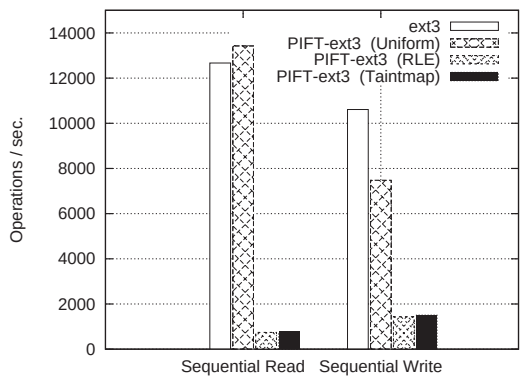
Figure 4.18(a) presents the throughput measurements for the *RLE* and *Taintmap* configurations. For a single-threaded workload, our implementation achieves 711237 and 155957 requests per second in these configurations, respectively. Under concurrent workloads, our system can take advantage of additional processor cores, demonstrating near-linear speedup to three processors for the *RLE* configuration. These results indicate that our method of concurrency control — a combination of coarse-grained spinlocks (held for very short periods of time) and more granular block-level mutexes (held across disk access operations) — enables a reasonable degree of parallelism and does not overburden the system with synchronization overhead in these scenarios.

Figure 4.18(b) shows the performance in a scenario, where each cached BTD initially carries a *Uniform* taint label and each update request replaces this block-level label by another randomly-chosen label. In this configuration, the performance characteristics of our implementation are quite different: while our system can process uniformly-labeled data blocks very efficiently in absolute terms, achieving an order-of-magnitude improvement over the *Taintmap* configuration, we observe that the aggregate throughput *decreases* as the level of concurrency grows. This result is noteworthy, but hardly surprising — we expect uniformly-labeled file blocks to be a very common case and our system has been carefully tuned to handle this scenario efficiently. In our current implementation, processing a read/update operation on a cached *Uniform* BTD requires only a few dozen machine instructions and consumes around $280ns$ of CPU time on our test machine. However, each operation must acquire a spinlock and a block-level mutex in order to coordinate concurrent access to shared state and these synchronization actions account for a significant fraction of the total cost. We observe from these results that any performance improvement we obtain by parallelizing across multiple hardware contexts is overshadowed by the significant costs of synchronization and cache coherence traffic on the memory bus.

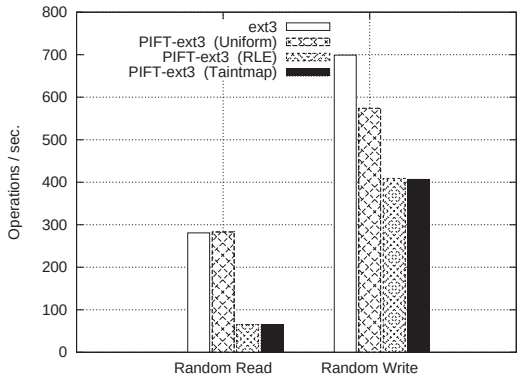
We note that despite this intrinsic overhead, our implementation handles uniformly-labeled BTDs efficiently relative to the other descriptor formats even under concurrent workloads. With 4 parallel client threads, our cache module achieves 1126650 requests per second for uniformly-tainted descriptor blocks and, by comparison, 234907 requests per second with the *Taintmap* format. In order to remedy the slowdown observed in 4.18(b), future versions of our filesystem may implement the BTD hash table using lock-free data structures or use more granular (per-bucket) spinlocks.

The Disk I/O Overhead of Taint Label Manipulation

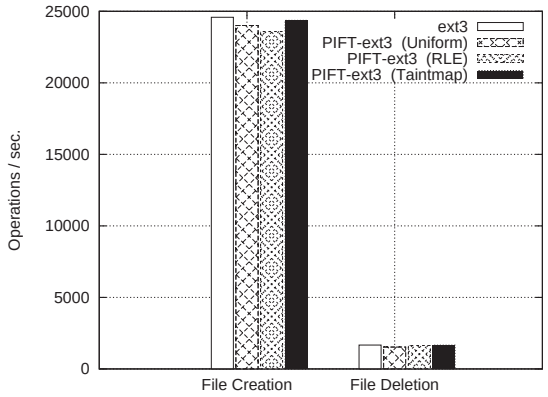
In the next set of experiments, we measure the additional disk overhead imposed by PIFT-ext3 and its effects on the overall completion time of file access requests. We compare the performance of PIFT-ext3 and the unmodified ext3 filesystem, both running inside



(a) Sequential file access (4KB request size, 100MB file size).



(b) Random file access (4KB request size, 100MB file size).



(c) Metadata workload.

Figure 4.19. Performance of the on-disk filesystem under data- and metadata-intensive workloads.

a native Linux kernel on "bare-metal" hardware without the hypervisor. For PIFT-ext3, we additionally modify its VFS operation callbacks to generate synthetic taint maintenance requests that reflect the effects of the respective file operations. Thus, when handling the `writepage` VFS callback, we generate a random page-sized taint descriptor and invoke `pift_ext3_set_filerange_taint` on the corresponding file range. Analogously, for `readpage` and `readpages` callbacks, we invoke `pift_ext3_get_filerange_taint` to read the taint descriptor for the corresponding file block from disk. As in the previous experiment, we measure and report the performance for each of the three taint descriptor formats. Each iteration of the experiment starts out with a cold page cache and a cold BTD cache. At the end of each run, we issue a `sync` command to write back any dirty data that remains in these caches and count its completion time towards the total running time.

In this experiment, we use FileBench [35] to generate the load and we chose this benchmark due to its flexibility, accuracy, and ease of configuration. Filebench was developed by

Sun Microsystems and was used for performance analysis of the Solaris OS [52], as well as in numerous recent academic studies [33, 39].

Figure 4.19 presents the results of this experiment. Subfigures (a) and (b) report the performance of basic file access operations (reads and writes) on a pre-existing file, measuring sequential and random access, respectively. As expected, the magnitude of the overhead is highly dependent on the size of a taint descriptor block, which is determined by its format. Most strikingly, the filesystem suffers an 85% drop in throughput on sequential reads with RLE and Taintmap descriptor formats. This is to be expected, since in our evaluation environment PIFT-ext3 transforms sequential file access into a stream of random-access disk requests. To understand this phenomenon, recall that for each file data block accessed by an application, PIFT-ext3 must access the corresponding BTD on the auxiliary taint descriptor partition. In our environment, both partitions reside on the same disk drive and hence, each data block access incurs the cost of two inter-partition disk seeks. As expected, the overhead is much less noticeable with random file access, which naturally suffers the disk seek overhead on every data block access. In the random read measurement, our extensions to ext3 decrease the throughput by 76% and this result reflects the reduction in the effective disk bandwidth — for every file data block of size 4KB, PIFT-ext3 must read the corresponding BTD, which occupies 16KB of disk space in both representations (RLE and Taintmap). Random file writes are considerably less expensive than random reads, since they benefit from asynchronous cache writebacks and request reordering at the device level, whereas read operations are fundamentally synchronous in our experiments.

Moving the auxiliary partition to a separate disk drive would allow PIFT-ext3 to preserve sequentiality and service BTD accesses in parallel with data block transfers, thus reducing latency and increasing the effective disk bandwidth. We plan to evaluate this optimized configuration in future work.

Note that the performance of PIFT-ext3 on uniformly-tainted file blocks is comparable to that of the unmodified ext3 implementation, as we expect. In our current design, uniform block taint descriptors are maintained in leaf indirect blocks, which both implementations must access on every file operation in order to obtain the data block pointer. Hence, file read operations on uniformly-tainted data blocks suffer no additional latency or bandwidth penalty. File writes are slightly more expensive in PIFT-ext3, since they require updating the indirect blocks and periodically flushing them to disk. Viewed collectively, these results confirm that our design is generally successful at minimizing the overhead associated with access to uniformly-tainted file data, which we expect to be the most common case.

Figure 4.19(c) compares the performance on metadata-intensive workloads. In this experiment, we first populate an empty directory with 100000 files and then delete these files by executing "`rm -rf *`". Our design does not alter the processing of metadata-related tasks, such as these, and hence imposes no additional overhead, as evidenced by these results.

Overall Performance of the Storage Subsystem

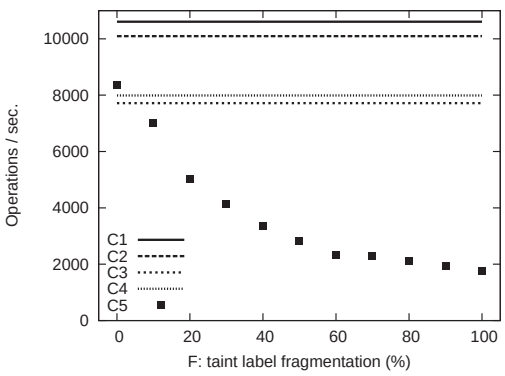
In our final set of experiments, we take a more macroscopic view and evaluate the performance of the entire storage stack, which combines the on-disk filesystem, the NFS wrappers, and the shared-memory RPC transport layer. Our goal is to assess the overall performance of the storage component in a fully-featured PIFT configuration and measure the worst-case overhead relative to "plain" paravirtualized and non-virtualized environments. More specifically, we compare the following configurations:

- ***C1***: A "bare-metal" configuration running unmodified Linux and ext3 without the hypervisor.
- ***C2***: A paravirtualized configuration running Linux and Xen. The control VM exposes one disk partition as a paravirtualized block device; the guest VM mounts this device as ext3.
- ***C3***: A paravirtualized configuration running Linux and Xen. The control VM mounts one of the disk partitions as ext3 and exposes it to the guest through unmodified NFS with TCP transport.
- ***C4***: A paravirtualized configuration running Linux and Xen. The control VM mounts one of the disk partitions as ext3 and exposes it to the guest through NFS and our shared-memory transport layer (Xen-RPC).
- ***C5***: A fully-featured PIFT configuration. The control VM mounts one of the disk partitions as PIFT-ext3 and exposes it to the protected VM through NFS and Xen-RPC. Since our focus in this section is on evaluating filesystem performance, as opposed to the computational overhead of taint tracking, we instrument this configuration in a manner that allows the protected VM to submit file I/O requests that operate on tainted data without entering emulation. To accomplish this, we instrument Xen-RPC to assign synthetic taint labels to all memory buffers that carry file data once they have been submitted for transmission by the client-side NFS endpoint. To simulate varying levels of taint label fragmentation, a fraction of the data buffers (F) is assigned a randomly-generated *Taintmap* descriptor and the rest are assigned a random *Uniform* descriptor.

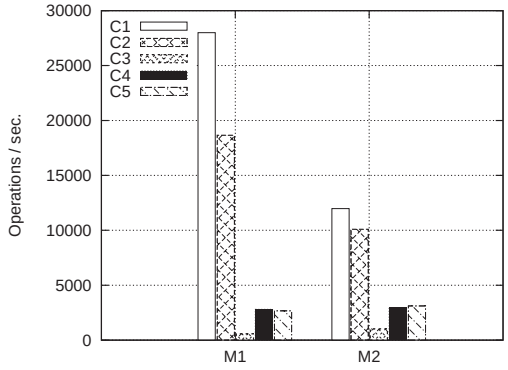
In our first experiment, we test the performance under the sequential write workload (described in the previous subsection) in these five configurations, varying the level of label fragmentation in *C5* between 0% and 100%. Table 4.5 shows the throughput in configurations *C1*-*C5* with the value of F fixed at 0%. Figure 4.20(a) plots the performance in *C5* for varying values of F. We observe that at low levels of label fragmentation, the throughput achieved by PIFT is comparable to that of the basic ext3/NFS configuration (*C4*), but both are measurably slower than the non-virtualized configuration (*C1*). The difference between *C1* and *C4*/*C5* is attributable to the overhead incurred by NFS and inter-domain communication, which are inherent in our design.

	C1	*C2*	*C3*	*C4*	*C5* with F = 0%
Operations / sec.	10609	10094	7715	7989	8354
Slowdown relative to *C1*	0.0%	4.9%	27.3%	24.7%	21.3%

Table 4.5. Operation throughput for sequential file writes (4KB request size, 100MB file size) across all five benchmark configurations. In *C5*, each data block carries a *Uniform* taint label.



(a) Performance of sequential file writes (4KB request size, 100MB file size) across all five configurations. In configuration *C5*, we vary the level of taint label fragmentation (F) between 0% and 100%.



(b) Metadata benchmark performance across all five configurations.

Figure 4.20. Overall performance of the PIFT storage subsystem under data- and metadata-intensive filesystem workloads.

As expected, the overall application-level throughput exhibits significant dependence on the degree of taint label fragmentation. In the degenerate scenario of $F = 100\%$, our storage subsystem achieves roughly $1/6^{th}$ of the maximum attainable throughput and this result is also consistent with our expectations. Loosely speaking, writing a single file data block that carries a *Taintmap* BTD incurs the cost of six disk writes in our current design: writing the data block itself, writing the associated BTD (which occupies four filesystem blocks), and updating the leaf indirect block with a pointer to the new BTD.

Our second experiment examines the performance of metadata-related operations using two synthetic benchmarks:

- ***M1***: Create a directory tree with depth 6 and fanout 6.
- ***M2***: Delete the directory tree created in *M1*.

Figure 4.20(b) shows a side-by-side comparison of metadata operation performance and the results clearly indicate a significant level of variability among the five configurations. The extreme slowdown observable in configuration *C3* indicates the magnitude of the overhead incurred by the NFS layer. In this experiment, our benchmark is best viewed as

a CPU-bound workload consisting of a large number of fine-grained operations (directory insertions and removals). Each such operation involves only several updates to in-memory filesystem data structures and thus represents a relatively small unit of work. Focusing on the performance gap between *C2* and *C3*, these configurations differ mainly in how they distribute this work among the two VMs. In *C2*, the guest VM mounts the paravirtualized block device and the ext3 filesystem layer runs directly inside the guest kernel. Hence, each directory update operation is processed locally by the guest VM and the costs of inter-VM communication are incurred only occasionally, when the guest kernel decides to flush the updated directory blocks to disk. In contrast, *C3* deploys the on-disk filesystem in the control VM and the guest mounts it remotely via NFS. In this two-sided configuration, the guest must relay each directory update request to the control VM via the NFS stack. Each such request must be marshalled into an RPC message, copied into a network-level buffer, and transmitted to the server via TCP/IP. As our results suggest, these manipulations become the dominant source of overhead and cause a drastic slowdown.

Next, comparing *C3* to *C4*, the difference indicates the reduction of costs achieved by replacing TCP with our custom shared-memory transport module. In *C4*, each directory update must still cross the VM boundary, but the system bypasses the TCP layer and instead allows the server to directly access the client-side RPC buffer by setting up a shared mapping.

In *C5*, the on-disk filesystem in the control VM is replaced by our label-aware implementation. Since the scope of the workload in this experiment is limited to directory operations, PIFT-ext3 does not need to update taint labels on file data blocks and hence does not impose a significant amount of additional load on the CPU or the storage device. As a results, the performance achieved in this configuration is comparable to that of *C4*. However, both configurations are noticeably less efficient than *C1* and *C2* and this difference is attributable to the costs of the NFS protocol and inter-domain signaling, which are inherent in our client-server design.

Evaluation Summary

In summary, the microbenchmark results presented in this section and our usage experiences lead us to conclude that our label-aware storage module can serve as a viable building block for a comprehensive information flow tracking substrate such as PIFT. Although the design of our filesystem was guided by a set of practical compromises that impose non-essential overhead, the evaluation demonstrates that our storage layer can deliver competitive performance even under significant levels of stress.

The BTD cache offers a crucial optimization, allowing us to reduce the number of accesses to the taint descriptor partition and thus improve the effective disk bandwidth. The challenge lies in scaling the cache implementation in a manner that would allow it to achieve efficiency on parallel application workloads — an increasingly important consideration in light of the inevitable and rapidly growing adoption of multi-core architectures.

The disk I/O overhead of accessing taint labels is negligible for metadata operations.

For operations that manipulate file data, the disk overhead varies and depends on the level of label fragmentation. Files and individual data blocks that are tainted uniformly with a single label can be accessed very efficiently and PIFT-ext3 incurs almost no additional performance costs. Conversely, accessing data blocks that carry fragmented taint labels is considerably more expensive due to the additional disk transfers. Most alarmingly, our current design tends to transform sequential file workloads into random disk workloads and future improvements to PIFT-ext3 will focus on reclaiming sequentiality.

Finally, the client-server design was introduced in order to minimize the required set of changes to the protected VM's software stack. The NFS layer imposes a noticeable cost resulting from RPC marshalling and inter-VM communication, but our novel shared-memory transport module helps mitigate this overhead.

4.4 Policy Enforcement

The previous sections have focused on the core mechanisms for tracking the flow of tainted information and storing it persistently on disk. It remains to describe one more essential component of our implementation; namely, the module responsible for evaluating policies and enforcing restrictions on the dissemination of sensitive information.

As we explain in Chapter 3, PIFT delegates policy enforcement to a set of user- or administrator-defined modules called *enforcement handlers*, whose specifications and implementation are external to PIFT and are beyond the scope of this dissertation. When PIFT detects that the protected VM has made a request to externalize tainted data through a virtual I/O device, it intercepts the request and invokes a device-specific enforcement handler, supplying a 32-bit taint label that represents the data being externalized. It is the responsibility of the enforcement handler to resolve this opaque token into a decentralized data label and evaluate its constituent policies.

Our current Xen-based prototype implements request interception for two types of virtual I/O devices: the standard paravirtualized block storage device and the standard paravirtualized network interface. The former can be used to control the transfer of sensitive data to removable hard drives, USB storage keys, and other secondary storage devices that are not directly managed or controlled by PIFT. The latter mechanism provides a means of controlling the transfer of information between principals over the network, as well as restricting its release to external networks, such as the public Internet.

In the paravirtualized model, on which our implementation is based, virtual I/O device abstractions are implemented using the *split-driver* scheme. In this scheme, the *front-end* component (operating in the protected VM) collects I/O requests issued by the guest kernel and forwards them to the *back-end* component through a shared ring buffer. The back-end operates in the kernel space of the control VM and is responsible for relaying the protected VM's requests to the actual hardware device via the standard kernel-level device interface. A data transfer request typically specifies a source or a destination buffer, which resides

in protected VM's memory, and in order to make this buffer available for DMA from the control VM, the backend makes a hypercall to Xen, requesting this buffer to be temporarily mapped into the address space of the control VM.

The back-end driver, operating in the control VM, provides a convenient point of interposition and request interception. Before relaying a `WRITE` or `TRANSMIT` request to the hardware device, our modified back-end drivers make an upcall to the user-level emulator (described in Section 4.2) to obtain the taint labels associated with the outbound data buffers. If the data carries a non-empty taint label, the back-end invokes a device-specific enforcement handler, which runs as a kernel module in the control VM and is registered with PIFT at the time of system startup. The enforcement handler examines the supplied taint labels, resolves them into decentralized data labels, and evaluates the corresponding policies. Based on these policies, the handler can decide to allow or block the release of tainted information and this decision is signaled back to the driver via a status code.

Next, we discuss the functional aspects of back-end drivers for storage and network devices, as well as our modifications to these modules, in further detail.

4.4.1 Enforcement for Virtual Block Devices

The core back-end driver functionality for paravirtualized storage devices is implemented in `linux/drivers/xen/blkback/blkback.c`. This file contains, among others, the main request dispatch routine (`dispatch_rw_block_io`), which is invoked in the context of a kernel worker thread when a new disk request appears on the shared ring. The request (defined by the parameter `blkif_request_t *req`) specifies a set of page-level segments, which reference the data buffers, and these segments constitute the basic units of inter-VM sharing. The dispatch routine invokes the `GNTTABOP_map_grant_ref` hypercall to map these foreign buffers into the local address space of the control VM. Once these mappings have been established, our modified implementation examines the operation type (`req->operation`) and, for `WRITE` operations, makes an upcall to the emulator to obtain an aggregated taint label for each of the segments. This exchange is performed though a client-server protocol running on top of Netlink, as discussed in Section 4.2.7. The request message sent by the kernel-level driver includes the segment's physical page number, its offset within this page, and the length of the segment. Our extended emulator looks up the page taint descriptor (PTD) for the specified page number and returns its taint label. If the PTD specifies a non-uniform (i.e., fragmented) taint label, the emulator aggregates the individual byte-level labels by invoking the `merge_labels` function (described in Section 4.2.5) and returns a single aggregated taint label that represents the union of all policies.

Another point worth mentioning about the taint label lookup procedure is that if the protected VM is running in the emulated mode at the time when the dispatch routine is invoked and if the asynchronous mode of taint tracking is enabled, QEMU may need to temporarily suspend the emulated guest context in order to allow the taint processor to drain its log and bring the state of memory taint labels up-to-date. This is one of the very

```
typedef uint32_t taint_label;

/* The enforcement handler for WRITE requests to a paravirtualized block
   storage device */
int evaluate_policies_bdev(
      struct block_device *dev,   // (In) Description of physical block device
      taint_label labels[],       // (In) Array of segment-level labels
      unsigned int num_segments); // (In) Number of segments

/* The enforcement handler for TRANSMIT requests to a paravirtualized network
   interface */
int evaluate_policies_netif(
      struct net_device *dev,   // (In) Description of physical netw. interface
      taint_label label,        // (In) Packet taint label
      void *payload);           // (In) Pointer to the packet payload
```

Figure 4.21. Function prototypes of policy enforcement handlers for paravirtualized disk and network devices.

few scenarios in PIFT that demand explicit synchronization between the emulator thread, which supplies blocks of taint tracking code, and the taint processor, which consumes these blocks asynchronously and advances the view of taint label assignments. From an application's point of view, this synchronization period can be seen as a form of buffering and manifests itself as a brief delay between the issuance of a `WRITE` request and its subsequent release to the physical disk drive. We do not expect this delay to pose a significant practical challenge, since current applications that exhibit sensitivity to the timing of disk operations must already be equipped to deal with the effects of request buffering and reordering at various levels in the stack — a standard practice in modern operating systems.

When the kernel-level dispatch routine regains control, it invokes the external enforcement handler function for block storage devices, whose full prototype is shown in Figure 4.21. In the current implementation, this function accepts three parameters: the `block_device` structure that identifies the destination physical device, the array of aggregated segment-level taint labels obtained from the emulator in the previous step, and the size of this array. The enforcement callback evaluates the policies defined by the segment-level labels, decides whether the data contained in these segments can be safely externalized to the specified storage device, and signals its decision to the driver via the return code. If the enforcement handler returns 0, the dispatch routine continues along the normal request processing codepath; it constructs a low-level disk request descriptor (`struct bio`) and invokes the `submit_bio` kernel function, which propagates this request to the generic physical block device layer. Otherwise, it immediately cancels the request and signals an error condition (`BLKIF_RSP_ERROR`) to the front-end component.

4.4.2 Enforcement for Virtual Network Interfaces

The central component of the back-end driver for virtual NICs is implemented in `linux/drivers/xen/netback/netback.c` and the general procedure for intercepting network output and enforcing policies is analogous to the one used for virtual storage devices. When the backend receives an event notification from Xen indicating the appearance of a `TRANSMIT` request on the shared ring, it schedules the `net_tx_action` routine for execution within a kernel-level tasklet. This routine copies the request control information from the shared ring onto the local stack, allocates a generic kernel socket buffer (`struct sk_buff`), and instructs the hypervisor to map the foreign buffer containing the packet payload into the address space of the control VM.

Our modified version then issues an upcall to QEMU and requests an aggregated taint label for this foreign data buffer. The action routine then invokes the administrator-supplied network enforcement handler, passing it the identifier of the physical interface (`struct net_device`) and the taint label, as shown in Figure 4.21. Note that since Xen guarantees that the protected VM's data buffer does not span page boundaries, it suffices to supply only one taint label value (as opposed to passing an array of segment-level labels, as we do for disk requests). However, in our current design the network enforcement handler accepts one additional parameter; namely, a pointer to the local mapping of the packet's data buffer. Exposing the data payload to the enforcement module appears to be advantageous (and perhaps even essential) in this particular case, since this information can be gainfully exploited to attain finer control and implement a wider range of useful policies. For instance, exposing the packet header information allows the enforcement module to restrict information flow on the basis of Ethernet- or IP-level destination addresses. Revealing the full packet enables an even broader range of policies that can use information contained in the TCP header or even perform deep packet inspection.

If the enforcement handler approves the attempted `TRANSMIT` operation, the action routine proceeds to copying the payload from the foreign mapping into a local kernel socket buffer and then passes this buffer to the `netif_rx` function. Otherwise, the back-end driver drops the request and invokes the `netbk_tx_err` routine, which constructs an error response, posts it to the shared ring, and signals the front-end component.

4.5 Extending PIFT to a Distributed Environment

The material presented in the preceding sections describes how PIFT tracks the flow of information and enforces policies within the boundaries of a single machine. The last piece of the architectural puzzle to be specified is how to extend the single-node platform in manner that would enable us to track information flow between machines and principals in a networked environment. That is, when an application running on machine M_a sends a message containing some tainted data D over the network to another machine M_b, it is not enough to simply invoke the enforcement handler and verify that M_b is authorized to access

D. We must also ensure that when M_b receives this message, its copy of D retains the taint status and the confidentiality policies associated with the original copy held by M_a.

Extending the single-node PIFT design to track taint labels across network transfers is conceptually straightforward: the hypervisor on the sender's machine can simply annotate each outbound network packet with a concise encoding of its taint labels. Analogously, upon receiving a packet from the network, the hypervisor can detach the labels and propagate them to the memory buffers holding the payload.

To implement this feature, we make another functional extension to the back-end component of the network driver. On the sender's side, if the back-end receives permission from the enforcement handler to externalize the data to the receiver, it makes another request to QEMU to obtain the fine-grained non-aggregated taint labels for the memory buffer holding the payload. Using this information, the back-end constructs a *packet taint descriptor* — a data structure that seeks to concisely describe the byte-level taint status of a packet. This data structure is a direct analogue of the page taint descriptor (Section 4.2) and the block taint descriptor (Section 4.3), which store taint metadata for memory pages and filesystem blocks, respectively. The fine-grained byte-level view is represented using one of three different formats (*Uniform*, *RLE*, or *Taintmap*) and the choice of format is determined by the level of taint fragmentation within the packet. The back-end driver then transfers the packet data and the associated taint descriptor to the *communication daemon* — a lightweight user-level process that runs in background in the control VM. This daemon concatenates the packet payload with the taint descriptor and forwards the resulting message to its peer on the receiver's machine through a TCP/IP tunnel.

When the packet reaches the destination machine, the control VM's networking stack demultiplexes it based on the outer packet headers and forwards the concatenated message to the user-level communication daemon, which, in turn, relays it to the back-end driver that implements the virtual NIC. The back-end detaches the packet taint descriptor from the inner packet, injects the latter into the protected VM's stack, and makes an upcall to QEMU, instructing it to taint the destination memory buffer accordingly.

Overall, our strategy of relaying traffic through a user-level daemon is a well-established technique and we implement it using mechanisms similar to those used by the TUN/TAP [82] kernel driver, which enables packet tunneling through a user-level process on traditional non-virtualized Linux platforms. Unsurprisingly, our implementation faces similar limitations, the most significant of which is the increased load on the CPU and the memory bus resulting from the copying of packet data between kernel- and user-space contexts. In the next version of our prototype, we plan to eliminate this non-essential overhead by pushing the tunneling functionality directly into the kernel.

4.5.1 Bandwidth Overhead Evaluation

The ability to track the flow of sensitive data across machines in a networked environment does not come for free; our extensions to the networking stack consume a certain

fraction of the available network bandwidth due to tunneling and taint label annotations. We conducted a simple experiment to study the effects of these extensions on the effective bandwidth and measure the extent to which they degrade the overall network performance.

In this experiment, we use the `iperf` [43] network benchmark to measure the effective network bandwidth between a pair of PIFT-enabled machines connected by a point-to-point 100Mbps Ethernet link. In order to isolate the performance impact of tunneling from the overhead of taint annotations, we run `iperf` and measure the resulting performance in three different configurations, specifically:

- ***C1***: A "bare-metal" configuration running unmodified Linux without the hypervisor. `iperf` is configured to communicate directly over the physical network interface.
- ***C2***: A "bare-metal" configuration running Linux without the hypervisor. `iperf` is configured to communicate over a virtual interface that binds to a TUN/TAP kernel device. This device relays all network packets to a minimal user-level communication daemon, which transmits them to the remote endpoint via a TCP/IP tunnel.
- ***C3***: A fully-featured PIFT configuration. `iperf` runs as a user-level process in the protected VM and communicates via a paravirtulized network interface. The back-end driver redirects all outbound packets to a user-level communication daemon, which transmits them to the remote endpoint through a TCP/IP tunnel. In this configuration, we also instrument the communication daemon to annotate a random sampling of the outgoing packets with PTDs, varying two parameters: taint prevalence (P) and the level of label fragmentation (F). The first parameter determines the fraction of packets that are marked as containing tainted data. A fraction (F) of these tainted packets is assigned a randomly-generated *Taintmap* descriptor and the rest are assigned a random *Uniform* descriptor. Since the intent of this microbenchmark is to measure the network bandwidth penalty due to packet annotations, rather than the computational overhead of emulation and taint tracking, we also instrument the daemon to clear the taint labels on the receiving side prior to injecting packets into the protected VM's stack. This step is necessary because failing to clear the labels would cause the protected VM to process incoming packets in the emulated mode and this would interfere with our bandwidth measurements.

In all configurations, `iperf` is set up with a unidirectional TCP-based data stream. To estimate the effective network bandwidth, we record and plot the overall connection throughput, as reported by the client endpoint.

Figure 4.22 presents the results of our bandwidth measurements for three distinct values of P and varying F. Looking at these results, the difference between *C1* and *C2* reflects the overhead of packet encapsulation and tunneling. While certainly measurable, the reduction in effective bandwidth is relatively modest and is limited to the cost of the additional TCP, IP, and Ethernet headers (roughly 66 bytes per packet). In *C3*, the overhead is strongly dependent on the prevalence of tainted data, as well as the degree of label fragmentation, as we would expect. With $P = 50\%$ and $F = 10\%$, users would observe a

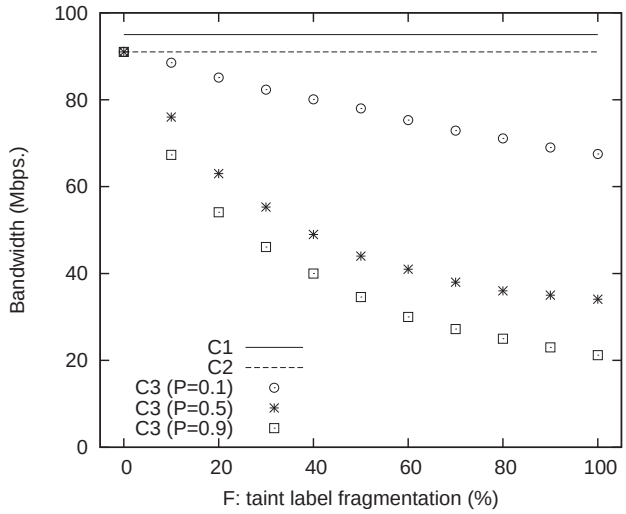


Figure 4.22. The effective network bandwidth, as measured by `iperf`, for varying amounts of tainted data (P) and at varying levels of label fragmentation (F).

20% reduction in effective bandwidth compared to the "bare-metal" configuration *C1*. In an extremely stressful scenario, where 90% of all packets contain sensitive data and each packet carries a highly-fragmented *Taintmap* descriptor, the sustainable bandwidth drops to 21.2Mbps. This can be seen as a dramatic penalty, but the result is fully consistent with our expectations: with a 32-bit label space, each tainted packet of length L carries a byte-level *Taintmap* descriptor of length $4L$ and hence, the application-level bandwidth is reduced to approximately $1/5^{th}$ of its original value.

Chapter 5

Full-System Performance Evaluation

In this chapter, we evaluate the performance of our PIFT prototype under a variety of workloads, which range from computationally-intensive microbenchmarks to interactive usage scenarios in graphical desktop environments. Our evaluation focuses on addressing the following key questions:

- How large is the performance penalty incurred by PIFT on worst-case computationally-bound workloads operating on tainted data (Section 5.1)?
- How effective are PIFT's on-demand emulation techniques at reducing the amount of time spent in the emulated mode (Section 5.1)?
- What are the benefits and limitations of asynchronous parallelized tracking and how does the size of the taint argument log affect our system's performance (Section 5.2)?
- How does our taint tracking platform affect interactivity and user productivity in graphical application environments (Section 5.3)?

To answer these questions, we ran a series of experiments and compared the overhead in the following configurations:

- ***NL***: A "bare metal" configuration running unmodified Linux on native hardware.
- ***PVL***: A paravirtualized configuration running Linux and Xen.
- ***Emul***: Linux running in a fully-emulated environment based on unmodified QEMU.
- ***PIFT-S***: Our prototype implementation of PIFT with synchronous taint tracking.
- ***PIFT-A(*x*)***: Our prototype implementation of PIFT with asynchronous parallelized taint tracking and x MB of physical memory reserved for the taint argument log.

5.1 Computationally-Intensive Workloads

In our first set of experiments, we evaluate the performance of our prototype under a variety of CPU-driven workloads, which intensively manipulate and perform computation on sensitive input data. Naturally, CPU-bound workloads that actively manipulate tainted data represent the most stressful scenario for PIFT, as each instruction that touches a sensitive value must be carefully analyzed and emulated. Hence, the experiments presented in this section can be viewed as measuring the worst-case performance overhead incurred by PIFT's emulation and information flow analysis components.

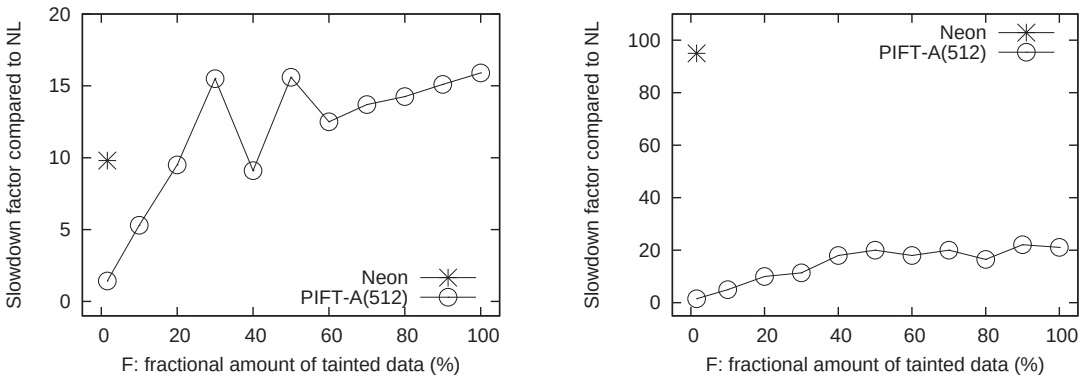
Our test machine for these experiments is a Dell Optiplex 755 with a quad-core Intel 2.4GHz CPU and 4GB of RAM. The hypervisor-level component of our prototype is based on Xen version 3.3.0. The augmented emulator (based on QEMU version 0.10.0) runs in the control VM as a multi-threaded user-level process. The protected VM is configured with 512MB of RAM and one VCPU, as our current implementation does not yet offer support for multi-processor guest environments. All tainted data files are accessed from a PIFT-ext3 filesystem, which the protected VM mounts remotely over NFS and Xen-RPC. PIFT-ext3 operates as a kernel extension in the control VM and is configured to access a 160GB 7200 RPM Seagate hard disk. Both virtual machines run the Fedora Core distribution of Linux with kernel version 2.6.18-8.

5.1.1 Copying and Compressing

We begin by considering two simple, but very common data manipulation activities — copying and compressing data files. In this experiment, we execute the following tasks from the command line, measure their running times, and report the slowdown incurred by PIFT:

LocalCopy: This task copies a sensitive file to another file in the same directory using the `cp` command from the GNU coreutils package [18]. In operational terms, copying a file on Linux involves a sequence of `sys_read` and `sys_write` system calls, which transfer the data from the source file buffer in the kernel-level page cache to an intermediate user-level buffer in the address space of the `cp` process, and from there to the destination file buffer in the page cache. From the vantage point of PIFT, the protected machine transfers tainted data between these memory buffers in 32KB-sized chunks using the `repz movsd` instruction, producing a memory-bound workload.

Compress: This task compresses a tainted input file using the GNU `gzip` command [40]. This command reads the contents of the input file into a user-level buffer via a sequence of `sys_read` system calls, compresses the data using a combination of Huffman coding [42] and LZ77 [99], and writes the results to a kernel-level page cache area that represents the output file using `sys_write`. At the instruction level, this operation involves a sequence of user-kernel data transfers using the `repz movsd`



(a) LocalCopy (b) Compress

Figure 5.1. Performance of *PIFT-A(512)* and Neon on file copying and compression tasks with a varying amount of tainted data (F).

instruction, as well as a nontrivial amount of user-level activity associated with computing the Huffman tree and identifying redundancies in the input byte stream.

This choice of benchmarks also allows us to compare our results with those in Neon [98]. Neon builds on top of the on-demand emulation primitive developed by Ho *et al.* [41] and hence provides a meaningful comparison to both systems most closely related to ours. To match Neon, we use a 4MB input file, measure the command completion time, and report the slowdown relative to the native unmonitored configuration (*NL*). Before the start of each measurement, we pre-stage the input file into filesystem buffers in the protected VM. This allows us to factor out the overhead of disk I/O (which remains constant across all configurations) and measure the fundamental overhead of taint tracking in the most stressful scenario — a CPU-bound task. In the PIFT configuration, we apply a non-zero taint label to one contiguous subregion of the input file starting at a random offset. Note that while the results reported in the Neon study focus on the case of sparsely-tainted input files, we are also interested in understanding the worst-case performance impact of taint tracking. To this end, we repeat the experiment multiple times, varying the length of the tainted subregion (F) between 0% and 100%.

Figure 5.1 reports the results of this experiment, presenting a comparison between Neon and PIFT with asynchronous tracking and 512MB of memory reserved for the taint argument log. The overhead is expressed in terms of the slowdown factor relative to the native Linux configuration (*NL*). Looking at these results, *PIFT-A(512)* increases the running time by 5.3$\times$ for file copy and 5.8$\times$ for file compression on lightly-tainted input files ($F = 10\%$). As the amount of tainted data grows, our system must spend more time in the emulated mode and the slowdown becomes more noticeable. In the extreme case of a fully-tainted input file, our implementation incurs slowdowns of 15.9$\times$ and 21.1$\times$ for copying and compression, respectively. The copy operation involves no computation on tainted data — it merely transfers file data between user- and kernel-level memory buffers

	NL	PIFT-S	PIFT-A(512)
LocalCopy	11.0	16.4	15.8
Compress	62.0	96.5	92.8

Table 5.1. Command completion time (in ms) for LocalCopy and Compress with $F = \frac{1}{64}$.

using the `repz movsd` instruction. The page-level taint transfer optimization described in Section 4.2.2 allows the taint processor to handle this scenario efficiently. File compression using `gzip` represents a somewhat more stressful scenario and its algorithmic components, Huffman coding and LZ77, produce a nontrivial amount of computational activity, which PIFT must carefully analyze.

This experiment exercises the ability of PIFT to transition efficiently between virtualized and emulated execution modes. Ideally, one would expect the slowdown to scale linearly with the length of the tainted file region, since the amount of taint should dictate the amount of time spent in the high-overhead mode. Our system does not exhibit fully linear scaling because the heuristics for transitioning are not well tuned in our current prototype. These heuristics err on the conservative side, keeping the system in the emulated mode even if one could have transitioned back to native execution a bit earlier.

Although we do not have enough data to draw definitive conclusions due to fundamental system differences and limitations in available data, we believe that these results are promising and yield a favorable comparison to Neon. In a scenario with $F = \frac{1}{64}$, the only data point available to us for comparison, Neon reports slowdown factors of $10\times$ and $95\times$ for file copy and compression, respectively[1]. As Table 5.1 shows, the overhead in PIFT with this amount of tainted data is only $1.5\times$ over native execution for both operations.

Although PIFT and Neon are quite similar in terms of the overall architecture, two fundamental aspects of our design tilt the results in our favor. First, PIFT tracks the flow of tainted data at a higher level of abstraction (guest x86 instructions, as opposed to QEMU micro-instructions). Second, PIFT leverages asynchrony by explicitly separating the taint tracking computation from the main emulation workload and executing these tasks concurrently on two processor cores.

5.1.2 Text Search

In the next experiment, we consider another common operation — text search. Our input dataset is a 100-MB sample of the Enron corporate e-mail database [29] spread across 100 equal-sized files and all files reside on disk at the start of the experiment. In the PIFT configuration, we also mark a fraction (F) of the files as sensitive, assigning them

[1]The Neon paper [98] presents the results of this experiment in Table 3. Note, however, that the textual summary of the experiment provided in the accompanying text in Section 5.2.1 reports different levels of overhead, which are inconsistent with the numeric results. The authors confirmed to us that the values in Table 3 are the correct reference results, and not the ones provided in the text.

	NL	*PVL*	*Emul*	*PIFT- S*	*PIFT- A(512)*
Completion time (s)	2.42	2.87	18.45	58.87	25.57
Slowdown relative to *NL*	1.00×	1.19×	7.62×	24.33×	10.57×

Table 5.2. Command completion time (in seconds) for a text search task with fully-tainted inputs ($F = 100\%$).

unique uniform taint labels. We use the GNU `grep` command [38] to search this sample for a single-word string and measure the running time. We repeat the measurement thrice with different search keywords, compute the average running time in all configurations of interest, and report the performance in terms of the slowdown factor relative to *NL*.

`grep` represents a somewhat more complex and diverse workload that includes disk reads, a text search computation based on the Tuned Boyer-Moore pattern matching algorithm [3], and transmission of output (lines of text that contain the search keyword) to the paravirtualized console. Note that since PIFT does not monitor and intercept the externalization of tainted data through the local console, this experiment does not incur the costs of suspending the emulator to synchronize the taint label state in the asynchronous configuration.

Table 5.2 reports the results of this experiment in the most stressful scenario, where all files in the input dataset are tainted ($F = 100\%$). We observe that our implementation imposes a noticeable performance penalty in this worst-case configuration — a factor of 10.6× with asynchronous parallelized taint tracking and 512MB of memory reserved for the taint argument log. Still, the slowdown is much lower than in the previous set of experiments, since the workload is not fully CPU-bound. The cost of a pattern matching computation (even including the associated overheads of emulation and information flow analysis) fades in comparison to the latency cost of an I/O request to fetch input data from disk and the latter is constant across all configurations. With a more modest amount of tainted data ($F = 10\%$), the slowdown is even less noticeable — only 1.46× relative to native execution with *PIFT-A(512)*.

Table 5.2 also helps us quantify the performance gains achieved by parallelizing the IFT computation and executing it asynchronously. The results indicate that `grep` runs $24.3\times$ slower with synchronous taint tracking on the same processor core and tracking asynchronously on another core reduces the running time by a factor of 2.3×.

5.2 Benefits and Limitations of Asynchrony

The results in the previous section provide evidence for our hypothesis that asynchronous tracking can greatly improve performance, offering the ability to move the IFT computation out of the critical path and preventing it from slowing down the execution of the emulated machine. Of course, the size of the taint argument log is a crucial parame-

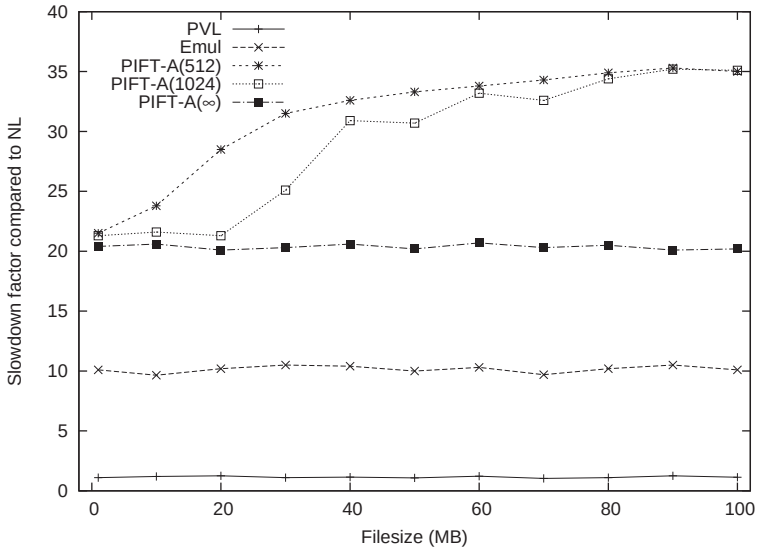
ter, which largely determines the degree of improvement we can obtain by tracking asynchronously. In Section 5.1.1, a 512MB log provided sufficient space to absorb the entire taint tracking computation for both operations (copy and compress) and thus, the impact of information flow analysis on their completion times was minimal. Conversely, a small log would make it difficult to absorb long bursts of computation, requiring the emulator to stall and wait for the taint processor thread to make progress and release space. In situations, where PIFT is routinely tasked with analyzing large CPU-bound workloads, the "log full" condition can become prevalent and effectively cause the system to transition back to the synchronous mode, in which the emulator and the taint processor operate in lockstep.

In the next experiment, we examine this transition phenomenon and evaluate PIFT's sensitivity to the size of the taint argument log. We focus on the absolute worst-case scenario — a CPU-bound workload operating on fully-tainted input files that have been pre-staged into memory buffers. We consider two such workloads: compressing a sensitive file using `gzip` and sorting an array of integers using the `qsort` library routine. In this experiment, we measure the running time for varying input sizes and, in PIFT configurations, several different sizes of the taint argument log. We also evaluate a special *PIFT-A(∞)* configuration, in which the taint processor implementation is modified to consume taint tracking blocks instantaneously, without executing the instructions. As a result, the main emulation thread never stalls due to lack of log space in this configuration. While *PIFT-A(∞)* does not usefully track the propagation of taint labels, it enables us to establish the absolute upper bound on the degree of performance improvement attainable by optimizing the taint tracker. Viewed from a different angle, the performance difference between *PIFT-A(∞)* and *Emul* quantifies the overhead our implementation adds to the critical emulation codepath on the producer side. This overhead is associated with computing intermediate values (such as physical memory addresses) and communicating them to the consumer via the taint argument log.

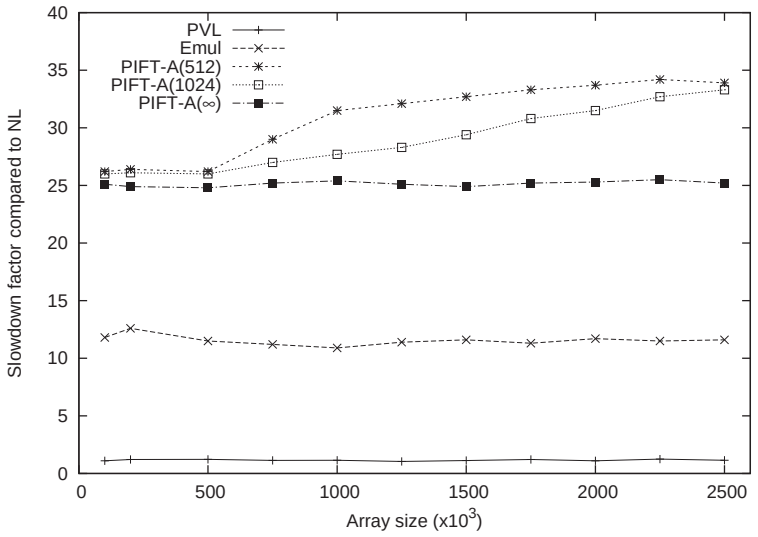
Figure 5.2 presents the results of this experiment, plotting the slowdown relative to *NL* for the compression (a) and sort (b) workloads. *PVL* shows the baseline performance on unmodified Xen and reflects the overhead of paravirtualization. *Emul* isolates the impact of basic emulation and, as we can see, both workloads suffer a slowdown of 10-12$\times$.

Turning our attention to the performance of PIFT, compression runs $22.4\times$ slower on *PIFT-A(1024)* and 22.5$\times$ slower on *PIFT-A(512)* for a 1MB input file. As we expect, these numbers start to diverge as input size increases. With a 20MB input file, *PIFT-A(1024)* still offers sufficient log space to absorb most of the overhead and thus, computation proceeds at a rate close to the upper bound given by *PIFT-A(∞)*. An input file of size 20MB appears to be the point of transition for *PIFT-A(1024)*, beyond which the producer starts stalling on log space for non-negligible periods of time. In the *PIFT-A(512)* configuration, this transition occurs around the 5MB mark according to our estimates. We can see that in both configurations, there is an asymptotic penalty as we increase the input size, plateauing at a $35\times$ slowdown. Viewed collectively, these results validate our intuition that *PIFT-A* can take advantage of underutilized memory resources to alleviate the computational burden of taint tracking.

The results from the integer sort benchmark reveal slightly higher levels of baseline



(a) File compression benchmark



(b) Integer sort benchmark

Figure 5.2. Performance on worst-case CPU-bound computational tasks in all configurations of interest.

emulation overhead, but are not qualitatively different. The taint tracking log provides enough space to absorb the costs of sorting up to 500000 integers and we see that performance starts to degrade beyond this point in both PIFT configurations. Still, we observe that our implementation can gainfully exploit additional memory resources to improve performance. Looking at the costs of sorting 1000000 array entries, PIFT configured with a 512MB log suffers a slowdown of 31.5×, while *PIFT-A(1024)* achieves a slowdown of

27.7×. The worst-case performance penalty, incurred for large input sizes that cause most of the taint tracking computation to be done synchronously, is around 34× in both configurations.

Unfortunately, there does not exist much data on how previous dynamic taint tracking systems behave at this level of stress, which makes it difficult to provide a direct comparison. We hope that the following data points, which we were able to collect from the literature, can help initiate such a comparison:

- Neon [98] reports overheads ranging from 10× to 95× for CPU-bound workloads when $1/64^{th}$ of the input file is tainted.
- The initial implementation of taint tracking using on-demand emulation with Xen [41] reports a 155× slowdown for what appears to be a CPU-bound task operating continuously on tainted data.

5.3 Interactivity and User Productivity in Graphical Environments

While performance on isolated CPU-intensive tasks allows us to assess the fundamental computational costs of taint tracking, our system aims to provide a general-purpose IFT substrate suitable for pervasive deployment in enterprise environments. Today, most enterprise users interact with computers through a graphical interface and routinely rely on large and complex applications, such as spreadsheets and word processors, to generate and manipulate data. Can PIFT deliver the level of interactivity and performance users have come to expect from these sophisticated graphical application stacks running on modern hardware? In this section, we evaluate the performance of our prototype in a graphical environment with the goal of understanding how our IFT primitives affect user experience and productivity on common tasks in widely-used applications.

Interactive graphical environments present a set of additional challenges for dynamic taint tracking systems such as PIFT. Although such environments rarely impose high computational demands and typical workloads tend to be interrupt-driven, we found that the task of rendering tainted data on the screen sometimes leads to undesirable oscillation between native and emulated modes.

To understand the nature of this problem, consider a basic scenario, where a user is editing a sensitive document in a word processor and entering text from the keyboard. Each keystroke triggers an interrupt that propagates through the layers of the GUI stack and eventually causes the text area widget to repaint itself. The widget's screen image is a rectangular array of pixels, whose values are computationally derived from tainted textual data that corresponds to the visible region of the document. Thus, the protected VM must access tainted data in the course of repainting the window, triggering a page fault and a

transition to the emulated mode. When it finishes computing the new window contents (reflecting the keystroke) and relinquishes the CPU, we switch back to the native mode, but find ourselves re-entering emulation once again upon the next keystroke. This behavior easily leads to thrashing and significantly impairs interactivity, which leads us to conclude that the on-demand emulation technique, as presented in Chapter 3, may not be directly applicable to interactive graphical environments. We found that undesirable oscillation can be avoided and the overall usability of the system can be greatly improved with a simple workaround: *persistently* switching to the emulated mode of execution and remaining in this mode for as long as tainted data remains on the screen.

Keeping the system persistently in emulation also enables us to leverage significant benefits from asynchronous parallelized tracking. In fact, interactive graphical environments seem to be a highly compelling use case for the asynchronous mode. Since the guest workload is interrupt-driven and proceeds mostly at human timescales, the taint processor can easily keep up with the producer and the log helps absorb the short bursts of computation that emerge as result of user activity.

Persistent emulation with asynchronous taint tracking led us to a fully-operational and usable graphical environment. In this environment, users observe minimal or no perceivable degradation of interactivity for simple UI actions, such as moving the mouse pointer and scrolling. In this section, we evaluate PIFT's performance on these and longer, more complicated, user activities. First, we evaluate how PIFT affects the start-up times of several large and widely-used interactive applications. Then, in order to understand and quantify PIFT's impact on user experience and productivity, we examine how our system affects users' performance on two specific interactive tasks: typing text into a word processor and editing a spreadsheet document.

Our benchmark machine for interactive graphical tests is a Lenovo H320 with an Intel Core i3 540 processor and 6GB of RAM. As before, the guest environment is configured with 512MB of physical memory and one VCPU. The guest runs Fedora Core Linux with kernel version 2.6.18-8 and the graphical interface is provided through the GNOME desktop environment. To prevent the abovementioned thrashing behavior, we disabled PIFT's on-demand emulation facility and the tests took place entirely in the emulated mode. In the asynchronous parallelized PIFT configuration, the taint argument log size is fixed at 1024MB.

To the best of our knowledge, PIFT is the first online taint tracking system to demonstrate support for interactive workloads in a graphical desktop environment and hence, we are unable to directly compare our substrate to previous work. When exploring performance on realistic user tasks, we compare PIFT's results to those achievable in an unmodified emulated environment (*Emul*) and in a native Linux environment operating on bare hardware (*NL*).

Graphical application launch: In the first batch of experiments, we measured the time it took to launch an application, render its graphical components, load a document from disk, and then close the program. The programs we measured were Abiword [1] (an open-source

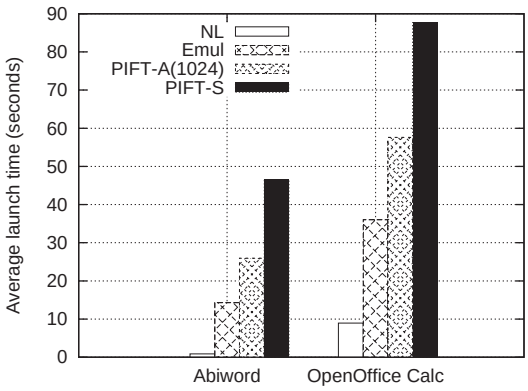


Figure 5.3. Application launch time for Abiword and OOCalc in all configurations of interest.

	Abiword		OOCalc	
Configuration	Slowdown rel. to *NL*	Slowdown rel. to *Emul*	Slowdown rel. to *NL*	Slowdown rel. to *Emul*
PIFT-A(1024)	29.51×	1.82×	6.43×	1.60×
PIFT-S	52.80×	3.25×	9.79×	2.44×

Table 5.3. Slowdown relative to *NL* and *Emul* in the application launch experiment.

word processor) and OpenOffice Calc [65] (the spreadsheet component of the OpenOffice.org productivity suite [67]). All tests were performed with cold filesystem caches and the time measurements were taken with the help of a stopwatch.

Figure 5.3 plots the average completion time of the application launch task in all configurations of interest and Table 5.3 reports the slowdown incurred by PIFT relative to *Emul* and *NL*. Looking at these results, we note that slowdowns are generally moderate and that asynchronous parallelized tracking offers significant savings. Abiword launch appears to be the more stressful scenario for our system, suffering a 29.5× slowdown with *PIFT-A(1024)*. However, the slowdown relative to the purely emulated configuration is only 1.8×, indicating that much of the overhead is attributable to the costs of basic emulation. In the OpenOffice Calc (OOCalc) experiment, the overall slowdown is even less noticeable and we attribute this difference to the fact that OOCalc has a larger codebase and a more complex set of external dependencies. Launching OOCalc requires loading a large number of shared libraries and other external components and, as a result, launch time is dominated by the costs of disk I/O, which are roughly equal in all configurations.

Entering text: In the next experiment, we evaluated the user-perceived slowdown on one of the most common user activities — entering text from the keyboard. We used `vncplay` [96, 95] to record the entry of a long passage of text into OpenOffice Writer [66] (the word processor component of the OpenOffice.org productivity suite [67]). The log

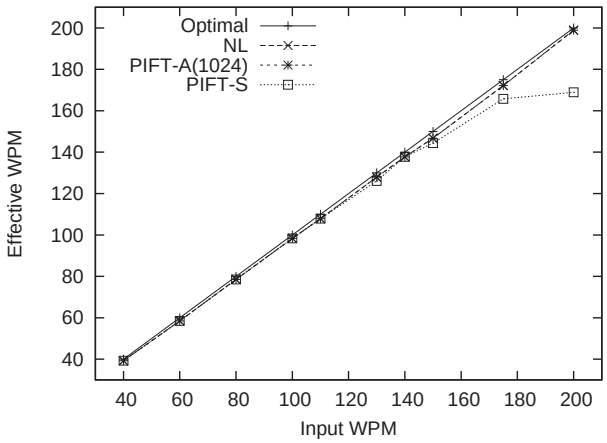


Figure 5.4. User-perceptible overhead in the text entry experiment.

contained a total of 749 distinct keystroke events and included backspaces, which corrected typing mistakes. We then replayed the log in all configurations of interest, adjusting the delay between successive keystrokes to simulate typing speeds ranging from 40 words per minute (WPM) to 200 WPM. During replay, the passage was "typed" into the beginning of a document that already contained text and, in both PIFT configurations, was tainted with a non-zero uniform label. We measured the time for the entire passage to appear on the screen and used this measurement to calculate the *effective WPM*.

Figure 5.4 reports the results of this test, plotting the effective WPM as a function of the input WPM. In the optimal scenario, these two quantities are exactly equal, which means that the system can "keep up" with the user at any typing speed. The difference between these values indicates the magnitude of user-perceived overhead incurred by the system and estimates the loss of productivity on typing.

Our results suggest that both PIFT configurations can sustain near-optimal performance at "normal" typing speeds up to 140WPM. Beyond this threshold, *PIFT-S* suffers a fairly dramatic performance drop-off, achieving only 84% of the optimal speed at 200WPM. On the other hand, the asynchronous parallelized PIFT configuration imposes no such overhead and can render text without observable delay even at extreme typing speeds. In both PIFT configurations, we noticed a short "warm-up" period, wherein keystrokes were rendered slowly for 2-3 seconds before catching up and matching the input speed imperceptibly.

Editing a spreadsheet: In our final experiment, we asked a human user to launch OpenOffice Calc and perform a series of simple spreadsheet editing tasks. They included opening a document, formatting cells, typing referential formulas, copying cells, and navigating the spreadsheet using both mouse and keyboard. We then measured the time that it took the user to complete these tasks in each configuration using a stopwatch. Compared to the keyboard text entry experiment, this test produced a more diverse and somewhat more stressful interactive workload, since it involved significant mouse movement, as well as switching between mouse and keyboard. Several tasks required the user to activate and

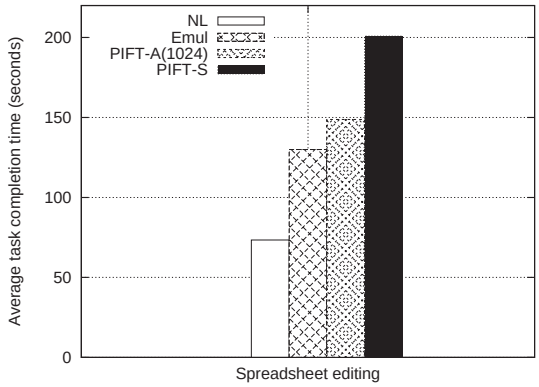


Figure 5.5. Time taken to complete the spreadsheet editing task in all configurations of interest.

Configuration	Slowdown rel. to *NL*	Slowdown rel. to *Emul*
PIFT-A(1024)	2.03×	1.14×
PIFT-S	2.74×	1.54×

Table 5.4. Slowdown relative to *NL* and *Emul* in the spreadsheet editing experiment.

interact with dialog boxes (for example, to modify cell border appearance) or to navigate file menus.

Figure 5.5 plots the task completion times in this experiment and Table 5.4 reports the slowdown incurred by both PIFT configurations. In the native Linux configuration (*NL*), the editing session took roughly 74 seconds and it took the user 201 seconds (or 2.7× longer) to complete the session in the *PIFT-S* configuration. As before, asynchronous parallelized tracking offered significant improvements, reducing the completion time to 149 seconds.

Looking at these results, it is essential to note that the dominant component of the overhead in *PIFT-A(1024)* is associated with the costs of basic emulation and our IFT extensions add only 14% to these baseline costs. This results is remarkable, as it suggests that with our performance improvements, information flow analysis ceases to be the principal performance bottleneck, at least as far as its impact on user productivity is concerned, and does not present a serious obstacle to usability. This observation leads us to believe that the next major performance improvement will come from optimizing the core mechanisms of emulation and dynamic code translation within QEMU.

Finally, we note that this test demanded agility from the user, requiring him to complete a series of mechanical tasks as quickly as possible, and did not allow for “think time”. In more realistic scenarios, users tend to pause between their actions and this factor would further reduce the observable performance impact.

5.4 Evaluation Summary

This chapter examined PIFT's performance characteristics in a variety of contexts ranging from CPU-intensive microbenchmarks to interactive applications in a graphical desktop environment. We believe that our overall results are encouraging and while there clearly remains room for further improvement, these results demonstrate the effectiveness of our performance optimization techniques, which are among the core technical contributions of this dissertation.

It was difficult for us to provide direct comparisons with previous work due to fundamental design differences and limited availability of performance data for previous systems. In the two specific cases where we could do so, our implementation achieved a slowdown of roughly 1.5× and demonstrated a major improvement over previous efforts, in which the two comparison cases suffered slowdowns of roughly one and two orders of magnitude, respectively. PIFT's performance advancements are attributable to a combination of high-level information flow instrumentation and asynchronous parallelized execution of taint tracking.

The copy and compression experiments presented in Section 5.1.1 exercise PIFT's ability to effectively switch between native unmonitored execution on the host CPU and emulated execution. Ideally, on linear workloads one would expect the slowdown to scale linearly with the amount of tainted data, since the latter dictates the amount of time spent in the high-overhead emulated mode. The current prototype does not achieve fully linear scaling, since our mode switching heuristics keep the protected VM in emulation a bit longer than strictly necessary in an effort to keep the aggregate costs of context switching at a manageable level and prevent thrashing. Under worst-case conditions, where the entire input file is marked as tainted, PIFT incurs slowdowns of roughly 16× and 21× on copying and compression tasks, respectively.

The text search experiment described in Section 5.1.2 demonstrates that the slowdown is even less noticeable for workloads that mix computational activity with disk I/O. With a fully-tainted input dataset, `grep` runs 11 times slower in a PIFT-managed environment with asynchronous parallelized tracking. The ability to track information flow asynchronously on a dedicated CPU core reduces the observable slowdown by a factor of 2.3.

While our results unambiguously demonstrate the utility of asynchronous tracking, it would be a mistake to view asynchrony as a panacea for taint tracking performance challenges. It merely provides the ability to buffer a burst of taint tracking activity and execute it opportunistically, so as to minimize the impact on the critical path of emulation. As expected, the degree of improvement is largely determined by the amount of memory allocated for the taint argument log. The results of Section 5.2 suggest that log sizes of 1024MB and higher tend to work well in practice and deliver substantial gains on non-trivial workloads. Exploring log compression mechanisms and other techniques that could reduce PIFT's memory consumption would be an interesting direction for future work.

To the best of our knowledge, PIFT is the first dynamic taint analysis platform to demonstrate support for interactive workloads in a graphical desktop environment. Al-

though the presence of tainted data on the screen forces the system to stay in emulation for extended periods of time, PIFT succeeds in masking the overhead through asynchrony. For usage scenarios that impose a minimal computational load, such as entering text from the keyboard, the usability impact is imperceptible. For more intensive operations, such as editing a sensitive spreadsheet document, it takes a user two times longer on average to complete the desired set of tasks in a PIFT-managed environment. We have not quite reached the point, where the user-perceptible overhead of dynamic taint analysis and its impact on productivity can be dismissed as negligible, but we believe that PIFT represents a substantial step towards this goal. As evidenced by the results of Section 5.3, most of the remaining overhead can be attributed to the fundamental costs of QEMU-based emulation and our system is well-positioned to take advantage of further improvements in emulation technology — an orthogonal, but important direction for future work.

Chapter 6

Correctness of Taint Label Propagation

In the previous chapter, we evaluated the runtime overhead of PIFT and assessed the effectiveness of our performance optimizations using a combination of microbenchmarks and simulated usage scenarios. Although this dissertation focuses predominantly on addressing the issues of runtime performance, we must also examine a separate, but equally important question: does PIFT offer an *effective* tool for confining the flow of sensitive information and do our mechanisms track its propagation *correctly*? After all, any effort aimed at reducing the runtime performance costs would be an exercise in futility if the resulting system does not track the flow of sensitive data in a manner that properly reflects users' actions, avoiding loss of sensitivity status and overtainting.

The question of effectiveness is a difficult one and a central challenge lies in expressing the criterion of correctness in precise terms. It is crucial to note that in this context, correctness is fundamentally a subjective and user-centric notion — our IFT platform behaves correctly if and only if it tracks the propagation of taint labels and enforces policies in a way that is consistent with users' intentions and expectations.

Although an in-depth investigation of taint propagation correctness is beyond the scope of this dissertation and would be an appropriate topic for a follow-on study, this chapter presents some of our preliminary findings in this area. These findings have emerged from our initial experimentation with the PIFT prototype and are based on a range of straightforward usage scenarios, for which an intuitive and unambiguous definition of correct behavior is easy to identify.

In general terms, these initial explorations have yielded mixed results. While fully correct behavior was observed in a subset of scenarios, we have also found evidence of problematic taint propagation dynamics, including drastic over-tainting of user information and taint poisoning of control data structures within the OS kernel. These alarming findings lead us to suggest that the current implementation of PIFT's information flow analysis mechanisms may not be directly applicable to certain classes of applications and user interfaces. These instances of false tainting must be studied and appropriate countermeasures

must be developed before fine-grained (byte- and instruction-level) information flow analysis can become truly practical. We hope that the initial set of results, which we present in this chapter, can serve as a starting point for such a study and help focus subsequent efforts.

Section 6.1 presents the key results from our study of taint propagation, providing representative examples of both correct and problematic behavior. The latter suggest that the hypervisor-based approach to IFT faces several fundamental limitations, which we try to articulate in Section 6.2. Finally, in Section 6.3, we sketch a methodology for reducing or altogether eliminating taint explosion in legacy codebases. As a case study, we demonstrate how this methodology can be applied to address the taint explosion problem in the Linux kernel.

6.1 Experimental Results

In this section, we present a representative set of results from our taint propagation experiments. All of these tests employ the "black box" methodology: we start with an input dataset that includes a combination of tainted (sensitive) and non-tainted data, execute a series of high-level application-specific data manipulation tasks, and then examine the output to determine whether its components are tainted in a manner consistent with our expectations.

(1) Command-line data manipulation tools: In this experiment, we used a command-line toolchain to execute a series of transformations on a text file. These transformations included copying the input file (using the `cp` command from GNU coreutils [18]), searching the resulting copy for a keyword using GNU `grep` [38], sorting the output alphabetically using the `sort` command from GNU coreutils, and finally compressing the output using `gzip` [40]. The initial input file (`F.txt`) contained 1MB of English-language text and we created two additional copies of this file (`Fs.txt` and `Fns.txt`), representing sensitive and public data, respectively. `Fs.txt` was assigned a uniform non-empty label and `Fns.txt` was tainted with $L_\emptyset$. After assigning taint labels, we executed the following sequence of commands in the protected VM operating on top of the PIFT platform:

```
(1) cp Fns.txt F2ns.txt
(2) grep 'the' F2ns.txt | sort | gzip > F3ns.txt; sync
(3) cp Fs.txt F2s.txt
(4) grep 'the' F2s.txt | sort | gzip > F3s.txt; sync
(5) cp Fns.txt F4ns.txt
(6) grep 'the' F4ns.txt | sort | gzip > F5ns.txt; sync
```

Note that commands `(3)` and `(4)` operate on sensitive inputs and, as a result, we expect PIFT to propagate the non-empty taint label into the corresponding output files (`F2s.txt` and `F3s.txt`). Conversely, commands `(1)`, `(2)`, `(5)`, and `(6)` operate on the public version of the input file and we thus expect their outputs (`F2ns.txt`, `F3ns.txt`, `F4ns.txt`, and `F5ns.txt`) to carry the empty taint label.

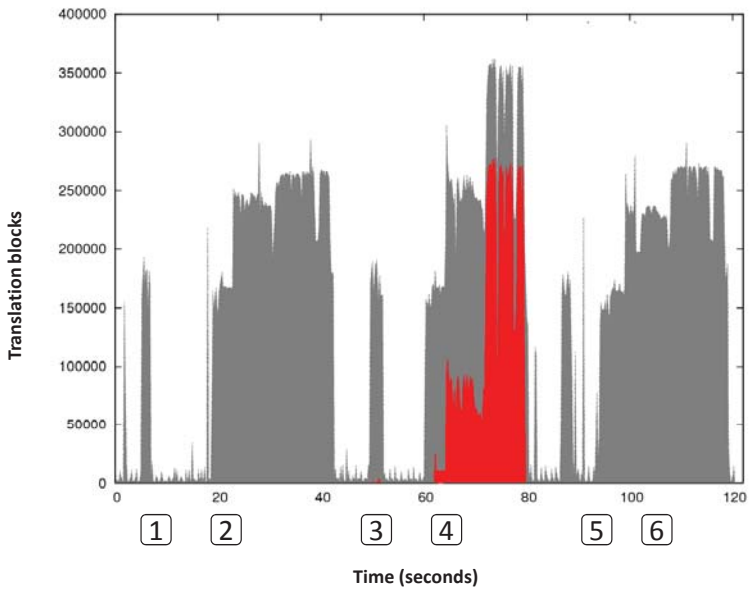


Figure 6.1. A time series showing the level of computational activity within the protected VM in Experiment 1. The highlighted overlay illustrates the number of basic blocks that touch at least one tainted operand.

After executing the above sequence of commands, we inspected the taint status of the output files and verified that the labels were propagated according to our expectations: `F2ns.txt`, `F3ns.txt`, `F4ns.txt`, and `F5ns.txt` were tainted uniformly with $L_\emptyset$, whereas `F2s.txt` and `F3s.txt` carried a uniform non-empty taint label that matched the label of the initial sensitive input file `Fs.txt`. These results lead us to conclude that PIFT exhibits correct taint propagation behavior in this particular scenario.

Figure 6.1 offers additional evidence to support our claim of correctness, providing a more fine-grained view of label propagation dynamics in this experiment. We repeated the experiment in a fully-emulated environment and instrumented QEMU to track the total number of basic code blocks executed by the guest VM, as well as the number of blocks that touch at least one tainted operand in a CPU register or a memory location. The figure plots these values as a time series: each data point corresponds to a 100ms time interval and the vertical axis shows the number of basic code blocks executed in each interval. Looking at this figure, we note that commands `(1)` and `(2)` produce substantial computational activity, but do not touch tainted data, as expected. Commands `(3)` and `(4)` perform operations on the tainted copy of the input file and the time series reveals that, as expected, a significant fraction of the corresponding code blocks manipulates tainted data values. Interestingly, the spike of activity representing the file copy operation `(3)` shows that only a minuscule fraction of the corresponding code blocks accesses tainted data and the cause of this counterintuitive behavior becomes apparent once we consider how this operation is implemented. Copying a file on x86-based Linux involves a series of data transfers between user- and kernel-level data buffers and modern implementations rely on the `repz movsd`

instruction to perform these transfers efficiently. These heavyweight instructions transfer tainted file data in 32KB-sized chunks and are the only instructions to touch tainted data during the execution of command `(3)`. Finally, commands `(5)` and `(6)` repeat the sequence of operations on the non-tainted version of the file. The time series reveals no traces of taint manipulation activity, indicating that the taint status of the affected memory areas and CPU registers has been properly reset.

(2) Compiling a tainted source code tree: In our second experiment, we attempted to compile PostgreSQL [69] (an open-source database management system) from source code using the standard GNU toolchain. As in the previous experiment, we created two separate copies of the source code tree: `Ts` (in which each file is labeled uniformly with a non-empty taint label) and `Tns` (in which each file is labeled with $L_\emptyset$). We then executed `make` within the tainted tree to compile the database server. We confirmed that the resulting executable was tainted with the correct label, but also observed that the system failed to return from the emulated mode upon the completion of compilation. Further investigation revealed a dramatic taint explosion scenario within the Linux kernel, which caused numerous control data structures in the kernel address space to pick up taint. As we explain in Section 6.3.1, this behavior is initially triggered by the propagation of taint through system call arguments. For instance, supplying a tainted filename string (derived from the contents of a tainted makefile) as an argument to the `sys_open` system call deposits taint labels into the kernel and eventually causes a number of `dentry` structures on various kernel-level lists to become tainted. Subsequently, any other user-space process that interacts with these kernel data structures becomes tainted as well. Most alarmingly, subsequent attempts to compile from `Tns` (the non-tainted version of the source code tree) produce tainted executables.

(3) Editing text using a word processor: In our final test, we experimented with Abiword [1] (an open-source graphical text editor) running in a GNOME-based interactive desktop environment. We began by applying a uniform non-empty taint label to a small text document (`Ds.txt`) and then launched the Abiword editor. We opened the tainted document in Abiword, then immediately closed it (without modifying or saving its contents), and issued a command to create an empty new document. We then entered several lines of text from the keyboard into this newly-created document, saved it under a different filename, and closed the Abiword application window. We inspected the taint status of the newly-created file and found that it carries the taint label assigned to `Ds.txt` — an unexpected result, considering that no data has been explicitly transferred between these two documents. We conjecture that opening a tainted file in the first step of this experiment caused taint labels to propagate from `Ds.txt` into Abiword's shared internal data structures and from there poison the new document. Further investigation revealed that taint labels propagated into several other files, including:

```
(1) /home/user/.AbiSuite/AbiWord.Profile
(2) /home/user/.AbiSuite/AbiCollab.Profile
(3) /home/user/.config/gtk-2.0/gtkfilechooser.ini
(4) /home/user/.recently-used.xbel
```

Files `(1)` and `(2)` maintain user-specific profile data for Abiword, while `(3)` and `(4)` store a list of recently-used files for the "file open" dialog box provided by the GTK+ library.

The observed behavior is clearly unusual, as we would not expect the taint status of these auxiliary files to be influenced by the simple act of opening a tainted user document. Further experimentation revealed that once tainted data has entered Abiword through this action, all subsequent "file save" operations within Abiword produce tainted output files and this behavior persists across machine reboots. These unexpected results lead us to suggest that fine-grained information flow analysis tools such as PIFT do not yet offer natural support for complex applications such as Abiword and attaining a practical solution will require further investigation into the causes of this taint poisoning phenomenon.

6.2 Fundamental Limitations

While a comprehensive evaluation of taint propagation behavior would be an appropriate topic for a separate in-depth study, we believe that our preliminary results paint a useful picture and can help understand some of the fundamental limitations associated with our approach.

On a fundamental level, interposing at the software-hardware boundary using a hypervisor allows us to maintain full compatibility with legacy software stacks, while retaining complete control over the protected VM and its interactions with external entities. This point of interposition happens to be highly advantageous for the purposes of IFT, as it allows us to track data flow *transparently* and *comprehensively* across application- and OS-level contexts.

At the same time, the capabilities of a hypervisor-level solution are inherently restricted by a *semantic gap* between application-specific data units and actions on the one hand and the low-level architectural state of the underlying hardware platform on the other. In modern interactive computing environments, users manipulate information on the basis of abstract human-centric entities and data types (*e.g.*, pages and paragraphs in a text document or cells in a spreadsheet). The hypervisor is oblivious to these concepts and observes users' actions as sequences of machine instructions that operate on the contents of physical memory and CPU registers, but carry no inherent meaning beyond describing specific low-level bit manipulation tasks.

The Problem of Over-Tainting

This semantic gap can make it challenging for the hypervisor to track the propagation of sensitive data *effectively*, in a manner that meaningfully captures users' intended actions and agrees with their expectations. Observing nothing other than a stream of low-level machine instructions, the hypervisor is fundamentally incapable of meaningfully differentiating between explicit information transfers (such as cutting and pasting text between

documents) and accidental ones (such as tainting the kernel's control data structures). The latter can arise as unintended side-effects of explicit data manipulation activities and our initial results suggest that such cases do, in fact, arise in a variety of common scenarios. Left unchecked, these accidental data flows can reduce the fidelity of PIFT's information flow tracking primitives and, in extreme cases, render them completely ineffective through taint explosion.

Lacking exposure to higher-level data types and application-specific knowledge, the hypervisor must rely on the layers above it — applications, supporting libraries, and the operating system — to confine intra-VM information flow to a set of well-defined channels and avoid accidental contamination of control data structures. The results of our initial studies suggest that some applications are not well-behaved in this sense and thus cannot directly leverage our taint tracking infrastructure. In order to attain compatibility with PIFT, these applications must be restructured or otherwise modified to manipulate data in a more orderly fashion.

Note that PIFT's exposure to the semantic gap is a fundamental consequence of our key design choices. An OS-level solution would have been inherently more disruptive to the protected software stack, but would have allowed us to track data propagation on the basis of OS-level primitives (such as files, sockets, and processes) and easily protect the kernel from taint poisoning. Yet another alternative would be to focus on applications written in a specific type-safe language, such as Java or Python, and implement tracking mechanisms within a managed language runtime, as in RESIN [92]. This approach would provide an even better view of how applications manipulate information, allowing the tracking substrate to monitor the propagation of labels on the basis of language-level variables and, at least in principle, reliably differentiate between intended and accidental propagation of taint. However, this approach forfeits the generality of a hypervisor-level design and requires additional mechanisms to track the flow of data outside the boundaries of a single application process.

It is also important to note that the taint explosion phenomenon can be viewed as a direct manifestation of *label escalation* (or *label creep*) — a general problem that has been observed in nearly all previous DIFC-based systems. Label creep refers to the notion that once a variable's data value has been tagged as "sensitive", this tag must be retained throughout the execution to ensure confidentiality, while reverse actions (clearing labels by overwriting sensitive values with non-sensitive constants) tend to occur infrequently. As the computation proceeds, label merge operations produce new labels with increasingly restrictive policies, coercing them in the direction of confidentiality. As a result, an application's internal state experiences a monotonic and continuous increase in sensitivity and information becomes increasingly difficult to externalize.

The threats and implications of label escalation were first observed decades ago in Denning's pioneering work [22] and, to this date, have not been fully resolved. These issues could be among the key reasons why multi-level information flow control has not yet reached widespread acceptance. Today, most DIFC-enabled systems address label escalation through controlled declassification, in effect providing application developers with an explicit "escape hatch" from strict information flow control rules. Declassification enables

downward label movement, allowing applications to release sensitive information through careful relabeling of data values. While necessary in practice, declassification is difficult to do correctly, makes it challenging to reason about the information flow behavior, and can be abused for malicious purposes.

In PIFT, label escalation manifests itself as over-tainting and the semantic gap arguably exacerbates the problem. As we explained above, the hypervisor's inability to differentiate between explicit and accidental data transfers can easily lead to the contamination of control data structures with taint. Once the initial contamination occurs, the sensitivity status spreads and moves upwards due to label creep, leading to full-scale taint explosion. It can be noted that the solution we propose in Section 6.3 to address kernel-level taint explosion is essentially a form of explicit controlled declassification.

The Challenges of Debugging Information Flow

Another fundamental issue resulting from the semantic gap is the difficulty of tracking causal dependencies between taint propagation events, as well as *explaining* anomalous behavior such as contamination of control variables. The PIFT platform monitors the execution at the level of machine instructions and the current design does not provide mechanisms for mapping the low-level architectural state (registers and memory words) onto higher-level constructs (variables and data structures). In the absence of such mappings, it is immensely difficult to "debug" information flow problems and explain unusual behavior, such as the incidents of taint poisoning described in the previous section.

On the one hand, a linear instruction-level trace of execution fully captures the taint propagation activity within the guest environment and, in principle, provides sufficient information for a variety of useful studies. In practice, techniques that are based on tracing the execution at the instruction level tend to produce very large amounts of data, making it difficult to capture, store, and subsequently analyze the interesting fragments. As an anecdotal example, after observing unexpected taint propagation activity in Scenario 3 (editing tainted documents in Abiword), we attempted to analyze the behavior and identify the likely causes of taint explosion. We used our QEMU-based tracing infrastructure to capture an instruction-level trace of taint propagation activity resulting from a short sequence of user actions; namely, invoking the "file open" command, selecting a tainted plain-text document from the file dialog box, and clicking the "open" button to load the selected document into the text editor. This elementary sequence of actions produced over 15GB of linear instruction-level execution traces and our attempts at identifying the sources of taint explosion were hindered by the size and complexity of these resulting traces, even after we have excluded the "non-interesting" segments that record null-to-null label propagation. At such scale, execution traces are clearly not amenable to manual inspection and fine-grained data flow analysis is computationally onerous.

These experiences lead us to suggest that instruction-level tracing may not be the most appropriate tool for studying the correctness properties of dynamic IFT systems and we believe that developing more effective analysis, introspection, and debugging tools is an

important direction for further research. Of particular value would be tools that can help in identifying the complete causal paths between interesting IFT events (such as specific files on disk getting tainted) and the initial access to tainted data. To facilitate analysis, such tools must provide cross-layer visibility and allow researchers to correlate interesting IFT behavior with guest machine instructions and the associated source code.

6.3 Eliminating Taint Explosion

Given the mixed results from the taint propagation study, one could be inclined to question the main hypothesis of this dissertation and ask whether PIFT, or perhaps hypervisor-based taint analysis systems in general, are indeed suitable for the role of a comprehensive and reliable data confinement platform. In particular, one could conclude that, lacking the ability to reliably differentiate between explicit information transfers and accidental poisoning, hypervisors are simply not the right tools for the task at hand. Perhaps they are limited to supporting a narrow class of well-behaved applications, which propagate sensitive information according to carefully-defined data flow rules, but cannot reliably track information flow in arbitrary executables.

Although our work merely scratches the surface of the problem and invites further research on the accuracy of taint tracking, we believe that categorical and pessimistic conclusions of this nature would be imprudent. To see why, we must consider the fact that few, if any, of the widely-deployed legacy applications were designed to function in a DIFC-enabled environment. While information flow control has a rich history in the research community and its principles are fairly well-understood, IFC techniques have not yet, at the time of writing, reached widespread adoption within the broader computing community. In particular, the principles of information flow control are not widely known within the industry of consumer software development and do not form a standard component of the general computer science curriculum. As a consequence, programmers rarely take information flow considerations into account when developing applications and systems software stacks. The problem is exacerbated by modern compilers, which, in an effort to squeeze out every last bit of performance, routinely transform code in a manner that causes sensitive information spillage and taint poisoning, even in well-structured applications that carefully confine information flow at the source code level. In light of these issues, it may be unreasonable to expect unmodified legacy application stacks to cooperate and be fully compatible with our low-level application-independent IFT infrastructure.

We believe, however, that the limitations we observed when applying PIFT to legacy applications can be eliminated by identifying the channels of information leakage and closing them via a limited number of simple and localized source code transformations that do not affect high-level program behavior. The general methodology for finding and eliminating the culprits of overtainting would involve first identifying the set of high-level containers, which are used by the application as explicit storage vessels for sensitive user data, and then examining all channels of information exchange between these containers. To make

this discussion somewhat more concrete, in a typical object-oriented application information containers would likely be represented by object classes that encapsulate user data, while the channels would correspond to the public functional interfaces between these classes. These interfaces can be then examined and classified into *explicit information transfer channels* and *spillage channels* according to their high-level taint dissemination effects.

Having identified the set of spillage channels, we can mitigate or altogether eliminate their undesirable taint propagation effects by removing ("*scrubbing*") the taint label at the destination endpoint. This can be done in a variety of ways and our current implementation facilitates this step by exposing a new hypercall (`__HYPERVISOR_scrub_taint`). Applications can invoke this hypercall to explicitly label a region of their virtual address space as non-sensitive, causing PIFT-Xen to replace the corresponding memory taint labels with $L_{\emptyset}$. Note that while this approach requires modifying and recompiling the application codebase, these modifications are minimal and non-intrusive in their nature. Crucially, these changes are limited to inserting a set of hypercall invocations and do not affect the program's data flow, control flow, or the overall application logic in any manner. An alternative approach would be to trap the execution of functions that transfer data across spillage channels within PIFT and automatically clear the taint labels at the destination endpoint without explicit signaling from the application. This approach is applicable in cases where compatibility with existing binaries is an absolute requirement, but is non-portable and involves pushing a certain amount of application- and/or OS-specific information (such as the addresses and signatures of functions that constitute spillage channels) into the hypervisor.

How much work would be involved in identifying and fixing all the spillage channels that currently exist in large and widely-deployed legacy programs? After all, the proposed methodology for eliminating such information flow leaks demands a non-trivial amount of manual analysis and requires expert understanding of a program's structure and data flow properties. Might it not be more prudent to discard the existing implementations and rebuild these programs from the grounds up, espousing the principles of information flow control and carefully restricting the dissemination of sensitive data values to avoid overtainting?

We conjecture that even in very large and complex legacy codebases, the main culprits of taint poisoning and explosion are represented by a small number of easily-detectable spillage channels that can be intercepted and securely scrubbed. As an initial step in assessing this conjecture, we undertook a systematic study of taint explosion in one of the most complex, yet universally deployed components of the software stack — the operating system kernel. Since our prototype implementation of PIFT was tested and evaluated with a paravirtualized (PV) Linux guest environment, we chose to focus our investigation on the PV Linux kernel version 2.6.8-18, although we expect our results to be fully applicable to other recent versions of Linux. Focusing on execution scenarios that result in severe kernel-level taint explosion, we found that the causes of initial kernel contamination can be traced to two specific spillage channels situated at the user-kernel boundary. By invoking `__HYPERVISOR_scrub_taint` at these two locations (which required adding 110 lines of new code to the standard Linux kernel), we were able to close these information flow gaps and eliminate kernel taint explosion for all practical purposes.

6.3.1 Case Study: Eliminating Taint Explosion in the Linux Kernel

One of the most common and problematic cases of taint explosion involves accidental propagation of taint labels into the internal data structures of the OS kernel and our early experimentation with the system revealed that the Linux kernel is highly susceptible to this phenomenon.

Recall from Section 6.1 that in one of our initial experiments we attempted to compile an executable from a tainted C-language source code tree using the standard GNU toolchain. In that experiment, we observed a dramatic taint explosion within the kernel, which was triggered by the propagation of tainted data through system call arguments. As an illustrative example, supplying a tainted filename string (derived from the contents of a tainted makefile) as an argument to the `open` system call deposits taint into the kernel and eventually causes a number of `dentry` structures on various kernel-level lists to acquire taint [1]. Subsequently, any other user-space process that interacted with the kernel would get tainted from these data structures and cause taint to propagate further along the execution path. Upon the completion of compilation, PIFT failed to return the protected VM from the emulated mode and subsequent attempts to build from non-tainted source code produced tainted binaries.

One way of approaching the problem of kernel taint explosion is by observing that for the purposes of information flow tracking, the propagation of taint from the application address space into kernel-level memory can *almost in all cases* be viewed as anomalous and undesirable behavior. In a general-purpose operating system such as Linux and Windows, the kernel provides a general-purpose substrate that exposes a specific set of system services through a well-defined and relatively narrow interface defined by system calls. By design, these services are *application-independent* and are expected to function without any knowledge of application-level data semantics. It follows that transfers of taint labels from an application-level address space to kernel-level control data structures should not occur under normal operating conditions. As a result, any information channel that permits tainted information to cross the user-kernel boundary and flow into kernel-level data structures can be viewed as a spillage channel and treated as such.

If we take this view, we can tackle the problem by identifying these accidental channels of taint propagation and closing them, while retaining the legitimate channels of information transfer the kernel is expected to provide as part of the system call contract. To localize the codepaths in the kernel where such accidental tainting happens, we undertook a systematic empirical study of taint propagation for a Linux-based guest environment.

Identifying and Closing Spillage Channels

We began by capturing an instruction-level trace of a usage session, which included the initial stages of the compilation experiment and resulted in severe kernel-level taint poisoning. Using this trace, we constructed a data flow graph illustrating the execution

[1] A detailed explanation of this taint poisoning scenario can be found in an earlier study [80].

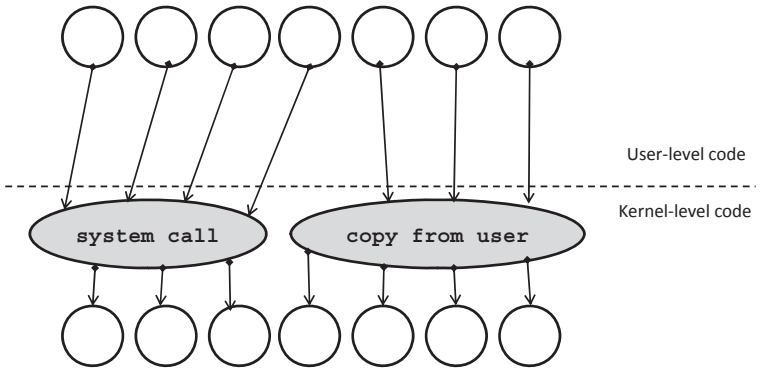


Figure 6.2. A schematic depiction of the data flow graph that emerged from our empirical study of kernel taint explosion. The shaded nodes depict the two kernel-level functions that are responsible for the initial taint poisoning.

paths that resulted in kernel taint poisoning. In this graph, nodes correspond to individual machine instructions and there is an edge from I_1 to I_2 if I_1 writes data to a memory location that is subsequently read by I_2.

This data flow graph helped us understand the general properties of execution paths that deposited tainted application data into kernel-level memory areas. While the data flow graph cannot be presented here in complete detail due to its size and complexity, Figure 6.2 depicts this graph in schematic form, which highlights our most significant finding. Specifically, we found that in all execution paths that deposit taint into the Linux kernel, only two kernel-level code blocks serve as entry points and enable the initial transfer of tainted data from application-level memory.

In the first case, the transfer occurs when the system call entry routine (`system_call` in `linux/arch/i386/kernel/entry-xen.S`) writes the user-level CPU register values (some of them holding system call arguments) to the kernel-level stack and the spread of taint starts when the system call handler fetches these arguments from the stack. In the second case, the transfer happens via the `copy_from_user` routine and its variants. The kernel invokes this function to fetch additional arbitrary-length system call parameters from application-level memory buffers and this action can cause transfers of taint into kernel-level stack and heap areas.

While the presence of these taint poisoning channels is not at all surprising, the interesting fact that emerged from our analysis is that no other channels exist. As a result, we can apply the methodology introduced above and easily stop kernel-level taint poisoning by closing these two spillage channels. To close the first channel, we modify the `system_call` entry routine and invoke the taint-scrubbing hypercall in the very first steps of system call handling, instructing PIFT-Xen to clear the taint status of system call arguments once they have been transferred to the kernel stack. Note that since we do not wish to change the application-level taint propagation behavior, we must preserve the original user-level register taint labels and restore them upon return from the system call handler. We accomplish

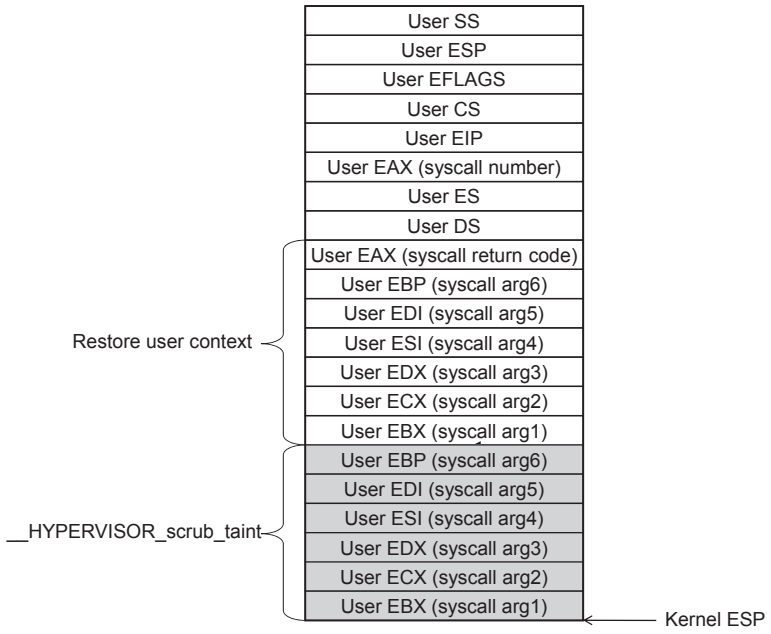


Figure 6.3. The contents of the kernel-level stack upon system call entry in a paravirtualized Linux guest environment with our taint scrubbing modifications.

this by storing a second copy of the user-level register context on the kernel-level stack, as illustrated in Figure 6.3. This second copy is scrubbed and subsequently accessed by the kernel-level code to obtain the system call arguments, while the first (tainted) copy is used to restore the application-level registers upon the return from the kernel. Using similar techniques, we modify the `copy_from_user` routine (and its variations) to scrub the destination kernel-level memory area once tainted data has been copied from a user-space buffer.

Identifying Legitimate Channels of Information Transfer

In practice, most operating systems provide a number of explicit and well-defined information transfer channels, which enable applications to communicate and store data. While the system call scrubbing technique described above offers an effective defense mechanism against accidental kernel taint contamination, we must, of course, ensure that any mechanism we adopt to address this issue does not interfere with these legitimate information channels. We performed an in-depth inspection of the Linux system call interface and identified the following *explicit* channels of information transfer from application-level memory to the kernel address space:

1. **The `write` system call (and its variations):** Like most modern OSes, Linux implements a kernel-level file block caching facility to speed up file access. The system call handler for `sys_write` transfers file data from an application-space buffer into

this kernel-level cache by invoking the `copy_from_user` routine. In this situation, we must avoid scrubbing the kernel-level copy in order to ensure that taint labels get preserved across file I/O operations.

2. **The `send` system call (and its variations):** Analogously, the handler for `sys_send` may invoke `copy_from_user` to transfer application data into kernel-level socket buffers and the taint labels must be propagated accordingly.

3. **Inter-process communication (IPC) system calls**: The Linux kernel provides temporary storage for user-level data buffers to support message-based IPC.

In each of the above cases, we modified the relevant codepaths in the Linux kernel to invoke an alternate version of `copy_from_user`, which does not scrub the kernel-level memory buffer.

Encouragingly, these modifications allowed us to eliminate all symptoms of kernel taint explosion and prevent subsequent inter-process taint poisoning. To test our solution, we re-executed the compilation experiment described above. We invoked 10 iterations of `make`, alternating between tainted and non-tainted copies of the source code tree, and verified that the resulting binaries are tainted in the expected manner. We then ran a series of other control experiments, which included copying, compressing, searching, and editing text using the `emacs` editor, and confirmed that taint labels do not leak into kernel-level memory areas of the protected VM.

Discussion

In summary, our approach to the problem of kernel taint poisoning involves identifying and closing all taint spillage channels, while making exceptions for a small number of legitimate information channels that the kernel explicitly exposes through the system call interface. Admittedly, our solution rests on the assumption that the user-kernel interface is relatively narrow and can be manually audited to identify the set of legitimate channels. Our approach also assumes that applications are "well-behaved" in the sense that they always confine explicit transfers of sensitive user data to these legitimate channels.

Of course, the second assumption may not always hold and it is quite easy to come up with a counterexample. Consider a scenario, in which two processes communicate tainted information by reading and writing the contents of a directory file: P_1 generates empty files and encodes tainted data into their filenames, while P_2 fetches this data by issuing `readdir` system calls. This is an example of an *implicit* information flow that would not be captured by our taint tracking substrate. While theoretically possible, we expect that such scenarios will rarely, if ever, arise in practice in the absence of malicious activity. In other words, we expect that in the vast majority of cases, applications will communicate data through the three explicit channels we have identified: file operations, socket operations, and message-based IPC.

	`no-op`	`stat`	`fork`
PIFT-PVL (without scrubbing)	0.248	0.952	242.21
PIFT-PVL (with scrubbing)	0.557	1.593	243.00

Table 6.1. Latency (in μs) for several system calls in a paravirtualized Linux guest environment with and without taint scrubbing.

Analogously, it is difficult to argue that kernel-level control data structures will never get contaminated through the channels we chose to keep open. In some scenarios, kernel-level code may need to examine and perform simple computations on tainted application data residing in a kernel-level cache, as in the case of filesystems that compute block-level checksums or hashes before writing data blocks to disk. While it is conceivable that a certain sequence of operations could trigger kernel-level taint explosion originating from a tainted checksum value, we found no evidence of such activity in the course of extensive experimentation with our Linux-based prototype.

Finally, one could note that our techniques are ineffective in the presence of malicious or compromised applications that want to circumvent the information flow tracking mechanisms. In particular, malicious code could exploit our kernel taint scrubbing functions to create an exfiltration channel for sensitive data. We respond by noting that PIFT focuses on tracking the flow of information in a benign environment, while protecting the integrity of taint labels in the presence of malware is an explicit non-goal. Even without our kernel scrubbing modifications, malicious software intent on stealing sensitive information has a myriad other options, including *implicit flow channels* and *covert channels*. Detecting and preventing data exfiltration through these channels is technically difficult for all existing information flow tracking systems and we do not attempt to close these gaps with PIFT.

Performance Overhead of Scrubbing

To see if our explosion elimination measures impaired performance, we used LMBench [51] to measure the latencies of three distinct system calls: `no-op`, `stat`, and `fork`. We performed these measurements in our PIFT-enabled paravirtualized Linux configuration both with and without the taint scrubbing extensions. Table 6.1 reports the results of this comparison. We see that for the `no-op` system call, the cost of an additional hypercall represents a significant (factor of $2.2\times$) penalty, but this overhead is much less noticeable for non-trivial system calls.

Chapter 7

Summary and Conclusions

In this dissertation, we presented Practical Information Flow Tracking (PIFT) — a novel information security architecture for enterprise environments that monitors the flow of sensitive data and restricts its dissemination by enforcing high-level security policies. In contrast to most previous efforts in this area, PIFT seeks to achieve full binary-level compatibility with existing software stacks, including widely-deployed legacy applications and operating systems.

Our exploration begins with the intuition that a hypervisor — a thin layer of virtualization software that can be interposed between the OS kernel and the hardware layer — can serve as a robust foundational building block for a comprehensive information security platform. Hypervisors are, by and large, compatible with existing software stacks, yet provide sufficient control over the execution of the guest VM and can easily intercept its interactions with external entities. Our hope was that by extending an off-the-shelf hypervisor implementation with dynamic taint analysis and policy enforcement capabilities, we could attain a robust security platform that tracks information flow and enforces end-to-end policies in a comprehensive manner.

Dynamic taint analysis is inherently a computationally-intensive task and its broad adoption has been hindered, in part, by the enormous levels of runtime overhead. Previous implementations of byte- and instruction-level taint tracking incur slowdowns of up to $20\times$ (for user-level code) and $100\times$ (for full-system emulation). While perfectly adequate for the contexts, in which these systems were proposed, such levels of overhead make previous implementations unsuitable for our purposes, i.e., real-time monitoring of information flow in interactive user-facing applications.

This dissertation proposes two novel algorithmic techniques, whose collective effects allow us to reduce the runtime performance penalty to a much more manageable level. First, we track information flow at the level of native machine instructions, without first decomposing them into emulator-specific bytecodes. While this technique requires a much more significant up-front engineering effort, it also enables a range of low-level optimiza-

tions that are difficult or altogether impossible to apply at the bytecode level. Second, we observe that emulation and fine-grained information flow tracking can be viewed as two separate and, for the most part, mutually independent computations. As a result, PIFT can improve performance by batching and delaying the processing of taint label updates and handling them "opportunistically" in an asynchronous manner. On multiprocessor CPUs, our system attains further overhead reductions by offloading the taint tracking computation to a separate processor core, allowing it to proceed asynchronously and in parallel with the main execution stream of emulated instructions.

The combination of these two optimizations yields a significant reduction in runtime overhead and allows PIFT to advance the state of the art in taint tracking performance, in some cases achieving an order-of-magnitude improvement over the best previously-published results [98]. To the best of our knowledge, PIFT is the most efficient implementation of whole-system byte-level taint tracking available at the time of writing and is the only system to demonstrate a fully-usable interactive graphical environment. While the overhead on stressful CPU-bound microbenchmarks is still relatively high (up to $35\times$ in worse-case scenarios), we were encouraged by the results of our usability study, which indicated that the user-perceivable delays and PIFT's overall impact on user productivity are much less noticeable (no more than $2\times$). Notably, most of the remaining overhead is attributable to the fundamental performance costs of emulation using QEMU, while the additional slowdown due to taint analysis is almost negligible.

Our techniques and results are, of course, not without limitations. A hypervisor-based architecture is inherently limited by a semantic gap between high-level context (OS- and application-level data constructs) and the low-level state of the virtual machine (processor registers and memory locations). This mismatch makes it challenging for PIFT to differentiate between intended transfers of sensitive information and accidental ones, the latter of which oftentimes arise as unintended side-effects.

Our empirical study of taint propagation behavior in legacy Linux applications revealed an alarming number of false tainting scenarios, whereby taint labels were superfluously propagated from sensitive data files into various shared application- and OS-level data structures. This behavior leads to overtainting and causes unrelated data files to get contaminated with sensitivity policies. In some cases, it leads to full-scale taint explosion — a degenerate state, in which nearly all components of the VM memory image acquire non-empty taint labels. In this state, the system erroneously prevents users from externalizing *any* data, regardless of the original policy specification and at that point, information flow analysis ceases to be useful or meaningful.

These results lead us to conclude that machine-level IFT primitives, such as the ones we explore in this dissertation, are not fully compatible with off-the-shelf legacy application binaries and further research into the dynamics of taint propagation is needed before the vision of a robust and transparent data security platform, as outlined in the introductory chapter, can be fully realized. As we explain in Chapter 6, these limitations are partly an implication of the semantic gap, which renders the hypervisor incapable of differentiating explicit information transfers from accidental "spilling", and partly a consequence of the

fact that today's application developers are largely unaware of IFT and rarely take proper precautions to confine sensitive data flow *within* an application to appropriate channels.

We believe that the current instantiation of PIFT can still serve as a useful tool for tracking information flow in legacy software, but in light of these limitations, a certain amount of application-level restructuring or modification appears inevitable. To assist software developers with this task, we have sketched a methodology for identifying the root causes of taint label spillage through source code analysis. Once these channels have been identified, they can be closed by issuing an explicit taint scrubbing request to PIFT or, alternatively, augmenting the hypervisor with a certain amount of application-specific logic that enables it to recognize transfers of information across these channels and scrub them automatically, without involving the application. We conjecture that in most legacy software systems that are susceptible to taint explosion, the initial contamination of non-sensitive state occurs through a small number of leakage channels, which can be identified and closed without excessive manual effort. As a feasibility study, we investigated the problem of taint contamination and subsequent explosion within the Linux kernel. By applying our methodology, we were able to identify and close all existing channels of kernel-level contamination, which required modifying only two kernel source files and adding 110 lines of new code. While these initial results are clearly encouraging, we hasten to note that they are of preliminary nature and further investigation is needed to assess whether the proposed methodology generalizes to other classes of legacy software.

Taking a broader perspective, the aim of this dissertation was not to present dynamic taint analysis as a panacea for information security problems within an enterprise, but rather to explore and evaluate a novel hypervisor-based security architecture that comprehensively tracks information flow and enforces end-to-end policies that are beyond the capabilities of today's DAC-based security mechanisms. By doing so, we hoped to assess the potential of hypervisors in the domain of information security and develop understanding of their unique strengths in this domain, as well as their limitations.

As a final point for discussion, we return to our original hypothesis — that a hypervisor augmented with byte-level taint analysis primitives can serve as a foundation for a robust and practical information flow tracking platform, which provides the above capabilities. At this juncture, it is natural to ask: how well did this hypothesis hold?

The non-intrusive nature of hypervisors and the promise of full compatibility with existing application binaries are, quite clearly, a major advantage and allow us to interpose IFT functionality in a fully-transparent manner. Further, while the computational overhead of dynamic taint analysis has been a major obstacle for all previous IFT implementations, this dissertation contributes several novel optimization techniques, which greatly reduce the runtime performance costs. On-demand emulation, coupled with asynchronous parallelized taint tracking, allow PIFT to reduce the performance impact to a point, where the runtime overhead is no longer a significant practical concern.

On the other hand, the lack of visibility into higher-level data constructs turns out to be a major practical limitation for hypervisor-based systems such as PIFT and, in some scenarios, affects their ability to track the dissemination of sensitive data correctly. There

remains more work to be done in understanding the causes of taint explosion, as well as mitigating its effects, and we hope that subsequent research efforts will help overcome these remaining hurdles.

In the final analysis, we believe that this dissertation contributes to a better understanding of hypervisors from the angle of enterprise security and makes a strong argument that hypervisor-driven IFT systems hold a significant promise. We also believe that such systems will become fully practical and ready for broad adoption once the issues of taint explosion are fully understood and resolved.

7.1 Directions for Future Work

In some sense, the core technical contribution of this dissertation is very narrow in scope, focusing predominantly on improving the runtime performance of dynamic taint analysis. Performance overhead was a major stumbling block for all previous dynamic IFT proposals and has contributed to a widespread perception that dynamic taint analysis is simply too slow and, for this reason, fundamentally unsuitable for real-time tracking. Our work has demonstrated that these pessimistic notions are unfounded. A detailed analysis of the existing bottlenecks, combined with several algorithmic advancements and careful engineering, has enabled us to achieve substantial overhead reductions. The end result of our efforts is a new and highly-efficient IFT implementation, which proves that fine-grained (byte- and instruction-level) taint analysis can be performed in real-time on commodity hardware with negligible impact on the user experience.

At the same time, it is abundantly clear that runtime performance is not the *only* major obstacle on the path to broad acceptance and other challenges must be overcome before dynamic IFT systems (and the design principles they embody) can be considered truly practical. This dissertation leaves many questions unanswered, but lays the groundwork for further investigation and we plan to continue using our PIFT prototype as a platform for taint tracking research. We conclude this dissertation by outlining what we believe to be the most important and promising directions for further inquiry.

7.1.1 Bridging the Semantic Gap

We believe that addressing the implications of the semantic gap between application-level constructs and VM-level state is among the most critical directions for future work, and one that will likely determine the practical viability of systems such as PIFT. As we have shown in this dissertation, state-of-the-art hypervisor-level IFT implementations do not always track the flow of taint labels accurately in legacy software stacks and exhibit a tendency to overtaint. Understanding the root causes of taint explosion in existing applications is an essential direction for future work. This will require developing an array of robust tracing, introspection, and data flow analysis tools, which would enable researchers

to study the dynamics of taint label propagation and correlate instances of unusual activity with application-level events. Once the dominant causes of taint poisoning and explosion are sufficiently well understood, the next step would be to identify effective techniques for eliminating taint explosion or mitigating its effects and the scrubbing technique introduced in Chapter 6 might serve as a useful starting point. At that stage, developing a set of meaningful quantitative metrics for evaluating the efficacy of such techniques and formulating a precise definition of *taint propagation accuracy* would be highly beneficial.

Another significant issue resulting from the semantic gap, and one that we largely sidestep in this dissertation, is the question of initial data labeling. Human users transact at the level of documents, e-mail messages, text paragraphs, and spreadsheet columns, yet, as we discussed, these notions remain largely invisible to PIFT. Conversely, the hypervisor tracks information flow and assigns policy labels on the basis of machine registers and memory addresses, but these low-level primitives are hardly meaningful to a typical desktop application user. Given this mismatch, how would a user indicate to the hypervisor that a specific column in her spreadsheet contains sensitive data and should be labeled with policies? Developing a robust data labeling mechanism, which does not require significant changes to the application layer and does not drastically alter the user experience, is another essential step on the path towards practicality.

7.1.2 Further Performance Improvements

While progress has been made on improving the runtime performance of taint tracking, our work leaves numerous opportunities for further optimization and we believe that careful analysis of the remaining bottlenecks, coupled with diligent engineering, can lead to further breakthroughs.

Our experimental results demonstrate that a significant fraction of the remaining overhead (around 50%) is attributable to the baseline costs of emulated execution in the current version of QEMU and improving the efficiency of core emulation mechanisms represents a clear "low-hanging fruit". While QEMU's just-in-time (JIT) code translation mechanisms are relatively efficient and offer a big improvement over plain binary interpretation, they do not employ any of the advanced dynamic compilation and optimization techniques found in today's state-of-the-art VM runtime environments, such as the Java HotSpot Server Compiler [68]. HotSpot translates from architecture-independent Java bytecode and achieves near-native performance on commodity hardware, employing an array of advanced compiler optimization techniques [17, 84]. Retrofitting some of these optimizations into QEMU may greatly improve performance for all applications of emulation, including PIFT.

Going further, the computational overhead of taint analysis could be reduced by JIT-compiling the taint tracking instruction handlers. Recall from Chapter 4 that our current implementation handles these instructions by repeatedly invoking pre-compiled instruction-specific handler routines. While straightforward to implement, this approach adds the overhead of a function call to each taint tracking instruction and precludes macroscopic code optimizations that span multiple instruction handlers. Instead, we could aggregate taint

tracking instructions into larger blocks and JIT-compile them to native code for the host CPU. During this step, we could apply register allocation and dead code elimination optimizations in hopes of reducing the amount of computational work for the taint processor.

Parallelized execution is another interesting direction for future work, and one that holds numerous promises and challenges. The current design of PIFT utilizes two processor cores, providing the ability to execute the taint tracking instruction stream asynchronously and in parallel to the main emulated context. This represents a clear step forward, but might it be possible to achieve further speedups by parallelizing across a larger number of cores? In the current design, the actual execution of taint tracking instruction handlers proceeds sequentially, mirroring the instruction stream in the emulated CPU, but the ordering of instructions within a basic block is determined by data dependencies and is only a partial ordering. Would it be possible to identify sets of mutually-independent guest CPU instructions at the time of code analysis and translation, and partition the output taint tracking code block into several independent sub-blocks, which could be executed in parallel on a multi-core host processor. We believe that the answer is affirmative — data dependency tracking is a routine activity in compiler design and techniques for identifying independent instruction schedules are readily available. A less obvious question is whether these parallelization opportunities can be leveraged in practice to achieve speedups and one of the challenges lies in overcoming the overhead of synchronization.

Finally, it may be beneficial to explore the potential of specialized hardware extensions for information flow analysis. Recall that although compatibility with unmodified hardware platforms was an explicit objective for our current implementation, at the core of our proposal lies a new processor architecture and a new ISA specification for a specialized taint analysis unit. The current implementation of PIFT emulates its functionality in software to achieve compatibility with existing platforms, but a true hardware-based taint processor implementation could yield significant performance gains. While instantiating the full taint processor design in hardware would be, without a doubt, an extremely ambitious project, it is likely that a significant fraction of the gains could be realized by implementing only a modestly-sized subset of the desired features and retrofitting them as incremental extensions. As a specific example, consider the `PageTaintSummary` lookup procedure described in Section 4.2.5, which translates physical page numbers into concise page taint descriptor summaries using a two-level nested table. As it happens, this lookup procedure is highly analogous to a standard page table lookup — an operation that translates from virtual to physical page numbers and is typically handled by the hardware memory management unit (MMU). The two-level nested structure illustrated in Figure 4.10 bears a strong resemblance to a standard page table and the auxiliary cache of `PageTaintSummary` structures is a direct analogue of a TLB. Hence, adding a new MMU instance for efficient handling of `PageTaintSummary` lookups — an incremental and relatively straightforward extension — could help us improve performance by eliminating 2-4 software-based memory accesses per taint tracking instruction.

7.1.3 Other Uses of Dynamic Taint Analysis

While tracking the flow of sensitive user data through third-party legacy applications represents the primary focus of our efforts in this dissertation, we believe that PIFT can be applied to a broader range of problems. Exploring additional use cases for the core set of taint analysis technologies we have developed in the context of PIFT can a fruitful avenue for further investigation. As a specific example, which emerged from discussions with our industry partners, it would be interesting to explore the application of fine-grained IFT techniques to the development and functional verification of large-scale Web applications.

Internet companies such as Google, Facebook, Amazon, eBay, and others rely on the ability to store and manipulate large amounts of information about their users and customers to drive their core business. These companies implement their services through an array of sophisticated internally-developed applications and typically expose these services to their users through a Web-based interface, while storing sensitive user-specific data in a back-end database tier. Social networking sites such as Facebook collect and store detailed information about their users' interests and activities, as well as detailed histories of communication with other users. Analogously, e-commerce sites such as Amazon face the burden of securely storing and accessing users' personal and financial data, including credit card numbers, billing addresses, product preferences, and detailed order histories. Ensuring the safety of users' private data and preventing its unauthorized disclosure to third parties is a major concern for these companies, and one whose importance will only continue to grow with time. In these scenarios, bugs in application-level logic, misconfigurations, or imprecisely-specified SQL queries can trigger catastrophic violations of privacy policies, such as when a user's credit card information gets mistakenly revealed through the "account settings" page to another user.

We believe that fine-grained information flow analysis tools such as PIFT can be of great value in this setting by providing an auditing tool or an early warning system for policy violations. One particularly interesting usage scenario would be to employ PIFT as a "data flow debugger" during application development and testing to obtain insight into the application's information flow behavior and flag potential violations of policy. An example policy might state that a user's credit card data may be externalized from the front-end Web tier only through an authenticated HTTP session that bears the credentials of the same user. During testing, developers would load the database with a collection of synthetically-generated user profiles, including credit card information, and taint them with desired policies using PIFT. Later, they would exercise the application using synthetic request workloads and observe how sensitive profile data disseminates throughout the components of the system. The client-facing module in the Web tier could be instrumented to query the taint labels in outgoing data packets and verify whether they are consistent with the active privacy policies.

This use case is different from the central scenario of enterprise data confinement in several crucial respects. First of all, the information flow tracking substrate would be deployed in a development and testing environment with synthetic user data, as opposed to a live production setting with real users. We expect that in this environment, the slowdown

resulting from emulation and information flow tracking would not be perceived as a major practical challenge, since it is well-understood that debugging and program analysis tools inevitably introduce a certain amount of overhead.

Further, we believe that adopting a tool such as PIFT and applying it in a disciplined manner would enable developers to study the application's information flow properties from the earliest stages of the development process and easily prevent the appearance of leakage channels. As our study has shown, leakage channels naturally lead to taint poisoning and explosion, but detecting and closing these channels after the fact in large applications can be highly challenging. On the other hand, applying IFT tools *during* the development of an application would help eliminate leakage channels as they appear and ensure that the end results of information flow analysis are fully accurate and meaningful.

Finally, we conjecture that the computational workloads associated with today's Web applications are inherently more amenable to instruction-level IFT and less susceptible to taint poisoning than legacy desktop applications, such as the ones we evaluated in Chapter 6. While Internet companies such as eBay and Amazon routinely store large volumes of potentially sensitive user data (such as names, addresses, and credit card information), their applications rarely require the server-side component to manipulate or perform substantial computation on such data. In most cases, the computational task involves no more than basic string construction and copying — the application fetches the relevant user data fields from the database, converts them to string format, and combines these text strings into a larger string that represents the entire output page, interspersing them with HTML tags and non-sensitive data values. We hypothesize that due to the constrained nature of computational workloads, accidental taint propagation and explosion will be less prevalent in this scenario.

The above factors highlight some of the important distinctions between the two use cases of fine-grained byte-level IFT: tracking information flow in legacy enterprise applications (the central focus of this dissertation) and debugging data flow in Web apps. It would be interesting to fully explore the applicability and limitations of PIFT in the latter scenario and we have already taken several exploratory steps in this direction.

7.2 Final Remarks

Securing the flow of sensitive data in a distributed environment is a challenging problem, and one whose importance will only continue to grow as our society increases its reliance on computerized information storage and processing systems. This dissertation lays the groundwork for a new class of information security architectures, which utilize an augmented hypervisor to track the flow of data in a comprehensive and transparent manner. PIFT is a concrete instantiation of the general design principles and represents the core technical contribution of this dissertation. In the course of developing and evaluating PIFT, we have gained valuable insights regarding the unique strengths of hypervisors in the context of information security, as well as an understanding of what goals and tradeoffs are not

achievable with our approach. Our successes to date suggest that PIFT offers a set of powerful security primitives, which strike a meaningful balance between comprehensiveness, deployability, and performance.

While fine-grained information flow tracking has traditionally been considered a heavyweight and expensive tool suitable only for offline analysis, we have demonstrated that online real-time tracking on commodity hardware is within the realm of being practical. We also hope that our work will help expose the principles of dynamic taint analysis more broadly to the application developer community and that our results will help guide further explorations of this topic.

Bibliography

[1] AbiWord. `http://www.abisource.com`.

[2] Kent Anderson. Dysfunctional operations in IT. *Information Systems Control Journal*, 1, February 2008.

[3] Hume Andrew and Daniel Sunday. Fast string searching. *Software - Practice and Experience*, 21:1221–1248, November 1991.

[4] Android. `http://www.android.com`.

[5] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, July 2004.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Symposium on Operating Systems Principles 2003 (SOSP '03)*, pages 164–177, New York, NY, USA, October 2003. ACM.

[7] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference: FREENIX Track*, pages 41–46, Berkeley, CA, USA, October 2005. USENIX Association.

[8] Kenneth J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, The Mitre Corporation, April 1977.

[9] William J. Bolosky and Michael L. Scott. False sharing and its effect on shared memory. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 57–71, Berkeley, CA, USA, September 1993. USENIX Association.

[10] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*, pages 621–634, New York, NY, USA, November 2009. ACM.

[11] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings*

of the 14th ACM Conference on Computer and Communications Security (CCS '07), pages 317–329, New York, NY, USA, October 2007. ACM.

[12] Brent Callaghan, Brian Pawlowski, and Peter Staubach. RFC 1813 - NFS version 3 protocol specification, June 1995. `http://www.ietf.org/rfc/rfc1813`.

[13] Haibo Chen, Xi Wu, Liwei Yuan, Binyu Zang, Pen-chung Yew, and Frederic T. Chong. From speculation to security: Practical and efficient information flow tracking using speculative hardware. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*, pages 401–412, Washington, DC, USA, June 2008. IEEE Computer Society.

[14] Shimin Chen, Babak Falsafi, Phillip B. Gibbons, Michael Kozuch, Todd C. Mowry, Radu Teodorescu, Anastassia Ailamaki, Limor Fix, Gregory R. Ganger, Bin Lin, and Steven W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID '06)*, pages 63–65, New York, NY, USA, October 2006. ACM.

[15] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*, pages 377–388, Washington, DC, USA, June 2008. IEEE Computer Society.

[16] Christopher Clark, Keir Fraser, Steven H, Jakob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, pages 273–286, Berkeley, CA, USA, May 2005. USENIX Association.

[17] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17:181–196, March 1995.

[18] coreutils - GNU core utilities. `http://www.gnu.org/software/coreutils`.

[19] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual volume 2A: Instruction set reference, A-M. `http://www.intel.com/Assets/PDF/manual/253666.pdf`.

[20] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 221–232, Washington, DC, USA, December 2004. IEEE Computer Society.

[21] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*, pages 482–493, New York, NY, USA, June 2007. ACM.

[22] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[23] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20:504–513, July 1977.

[24] *Department of defense trusted computer system evaluation criteria*. DOD 52800.28-STD (the Orange book) edition, December 1985.

[25] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 17–30, New York, NY, USA, October 2005. ACM.

[26] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In *Proceedings of the 2007 USENIX Annual Technical Conference (ATC '07)*, pages 1–14, Berkeley, CA, USA, June 2007. USENIX Association.

[27] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*, pages 1–6, Berkeley, CA, USA, October 2010. USENIX Association.

[28] Paul England and John Manferdelli. Virtual machines for enterprise desktop security. *Information Security Tech. Report*, 11(4):193–202, January 2006.

[29] Enron email dataset. `http://www.cs.cmu.edu/~enron`.

[30] Andrey Ermolinskiy. Master's report: The design and implementation of free riding multicast. Technical report, UC Berkeley, May 2007.

[31] Andrey Ermolinskiy, Daekyeong Moon, Byung-gon Chun, and Scott Shenker. Minuet: Rethinking concurrency control in storage area networks. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST '09)*, pages 311–324, Berkeley, CA, USA, February 2009. USENIX Association.

[32] Andrey Ermolinskiy and Scott Shenker. Reducing transient disconnectivity using anomaly-cognizant forwarding. In *Proceedings of the 7th ACM Workshop on Hot Topics in Networks (HotNets-VII)*, pages 91–98, New York, NY, USA, October 2008. ACM.

[33] Andrey Ermolinskiy and Renu Tewari. C2Cfs: A collective caching architecture for distributed file access. In *Proceedings of the 2009 International Workshop on Network Storage and Data Management (HPCC '09)*, pages 642–647, Washington, DC, USA, June 2009. IEEE Computer Society.

[34] Fedora project. `http://fedoraproject.org`.

[35] Filebench. `http://www.solarisinternals.com/wiki/index.php/FileBench`.

[36] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Internet Society's 2003 Symposium on Network and Distributed System Security (NDSS '03)*, pages 191–206, Reston, VA, USA, February 2003. Internet Society.

[37] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, September 1992.

[38] grep - GNU project - free software foundation. `http://www.gnu.org/software/grep/`.

[39] Ajay Gulati, Manoj Naik, and Renu Tewari. Nache: design and implementation of a caching proxy for NFSv4. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*, pages 199–214, Berkeley, CA, USA, February 2007. USENIX Association.

[40] gzip - GNU project - free software foundation. `http://www.gnu.org/software/gzip/`.

[41] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '06)*, pages 29–41, New York, NY, USA, April 2006. ACM.

[42] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40:1098–1101, September 1952.

[43] iperf. `http://sourceforge.net/projects/iperf`.

[44] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: a hidden code extractor for packed executables. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode (WORM '07)*, pages 46–53, New York, NY, USA, November 2007. ACM.

[45] Hari Kannan, Michael Dalton, and Christos Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '09)*, pages 115–124, Washington, DC, USA, June 2009. IEEE Computer Society.

[46] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. A data-oriented (and beyond) network architecture. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '07)*, pages 181–192, New York, NY, USA, August 2007. ACM.

[47] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, pages 321–334, New York, NY, USA, October 2007. ACM.

[48] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16:613–615, October 1973.

[49] Global data leakage survey. February 2006. `http://www.securelist.com/en/analysis/204791919/Global_Data_Leakage_Survey_2006`.

[50] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: a platform for secure distributed computation and storage. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP '09)*, pages 321–334, New York, NY, USA, October 2009. ACM.

[51] LMbench - tools for performance analysis. `http://www.bitmover.com/lmbench`.

[52] Richard McDougall, Jim Mauro, and Brendan Gregg. *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris, 1st ed.* Prentice Hall, July 2006.

[53] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2:181–197, August 1984.

[54] Bob Sullivan (MSNBC). Government agency exposes day-care data (daily whereabouts of hundreds of children posted on public web site), February 2004. `http://www.msnbc.msn.com/id/4186130`.

[55] Andrew C. Myers. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL '99)*, pages 228–241, New York, NY, USA, January 1999. ACM.

[56] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 129–142, New York, NY, USA, October 1997. ACM.

[57] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 186–197, Washington, DC, USA, May 1998. IEEE Computer Society.

[58] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9:410–442, October 2000.

[59] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pages 89–100, New York, NY, USA, June 2007. ACM.

[60] Nettop: Commercial technology in high assurance applications. `http://www.vmware.com/pdf/TechTrendNotes.pdf`.

[61] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 2005 Network and Distributed System Security Symposium (NDSS 2005)*, Reston, VA, USA, February 2005. Internet Society.

[62] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 191–205, New York, NY, USA, October 2005. ACM.

[63] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, pages 308–318, New York, NY, USA, March 2008. ACM.

[64] Donald A. Norman. The way I see it: When security gets in the way. *Interactions*, 16:60–63, November 2009.

[65] Openoffice.org calc. `http://www.openoffice.org/product/calc.html`.

[66] Openoffice.org writer. `http://www.openoffice.org/product/writer.html`.

[67] Openoffice.org - the free and open productivity suite. `http://www.openoffice.org`.

[68] Michael Paleczny, Christopher Vick, and Cliff Click. The Java Hotspot server compiler. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM '01)*, pages 1–12, Berkeley, CA, USA, April 2001. USENIX Association.

[69] PostgreSQL. `http://www.postgresql.org`.

[70] Qemu. `http://www.qemu.org`.

[71] Qemu internals. `http://www.weilnetz.de/qemu-tech.html`.

[72] Feng Qin, Shan Lu, and Yuanyuan Zhou. Safemem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA '05)*, Washington, DC, USA, February 2005. IEEE Computer Society.

[73] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*, pages 135–148, Washington, DC, USA, December 2006. IEEE Computer Society.

[74] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems, 3rd ed.* McGraw-Hill, August 2002.

[75] Sylvia Ratnasamy, Andrey Ermolinskiy, and Scott Shenker. Revisiting IP multicast. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '06)*, pages 15–26, New York, NY, USA, September 2006. ACM.

[76] Grant McCool (Reuters). At trial, ex-SocGen trader admits secret code theft, November 2010. `http://in.reuters.com/article/idINIndia-52984120101117`.

[77] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, pages 63–74, New York, NY, USA, June 2009. ACM.

[78] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing dynamic information flow tracking. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '08)*, pages 35–45, New York, NY, USA, June 2008. ACM.

[79] Jamal Hadi Salim, Hormuzd M Khosravi, Andi Kleen, and Alexey Kuznetsov. Linux netlink as an IP services protocol, 2003. `http://www.ietf.org/rfc/rfc3549.txt`.

[80] Asia Slowinska and Herbert Bos. Pointless tainting? evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*, pages 61–74, New York, NY, USA, April 2009. ACM.

[81] Raj Srinivasan. RFC 1832 - XDR: External data representation standard, 1995. `http://www.ietf.org/rfc/rfc1832`.

[82] Florian Thiel. Universal TUN/TAP device driver, January 2002. `http://www.kernel.org/doc/Documentation/networking/tuntap.txt`.

[83] Herbert H. Thompson. Visual data breach risk assessment study, 2010. `http://www.3mprivacyfilters.com/whitepapers`.

[84] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*, pages 142–151, New York, NY, USA, June 1998. ACM.

[85] Sensitive TSA manual posted on web, December 2009. `http://www.usatoday.com/travel/flights/2009-12-08-airport-security_N.htm`.

[86] Joseph Tucek, James Newsome, Shan Lu, Chengdu Huang, Spiros Xanthos, David Brumley, Yuanyuan Zhou, and Dawn Song. Sweeper: a lightweight end-to-end system for defending against fast worms. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '07)*, pages 115–128, New York, NY, USA, March 2007. ACM.

[87] Stephen C. Tweedie. Journaling the Linux ext2fs filesystem. In *Proceedings of the 4th Annual LinuxExpo (LinuxExpo '98)*, May 1998.

[88] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the 2002 USENIX Annual Technical Conference (ATC '02)*, Berkeley, CA, USA, June 2002. USENIX Association.

[89] Xen hypervisor. `http://www.xen.org`.

[90] Heng Yin, Zhenkai Liang, and Dawn Song. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS '08)*, Reston, VA, USA, February 2008. Internet Society.

[91] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*, pages 116–127, New York, NY, USA, October 2007. ACM.

[92] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, pages 291–304, New York, NY, USA, October 2009. ACM.

[93] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 263Ű–278, Berkeley, CA, USA, November 2006. USENIX Association.

[94] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI '08)*, pages 293–308, Berkeley, CA, USA, April 2008. USENIX Association.

[95] Nickolai Zeldovich and Ramesh Chandra. Vncplay. `http://suif.stanford.edu/vncplay`.

[96] Nickolai Zeldovich and Ramesh Chandra. Interactive performance measurement with VNCplay. In *Proceedings of the 2005 USENIX Annual Technical Conference: FREENIX Track*, pages 189–198, Berkeley, CA, USA, April 2005. USENIX Association.

[97] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware enforcement of application security policies using tagged memory. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*, pages 225–240, Berkeley, CA, USA, December 2008. USENIX Association.

[98] Qing Zhang, John McCullough, Justin Ma, Nabil Schear, Michael Vrable, Amin Vahdat, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Neon: System support for derived data management. Technical Report CS2008-0934, UC San Diego, December 2008.

[99] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.