Analysis and Experiences with Information Flow Tracking as a Practical Means to Prevent Data Leakage

Lisa L Fowler

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-126
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-126.html

December 9, 2011

Analysis and Experiences with Information Flow Tracking as a Practical Means to Prevent Data Leakage

by Lisa L. Fowler

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Scott Shenker
Research Advisor

(Date)

* * * * * * *

Professor Ion Stoica
Second Reader

(Date)

Analysis and Experiences with Information Flow Tracking as a Practical Means to Prevent Data Leakage

Lisa L. Fowler
University of California at Berkeley

Abstract

Data leakage is a primary concern for companies and governmental agencies, for which information flow is means for mitigation. A popular technique used for evaluating pre-existing binaries is "taint tracking," but such approaches in real-life application were impractical due to excessive performance costs and numerous false positives due to taint explosion. The novel techniques used by our system PIFT directly eliminated these concerns, but revealed new deeper and more troubling concerns. In using PIFT successfully for information flow tracking on a commodity GUI-based operating system, and upon further inspection, we discovered that the applications and operating systems that we use in day-to-day practice are imperfect components that violate the basic tenets of information flow. In this thesis, we explore information flow tracking as a whole, elaborate on these troubling discoveries, and argue that no matter the performance improvements or adjustments made to correct the metadata for information flow tracking, it will be impossible to provide useful data for the prevention of data leakage without addressing and resolving common practices prevalent across legacy software.

1 Introduction

Controlling *data leakage* is one of the top security concerns today. For many industries, data loss is the top corporate security concern [2, 20, 55]; a concern that has increased as companies move to trusting third parties with their data, namely with the increase in reliance on cloud computing [45, 56]. In order to prevent and mitigate data loss, the concerned party must ensure that the target data and documents are stored only at authorized locations and are disseminated only to the appropriate parties. The target data can be restricted in various ways, such as only editable by a small group of corporate officers, or viewable only those with a particular security clearance, or forwarded to anyone within the organization but not the general public. Unfortunately, the recent history of major data leaks[1] suggests that most organizations are severely deficient in this regard, including those in government [37, 49, 50, 54], education [8, 44, 61], and the commercial world [19, 38, 43, 57, 59].

There are three main human vectors for data leakage:

- Malicious external party
- Malicious internal party
- Well-meaning internal party

In practice, it is for all intents and purposes impossible to prevent a determined internal party with otherwise legitimate access to the target data from exfiltrating that data. The determined malicious insider can utilize any number of out-of-band techniques for leaking the data, e.g., printing the documents, taking pictures of the documents, memorizing the data, sending information encoded as timed requests to a target website, and so on. Given that a malicious insider is limited only by the extent of his imagination, we do not attempt to address this threat, and instead seek to make their task more difficult.

We have seen corporate insiders intentionally leaking sensitive data in recent news [17], however none of the incidents cited earlier were due to malicious employees: The leaks were the result of external attackers and/or carelessness on behalf of well-meaning employees. In many cases, the organization had already devoted significant resources to improving their security prior to the leak. Furthermore, even after detecting a leak and improving security, additional security breaches often still occur [3, 61]. It is apparent that data confinement is a serious problem and the blame cannot be placed on organizational complacency or internal malfeasance.

1.1 Imperfect components

Despite the clear threat, and the money and effort devoted to developing new security technology, it is

[1]See http://blog.dataleaktoday.com for a litany of such incidents.

evident that it is hard to prevent leaks of sensitive information even in well-managed organizations with well-intentioned employees. In short, as we address herein, it is because most current efforts to confine sensitive data rely on imperfect components, and furthermore, it is because of these imperfect components that any effort to automatically detect or prevent data loss has remained fruitless.

In the case of a *malicious external party*, the attacker must exploit a vulnerability in the system. The vulnerability can be any number of: a *security policy oversight*, a *social vulnerability*, or a latent *technical vulnerability* in the operating system or application software. For simplicity, we assume a robust security policy, however this does not make the task any easier.

Consider the second vulnerability: the human component. The attacker can trick otherwise well-meaning internal parties into revealing confidential information. In this case, the attacker might impersonate a friendly party or someone in authority in order to gain the target employee's security credentials, or to trick the employee into installing malicious software on their otherwise secure workstation, etc. Regardless of the particular attack, the outcome in this scenario is the same: the sensitive data will be exfiltrated from the entity's domain of control.

Next, we examine the technical vulnerabilities in the systems that we are trying to protect. In order to prevent leakage of the target computer data, we require the operating system and application software to be resilient against attacks. However, most software in use today is rife with bugs that can easily be exploited for exfiltration [9]. This is an exceptionally effective vector for attack, and unfortunately there is little hope that this will be rectified in the near future: Legacy code will remain in use for a long time and even modern software has significant vulnerabilities. If we insist on using legacy software while attempting to defend against data leakage, we must build a separate supporting mechanism that compensates for the extant bugs and vulnerabilities.

This system must also be able to detect when unauthorized parties (such as malicious code, or personnel lacking the required clearance) have accessed sensitive data. This is limited in part by the efficacy of the security policy and how well access control mechanisms have been implemented. However, we consider again that we are attempting to implement access control on legacy code, which may or may not support such a mechanism. Furthermore, access control and data obfuscation techniques (such as encryption) often protect the data during only *part* of its lifetime. For example, even though a credit card number is encrypted and protected by ACLs while stored in the database, it will have to be decrypted, stored in memory, perhaps written to cache, and processed in the clear. In our current operating systems and applications, any of those intermediate states is likely not protected by the same level of strict security. Our modern operating systems simply do not support protecting data throughout its lifetime.

In the case of a *well-meaning internal party*, we still encounter hurdles. Confining sensitive data requires users to obey dissemination restrictions on documents and datasets. Yet we know that otherwise well-meaning users can be careless and occasionally email sensitive documents to the wrong parties, transfer data to insecure machines, or otherwise inadvertently allow data to leak.[2]

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" [34]. For example, to get around data restrictions implemented on corporate computing systems, users might use a USB key to transfer the data to their personal laptop, thereby creating an even worse security situation [50].

More generally, when given a choice between security and convenience, both the market and individual users will choose the latter. For instance, one of the biggest security problems is users running executable code downloaded from an external source. However, this is a feature often required by web sites, so if security regulations forbid users from doing so, the users will likely find some way around those regulations to continue doing what they wish to do.

Applications can even use legitimate techniques to achieve their goals—techniques that could later cause security holes. For example, Skype allows file transfers [24]. For legitimate reasons, all of Skype user session traffic is encrypted, including the file transfers. However, this end-to-end encryption means that no intermediate party can inspect the traffic for viruses or confidential data leaks. Skype currently provides no way of externally disabling file transfers; If one

[2]This does not take into account willful data theft, which is a case that we find out of scope for this analysis. Beyond the now famous Wikileaks events [17], we have also seen that lax data protection systems have enabled employees to steal sensitive company data prior to leaving the company. In a recent survey, 59% of ex-employees willfully took data without an employer's permission, with only a little over half of those data thieves reporting unfavorable views of their former employer [25].

wanted to prevent Skype file transfers, one must block Skype entirely—which is a technique that companies utilize for exactly this reason.[3]

1.2 Reducing the problem

It is clear that we must find a solution to data loss prevention that supersedes the imperfect components. We see that ultimately the outcome in all of the above circumstances is that sensitive data has left its protected systems in one form or another. We reduce the problem to addressing just this outcome, while considering the nuances of each vector.

Fundamentally, the goal in preventing data leakage is *understanding data movement and data lifetime* despite these imperfect components. In order to do so, we must know what is happening to the target data, presently or in the past:

1. *Where are that data and its derivatives right now?*
2. *How are the data and its derivatives accessed?*

A separate but no less important challenge is achieving determining this with minimal intrusiveness in order to foster adoption and encourage continued use.

Many systems have attempted to provide correct information flow tracking with varying levels of success, as we discuss in Section 2. However, as of yet, no system has proven ideal. In our own attempt to build a data loss prevention system, we believed that the first two challenges of tracking data movement were solved problems and the main hurdle was instead reducing intrusiveness by decreasing the perceived user slowdown. As such, we strove to significantly improve performance beyond the state of the art into a practical, usable, information flow tracking substrate. After doing so, we discovered that in fact, answering the first two questions correctly was in fact *nearly impossible* due to our underestimation of the inherent flaws in our legacy systems. We believe that these inherent flaws in our imperfect components means that any efforts to develop a whole-system real-time system that sufficiently captures data lifetime based on legacy systems will remain fruitless.

We provide herein an introduction to information flow tracking, as well as a discussion of the varied reasoning, techniques, and use-cases for information flow tracking. We briefly describe our own solution and detail our experiences applying an information flow tracking technique for data loss prevention in modern applications and operating systems. We continue with pinpointing the flaws in the general approach of dynamic analysis for information flow tracking, and conclude with our recommendations for properly addressing data leakage prevention in future systems.

[3]C.f., promotional material for security products Watchguard Firebox X Core `http://www.watchguard.com/products/core-e/overview.asp` and Trend Micro LeakProof `http://www.vmware.com/appliances/directory/193163`

2 Information flow tracking

One way to gain insight into potential leaks of sensitive data is to measure *information flow*, namely the derivation of information in one data object o into another data object s. Among other uses, graphs of information flow can be used to check *confidentiality* (if a path exists between a confidential s and an unauthorized o, there is a leak) and *integrity* (e.g., discovering if a path exists between a trusted s and an untrusted o).

Improved security of information flow can be achieved manually by the programmer, wherein the programmer directly makes each and every assertion necessary to enforce the desired data flow. However, this process is onerous, difficult to verify, and any oversights lead to critical vulnerabilities [9]—and does not take into account the dependencies between different co-existing systems and applications. In an effort to alleviate this problem, researchers began exploring providing information flow control (IFC) with the application of information flow models [1, 12, 13].

In short, keeping track of the original sensitive data and any of its derivatives enables us to see when that data is destined to an unapproved site or recipient.

2.1 Applications of IFT

Information flow tracking (IFT) systems and approaches currently have four main application domains:[4]

I Application security
Attack detection and prevention; unknown vulnerability detection; measure the behavior of foreign applications; detect malware (e.g., keyloggers); automatic input filter generation (e.g., defending against command injection attacks). Examples include Panorama [69], CWSandbox [67], TaintCheck [31].

II Data lifetime
Provide assertions on data deletion; enforcing information flow policies; auditing policy-

[4]The provided examples are not an exhaustive list.

compliant data usage.[5] Examples include TaintBochs [6].

III Software testing & debugging
Find data leaks; discover input that causes a failure; test case generation; verify or ensure policy-compliant data usage; ensure correctness. Examples include PQL [23].

IV General use
Any combination of the above. Examples include TaintDroid [15], Neon [75], RESIN [70], Loki [72], DStar [74] and HiStar [73].

Despite the common goal of providing information flow tracking, systems built for a target application domain will often prove insufficient for another domain, and even the general use IFT systems are often limited in their application to different scenarios. Clearly, a system developed for automatic test case generation will likely be insufficient for capturing real-time command injection attacks. These differences will prove to be critical when we later explore the efficacy of IFT in the different scenarios.

2.2 Additional considerations

Regardless of the target application, the IFT system designer must consider additional tradeoffs. These decisions have a direct impact on the completeness of the approach. A system might choose to focus on a narrow problem, ignoring more general cases, or vice versa. We enumerate several common, but critical, tradeoffs.

Online or offline Is the target system to be protected an interactive system? Does the IFT system need to measure real-time events? Will the analysis be performed on a test machine or in an otherwise out-of-band manner? Section 2.2.1.

Restrictions on modifications Is modifying the application/operating system/runtime/etc an acceptable assumption? In particular: Can we make changes to the hardware or are we limited to only software changes? To what extent can we modify the target system? What is the smallest data representation that we are interested in tracking? Section 2.2.2.

Whole-system analysis Are we only concerned with behavior within a single application (e.g., sanitizing user input in a particular web application such as in [5]); or do we seek to provide derived data management across multiple applications or networked systems? Section 2.2.3.

Permanence Will it be necessary to maintain metadata beyond a particular transaction or session? Should that metadata be maintained across systems or instances? Section 2.2.4.

Policies How much should the system understand "policies"? Are actions on all sensitive items treated the same way? Will the system be required to support an externally defined policy, or should it enable the definition of new policies on the fly (e.g., the marking of a particular email as "Do not forward")? Section 2.2.5.

We explore each of these tradeoffs in more detail.

2.2.1 Online or offline

Based on the expectations of interactivity and performance, information flow tracking can be done online or offline. It is necessary to perform IFT in real time (online) when you wish to take immediate action on information flow events, such as blocking the network transmission of confidential records, or preventing an employee from downloading a copy of the source code to a USB drive.

In contrast, offline IFT can be done on a separate test machine, or when performance is not a priority. Offline IFT is often simpler due to these lower performance demands, more tolerance to false positives and negatives (since the results can be verified or repeated), and better isolation. Offline IFT is particularly appropriate for malware analysis [67, 69], as well as software testing and debugging.

IFT can also be implemented partially online and offline. For example, a web application request could be duplicated and a copy sent to a single "canary" machine out of thousands of datacenter servers in order to perform the slower analysis in a non-disruptive manner. Similarly, one may also record live system events and replay those events on an offline IFT system to log any questionable events during an audit. It is also possible to delay certain IFT calculations until a later point in time if the approach can tolerate such a delay [46]. In this case, IFT can be delayed until a particular event triggers the need for verification, such as a user submitting a form for processing.

2.2.2 Restrictions on modifications

Depending on the assumptions regarding modifiability, the IFT system can have very different

[5]For example, detecting regulatory compliance violations, or providing assurances to customers that data is being handled in a particular way, such as promising that no user data will be sent to third party applications [43].

incarnations. If one assumes that it is impossible to change the hardware or to add supplemental hardware support, we are clearly restricted to place all IFT support only into the software. Other scenarios allow for changes to the software, but only via the runtime; or allow such extreme changes as rewriting the entire operating system (e.g., Asbestos [63], HiStar [73], and LoStar [72]); or supports writing or rewriting programs in a different programming language that natively supports IFT.

Can we make changes to the software or hardware? Many researchers choose to build hardware-based solutions for IFT [4, 11, 21, 46, 47, 53, 58, 62, 64]. By providing IFT support in hardware, performance penalties could be reduced, and absolutely no changes would be necessary for the application and operating system. In fact, the new architectures often incur low performance overheads and are completely transparent to the existing programs

Of course, a hardware-based solution is sometimes impossible if we are unable to make changes to the hardware, such as in the case of a deployment on commodity hardware. Some software-based solutions attempt to emulate these hardware changes through the use of emulators (such as QEMU) and software-based shadow memory [18, 52, 69].

Currently, commodity hardware does not support the basic notions of information flow tracking. While a hardware-based solution might be excessive when one is merely verifying user input, it is useful to consider the fundamental primitives that could be built into hardware that would support IFT solutions across application domains.[6]

To what extent can we modify the target system? The approaches available to the system designer will be very limited by where and how much they can modify the target system.

Some of the more promising IFT systems used a custom operating system with IFT primitives provided natively [22, 63, 70, 73]. If it is not possible to modify the operating system and provide IFT primitives, the system designer must decide where in the application stack to insert the IFT mechanism.

If we cannot edit or view the source code, we are limited to binary or *dynamic analysis*, which is performed on an executing program on real or virtual hardware and can be done without access to the source code.[7] Examples of this particular scenario are when we wish to provide IFT in commodity off-the-shelf proprietary software or downloaded foreign executables. Systems that use dynamic analysis include [4, 7, 14, 16, 65, 69], whose target domains vary from security to software verification, and more.

If we can edit the source code of the target program, we can use *static analysis*, which can be performed without executing the program, and requires access to the source code and/or object code. Some examples include [28, 40, 66], which use static analysis for software verification and testing. Both methods have advantages and disadvantages, and some approaches espouse a combination of both static and dynamic analysis.

Additionally, the inter-positioning point for the IFT mechanism can vary based on the approach. Some techniques provide augmented libraries that can be used on the fly [5, 36]. Other approaches might require recompiling the entire program, or even rewriting the program in an advanced IFT-aware programming language.

Over time, it has become evident that hybrid approaches often are more fruitful than a system that depends only on dynamic analysis or only on static analysis. Dynamic analysis suffers greatly from both false positives and false negatives (as we explore later in this paper), and static analysis cannot be done on foreign binaries such as malware. Another technique that has shown great promise is improving isolation (such as sandboxing the software that you cannot inspect properly) and performing IFC at these boundary points, such as with [35, 42].

What is the smallest data representation that we are interested in tracking? Depending on the approach used, IFT can be simplified when considering larger objects (process or file object rather than single characters or bytes). However, there is a tradeoff, as having analysis available at a finer grain can enable a larger feature set. For some use-cases, an extremely fine grain might be unnecessary (e.g., for detecting command injection attacks)

IFT can be performed at any level in the spectrum of fine or coarse grained analysis. The granularity can have a large impact, both on performance and on supported features. For example, an IFT system that tracks data at the byte-level could support "auto-redaction" wherein certain phrases in a document could be censored or revealed depending on the recipient's security clearance level; such a task would be impossible for a system that tracked information

[6]We believe that this would be a ripe venue for future work.

[7]For an in depth discussion of dynamic analysis (including dynamic taint analysis and forward symbolic execution), please see Schwartz, Avgerinos, and Brumley's detailed guide, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask" [48].

flow to and from file objects alone.

2.2.3 Whole-system analysis

Another major decision for feature trade-off is in the choice between whole-system analysis, or targeted analysis. In the case of automatic input filter generation for a single application (such as for defending against command injection attacks), it might suffice to provide IFT for only those data used by that application, as in [5, 32, 39, 68].

Privacy Oracle [11] and TightLip [29] are lightweight tools that are capable of analyzing applications for information leaks without any application-level instrumentations. However, these systems are limited to the software whose outputs only depend on inputs (and externally controllable parameters such as time and system configurations) and not scalable to tracing multiple input data.

Other use-cases for IFT require a more complete understanding of data movement. In the case of malware analysis, it is critical to understand what other data the malware might be accessing, meaning that fundamentally, the scope is not limited to a single application. Another case is of detecting keyloggers, or other forms of unintended data leakage. The user wishes to know all the places where that data resides, and not just limited to the scope of a single application. An even larger scale is to consider the interaction across multiple systems, such as in [75] and [26]. Additional examples of systems that provide whole-system analysis are: Panorama [69], TaintBochs [6], and our own system [16].

2.2.4 Permanence

When considering a particular need for IFT, one must evaluate whether or not historical information flow metadata should be retained. Should the IFT be performed only on a per-session basis (such as in the case of evaluating command injection attacks), or should the metadata persist despite major state changes such as system shutdowns?

Clearly for many instances of software verification, it is sufficient to consider the information flow history for a limited amount of time. When using IFT to prevent data leakage, a confidential document should be treated as confidential over its lifetime, and thus the metadata must be kept in persistent storage and protected as much as the data itself.

2.2.5 Policies

Depending on the particular application for IFT, the policy needs can vary greatly. In a system with only one well-defined attack vector, such as with command injection attacks on a particular web form, or for determining if a particular piece of malware has accessed sensitive data, it is sufficient to treat all data as either suspect or not (or more generally, suspect, sensitive, and neither).

For other applications, the system might want to support elaborate policies encoded as metadata. For example, it might be necessary to flag a particular email as “Do not forward” and enforce that that email and its derivatives are never sent over the network interface, while at the same time supporting a “Do not print” notion.

Clearly, information flow tracking can also become information flow *control* if the system chooses to enforce the desired policies regarding information flow.

Another point of consideration for implementation is that of defining the policies themselves. Some systems, such as those performing offline analysis of particular software (either for testing or security evaluation), can have the policies defined *deus ex machina* and will never change throughout the IFT process, e.g., keystrokes coming over the network interface are to be treated as suspect and nothing else. Other systems might want to enable the user to declare policy on the fly, e.g., letting the user decide that a new document is confidential.

After this exploration of the history and tradeoffs in developing an information flow tracking system, we present our own approach to preventing data exfiltration, named *PIFT* for *Practical Information Flow Tracking*, and comment on our experiences using such a system.

3 System overview

Conceptually, our system architecture is analogous to a recently proposed IFT system (Neon [75]) and makes use of similar building blocks to it and its predecessor [18]—namely, a hypervisor and an augmented QEMU-based emulator that tracks information flow. 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. For example, even a simple computation on tainted data can incur a slowdown on the order of 95×[8] when only 1/64th of the input file is tainted [75] (no data is provided on how the system behaves with a more

[8] According to [75]-Table 3.

stressful amount of taint). Such a slowdown is unacceptable in practice and significantly hinders the adoption of dynamic taint tracking systems for everyday use.

2. To be comprehensive, taint tracking has to track information flow across pointer references, i.e., taint the referenced data with the same taint as the pointer. However, prior work [51] shows that this leads to accidental tainting of kernel data structures, meaning that soon any other application interacting with the kernel also acquires taint, and eventually, the taint status propagates to all data in the system. Such taint explosion renders the effort of IFT ineffective and substantially impairs the performance of the system.

In order for any data leakage prevention system to be successful in the face of such willful yet well-meaning circumvention:

- The system must minimize any impact to the user—the user should be able to continue behaving as they would otherwise, with minimal performance impact;
- The system must be fine-grained enough to support the desired policies—As in our Skype file transfers case, blocking Skype entirely would be undesirable; we merely wish to prevent the loss of sensitive data;
- The system must support legacy applications and operating systems.

We propose three novel techniques that help us address the above challenges and bring real-time taint tracking closer towards 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 changing each micro-instruction to propagate taint, which immediately incurs a significant but non-essential overhead. In contrast, PIFT tracks the flow of information directly at the native instruction level of the protected VM, which enables a range of optimizations that are difficult or impossible to apply at the micro-instruction level.

(2) PIFT performs taint-tracking asynchronously and in parallel with the main emulation codepath. The key insight is that up-to-date taint information is only needed under certain conditions (e.g., when switching from emulated to native execution or when invoking a policy). Hence, instead of tracking the propagation of taint labels synchronously and in lockstep with emulation, PIFT generates separate streams of taint tracking instructions and executes them asynchronously on another CPU core. In Section 5, we show that asynchronous taint tracking performed at a level of abstraction that directly matches the machine architecture can produce a 60× 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.

After building this system with a goal of fast performance with fine grained analysis on legacy software, we evaluated it on commodity off the shelf software. In doing so, we discovered that we had underestimated the final requirement of "supporting legacy application and operating systems." We continue to detail our approach for an IFT solution despite this setback, and address our discoveries after presenting our technique.

3.1 Design choices

As we detailed in Section 1, any developer of an IFT solution must consider particular tradeoffs with respect to what can or should be modified in the target system. We elaborate on our choices herein.

3.1.1 Software-based solution

Our solution must work on legacy systems, and on commodity off-the-shelf software as well as hardware. As such, we are limited to a software-based solution. Currently, we can assume that the hardware provides no IFT primitives, and thus we chose to use emulation in order to achieve those primitives.

3.1.2 Whole-system analysis

Information flow tracking across the entire operating system can enable a number of useful tasks. By en-

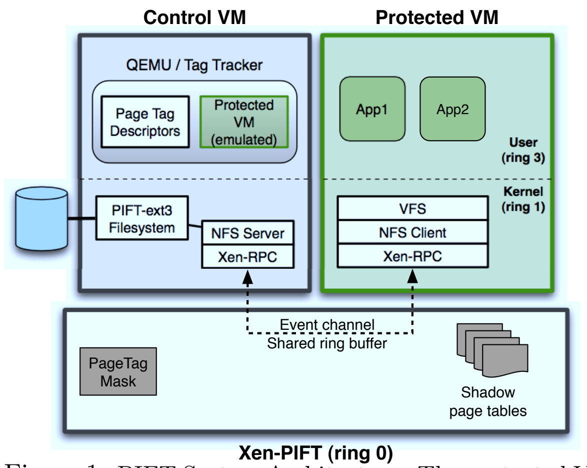

Figure 1: PIFT System Architecture. The protected VM is emulated via QEMU in the control VM when it accesses tainted data. The PTT-ext3 filesystem is located in the control domain and the protected VM communicates with it via a shared memory RPC mechanism.

abling the programmer to discover deviations from the expected flow logic, IFT-aware systems can provide tools for stronger security and debugging. IFT-aware systems can also provide support for stronger policies to protect sensitive data or intellectual property. For example, given a whole-system IFT mechanism, one could enforce a policy that no derivations of the contents of a particular sensitive file can be sent over any network interface.

For our target application of preventing data exfiltration, we must provide a whole-system fine-grained solution.

3.1.3 Dynamic analysis via taint tracking

A primary means for achieving IFT on binaries is a method wherein a label is recorded in shadow memory, and that label is propagated as contents of the memory are computed upon and moved. This label is sometimes referred to as a "taint" label, a term that originated from "taint mode" in Perl, wherein all user supplied input is treated as tainted and suspect unless the programmer explicitly approves the data [36]. Taint-tracking systems have been used for malware analysis [69], software verification [7], and protecting data of interest [71]. Some of these uses require online (or real-time) analysis [75], whereas others are suitable for offline analysis [30]. Like our predecessors [18, 75], we use this technique.

3.1.4 Prevent data leaks

There is a large literature of taint tracking techniques which were successfully applied for malicious code analysis (e.g., [5, 18, 30, 31, 52]). Recently, many approaches focused on information flow tracking for protecting user data or preventing data leaks [26, 75, 76]. In our PIFT system, we also strove to provide full-system fine-grained information flow tracking for daily use defending against data leakage.

3.1.5 Persistent metadata

For our use case of preventing data leaks, clearly a confidential document should always be treated as confidential, even if the user reboots their system. As such, we must require that any IFT metadata has permanence. We also wish to provide metadata that can be transmitted between distributed systems.

3.1.6 Policies

Following [33], at a conceptual level the behavior of our system is dictated by three policies:

- **Input Policy**: The input policy determines what data is tainted and what taint to apply. For example, an input policy might be to taint all data from a socket with a particular label, and track it as it flows through the system. Of course, users and administrators can always mark taint on files and data themselves.
- **Propagation Policy**: A propagation policy determines how taint is propagated from one location to another. By default, our propagation policy carries taints from data sources to data sinks for all data movement (`MOV, MOVS, PUSH, POP, ADD, CMOV`, etc.) instructions.
- **Assert Policy**: An assert policy determines when to take action and what actions to take, based on the taint. For example, for data exfiltration, the appropriate assert policy would be to check taints and prevent exfiltration only at exit points, such as writing to a network device. By default, we worked with an exfiltration-style assert policy.

Figure 1 sketches a high-level picture of our system architecture, and more detail can be found in [16].

3.2 Prior work

Our system builds on extensive prior work in information flow tracking (IFT) and dynamic taint analysis (or "taint tracking") in commodity operating systems and applications. Information flow tracking was introduced nearly 30 years ago as a technique for security policy enforcement [12, 13]. Early efforts focused on static analysis (such as the popular work in information flow control by Myers and Liskov [28]), with later techniques supporting dynamic information flow labels by combining static and run-time checking, such as [27] and [29]. Another recent work in dynamic information flow tracking [70] offers a language runtime that propagates policy objects along with data, and applies this policy by filtering objects at system I/O boundaries. However, each of these language-based techniques require significant alterations to application, and the static techniques cannot support dynamic changes to policy without recompilation.

PIFT associates taint labels with low-level elements of the machine state (individual bytes in CPU registers, in physical memory, and on disk) and tracks their propagation at the level of machine instructions. Previous systems such as Asbestos [63], HiStar [73], and LoStar [72] employ an alternate approach—attaching data labels to OS-level primitives such as file descriptors, sockets, address spaces, and processes.

In order to perform information flow tracking in environments that do not provide such language, compiler, or OS support, recent efforts have relied on binary rewriting at runtime as a way of introducing IFT into commercial applications and operating systems. This has been used to address numerous security-related issues. For example, one body of work [30, 60, 69] applies taint tracking mechanisms to the problem of malware detection and analysis. These projects impose significant slowdowns, but performance is a secondary concern for offline analysis. Other systems use taint tracking mechanisms to understand the issues of data lifetime and leakage (e.g., [6], [10], or to enforce security policies on the flow of sensitive user data in networked environments [75]).

Byte-level taint tracking faces significant performance challenges and a number of optimizations have been suggested in earlier work. Demand emulation and shadow page table trapping were first proposed in [18] and our system directly leverages these techniques. Similar in spirit to our work, the Log-Based Architecture proposed in [46] attempts to improve the performance of fine-grained information flow analysis through asynchrony and parallelism, but depends on a major hardware extension. Speck [33] proposes a set of techniques for parallelizing security checks (including taint tracking) on commodity hardware. Speck focuses on tracking within one user-level process and assumes OS-level support for speculative execution, while our work achieves full-system tracking using a hardware emulator.

4 Designing for practical information flow tracking

After building PIFT and addressing the performance hurdles that prevented other systems from being utilized in daily use, we explored in-practice using PIFT as an information flow tracking substrate. We present those findings herein, as well as reveal our insights into improving the state of the art in information flow tracking.

We illustrate the overall machinery of PIFT by considering a typical usage scenario—enforcing confidentiality policy on the contents of a sensitive file. For simplicity of exposition, we assume that the user taints the entire file with a single policy label. Initially, the tainted file resides on disk that is managed by the taint-aware filesystem and applications in the protected VM execute directly the native host CPU. When an application opens the tainted file, the call is routed via the hypervisor to the filesystem in the control VM. Before returning the file handle, the filesystem makes a call to the hypervisor, informing it of the file's taint label. In response, the hypervisor marks the memory pages where the file contents are stored as *tainted*.

Shortly thereafter, the protected VM tries to access the file data and since the corresponding pages are now marked as *tainted*, the hardware generates a page fault and delivers it to the hypervisor. The hypervisor saves the protected VM's context and switches to emulated execution in a user- space QEMU process within the control VM.

The emulator is instrumented to track the movement of taint labels in the manner that reflects the computation performed by the protected VM. As QEMU proceeds with the emulation, it executes additional logic to update data structures that keep track of taint labels for machine registers and memory addresses. When the protected VM ceases to manipulate tainted data, the QEMU suspends the emulated machine context and notifies the hypervisor, which reverts the protected VM back to native virtualized execution.

The final step is when the policy is invoked. For example, the policy might specify that an exception should be raised if tainted data is being externalized through a network interface or a removable block storage device, such as a USB stick. The exception thrown could be an existing processor fault (e.g., an invalid opcode) or a new specialized exception. The exception handler provides an opportunity to write custom policy filters, which can either deny the action, log it for audit purposes, or filter the content being externalized.

Joining the efforts to provide information flow tracking at a level that would reduce the hurdles to adoption, we created PIFT. Through our system PIFT, we provide users with a way of tagging files or parts of their files, track which (non-malicious) applications access that data and how the data is computed upon, and ultimately to where (if at all) the data or any derivative is shipped. Note that PIFT's goal is not to secure sensitive data—tracking in PIFT could be thwarted by additional focused effort from a malicious application. Rather, PIFT enables users to monitor their personal data and its interactions with commonly used benign applications. The goal is that this functionality be then used to analyze whether an application is accessing more data than it needs, help implement application access control, or to prevent users from accidentally breaking data restrictions (e.g., ensuring that a secret company file does not unknowingly get sent to an offsite email address).

The typical implementation approach to build such information flow tracking is to instrument hardware emulators such as QEMU with tracking instructions. Hardware emulation is extremely slow, so to improve performance, modern implementations [18] use emulation only for those regions of code that interact with tagged data. These efforts proved quite useful for binary analysis [30], but the performance is still not adequate for real-time use. In addition, fine-grained tracking often results in accidentally tagging control information such as kernel data structures, which then amplifies into a *taint explosion* that unnecessarily taints significant portions of data.

These two concerns, poor performance and taint explosion, have rendered full-system fine-grained information tracking impractical for deployment outside of a laboratory environment. This paper presents our progress in addressing these two problems and thereby making fine-grained taint tracking more practical. However, this advance enabled deeper testing and evaluation of this approach against legacy software, and presented some troubling results. We present our approach regardless, since we provided significant performance advances beyond the state of the art, and evaluate those troubling results in our analysis.

Our approach (which we call PIFT for *practical information flow tracking*) begins with completely standard building blocks: PIFT uses the Xen hypervisor to run the tracked system within a virtual machine, and (as in [18]) dynamically switches execution on to and off of an augmented QEMU hardware emulator as tracking is required.

To improve performance, we track tags at a higher abstraction level and in an asynchronous fashion. Specifically, instead of instrumenting the micro-instructions QEMU generates to track data, we create a separate stream of tag tracking instructions from the x86 instruction stream itself. This provides two benefits:

- For the tracking instructions, we avoid the amplification that occurs when we go from one x86 native instruction to multiple QEMU micro-instructions.

- Since this is a separate stream, the tracking can be done asynchronously and in parallel without stopping the emulation context.

In essence, this approach is both directly more efficient (because it generates fewer tracking instructions), and it also allows us to execute those instructions asynchronously; it is the combination of these two effects that give PIFT better performance.

Graphical environments present a non-trivial challenge for fine-grained taint tracking systems such as PIFT. Although such environments rarely impose high computational demands, the key challenge is dealing with window rendering systems when tainted data is present on the screen. A simple screen refresh will involve moving tainted data around, and hence lead to constant oscillating switches between native and emulated execution.

A direct application of PIFT to a graphical environment with tainted data on the screen leads to significant performance impairment, to the point that the user perceives serious interactivity problems. The reason is a thrashing behavior resulting from repeated screen repaint operations. In typical usage scenarios, basic user actions (such as mouse movements and keystrokes) trigger application-level events that cause the window image to be recomputed. For instance, if a text window is showing tainted data and the user is entering text from the keyboard, each keystroke cause a page fault and transition to emulation. When the guest system 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.

We found that the overall performance and usability of the system can be greatly improved in such situation by making a simple fix: *persistently* switching to emulation mode and remaining in emulation mode for as long as tainted data remains on 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 fairly compelling use case for asynchronous taint tracking. Since the guest workload is interrupt-driven and proceeds mostly at human timescales, the taint tracker can easily keep up with the producer and the log helps absorb the short burst of computation resulting from basic user activity.

Persistent emulation with asynchronous taint tracking led us to a fully-operational and usable graphical environment. In this environment, users observe almost no perceivable degradation of interactivity for simple UI actions (e.g., moving the mouse, entering text, scrolling), which we show in Section 6.

For taint explosion (a common problem for approaches such as ours), we show that a major source of the explosion is the tainting of kernel data structures via a few kernel entry functions. We show that the taiting is accidental, and does not reflect explicit information flow. PIFT leverages this insight to effectively eliminate kernel taint explosion by interposing and scrubbing taint labels at specific and appropriate points, but unfortunately reveals that taint explosion is a problem that persists into desktop applications.

The cumulative effect of our techniques is making fine-grained information flow tracking significantly more practical. In fact, PIFT is usable and supports a full graphical user interface, enabling real user activities—we edited portions of this paper with it! It is this substantial advance that enabled us to more deeply evaluate the technique of taint tracking as a means for information flow tracking on legacy software. From this analysis, we find that dynamic analysis for real-time information flow tracking is fundamentally flawed due to the reliance on the imperfect components in the target legacy code. There is more work to be done, however we wanted to report on our progress and findings so that the community can help in overcoming the remaining barriers.

4.1 Caveats

It was difficult for us to provide direct comparisons with previous work. In the two specific cases where we could do so, PIFT achieved a slowdown of roughly 1.4× (compared to native code execution), as opposed to previous efforts (in [75], based on [18]) in which the two comparison cases had slowdowns of roughly one and two orders of magnitude, respectively. We hasten to note that these two comparison cases were optimistic ones, and there are other cases where our slowdown is higher (roughly 20×).

As we noted, many similar approaches struggled with the problem of taint explosion [10, 18, 30, 51], wherein in certain paths of data flow, key system components become tainted, and spread that taint to irrelevant and unrelated data. For taint explosion, we asserted that the source is accidental tainting of kernel data structures via a few system calls. By interposing at these specific system calls, and securely scrubbing taint, we prevented accidental tainting of kernel data structures. This prevents incorrect taint propagation to other data, which we believed would eliminate kernel taint explosion in practice. Since this enabled us to analyze information flow in applications (rather than the operating system), we discovered that *not only* does taint explosion occur due to pointer tainting in the kernel, but common programming paradigms in legacy software also result in taint explosion.

While repeating the technique of interposing at key points in the application (just as we did with the kernel) to scrub the taint and prevent taint explosion would be successful, it becomes intractable, and forces us to augment the software, which is against our earlier assertions that we could not make such changes.

Before proceeding, we note that there is another problem with taint tracking: Implicit information flows (e.g., copying by branching) are not detectable by dynamic analysis and do not leave behind a taint trail. This can be a problem for certain applications of taint tracking such as data exfiltration prevention. However, here we are focusing on making taint tracking practical for applications such as auditing, debugging and data tracking where malicious taint scrubbing via implicit flows is not a concern. Note that researchers are attempting to address this particular flaw by providing a hybrid solution between dynamic analysis and static analysis or programming language techniques. Thus, while implicit information flows are an important issue for some applications of taint tracking, we do not address this problem here.

In our current design, PIFT tracks all explicit data flows resulting from variable assignments and arithmetic operations. The emulator monitors the computation on tainted data at the level of machine instructions and propagates the labels accordingly. We also track indirect flows that occur as the result of pointer dereferencing, whereby a tainted value is used as a base pointer or an offset to access another value in memory. However, like other similar systems, PIFT does not currently track the flow of information through implicit channels that arise from control flow dependencies, such as when the value of a tagged byte influences a conditional branch.

5 Performance evaluation

In this section, we evaluate the performance overhead of our prototype implementation under a variety of workloads, which include synthetic microbenchmarks and real-world applications. We find the following:

- For a system with 10% of the data tainted, the computational overhead over native execution is 40%. In the worst case (when all data is tainted), the overhead is 20×. Both of these results appear to be significantly better than prior work, though we can only make exact comparisons in two specific senarios.

- The performance improvement is the result of a combination of high-level taint flow instrumentation and asynchronous parallel execution of taint tracking.

- PIFT can support graphical windowing environments, ensuring a reasonable level of interactivity and application-level performance. 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.

5.1 Setup

Our test machine is a Dell Optiplex 755 with a quad-core Intel 2.4Ghz CPU and 4GB of RAM. The hypervisor-level component of our implementation is based on Xen 3.3.0. The emulator and taint tracking modules (based on QEMU v0.10.0) run in the privileged Xen domain as a multi-threaded user-level process. The guest domain is configured with 512MB of RAM and one VCPU (as our current implementation does not yet offer support for multi-processor guest environments). The guest runs a paravirtualized Linux kernel v2.6.18-8.

Our experiments evaluate the overhead in the following configurations:

- **NL**: Linux on native hardware
- **PVL**: Paravirtualized Linux on Xen hypervisor
- **Emul**: Linux in a fully-emulated environment using unmodified QEMU
- **PTT**: Our prototype implementation of Practical Taint Tracking
  - **- PTT-S**: Synchronous taint-tracking
  - **- PTT-A**(x): Asynchronous parallel taint tracking with x MB of memory reserved for the log. We explore using log sizes of 512MB, 1GB, and infinite (∞)

We compare our system primarily with Neon [75] when possible. Neon builds on top of the on-demand emulation-based taint tracking system developed by Ho et al. [18], and hence provides a comparison to both systems most closely related to us. However, we were unable to get the code for [18] working on our system since it is based on a heavily outdated version of QEMU and has fragile dependencies, and as such would not work on our test systems. Thus, the only way we could provide direct comparisons was to run the same experiments as reported on in [75] and use their published results to compare the two systems.

5.2 Application-level overhead (data processing tools)

We evaluate the overall performance penalty of taint tracking in common usage scenarios via microbenchmarks, as perceived by potential users of the system. The goal is to measure if our taint tracking substrate can be used for everyday computing activities.

5.2.1 Copying and compressing

We begin by considering two simple but very common data manipulation activities:

LocalCopy - Copying a partially-tainted file to another file in the guest filesystem using the `cp` command.

Compress - Compressing a partially-tainted input file using the`gzip` command. Compression represents a somewhat more stressful scenario as it involves a nontrivial amount of computation on the input data.

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 fraction of taint, since the amount of taint should dictate the amount of time spent in emulated taint tracking mode. Our system does not show linear scaling because the heuristics for transitioning are not perfect. The heuristics err on the conservative side, keeping the system in emulated mode even if one could have transitioned back to native virtualized mode a bit earlier. Hence, the overhead at low levels of taint is larger than the linear scaling would suggest.

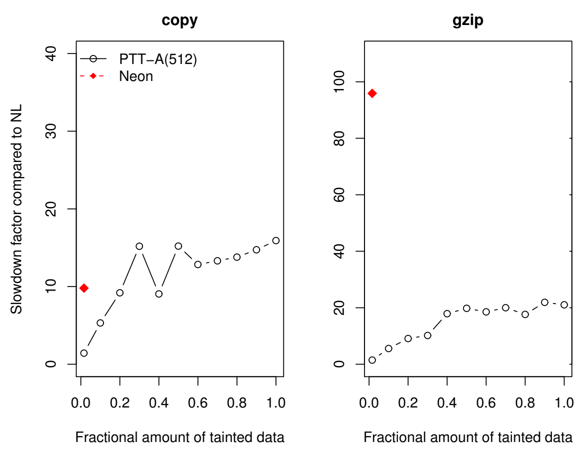

Figure 2: Performance overhead in the *LocalCopy* and *Compress* experiments.

Finally, although we do not have enough data to draw definitive conclusions, we believe that these results yield a favorable comparison to Neon. In a scenario with $f = 1/64$, where f is the fraction of tainted data, Neon reports slowdown factors of 10× and 95× for file copy and compression, respectively. This is the only data point for this experiment reported in [75] (Table 3). We see that taint tracking with PIFT incurs significantly less overhead. The overhead for both file copy and compression is only 1.4× over native execution. In Figure 2, we can see the performance of PIFT over the range of f, but have no corresponding data from [75] except for $f = 1/64$.

5.2.2 Searching

In the next experiment, we consider another common usage scenario: text search. We use the *grep* command to search the input data set for a single-word string and measure the overall running time. Our input dataset for this experiment is a 100-MB text corpus spread across 100 equal-sized files. These files reside on disk at the start of the experiment. We measure the command completion time in all configurations of interest and compute the slowdown relative to native execution, i.e., *NL*. For PIFT, we repeat the experiment multiple times, varying the number of files f marked as tainted. Unfortunately we cannot compare with Neon, since they do not report any measurements for this scenario.

Table 1 reports the results of this experiment for the most stressful scenario, i.e., when all files are tainted ($f = 1$). As expected, the performance is significantly worse compared to native execution for this worst case scenario. PIFT with asynchronous parallelized taint tracking is slowed down by a factor of 10×. For more modest amounts of taint, when only 10% of the files are tainted, the overhead is 46% over native execution.

	NL	PVL	Emul	PTT -S Synch	PTT -A(512) Asynch
Completion time (s)	2.42	2.87	18.45	58.865	25.57
Slowdown factor	1.00	1.18	7.62	24.32	10.56

Table 1: Text search performance in the worst-case scenario ($f = 1.0$).

Table 1 also quantifies the gain over synchronous taint tracking obtained by performing taint tracking in an asynchronous parallelized fashion. Asynchronous tracking using multiple cores roughly provides a 2.5× gain over synchronous sequential tracking on a single core. The best case gain that can be obtained from parallelization of taint tracking is if it creates an illusion that there is no taint tracking at all, i.e., it has the same performance as pure emulation. PIFT with parallelized tracking incurs a 40% overhead over pure emulation. This nontrivial overhead is to be expected, especially since the emulated instruction sequence has to be instrumented to write to the log operands that are needed by the taint tracking instruction sequence.

5.3 Graphical text editor

To measure the performance impact on graphical application-level operations, we instrumented the *gedit* text editor application and ran a simple user session. This session included launching the editor, opening a 1.2MB text file, making some changes, computing document statistics, and saving the file under a different name. Table 2 reports the results from this experiment and Figure 3 shows a trace of taint tracking log usage and CPU utilization in the control domain for this user session.

Although the text editor was fully usable and responsive during this user session, the measured performance degradation was somewhat higher than we expected. Notably, the component of the overhead due to taint tracking does not increase from previous experiments in the text console environment. At the same time, the costs of basic system emulation increase to about 20×. Further investigation revealed the likely source of this discrepancy. The GNOME graphical environment on x86 makes extensive use of

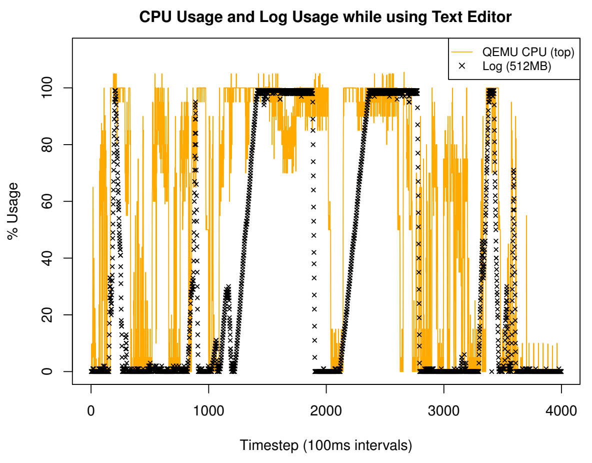

Figure 3: Time series of log usage and host CPU usage by the QEMU process. Note that the 100% CPU usage mark represents full utilization of both processor cores (producer and consumer).

Action	NL	Emul	PTT-A(512)
Launch editor	2403	7472	8741
Open file	302	2087	4634
Compute document stats	307	7309	12491
Save file	420	8300	15086

Table 2: Completion time (in *ms*) for a range of user actions in the *gedit* text editor experiment.

the SSE instruction set extensions and QEMU does not currently optimize the emulation of these processor features. Current versions of QEMU dynamically recompile arithmetic and memory access instructions into native code and try to optimize the use of host registers. At this time, QEMU does not perform JIT recompilation for SSE instructions and does not take advantage of the SSE registers available on the host processor. We expect that the overhead of our system will be reduced further with orthogonal improvements in emulation technology.

6 In-practice performance evaluation

In the earlier sections, we evaluated the raw performance of PIFT in several focused use cases. We now explore using PIFT as an information flow tracking substrate in interactive situations, on a system with a modern GUI, in order to see if our performance gains listed in Section 5 still persist when performing more complicated tasks.

We focus on the user-perceived overhead when performing common tasks in applications such as spreadsheets, word processors, web browsers, etc. We also evaluate the exact overhead of applying a policy against the taint labels in a real-world situation in order to enforce concepts such as, "You may not send files with label *xyz* over the network."

Graphical environments are difficult for fine-grained taint tracking systems due to their highly interactive nature. Additionally, keyboard and mouse actions will cause many window updates (which may contain tainted data) and thus may trigger many transitions to and from emulation. We found that the overall performance and usability of such a system can be greatly improved by making a simple fix: *Persistently* switch to emulation mode and remain in emulation mode for as long as tainted data remains on screen. Doing so also enables us to leverage significant benefits from asynchronous parallelized tracking. In fact, interactive graphical environments seem to be a compelling use case for asynchronous taint tracking. Since the guest workload is interrupt-driven and proceeds mostly at human timescales, the taint tracker can easily keep up with the producer and the log helps absorb the short burst of computation resulting from user activity. This is the primary deviation from the sources of performance gains listed in Section 5.

6.0.1 Graphical environment test setup

For the following series of performance tests in interactive environments, we used a Lenovo H320 with an Intel Core i3 540 processor and 6GB DDR3 RAM. Our guest environment was again configured with 512MB RAM, one VCPU, and a modified Linux kernel v2.6.18-8. The taint tracking log was fixed at 1GB for *PIFT-A*, and the tests took place entirely in emulated mode. We used the X Window server and GNOME desktop environment. We used *vncplay*[9] to measure and replay user input in order to evaluate the slowdown of various configurations. We provide the following new testing parameters:

PIFT-x/C	The guest is "clean" (no tainted data in the system).
PIFT-x/T	The application is instructed to open or manipulate a tainted file.

6.0.2 Graphical application launch test

In the first set of experiments in a graphical environment, we measure the time it takes to launch a program, render the graphical components, load a file, and quit. The programs we measured were Abiword (an open source word processing program) and

[9]`http://suif.stanford.edu/vncplay/`

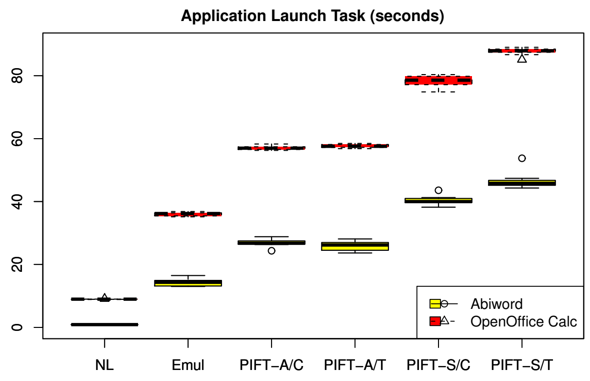

Figure 4: Completion time in seconds for starting an application and opening a file (tainted and clean), in both Abiword and OpenOffice Calc across all configurations.

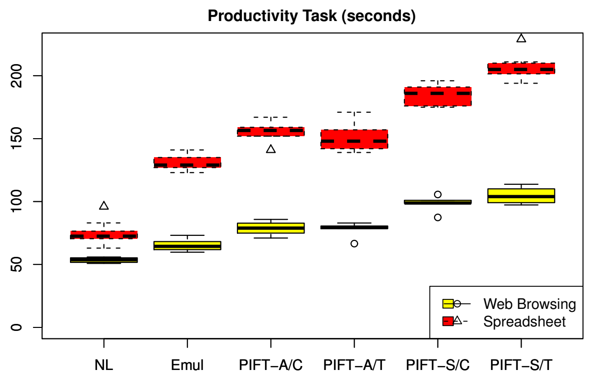

Figure 5: Time in *s* to complete the given productivity task (update spreadsheet in OpenOffice Calc, or browse and click through 10 web pages) in all configurations.

Ideal	*NL*		*PIFT-S/T*		*PIFT-A(1GB)/T*	
WPM	WPM	% of Ideal	WPM	% of Ideal	WPM	% of Ideal
60	58.51	98	58.51	98	58.51	98
110	107.86	98	107.86	98	107.86	98
140	137.72	98	137.72	98	137.72	98
150	146.75	98	144.39	96	146.75	98
200	198.93	99	168.91	84	198.93	99

Table 3: Time (in seconds) to type a passage at varying WPM, using *vncplay* to synthesize keypresses.

OpenOffice Calc (the spreadsheet component of the OpenOffice.org productivity suite). All tests were performed with cold filesystem caches.

In this computationally- and graphically-intensive situation, we confirm the benefits of asynchrony and see that slowdowns are moderate (Figure 4). Compared against emulation alone, asynchronous taint tracking results in less than a 2× slowdown for launching an application, even when opening a tainted document.

6.0.3 Common GUI-based task test

To measure the performance impact on user experience for common tasks, we measured the time that it takes a user to complete a series of tasks in GUI-based applications. We examined three interactive tasks in particular: Typing into a word processor, updating a spreadsheet, and browsing webpages.

Typing: To evaluate any perceived slowdown for typing, we used a slightly modified *vncplay* to record the entry of a passage of text into OpenOffice Writer and play back the entry of the passage with the timing adjusted to simulate typing at a number of different speeds (the *ideal* WPM). When replayed, we measured the time for the entire passage to appear and used this to calculate an *actual* WPM. During playback, the passage was "typed" into the beginning of a document that was already tainted (except in the *NL* case). The results of this test are in Table 3.

Beyond this, we experience *no perceptible difference* in text entry speed using PIFT, from 60 WPM through a very fast 110 WPM, and regardless of taintedness.[10] For *PIFT-S*, we see performance degradation begin around 140–150 WPM. *PIFT-A*, however, kept pace with *NL* throughout the entire test.

Spreadsheet: We measured the impact to user experience when performing a series of basic spreadsheet tasks, such as formatting and copying cells, typing referential formulas, and navigating the spreadsheet using both mouse and keyboard. We measured the time that it took a user to complete the overall task in OpenOffice Calc on each configuration. In *NL*, the complete task took roughly $1m15s$ to complete. It should be noted that this test focused on speed—the task being mechanically completed as quickly as possible, and did not allow for 'think-time', which would reduce the impact of the overhead.

Our tests showed that emulation itself causes roughly a 2× slowdown compared to *NL* (Figure 5), and *PIFT-A* only incurs approximately another 15% overhead regardless of taintedness.

Webpage Browsing: Finally, we evaluated browsing and scrolling webpages. We created 10 sample webpages of varying lengths, containing only very simple HTML, CSS, and JavaScript. We added 10-20 hyperlinks to each page, and gave the user the task of finding and clicking on a specific hyperlink on each page, which would then direct the browser to

[10]The difference of the measured WPM for all tests (including native Linux) from the ideal is attributable to overhead from VNC and *vncplay*. Indeed, before some modification to the timing code in *vncplay*, the difference between ideal and measured WPM was much greater.

another page in the series. The user used the Konqueror web browser with caching disabled and used only the mouse for the duration of the test. All files were local to the user.

We can see in Figure 5 that the user experiences a 1.96× slowdown for completing the web browsing task in the worst-case of *PIFT-S/T* compared to *NL*. With *PIFT-A(1GB)*, the task takes on average 1.48× longer. Like the spreadsheet updating task, this task was heavily dependent on screen updates, since the user was required to scroll through and search for a specific link on each page. As such, we suffer a performance penalty due to this dependency on graphical components, but it is no worse than 2× slower than *NL*.

6.0.4 Impact of policy checking

One use of information flow tracking is monitoring the flow of sensitive data, perhaps with the end goal of auditing the export of sensitive data, or blocking the export of such data outright. We challenge the benefits gained by asynchronous taint tracking by inserting a policy check into a regular program, such that all data will be checked against the policy and data flagged with certain taint labels will be blocked from export. We predict that the times that we require the logs to become current in order to perform the policy check will be so infrequent that the overhead of policy checking will be minimal.

In this test, we performed a scripted series of tasks, including copying a 20MB file, compressing a directory (23MB uncompressed), appending tainted data to another file, deleting files, and then attempting to export a potentially controlled file via a modified `tftp`. These tasks were carefully chosen to both explore the benefits of asynchronous taint tracking and to narrowly focus on the overhead of checking the data against a custom policy, such as, "You cannot send files with taint label *xyz*."

We performed this test in two forms: (A) `tftp` compiled from source as-is. (B) `tftp` is modified to support a semantic-aware policy checking mechanism. Six lines were added to the `tftp` source code, including a reference to the PIFT helper library. The outbound file is examined in 512B increments, with each block checked for a matching taint label. If any matching taint label is discovered, we present a notification and the file will be blocked and not sent.

In this test, we attempt to send a tainted file that will pass the policy check, so that the times for transfers will be included across all tests. We can see in Table 4 that adding policy checking increases the variance in time to completion by roughly 3×, and adds a 1% overhead to complete the task.

Environment	Policy Check?	Avg	Std Dev
Emul	N/A	19.69	0.75
PIFT-A(1GB)/T	Yes	36.72	2.44
PIFT-A(1GB)/T	No	36.35	0.91
PIFT-S/T	Yes	44.91	1.82
PIFT-S/T	No	44.38	0.52

Table 4: Evaluating the overhead of policy checking. Time (in seconds) to complete a sequence of basic tasks, with policy checking code enabled and disabled.

One should note that the version of `tftp` that performs policy checking innately requires more lines of code in order to perform the policy check and handle errors gracefully. It is possible that a developer could extend their target program with inefficient policy checking code. Since PIFT only provides the information flow tracking substrate, we cannot control how the final developer writes the policy checking code, and thus cannot estimate the final overhead for all programs under all circumstances. In this experiment, we see a very small overhead to perform the policy check, and we foresee that given enough care on behalf of the developer, future applications that leverage the information flow tracking benefits of our system will see similar overheads.

In summary, through three key improvements, PIFT provides performance benefits across all tasks when compared to prior work. For novel features, such as support of graphical environments, even though tainted data on the screen effectively forces the system to stay in emulated mode, we see encouraging results for confirming the benefit of asynchronous taint tracking. For graphically intensive tasks, we see that it will take the user no more than 2–3× longer to complete a task with taint tracking than without. For applications with minimal graphical updates (such as word processing), the impact is imperceptible. Furthermore, we see that when using asynchronous taint tracking for some CPU-heavy workloads, around 40–50% of the overhead is from the emulation and not the actual taint tracking. Thus, additional improvements to our performance in these workloads are possible through further optimization of the core emulation mechanisms used by QEMU.

Despite these advances, we discovered something troubling about our assumptions. We believed that performance was the key hindrance against adoption (solving which enabled us to test our system against a fully functional GUI), and that we had solved the kernel taint explosion problem by removing the extraneous taint labels from selected system calls (solving which enabled us to get useful results in practice). However, since we were able to test our system in practice on common desktop software, we discovered that our system was dependent on imperfect components, and these imperfections would prevent us from ever achieving our goals using this approach.

7 Imperfect components

Once we were able to get a usable system-wide information flow tracking system, we were able to explore the effectiveness of taint tracking information flow in commodity applications. When we started, we believed that *performance* was the primary hurdle preventing the adoption of IFT solutions. After our evaluation, we have decided that performance, while still important, is negligible when considered in the light of so many false-positives and false-negatives. *Current applications and operating systems consistently violate the basic tenets of information flow such that any IFT solution based on dynamic analysis is intractable.*

7.1 Implicit flows

We face the problem of *implicit flows* with the use of several common programming constructs, namely branching. Consider the following code sample:

```
int x = 0;
char ch = getchar();
if (ch == 'a')
    x = 1;
```

We can see that in this case, any taint associated with `ch` cannot be directly propagated to the variable `x` using our technique. `x` does not explicitly depend on `ch`, but could be used to leak information regarding the value of `ch`. We assume that the user is benign and will only use programs that they believe to be well-behaved, thus we do not concern ourselves with handling these cases directly. Instead, we argue that if the user or administrator determines that implicit flows such as these are essential to a program's function, he will instead opt to confine the execution of that program to a separate virtual machine. Any program that requires branching (e.g., `if` and `switch` statements) to perform its main function should be evaluated on a case-by-case and per-site basis.

Consider another case:

```
char ch = getchar();
int x = my_array[ch];
```

As described in Section 4, any taint associated with `ch` is propagated to `x`.

We seek to argue that the second case of using tainted data as an index into an array of potentially untainted data is in fact a more common technique than using branching to generate output.

Case study: `tr`

`tr` is a simple command-line text replacement tool. The basic function of this program is to replace a set of single characters in a given input string with corresponding characters, or delete those target characters entirely.

In the case of character replacement, the method is straightforward and taint-maintaining. `tr` creates an array, `string1[]` such that all 256 ASCII characters appear in the following manner: For every $a = 0..255$, where a is the numerical ASCII representation for a character, `string1[a] = a`. If a character a is due to be replaced with b, `tr` then sets `string1[a] = b`. After performing all the desired substitutions to this character index, `tr` then executes the replacement by merely reading the input string character by character, and using each character as an index into the `string1` array, creates a new string with the desired characters replaced. Since taints are propagated when using tainted data as an index to an array, we can still enforce data confinement.

```
while ((ch = getchar()) != EOF)
    (void)putchar(string1[ch]);
```

In the case of character deletion, we face the problem of implicit flows. In this case, `tr` sets the values of `string1[a]` to be either `1` or `0` indicating whether or not the character is due to be deleted. Then, upon reading the input string character by character, the new string is created by only those characters for which `string1[a] == 0`.

```
while ((ch = getchar()) != EOF)
     if (!string1[ch])
        (void)putchar(ch);
```

In this case, the taint on the deleted characters will not be propagated, due to the fact that the tainted character would only be used in the `if` statement, and as written, the tainted character will not appear in the output string.

While `tr` could be used in a manner that retained taint, one usage case could potentially leak information, and as such, we would leave it to the administrator to decide if this program should be limited to execution in a separate virtual machine.

Case study: TEX and `dvipng`

When rendering a DVI as a PNG, the process is very straight-forward. DVI commands corresponding to the numerical representation of the character, such as `SETC_127`, indicate that the program should typeset the character 127. Pixels are plotted directly from the information retrieved from the font table (e.g., `ptr=currentfont->chr[c];`), as indexed by the character's numerical representation. As such, the data retrieved from the font table is directly used to set pixels (Figure 6).

We can see that the conversion from DVI to PNG does not use any form of branching based on potentially tainted data, and will thus maintain taints.

Looking at the conversion of TEX to DVI, we have an implicit flow problem with the use of ligatures. TEX attempts to locate sequential characters that could be represented by a ligature, such as in the cases of "fi" and "ffi', and replaces them with the appropriate ligature, if supported by the desired font. Unlike with other ASCII characters, the original characters will not be transcribed directly into the output DVI file, and thus the taint for these few characters will disappear.

In summary, using an application such as `dvipng` would not be restricted, but an administrator might choose to execute `tex` and variants in separate virtual machines.

Case study: `gnuplot`

`gnuplot` is a popular open-source command-line data and function plotting utility. We explore how `gnuplot` decides which pixels to draw to the output image.

In evaluating functions, `gnuplot` uses a stack to hold intermediate values. Each incoming value is pushed to the stack, which is then popped from the stack and evaluated according to the desired mathematical function as a simple passed argument, with the output pushed back on the stack. This is easily handled with our technique for taint propagation. As such, calculating $sin(x)$ where x is tainted will result in a tainted output.

Results are stored in a struct that maintains the coordinates that will be plotted. Coordinates are converted to terminal coordinates, using the following macro code (where `variable` is the tainted value):

```
(int) ((axis_array[axis].term_lower) \
+ ((variable) - axis_array[axis].min) \
* axis_array[axis].term_scale + 0.5)
```

Taint is still propagated. The resultant values are then passed to the terminal driver.

If using X11, commands for graphing lines or plotting points are stored in a buffer, in well-defined formats, featuring the potentially tainted data directly. For example, a point would be retrieved as follows: `sscanf(buffer + 1, "%d-%d-%d", &point, &x, &y);` which would then be stored into a pixmap using the X11 function: `XDrawPoint`. This pixmap is later rendered using the X11 drivers. Advanced point representations such as diamonds and crosses are handled using simple arithmetic to create a series of segments, which are then stored in the output pixmap. Cairo performs its graphing functions similarly.

One might consider the information leaked by dynamic axes, but we find that the axes are expanded or contracted based on tracking the max and min values for each axis. As such, taint is maintained throughout. We find no instances where branching is essential to the flow of data through `gnuplot`.

7.2 Problematic programming paradigms

In our evaluation of commodity off-the-shelf applications, we discovered that some common programming paradigms severely inhibit our ability to perform useful information flow tracking. If our tolerance for errors (be they false positives or false negatives) is low, these paradigms will be found to be too common to enable correct and useful information flow tracking.

7.2.1 Shared libraries and components

For example, when exploring text editors, such as `gedit`, `medit`, and AbiWord, we discovered that merely loading a tainted file into memory would cause any subsequent file written by any other program in that set to be marked with the same taint label. What we discovered was that each of these programs used the glib+ libraries, and in particular, the *message bus* of glib had become tainted. All files written or loaded into memory by glib would gain that taint label, including user-level state files such as `~/.recently-used.xbel`, `~/gtkfilechooser.ini`, as well as several other metadata or session files. The use of common libraries and constructs such as a message bus is not that uncommon, and in fact is related to a general lack of isolation between distinct components. We believe that the GUI (and even GPU) will also be a place where IFT semantics can be abused.

```
for( y=0; y<ptr->h; y++) {
  for( x=0; x<ptr->w; x++) {
    if (ptr->data[pos]>0) {
      pixelgrey=gammatable[(int)ptr->data[pos]/2];
      bgColor = gdImageGetPixel(page_imagep, hh + x, vv + y);

      // snip set color of pixel

      gdImageSetPixel(page_imagep, hh + x, vv + y, pixelcolor);
    }
    pos++;
  }
}
```

Figure 6: **Case study:** TEX and `dvipng`

7.2.2 Branching

Another common programming paradigm is that of using switch statements. Unicode translation is such an example of branching that evades taint propagation. Mentioned as early as 2004, the researchers who developed TaintBochs discovered that the Windows 2000 kernel uses this form of branching to translate keyboard scancodes into unicode [6]. As such, they discovered that a tainted password would appear in memory in both a tainted form (scancode) and an untainted form (unicode). Researchers continued to experience this particular problem in Taint Eraser [76] and [30], among others. As of yet, this problem has not been resolved in any dynamic analysis system. Panorama [69] addressed this exact problem in Windows XP by specially instrumenting an instruction within the function _xxxInternalToUnicode.

While we used the technique of taint scrubbing or whitelisting specific system calls that we manually verified to not mandate the propagation of taint labels, since this pattern appears in the desktop applications as well, we find that we are at an impasse. Since we asserted that we are not allowed to change the desktop applications in order to provide IFT, we are unable to proceed, and consequently any technique with similar goals will also remain fruitless.

7.3 Lack of GUI isolation

The lack of GUI isolation has been known to be a security concern for many years [41, 42]. An example of this particular concern can be explored by the reader through `xinput`. A short recipe is as follows:

1. Install `xinput`.
2. *As user*, attach and run `xinput` to the system's keyboard
3. Open a terminal window as root
4. Type into the root-owned terminal window

The user terminal running `xinput` will reveal all keys entered in that root-owned terminal window.

The general theme of *lack of isolation* is a troubling one. The lack of isolation enables attackers to cross these boundaries between systems with potentially differing policy demands.

7.4 Additional threats

As the above examples demonstrate, information leakages happen because current systems provide a myriad number of channels through which data can be easily pilfered or lost. PIFT aims to prevent such leakage by allowing administrators to tag sensitive data, track it, intercept and enforce security policies on all exchanges between principals. However, we do not expect users to be malicious. An adversarial insider can always photograph a screen with sensitive data, an attack that any software based approach cannot defend against, yet we allow users to be careless. Users can access untrusted sites, download potentially unsafe external applications, and use all the normal means of communication they have come to expect (email, peripherals, etc).

PIFT treats the hypervisor, emulator and backend drivers as part of the trusted computing base, but it makes no assumptions about the operating systems or applications. Second, we assume that at startup, the hypervisor can be securely loaded before any other code executes. With the advent and wide availability of trusted computing hardware in computers, such an assumption is feasible. Finally, we assume that the existence of a key infrastructure within the trusted boundary, so hypervisors can sign and verify messages from each other.

8 Conclusion

We began seeking to address the problem of data exfiltration. Common approaches using information flow tracking seemed to be promising, with the biggest hindrance to adoption being the dramatic overhead incurred due to the taint tracking techniques used. We saw three possible areas for improvement: Track taint labels at a higher abstraction level, namely at the native instruction level of the protected VM; Perform taint tracking asynchronously, only evaluating the IFT metadata as needed according to the policy demands; and eliminate kernel taint explosion by separating the system calls into those which *should* logically propagate the taint labels vs. those that should not. These three changes proved to be those which decreased our performance overhead to a level that enabled real users to interact with a IFT-aware GUI-based commodity applications and operating system (with usable results), a feat we believe to be the first of its kind.

It is because of this new ability that we were able to evaluate the technique of taint tracking as a means to achieve information flow tracking on commodity applications and operating systems. What we discovered was that early results expressing concern about taint explosion where founded, but not sufficient in their scope. We discovered that our legacy applications and operating systems are imperfect components that rely on techniques that fundamentally violate the notions of information flow, particularly as revealed by dynamic analysis. If we seek to provide information flow tracking or information flow control to extant application binaries, we will most likely fail.

So what is a researcher to do? We need to think about information flow tracking at all stages of our design. Support for IFT needs to be built natively into the programming languages, with the needs of the programmer in mind, and IFT needs to be supported by the (trusted) hardware, and the operating system. More importantly, we need to enforce and support isolation. As our systems become more complex, it becomes even more important to sandbox and isolate foreign or untrusted code. Recent efforts to natively support isolation in the operating system as a security measure have shown great promise.

For future work, we recommend: Evaluating the CPU changes with the most potential for assisting with IFT; and exploring IFT only on the messages sent between isolated virtual machines, each of which enables legacy or suspect software to run unhindered and unattended. In the latter case, it could be conceivable to use programming language techniques to implement the IFT between the isolated components. We believe that isolation and programming language-based techniques for information flow tracking will be the correct path to achieving practical information flow tracking on a daily basis.

9 Acknowledgements

With sincere gratitude to Scott Shenker, Sachin Katti, Andrei Ermolinskiy, and James Murphy McCauley, without whose advice and hard work this would have been impossible; and to Kevin Jeffay for his constant support and encouragement. This work was funded in part by the National Science Foundation and the University of California.

References

[1] G. R. Andrews and R. P. Reitman and. “An Axiomatic Approach to Information Flow in Programs”. In: *ACM Trans. Program. Lang. Syst.* 2 (1 1980), pp. 56–76.

[2] H. Barwick. *Identity theft, e-fraud top Australian security concerns: Unisys.* Computerworld. May 2011. URL: `http://www.computerworld.com.au/article/385356/identity_theft_e-fraud_top_australian_security_concerns_unisys/`.

[3] T. Bradley. *Sony Hacked Again: How Not to Do Network Security.* PCWorld Business Center. June 2011. URL: `http://www.pcworld.com/businesscenter/article/229351/sony_hacked_again_how_not_to_do_network_security.html`.

[4] S. Chen. “Defeating memory corruption attacks via pointer taintedness detection”. In: *In IEEE International Conference on Dependable Systems and Networks (DSN.* 2005, pp. 378–387.

[5] E. Chin and D. Wagner and. “Efficient character-level taint tracking for Java”. In: *Proceedings of the 2009 ACM workshop on Secure web services.* SWS ’09. Chicago, Illinois, USA: ACM, 2009, pp. 3–12.

[6] J. Chow. “Understanding data lifetime via whole system simulation”. In: *SSYM’04.* San Diego, CA, 2004, pp. 22–22.

[7] J. Clause and A. Orso and. “Penumbra: automatically identifying failure-relevant inputs using dynamic tainting”. In: *Proceedings of the eighteenth international symposium on Software testing and analysis.* ISSTA ’09. Chicago, IL, USA: ACM, 2009, pp. 249–260.

[8] G. Cluley. *University of Florida warns students and staff of security breach.* Sophos. Feb. 2009. URL: `http://www.sophos.com/blogs/gc/g/2009/02/20/university-florida-warns-students-staff-security-breach`.

[9] *Common vulnerabilities and exposures (CVE) database*. The MITRE Corporation. URL: `http://cve.mitre.org/data/downloads/`.

[10] M. Dalton, H. Kannan, and C. Kozyrakis, and. "Tainting is not pointless". In: *SIGOPS Oper. Syst. Rev.* 44.2 (2010), pp. 88–92.

[11] M. Dalton, H. Kannan, and C. Kozyrakis, and. "Raksha: a flexible information flow architecture for software security". In: *Proceedings of the 34th annual international symposium on Computer architecture*. ISCA '07. San Diego, California, USA: ACM, 2007, pp. 482–493.

[12] D. E. Denning. "A lattice model of secure information flow". In: *Commun. ACM* 19.5 (1976), pp. 236–243.

[13] D. E. Denning and P. J. Denning and. "Certification of programs for secure information flow". In: *Commun. ACM* 20.7 (1977), pp. 504–513.

[14] P. Dhoolia. "Debugging model-transformation failures using dynamic tainting". In: *Proceedings of the 24th European conference on Object-oriented programming*. ECOOP'10. Maribor, Slovenia: Springer-Verlag, 2010, pp. 26–51.

[15] W. Enck. "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. Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–6.

[16] A. Ermolinskiy. *Towards Practical Taint Tracking*. Tech. rep. UCB/EECS-2010-92. EECS Department, University of California, Berkeley, 2010. URL: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-92.html`.

[17] L. Greenemeier and C. Q. Choi and. *WikiLeaks Breach Highlights Insider Security Threat*. Scientific American. Dec. 2010. URL: `http://www.scientificamerican.com/article.cfm?id=wikileaks-insider-threat`.

[18] A. Ho. "Practical taint-based protection using demand emulation". In: *SIGOPS Oper. Syst. Rev.* 40.4 (2006), pp. 29–41.

[19] S. R. Hunt. *Symantec leaks credit card data*. APC Magazine. Mar. 2009. URL: `http://apcmag.com/symantec-says-credit-card-data-could-have-leaked-from-india.htm`.

[20] *Indian companies find data loss biggest security concern*. CXO today. Nov. 2009. URL: `http://www.cxotoday.com/story/indian-cos-find-data-loss-biggest-security-concern/`.

[21] H. Kannan, M. Dalton, and C. Kozyrakis, and. "Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor". In: *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*. 2009, pp. 105 –114.

[22] M. Krohn and E. Tromer and. "Noninterference for a Practical DIFC-Based Operating System". In: *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 61–76.

[23] M. S. Lam and M. Martin and. "Securing web applications with static and dynamic information flow tracking". In: *In ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation*. 2008, pp. 3–12.

[24] H. Max and T. Ray and. *Skype: The Definitive Guide*. Que, 2006.

[25] *More than half of ex-employees admit to stealing company data according to new study*. Symantec Corp and the Ponemon Institute. Feb. 2009. URL: `http://www.symantec.com/about/news/release/article.jsp?prid=20090223_01`.

[26] Y. Mundada. *Practical data-leak prevention for legacy applications in enterprise networks*. Tech. rep. GT-CS-11-01. Georgia Tech, 2011.

[27] A. C. Myers. "JFlow: practical mostly-static information flow control". In: *POPL*. San Antonio, Texas, United States, 1999, pp. 228–241.

[28] A. C. Myers and B. Liskov and. "A decentralized model for information flow control". In: *SOSP*. Saint Malo, France, 1997, pp. 129–142.

[29] S. K. Nair. "A virtual machine based information flow control system for policy enforcement". In: *Electron. Notes Theor. Comput. Sci.* 197.1 (2008), pp. 3–16.

[30] J. Newsome and D. X. Song and. "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software". In: *NDSS*. 2005.

[31] J. Newsome. "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software". In: 2005.

[32] A. Nguyen-tuong. "Automatically Hardening Web Applications Using Precise Tainting". In: *In 20th IFIP International Information Security Conference*. 2005, pp. 372–382.

[33] E. B. Nightingale. "Parallelizing security checks on commodity hardware". In: *ASPLOS XIII*. Seattle, WA, USA: ACM, 2008, pp. 308–318.

[34] D. A. Norman. "When security gets in the way". In: *interactions* 16 (6 2009), pp. 60–63.

[35] I. Papagiannis. "PrivateFlow: decentralised information flow control in event based middleware". In: *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. DEBS '09. Nashville, Tennessee: ACM, 2009, 38:1–38:2.

[36] *Perl Security*. Perl. URL: `http://perldoc.perl.org/perlsec.html`.

[37] *Personal information: data breaches are frequent, but evidence of resulting identity theft is limited; however, the full extent is unknown*. US Government Accountability Office. June 2007. URL: `http://www.gao.gov/products/GAO-07-737`.

[38] L. Phifer. *Top 10 data breaches of 2010*. eSecurity Planet. Jan. 2011. URL: `http://www.esecurityplanet.com/features/article.php/3921656/Top-10-Data-Breaches-of-2010.htm`.

[39] T. Pietraszek. "Defending against Injection Attacks through Context-Sensitive String Evaluation". In: *In Recent Advances in Intrusion Detection (RAID*. 2005.

[40] M. Pistoia. *TAJ: Effective Taint Analysis of Web Applications*. 2009.

[41] A. Pretschner. "Usage Control Enforcement with Data Flow Tracking for X11". In: *Proc. 5th Intl. Workshop on Security and Trust Management (STM)*. 2009.

[42] *Qubes OS*. Invisible Things Lab. 2011. URL: `http://qubes-os.org/`.

[43] F. Y. Rashid. *Most websites regularly leak sensitive, personal data: survey*. eWeek.com. June 2011. URL: `http://www.eweek.com/c/a/Security/Most-Web-Sites-Regularly-Leak-Sensitive-Personal-Data-Survey-640359/`.

[44] D. Raywood. *University College Berkeley hit by second data breach in six months as details of almost 500 applicants are hacked*. SC Magazine. 2009. URL: `http://www.scmagazineuk.com/University-College-Berkeley-hit-by-second-data-breach-in-six-months-as-details-of-almost-500-applicants-are-hacked/article/146593/`.

[45] T. Ristenpart. "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds". In: *Proceedings of the 16th ACM conference on Computer and communications security*. CCS '09. Chicago, Illinois, USA: ACM, 2009, pp. 199–212. URL: `http://doi.acm.org/10.1145/1653662.1653687`.

[46] O. Ruwase. "Parallelizing dynamic information flow tracking". In: *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*. SPAA '08. Munich, Germany: ACM, 2008, pp. 35–45.

[47] H. J. Saal and I. Gat and. "A hardware architecture for controlling information flow". In: *Proceedings of the 5th annual symposium on Computer architecture*. ISCA '78. New York, NY, USA: ACM, 1978, pp. 73–77. URL: `http://doi.acm.org/10.1145/800094.803030`.

[48] E. J. Schwartz, T. Avgerinos, and D. Brumley, and. "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask". In: *In Proceedings of the IEEE Symposium on Security and Privacy*. 2010.

[49] *Sensitive TSA manual posted on web*. USA Today. Dec. 2009. URL: `http://www.usatoday.com/travel/flights/2009-12-08-airport-security_N.htm`.

[50] N. Shachtman. "Under Worm Assault, Military Bans Disks, USB Drives". In: (2008). URL: `http://www.wired.com/dangerroom/2008/11/army-bans-usb-d/`.

[51] A. Slowinska and H. Bos and. "Pointless tainting?: evaluating the practicality of pointer tainting". In: *EuroSys '09*. Nuremberg, Germany, pp. 61–74.

[52] K. Z. Snow. "SHELLOS: enabling fast detection and forensic analysis of code injection attacks". In: *Proceedings of the 20th USENIX conference on Security*. SEC'11. San Francisco, CA: USENIX Association, 2011, pp. 9–9.

[53] G. E. Suh. "Secure program execution via dynamic information flow tracking". In: *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*. ASPLOS-XI. Boston, MA, USA: ACM, 2004, pp. 85–96.

[54] B. Sullivan. *Government agency exposes day-care data*. 2010. URL: `http://www.msnbc.msn.com/id/4186130/`.

[55] *Survey: SMBs are getting serious about information protection*. Symantec. 2010. URL: `http://www.symantec.com/about/news/resources/press_kits/detail.jsp?pkid=smbsurvey2010`.

[56] K. Thomas. *Microsoft Cloud Data Breach Heralds Things to Come*. 2010. URL: `http://www.pcworld.com/businesscenter/article/214775/microsoft_cloud_data_breach_heralds_things_to_come.html`.

[57] T. Thorsen. *PSN data leak cost could top $24 billion*. Gamespot. Apr. 2011. URL: `http://www.gamespot.com/news/6310436.html`.

[58] M. Tiwari. "Complete information flow tracking from the gates up". In: *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*. ASPLOS '09. Washington, DC, USA: ACM, 2009, pp. 109–120.

[59] *Trend Micro hit by massive Web hack*. InfoWorld. 2008. URL: `http://www.infoworld.com/d/security-central/trend-micro-hit-massive-web-hack-212`.

[60] J. Tucek. “Sweeper: a lightweight end-to-end system for defending against fast worms”. In: *SIGOPS Oper. Syst. Rev.* 41.3 (2007), pp. 115–128.

[61] *University of California hit by data breach that may affect 160,000 past and present students.* SC Magazine. 2009. URL: `http://www.scmagazineuk.com/University-of-California-hit-by-data-breach-that-may-affect-160000-past-and-present-students/article/136499`.

[62] N. Vachharajani. “RIFLE: An architectural framework for user-centric information-flow security”. In: *In MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, 2004, pp. 243–254.

[63] S. Vandebogart. “Labels and event processes in the Asbestos operating system”. In: *ACM Trans. Comput. Syst.* 25.4 (2007).

[64] G. Venkataramani. “FlexiTaint: A programmable accelerator for dynamic taint propagation”. In: *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on.* 2008, pp. 173 –184.

[65] T. Wang. “TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection”. In: *Security and Privacy, IEEE Symposium on* 0 (2010), pp. 497–512.

[66] M. Weiser. “Program slicing”. In: *Proceedings of the 5th international conference on Software engineering.* ICSE ’81. San Diego, California, United States: IEEE Press, 1981, pp. 439–449.

[67] C. Willems, T. Holz, and F. Freiling, and. “Toward Automated Dynamic Malware Analysis Using CWSandbox”. In: *IEEE Security and Privacy* 5 (2007), pp. 32–39.

[68] W. Xu, E. Bhatkar, and R. Sekar, and. “Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks”. In: *In 15th USENIX Security Symposium.* 2006, pp. 121–136.

[69] H. Yin. “Panorama: capturing system-wide information flow for malware detection and analysis”. In: *CCS ’07.* Alexandria, Virginia, USA, pp. 116–127.

[70] A. Yip. “Improving Application Security with Data Flow Assertions”. In: *SOSP.* Big Sky, Montana, 2009.

[71] A. R. Yumerefendi, B. Mickle, and O. P. Cox, and. “TightLip: Keeping applications from spilling the beans”. In: *In Proc. 2007 NSDI.* 2007.

[72] N. Zeldovich. “Hardware Enforcement of Application Security Policies Using Tagged Memory”. In: *OSDI.* 2008, pp. 225–240.

[73] N. Zeldovich. “Making information flow explicit in HiStar”. In: *OSDI.* Seattle, Washington, 2006, pp. 263–278.

[74] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières, and. “Securing distributed systems with information flow control”. In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation.* NSDI’08. San Francisco, California: USENIX Association, 2008, pp. 293–308.

[75] Q. Zhang. “Neon: system support for derived data management”. In: *VEE ’10.* Pittsburgh, Pennsylvania, USA, 2010, pp. 63–74.

[76] D. Y. Zhu. “TaintEraser: protecting sensitive data leaks using application-level taint tracking”. In: *SIGOPS Oper. Syst. Rev.* 45 (1 2011), pp. 142–154.