# Information Flow Auditing In the Cloud

Angeliki Zavou

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2015

# ABSTRACT

## Information Flow Auditing In the Cloud

Angeliki Zavou

As cloud technology matures and trendsetters like Google, Amazon, Microsoft, Apple, and VMware have become the top-tier cloud services players, public cloud services have turned mainstream for individual users. Many companies and even governments are also adopting cloud services as a solution to reduce costs and improve the quality of their services. However, despite the appealing benefits of cloud technologies, inherent in the concept of cloud computing are also the risks associated with entrusting confidential and sensitive data to third parties, especially in the case of security-sensitive operations, i.e., banking, medical services. Therefore, it comes as no surprise that the most-often cited barriers against cloud computing are cloud users' *lack of trust* regarding *data confidentiality* as well as the risk of *unauthorized exposure and manipulation of sensitive user data* within the cloud, since such incidents cause catastrophic damages to the business interests of an organization, as well as affect the privacy individuals are entitled to.

Most cloud providers and service providers (e.g., Amazon, Google, Dropbox etc.) do take security precautions for protecting users' data and use service level agreements (SLAs) as a means to promise, among other features, availability, reliability and compliance with privacy standards (e.g., HIPPA, PCI-DCS, FISMA etc.). But very limited tools are currently available to cloud users to evaluate their effectiveness and incidents abound that fuel scepticism and distrust towards cloud computing. The many examples of security breaches in major cloud services, that reach the press from time to time, show that despite SLAs and good intentions from the providers' side, protection against data leakage remains a challenging task that needs to be addressed carefully to take full advantage of the cloud computing potential. In this setting, and in lack of a better alternative, other than not using cloud services at all, cloud users have to blindly trust the fate of their data to the

best efforts of service providers to achieve the promised security guarantees.

This dissertation aims to address security issues and concerns that affect cloud-hosted web services, whose providers do not have malicious intentions but which may be composed by buggy or misconfigured software, vulnerable to attacks and accidental data leaks. My approach was inspired by the observation that cloud users' security concerns could be alleviated if the SLAs between cloud services' providers and their users could be verifiable, at least to some extent. More specifically, since the verification of adherence to security constraints within cloud services is a very challenging and formidable task, users would benefit if they were offered the tools to monitor the high-level service behavior and at least promptly discover when the security measures are failing. Another important premise of our approach is that cloud providers and service providers faced with the requirement to satisfy customer security concerns, have the incentive to both make real investments in improving the security of their portion of the technology stack as well as incorporate the proposed techniques to their services. It is in their best interest to provide best security practises to maintain reputation as their business depends on this.

In this work, I propose a set of techniques that can be used as the basis for alleviating cloud customers' privacy concerns and elevating their confidence in using the cloud for security-sensitive operations as well as trusting it with their sensitive data. The main goal is to provide cloud customers' with a reliable mechanism that will cover the entire path of tracking their sensitive data, while they are collected and used by cloud-hosted services, to the presentation of the tracking results to the respective data owners. In particular, my design accomplishes this goal by retrofitting legacy applications with data flow tracking techniques and providing the cloud customers with comprehensive information flow auditing capabilities. For this purpose, we created CloudFence, a cloud-wide fine-grained data flow tracking (DFT) framework, that allows service providers to monitor and log the use of sensitive data in well-defined domains, offering additional protection against inadvertent leaks and unauthorized access. To achieve cloud-wide data tracking for legacy application without demanding emulation we built TaintExchange, a generic cross-process and cross-host DFT system, which was incorporated in the CloudFence framework. Besides cloud-wide information flow tracking for the service providers, we also built Cloudopsy, a service

that allows users to independently audit and get a better understanding of the treatment of their cloud-resident private data by the third-party cloud-resident services, through the intervention of the cloud infrastructure provider that hosts these services. While Cloudopsy is targeted mostly towards end users, it provides also the online service providers with an additional layer of protection against illegitimate data flows, e.g., inadvertent data leaks, through a graphical more meaningful representation of the overall service dependencies and the relationships with third-parties outside the cloud premises, as they derive from the CloudFence-generated audit logs. Experimental results are presented to support the effectiveness of these techniques. The results of my evaluation demonstrate the ease of incorporating these techniques on existing real-world applications, their effectiveness in preventing a wide range of security breaches, and their performance impact on real settings.

# Table of Contents

# List of Figures

# List of Tables

# Acknowledgments

First and foremost, I would like to thank my advisor, Angelos D. Keromytis, for enabling this thesis to even exist. I am deeply grateful to him for giving me the opportunity to be here and explore the world of research and security in particular, and for teaching me how to ask questions, identify interesting problems for my research, appreciate sushi, but most of all for challenging me everyday to become better.

I am also very grateful to Georgios Portokalidis, who guided me at my first steps in research, taught me that working hard will eventually pay off, and who remained an encouraging mentor, collaborator and friend throughout the PhD odyssey. I heartily thank also the other members of my defense committee, Steve Bellovin, Sal Stolfo, and Stelios Sidiroglou-Douskos for teaching me the security fundamentals throughout the years but also for their valuable feedback and support during my defense and this thesis writing and of course for getting me past “the finish line”. This thesis would not have been complete without their valuable feedback and their meticulosity and therefore I am sincerely grateful to them.

I am especially thankful to all my officemates throughout these years, Stelios, Mike Locasto, Brian Bowen, Vasilis Kemerlis, Dimitris Geneiatakis, Elias Athanasopoulos and Christopher Dall, who made going to the office so much more enjoyable. I am also thankful to all members of the Network Security Lab and especially Mariana Raykova, Hang Zhao, Mike Polychronakis, Georgios Kontaxis, Vasilis Pappas, Sambuddho Chakravarty, and Kangkook Jee for their, advice, friendship and support. Many of them have provided valuable feedback on my ideas and their comments have been useful in refining my research and publications.

But this journey would not have been possible without some people very important to me. I will always cherish the support and encouragement of Petros Mol, Christine

Karastathi, Elli Androulaki and Alexandros Iliadis, who made my phd life so much better even from far away. Our experiences, discussions and the laughs we shared even at very difficult times all these years were the reasons I managed to stay sane and complete this journey.

Last but not least, I would like to thank my parents, Polyxeni and Alexandros Zavos, who made me the person I am today and who taught me the value of education and instilled in me the desire to strive for the best. They, along with my brother, Dimitris, have always been my strongest supporters throughout my PhD journey and I am deeply indebted to them for their humor at hard times, and for their endless love and unwavering faith in me. Without them I wouldn't be able to make my dream come true and therefore I dedicate this thesis to my family.

*To my strongest supporters, my family ...*

# Chapter 1

# Introduction

Cloud computing has inarguably received enormous attention in the recent years from both businesses and individual users. At an unprecedented pace, the cloud service model is becoming the new paradigm for deploying software services for businesses, government, and individuals in the form of public clouds. Indeed, the world's largest technology trendsetters, e.g., Amazon, Google, Microsoft, IBM, and Apple have made the long-held dream of "*using computing as a public utility*" a reality. Many companies and even governments are adopting the cloud services as a solution to reduce costs and improve the quality of their services. Cloud-based services are also gaining traction among individual users. Applications such as Dropbox, Google Docs, iCloud, Vine, and Instagram have become an integral part of many people's daily lives, especially through the use of smartphones and tablets.

In today's interconnected and competitive world, cloud technologies provide dynamism, elasticity and such a range of services, that makes them too attractive for enterprises to ignore. Therefore, as cloud technology matures, and businesses and individuals increasingly rely on the cloud, an increasing number of critical applications will also be deployed and operated in these computational environments. As a result, some of their *private (or mission-critical) data* is handled and stored on the cloud systems outside of their administrative control. Access credentials, social security and credit card numbers, private files, and other sensitive data is temporarily or permanently stored in back-end databases and file systems, beyond their owners' control.

In reality, the cloud IT paradigm is a greater cultural transformation for enterprises rather than for individuals users, as the shift from server- to service-based IT requires the consigning of their computing services outside the enterprise's perimeter to the *cloud providers*. Since cloud providers are separate administrative entities than *cloud users*, including both individual users and enterprises, it is of critical importance to realize that the use of the cloud infrastructure and cloud services implies **diminishing control** over the data that are shared in the realms of these services. One fundamental aspect of this shift is that sensitive and mission-critical data are also outsourced to the cloud, which implies the inevitable (at least partial) release of control over its fate, which enterprises often find quite alarming. Therefore, although they may be willing to make certain compromises in order to lower costs and improve their services, they do not seem to be equally comfortable relinquishing control over their sensitive data to the third-party providers of cloud environments.

## 1.1 The Problem

Recent surveys [32; 33] found that **trust**, not money, will ultimately determine cloud computing's growing adoption and success. Trust is generally related to "*levels of confidence in something or someone*". Hence by trust in the cloud computing context, we refer to the customers level of confidence in using the cloud services, which is based on several factors, the most important of which are the **degree of control**, the **level of transparency** and the **provider's reputation**. In particular, the level of control clients retain over their valuable data assets as well as the degree of transparency they have over the services that handle them can affect how much trust they have in the provider and the service itself, which will seems to be the main factor for the future uptake of cloud services.

Moving confidential data to the cloud does not remove the requirement for its protection with the same diligence as before, if not more, and naturally poses concerns for their owners. This transfer outside of the secured corporate perimeter increases the complexity of protecting this data as well as the risk of compromise. Therefore, it is no surprise that the **inherent loss of control** over sensitive (or mission-critical) data in the cloud, due to its third-party nature, and consequently the **risk of unauthorized access** to it are cited [35;

47; 7] as the primary inhibitors to further adoption of cloud-based services, especially for security-sensitive operations. Also contributing to the exacerbation of this problem is the **lack of transparency** inside the cloud processing sites, which leads to the perception that the cloud is less secure than an in-house system. The latest Cloud Security Alliance's (CSA) report [7] on the most significant vulnerabilities in cloud computing identified **data breaches** as the most critical threat currently (in order of severity), moving up five places in the corresponding ranking of 2010. These concerns continue unabated as the number and frequency of real cases of security breaches at major cloud services [67; 22; 63; 12], that gets publicized in the media, increases. In 2011, hackers broke into the Sony PlayStation Network [30], exposing usernames, passwords, credit card details, security answers, purchase history and addresses of 77 million people around the world. Later in the same year, the cloud-storage provider Dropbox admitted that a bug in their authentication mechanism had disabled password authentication [73]. Hence, temporarily visitors were allowed to log in to any other of Dropbox's 25 million customers' accounts using any password or none at all. More recently, Adobe [62] suffered one of the worst security breaches in recent times, as almost 150 million "breached records" from a database of Adobe user data has turned up online at a website frequented by cyber criminals. And the predictions are not very optimistic regarding similar events in the future.

Briefly, such incidents where data of sensitive nature is exposed to the public, inadvertent or not, happen with astonishing regularity and the damages caused to companies and individuals are estimated to be in the range of millions of dollars. But apart from the direct financial damage these incidents also lead to catastrophic damages to the overall business reputation as well as affect the privacy individuals are entitled to. Consequently, especially in the cloud setting the concerns over **sensitive data confidentiality** become even more persistent and lead to strong hesitation from both enterprises and privacy-aware individuals regarding trusting their mission-critical and/or sensitive user data to such an abstract and opaque entity as the cloud.

Another significant conclusion that is drawn from the frequency of such incidents is that personally identifiable and sensitive information (PII) have become a target for attackers seeking financial gain through the misuse of such information. Cloud infrastructures represent a tempting and highly lucrative target for attackers with such incentives, given the concentration of services and data, often from different entities into a single location. Data leaks are often associated with malicious intent originating from individuals or organizations that aim to exfiltrate information of some value (e.g., trade secrets, personal sensitive data). Adversarial scenarios aside, in reality, about half of the data breaches reported (PII disclosure included) are unintentional, caused by vulnerabilities in applications software (e.g., bugs, sloppy administration practices or other operational problems) or inadequate security measures. Hence, the security measures in cloud computing infrastructures should be higher than those in traditional computing.

### 1.1.1 Misconceptions about cloud security

While there is probably some basis to users' expectation that the unified and concentrated administration and management in cloud computing environments as well as their specialized expertise enables the use of the latest sophisticated and usually expensive security measures, implying some form of security assurance in comparison to the (in)security of enterprise networks, unfortunately this does not always reflect the real world, especially for issues related to **data security** and **compliance with privacy policies**. There are at least three different delivery service models (Infrastructure-as-a-Service, Platform-as-a-Service, and Software-as-a-Service) in the cloud and depending on the delivery model used, the responsibility of security provisioning changes. The optimistic expectation regarding cloud security may be closer to reality in the case of SaaS services (e.g., Google Docs, Microsoft Office 365, Salesforce applications etc..), where the service provider is responsible for the whole software stack and the access to the application, including security, availability, and performance. Especially in the case of a reputable service provider, e.g., Google or Microsoft, that owns the resources and the specialized expertise needed to offer better security solutions. But even in this best case scenario, reality has proven that security glitches might still happen even when the provider controls and hosts a sophisticated stack. An illustrative example of such a security failure is the Google Docs' security glitch [66; 67] in 2009, where users had inadvertently shared some of their documents with contacts who were never granted access to them. The cloud security expectations are even further from the reality in the cases of the IaaS and PaaS delivery models, where the security responsibility lies more on the cloud customer's efforts.

Therefore, although the hosting setting has shifted from enterprises' servers to the cloud, which usually implies a more secure hosting environment, the common security vulnerabilities, pre-existing in the server-based web applications (e.g., cross-site scripting, injection flaws etc.), do not automatically disappear but on the contrary remain a significant problem. Experience has shown that software bugs and misconfigurations, which can be abused for acquiring unauthorized access to data, are inevitable. There is no such thing as bug-free code, and despite the fact that formal verification of software and systems has demonstrated some promising results [37], it has yet to be adopted by commercial off-the-self (COTS) software. Part of this is due to the significant scalability issues that formal verification faces (in terms of code size and system complexity), leaving aside the fact that it cannot handle legacy applications.

Altogether, the **responsibility** for the **application-level security** lies in the best efforts of the service providers, i.e., the application developers, who especially in the cloud context can be practically anyone and is not unlikely he might be lacking the knowledge, experience or incentive to build secure applications. Consequently, *when the development of web applications, cloud-hosted or not, is done with little or no security in mind, the presence of software security holes is still not mitigated.*

### 1.1.2 Current Measures

A common approach for dealing with data leaks, and degrading the impact of such incidents, is to require the important data in an encrypted form on the cloud side. Even though the encryption may help with the problem of secure storage in the cloud, it does not solve the security issues of remote processing of data inside the cloud. There are many other important security issues raised regarding management of encryption keys, processing the encrypted data in the cloud etc.. As a result, providers do not uniformly offer encryption as a solution for the data confidentiality concerns.

Another obvious step is to articulate clear policies that circumscribe the ways in which the sensitive data can be used. Most cloud providers and service providers (e.g., Amazon, Google, Dropbox etc.) take security precautions for protecting users' data and use service level agreements (SLAs) and third-party certifications as a means to promise, among other features, availability, reliability and compliance with privacy standards (e.g., HIPPA, PCI-DCS, FISMA, EU Data Protection Directive etc.). But these claims and legal agreements of a "secure cloud" are not sufficient for boosting clients' confidence. Their effectiveness is usually unproven and incidents abound that fuel scepticism and distrust towards cloud computing. Currently, the technological reality of cloud computing provides very limited tools to cloud clients to monitor how providers are achieving this and verify compliance with the promised security guarantees. In a a typical business environment, the customer is compensated according to the terms included in the signed SLA, in case the service is not delivered as expected. But in the cloud computing environments the verifiability of the promised security guarantees is at least challenging (if not impossible) and for most enterprises, a security breach resulting in the inadvertent exposure of their valuable assets, i.e., confidential data, is irreparable, because no amount of money can guarantee to restore the enterprise's reputation.

Other techniques also fall short of the requirements for cloud computing in different ways. Since the cloud platforms are general-purpose platforms, the solution should be able to accommodate all kind of applications that the cloud customers decide to run on cloud premises. This rules out application-specific techniques used in the traditional server model, that would require apart from software modifications also knowledge of the specific requirements of application internals.

Many users in lack of an alternative option (other than not using cloud services at all), eventually trust the service provider to properly handle their private data. Many companies though continue testing the waters with non-critical projects and still keep business-critical operations and data in-house. Unfortunately, relying solely on reputable service providers, legal agreements and the implied economic and reputational harm as a motive for service trustworthiness does not mitigate the risks. That being the case, the question that remains to be answered is what we can do to address this trust challenge and boost customer's confidence in cloud computing.

## 1.2 Research Approach

The adoption of cloud computing came before the appropriate technologies appeared to tackle the accompanying challenges of trust. To ease the tension between user data protection and the rich computation offered in the cloud and determine which is the best solution to address this problem and alleviate the security concerns, we explored the trust issues associated with this IT paradigm from both a technology and a business perspective.

Generally, trust is more of a social concept than a technological one, but we believe that technology could work as the base for building clients' trust towards the cloud services. It is not that much that the cloud clients do not trust the intentions of the service- and the cloud providers, but more that they *lack confidence that the cloud services will behave or deliver as promised*. In order to increase this confidence, there are both preventive and detective measures. Verifying the correct functioning of a subsystem and the effectiveness of security controls may not be feasible for the complex cloud-hosted services, however, and therefore focusing on increasing the accountability and transparency via less preventive approaches may be more appropriate to establish a level of trust. Therefore, instead of trying to fortify the software that operates on private user information [74], or striving to enforce data and network isolation [48], we propose a *data-centric* security approach for the cloud setting, as the sensitive data is the valuable aspect that needs to be protected in this setting.

Our approach is also build upon the observation that clients tend to trust a system less if it gives insufficient information about its internal procedures. Cloud services are too opaque, like a black-box, which certainly exacerbates users' uneasiness regarding the fate of their data after it enters the cloud premises. Therefore, users could benefit from knowing to what extent a cloud service delivers the promised security measures against such privacy risks over their data. More concretely, users want to have an insight and assurances (more than a signed legal agreement) that their data is actually used as expected by the cloud-hosted services, and also be offered the tools to identify incidents when this does not happen. Therefore, in this dissertation we claim that by providing auditing capabilities at a sufficient granularity and across the whole cloud infrastructure, would be a step towards easing the formidable user concerns regarding confidentiality breaches, information leaks and cloud transparency.

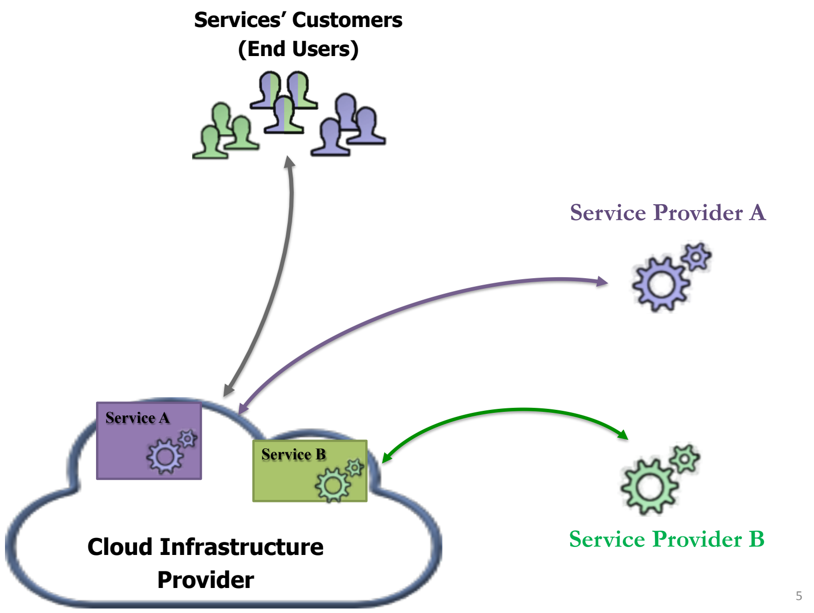


**Figure 1.1:** Service providers **A** and **B** run their respective services on the same cloud infrastructure, which are accessed by their customers (end-users) over the Internet.

### 1.2.1 Scope and Target Applications

The cloud means too many different things to have one solution, that could solve all security and privacy issues related to it. Figure 1.1 illustrates the target applications and the cloud computing model we are concerned with in this dissertation. Assume a service provider **A** is interested in running its service on the cloud that will be accessed by a group of end-users. For this purpose, service provider **A** will use a number of machines from the cloud provider (IaaS or PaaS model). Service provider **B**, which also wants to run its service on the cloud will act similarly with service provider **A**. The end-users can access both of these services via the Internet. Note that we chose to present a more complex scenario in Figure 1.1 but our approach would also be applicable in the case when the service provider and the cloud infrastructure are the same entity (e.g., Google and its various cloud services).

We chose to focus on an important class of widely-used cloud services, that meet the following criteria:

- applications that are available to their customers through websites,
- applications that provide services to a large number of distinct end users,

- applications that may consist of one or more components (e.g., web-server, database server etc.),
- applications whose developers (service providers) could be the same or different from the cloud infrastructure providers,
- applications whose developers do not have malicious intentions

### 1.2.2 Security Model

We assume **benign cloud infrastructure providers**, who recognize that having a *reputation for security* will attract customers and will be the key factor in determining the dominant players in the future of cloud computing. Our goal is to support **benign service providers**, who are willing to enhance the security of the provided services. While we do not expect that the applications are outright malicious, we assume that their complex software will very likely have **bugs or security vulnerabilities**. Note that his situation is typical for most cloud-based services:

- Most public cloud services are currently being hosted on the infrastructure of large relatively trustworthy organizations (e.g., Amazon, IBM, Microsoft, Google etc.), that have the means and the incentive to protect their reputation.
- Developers of cloud services are not necessarily security experts, and therefore regardless of their good intentions any misconfiguration, buggy code, or weak password could expose their services to critical data breaches.

In this setting, end users have to **implicitly trust** their data to both the service provider and the cloud hosting provider in order to use these services, although in most cases they are not even aware of the existence of the latter as they don't directly interact with the cloud hosting provider. (Note that the implicit and explicit trust relationships among the cloud entities are discussed in more detail in Chapter 2.)

Our approach aims to identify many classes of attacks that can lead to unauthorized data access (but which do not allow arbitrary code execution), such as SQL injection, command substitution, parameter tampering, directory traversal, and other prevalent web attacks that are seen in the wild. In case of attackers who gain arbitrary code execution, we can no longer guarantee accurate data tracking, since they can not only compromise our framework, but can also exfiltrate data through covert channels. Finally, besides protecting against external attacks, an equally important goal of ours is to bring into users' and service providers' attentions any unintended data exposure that may lead to unauthorized access. For example, sensitive data can accidentally be recorded in error logs or included into memory dumps after an application crash.

### 1.2.3 Thesis Statement

In this dissertation we argue that we can alleviate cloud users' trust concerns regarding sensitive data confidentiality, which hinder the adoption of cloud computing for security-sensitive operations, by providing more transparency on the use of their cloud-resident data. The main idea of this thesis is summarized in the following statement:

**THESIS STATEMENT**: *This dissertation examines the claim that information flow auditing mechanisms in the cloud can be used as the basis for increasing the transparency on the use of sensitive data by cloud services.*

Although the primary purpose of this thesis is to reinforce users' confidence in trusting the cloud services with their sensitive data, we also think that cloud hosting providers and service providers could also benefit from our proposed solution if they chose to adopt it because it could work as an additional level of protection against data breaches incidents and subsequently create a competitive business advantage for privacy-conscious cloud customers. To truly support our idea, we envision cloud providers offering our information flow auditing service in addition to their existing hosting environment, to both service providers and data owners, either as a *by default* mechanism, or as a *security-as-a-service* option. A large, reputable cloud provider could help leverage user confidence much more effectively than a lesser known service provider. Nevertheless, our approach allows service providers to provide, with minimal effort, an extra feature that reinforces the trust relationship with their users, knowing that they have an additional way to monitor what happens with their data. In addition, we empower service providers with the ability to build an additional level of protection and get a better understanding of how their services really work as they are given the tools to quickly identify inadvertent leaks or suspicious data flows.

### 1.2.4 Contributions

With data flow tracking as the basic underlying mechanism we designed and built the components that constitute the information flow auditing mechanisms, which will help towards the goal of elevating cloud customers' trust on cloud services. The main contributions of this thesis are summarized in the following:

- We designed and implemented **TaintExchange**, a *reusable cross-process* and *cross-host* taint tracking framework, which operates *transparently* on unmodified x86 Linux binaries, allowing real-world legacy applications to take advantage of our framework. To the best of our knowledge, our system is the **first** that allows **cross-host taint tracking without requiring full-system emulation**.
  - TaintExchange transparently enables cross-host and cross-process fine-grained taint tracking on the existing communication channels that matter to the target applications, rather than overloading every operation in the entire system with unnecessary heavyweight taint tracking operations.
  - TaintExchange offers flexible configuration of taint sources, in contrast to many other security-oriented DFT implementations that mark all incoming network traffic as tainted.
  - We evaluated the overhead imposed by TaintExchange, and showed that it incurs minimal overhead over the baseline DFT tool Libdft [36].
- We designed and implemented **CloudFence**, a novel data flow tracking framework for cloud hosting environments that provides *transparent*, *fine-grained* data tracking capabilities to both service providers, as well as their users.
  - Our prototype i) enables service providers to easily integrate data flow tracking in their applications through a simple API and verify the use of sensitive data within the well-defined domains of their services, offering additional protection against inadvertent information leakage and unauthorized access. ii) allows users,

to independently audit the use of their data by the the third-party cloud-hosted services.

  - CloudFence uses 32-bit wide tags per byte, and introduces new features such as lazy tag propagation and persistent tagging on disk and across the network It also incorporates the TaintExchange mechanism.

- We designed a novel mechanism to improve the effectiveness of our cloud-wide auditing mechanism and provide i) users, even the ones lacking any particular technical background to better understand the information collected in the audit log files regarding the treatment of their data by the cloud-hosted services, and ii) service providers, with an additional layer of protection against illegitimate data flows (e.g.,, inadvertent data leaks), by offering a more meaningful representation of the overall service dependencies and the relationships with third-parties out- side the cloud premises.

  - We designed and implemented **Cloudopsy**, a prototype implementation of this concept that through visualization mechanisms and automated analysis of the raw audit trails produces graphical representation of events and sensitive data flows, as they derive from the collected audit logs.

## 1.3 Dissertation Roadmap

The remainder of this dissertation is organized as follows: In Chapter 2 we introduce the core concepts that are discussed throughout this thesis. The following chapters are the building components of our solution. In Chapter 3 we discuss TaintExchange and how it implements transparent cross-host and cross-process information flow tracking on binaries, without requiring full-system emulation. In Chapter 4 we discuss CloudFence, a cloud-wide data flow tracking mechanism. We explain how it can transparently monitor the flow of user sensitive data within the cloud premises. In Chapter 5 we present Cloudopsy, a visual representation of the use of user data in the cloud premises. We explain how this contributes to alleviate data owners security-related concerns regarding their cloud-resident sensitive data used by the cloud-hosted services. Finally, the dissertation concludes with Chapter 6.

# Chapter 2

# Background

This dissertation has benefited greatly from a large body of prior work in the areas of data flow tracking and cloud computing security, which either served as inspiration for our design and/or as building blocks for our prototypes. This chapter intends to give an overview of the basic concepts required for the better understanding of the following chapters as well as provide a brief survey of the most relevant and influential research efforts in the related areas.

Section 2.1 gives a brief description of multi-level security and information flow control. Section 2.2 describes the basic mechanism of dynamic data flow tracking and its various forms, depending on the deployment platform and the level of tracking accuracy. Section 2.4 and its subsections describe the basic concepts in cloud environments and the most recent works in cloud security.

## 2.1 Multi-level Security

The root of concerns that lead to the design of multi-level security systems (MLS) in the 70's is once again current in the cloud computing environment. *Having information of different security levels and of different owners on the same computer systems poses a real threat.* MLS was introduced when *time-sharing* was starting to provide commercial customers the ability to share the leasing costs of IBM computers through simultaneous or sequential use of the expensive mainframe computers. Government and military were such customers that wanted to take advantage of this new capability, which promised great savings. But for those classified government and military circles, it was crucial that files and data of different security levels would be kept separate and with a high degree of confidence.

**The Bell and LaPadula model.** The Bell and LaPadula model (BLP) [9], also called the *multi-level security* model, was introduced as a response to the concerns regarding time-sharing mainframes with the primary goal of protecting data *confidentiality* by enforcing access control to classified data in government and military applications. In such applications, subjects (e.g., processes) and objects (e.g., files and user inputs) are often partitioned into different security levels and are assigned the proper security label. Security labels range from the most sensitive (e.g., "Top Secret"), down to the least sensitive (e.g., "Unclassified" or "Public"). The BLP model specifies how information can flow within the system based on the labels attached to each subject and object, i.e., a subject can only access objects at certain levels determined by his security level *clearance*. For instance, those with "Confidential" clearance are only authorized to view "Confidential" documents and they are not trusted to look at "Secret" or "Top Secret" information.

More specifically, a multilevel secure system for confidentiality enforces the following:

- The *simple security property*: no process may read data at a higher level. This is also known as *no read up*
- The **-property*: no process may write data to a lower level. This is also known as *no write down.*

The *-property was BLP's model critical innovation. It was driven by the fear that information might be leaked either intentionally by malicious code or unintentionally as a result of a bug, if applications could write down. These two properties provide the confidentiality form of a specific *mandatory access control* (MAC) security scheme.

Although the BLP model was originally proposed to support confidentiality in military systems, commercial systems now also use multi-level security policies. For instance, Red Hat Linux has incorporated MAC mechanisms with SELinux [4] and AppArmor [1]. The general idea is similar: subjects (e.g., processes, programs) and sensitive objects (e.g., files, user inputs) are assigned secutity labels, which represent their security sensitivity (level). The goal is again to make sure that the data are not accidentally written to unauthorized components.

One major disadvantage of access control of any kind, is that its scope is limited to the defined subjects, objects, and operations and also that the security labels can not be changed dynamically, which limits its applications.

**The IX system.** IX [44] is a MLS variant of Unix, which was built to offer practical security in the OS-level, for private- and public- sector nonmilitary critical uses. It shares the same high-level goal with the BLP model, i.e., that files and data of different security levels should be kept separate and with a high degree of confidence information should not be disclosed to unauthorized users. But it also has significant differences from the BLP model.

IX's features include *dynamic* security labels to classify information regarding privacy and integrity, *private paths* for the safe communication among privileged processes, and structured *privileges*.

Security labels are assigned to every file and process and represent its classification. These labels are checked at every system call involving data flow and are adjusted *dynamically* during the computation to guarantee that the labels on the output are at least as high as the labels of the inputs from which they derive. This is one of the main differences with the BLP model, where the labels of the files can not be changed while the file is in use, and therefore labels need to be checked only when files are opened. In contrast, IX dynamic labels require continual rechecking of labels on every data transfer, which as expected incurs overhead. Apart from the dynamic labels though, every process and every file system has a *ceiling*, which is also a label below which all transactions must stay. Process ceilings prevent processes from getting into overly sensitive places, and partly resemble the BLP subject labels.

Although data flow policy covers normal operations, there are some actions, e.g., document declassification, mounting file systems, opening external media that fall outside the territory of normal policies and need a *privilege* mechanism to be performed safely. Privileged processes administer their own security policies. Therefore the kernel and the privileged processes constitute the Trusted Computing Base (TCB). Privilege can be exercised only through trusted code.

Finally, for the safe communication among priviliged processes IX administers *private paths*. Briefly, a private path is a special state which exist so that the security is not compromised while waiting for some verification, i.e., authentication so that the user clearance is known.

In order to build the above features in IX the original Unix system had to be changed. Some special system calls were added, extra kernel memory was required for the labels, and the layout of the filesystem had to be changed to accommodate the labels and some utilities programs were written to set and retrieve the labels. Nevertheless, the dynamic nature of the labels that are updated and checked in every data transfer (through the system calls) as well as the end-to-end information flow tracking build in IX are techniques chosen aslo for the implementation of this thesis' data flow tracking mechanism. Such *data-centric* security mechanisms, which track or enforce information flow seem promising on detecting and preventing sensitive data leaks incidents, especially for the cloud computing environment where data flows across different services and the environment is more loosely connected than in the context of the military services.

## 2.2 Dynamic Data Flow Tracking

Dynamic data flow tracking (DFT), also referred to in literature as *dynamic taint analysis* (DTA), has been a prominent technique in the computer security domain, used independently or frequently complementing other systems. It has been shown to have a wide variety of applications including the detection of unknown exploits [53; 65; 59; 16], analysis of malware [76; 28], the prevention of information leaks [57; 29; 86] and many more, while researchers seem to continuously find new applications for it, even extending it to different domains [8; 49].

Dynamic data flow tracking is the mechanism of marking and tracking *selected* data, usually referred to as *tainted data*, as they flow during program execution. The mechanism is inductive. Broadly, dynamic taint analysis mechanisms operate by first identifying the *data of interest* according to predefined configurations, associating it with metadata, usually referred to as *taint tags* or *taint labels* and then track the flow of the labelled data (and the data they produce/influence) along with the relevant tags at runtime throughout the system.

The specifics of dynamic DFT implementations can vary significantly depending on ones goals, performance considerations, and deployment platform (see Section 2.3). Nevertheless, most dynamic data flow tracking implementations can be described by three properties: the *taint sources*, the *taint propagation rules* and the *taint sinks.*

**Taint sources.** Taint sources, i.e., the input channels for tainted data, are the locations of a system, where the *data of interest* enter the system and where the initial taint tag is assigned to it. These locations can be of different types, including program variables, functions, I/O streams (e.g., network connections, file system, keyboard) and many more. For instance, if a file and/or an IP address are defined as taint sources then the data read from this file or received from this IP address will be tainted, when it first enters the system.

**Taint propagation.** The propagation rules specify the *taint status* for data derived from one or more operations involving tainted data. Typically, there are two kinds of taint propagation: *data-flow* based and *control-flow* based. The former accounts only for *explicit* propagation of taint tags, which occurs through direct or transitive value assignments. The latter accounts for *implicit* propagation, which occurs due to control-flow dependences. The data- and control-flow based propagation, while useful in some domains, it induces a higher runtime overhead than propagation based on data-flow only and frequently leads to an explosion in the amount of tagged data as well as to incorrect data dependencies. Therefore, many DFT implementations [53; 65; 19] of the related work tend to focus on explicit taint propagation, whereas fewer [21; 34] consider also the implicit taint flows. The different goals DFT systems allow different taint propagation policies.

**Taint sinks.** Taint sinks are the locations, where checking operations on the taint status of data needs to be performed (e.g., the network interface). For instance, if the network is defined as a data sink, the process of sending tainted data to the network will trigger an action according to a predefined policy (e.g., issue a notification or halt program execution). More generally, user-specified policies may be enforced regarding the use of tainted data at these sinks, that will determine the further behavior of the monitored program.

Broadly, the purpose of DFT is to monitor the flow of data between taint sources and taint sinks. For a simple example of how taint sources, propagation policies, and taint sinks work together, consider the following pseudocode:

```
c = taint-source()
...
a = b + c
...
network-send(a)
```

`c` is tainted, i.e., is assigned a taint tag, *tag(c)=1*, since it is read from a taint source. Then `a` is also tainted (*tag(a)*=1), since it derives from the tainted variable `c` and the untainted `b` (*tag(b)*=0). Assuming that the taint sink is the network, when the execution reaches the `network-send()` function, it will check the value in *tag(a)* and act according to some predefined policy.

## 2.2.1 Taint Tags Granularity

In addition to specifying the data to be tainted, it is important also to define what taint tags should be associated with the selected data. In most cases one bit of taint is sufficient. Many existing dynamic taint analyses [59; 16; 19; 86], have been implemented using a single taint mode, i.e., *tainted* or *untainted* data. But there are also situations [36; 85; 84; 75]) where we need to discriminate between data read from different sources or to distinguish between trust levels. For instance, if we need to distinguish between data coming from network hosts with different levels of trust, the single taint mode, would not be sufficient. To support these applications multiple taint bits are necessary.

Taint tags are usually kept in a separate memory area, usually called *shadow memory* [51], which is inaccessible to the program under analysis. Taint is propagated through the system to all data derived from the tainted values in the shadow memory.

### 2.2.2 Taint Accuracy

The accuracy of taint is also a key challenge. By *taint accuracy* we refer to the smallest data unit for which we keep track of its taint status. This data unit could be as small as a single bit or much larger as contiguous chunks of memory or process-level taint tags [81]. The former enables more fine-grained and accurate DFT, while the latter implies coarser DFT and probably not suitable for the level of accuracy demanded for information leakage incidents, that are the focus of this work. For instance, with page-level taint tags, moving a single tainted byte into an untainted location will result into tainting the whole page that contains the destination, thus “polluting” the adjacent data. There is a trade-off between accuracy and performance when choosing extremely fine-grained over coarser tainting, as more memory space is needed for storing the tags and the propagation logic becomes more complicated. In this work we used *byte-level* tainting, since that was the necessary level of accuracy for detecting data leakage incidents.

## 2.3 Taxonomy of Dynamic DFT Systems

The various dynamic DFT tools can be categorized depending on the *deployment platform*, where the taint logic is implemented, and the *scope of the monitoring*, as shown in Figure 2.1.

### 2.3.1 DFT Deployment Platforms

One possible classification of existing dynamic DFT mechanisms can be made based on the *means* by which the *tracking logic* is augmented on regular program execution. Building the tracking logic necessitates integrating some monitoring facilities to the analyzed application. Such monitoring features have been integrated into full system emulators and modified virtual machines, retrofitted into unmodified binaries using dynamic binary instrumentation (DBI) and added to source codebases using source-to-source transformations. Proposals have also been made to implement it in hardware, although this had little appeal to hardware vendors.

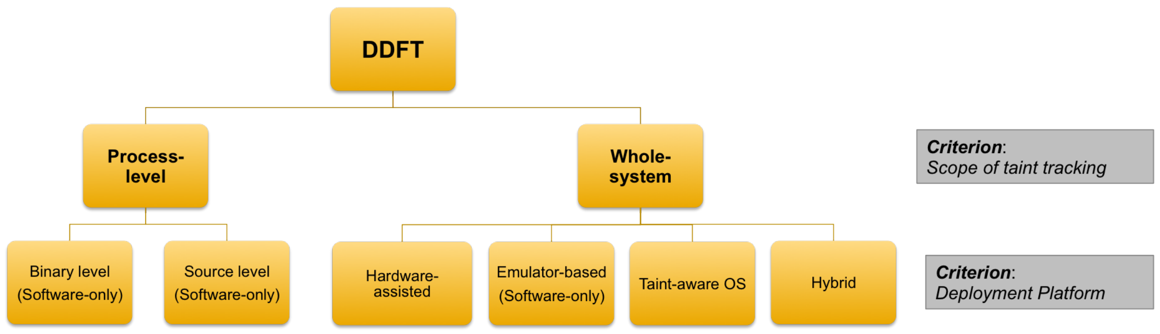

DDFT
Process-level
Whole-system
Binary level (Software-only)
Source level (Software-only)
Hardware-assisted
Emulator-based (Software-only)
Taint-aware OS
Hybrid
Criterion: Scope of taint tracking
Criterion: Deployment Platform

**Figure 2.1: Taxonomy of Dynamic DFT systems.**

#### 2.3.1.1 Software-based DFT

In order to enable data flow tracking facilities on software-only level, we need an environment where there is full control over the process under analysis. Such taint tracking systems can be based on source-level instrumentation [74], binary instrumenation ([59; 21; 53; 86; 36]) using DBI frameworks (see Section 2.3.3) or specialized whole-system emulators ([57; 20; 28; 76]).

Dynamic binary instrumentation of unmodified binaries is a very popular technique for implementing DFT tools, as it runs on existing hardware and can offer flexible and configurable solutions. It is applicable to all user executables and library binaries but it incurs high runtime overheads. This approach can neither support multi-threaded code nor track information flow across multiple processes. Usually for performance reasons, DBI systems support a single policy that protects against attacks on control data. TaintCheck [53], one of the foundational efforts in this area, aims to provide an effective defense mechanism against fast-spreading Internet worms through automated exploit detection and signature generation. TaintCheck operates by marking any data that originates from an untrusted external source (e.g.,, network sockets) as tainted and uses binary rewriting mechanisms to track the subsequent propagation of such data. To detect attacks, TaintCheck looks for dangerous and potentially illegitimate operations on tainted data, such as the use of a tainted value as the destination for a jump instruction, which would be suggestive of an attempt to redirect control flow. Its implementation is based on Valgrind [52] and can track the propagation of tainted inputs within the virtual address space of a single user-level process. TaintTrace [19], LIFT [59] and Dytan [21] are also built on DBI frameworks, DynamoRio, StarDBT and Pin respectively, but also propose optimization techniques to improve the efficiency and reduce the overhead from the instruction-level tracking, that make these tools prohibitive for large-scale real-world applications.

On the other hand, Panorama [76] uses full-system emulation and dynamic taint analysis to detect malicious access to sensitive user data and identify privacy-breaching malware. The taint tracking engine monitors how sensitive (tainted) information propagates within the system and flags any suspicious interaction between the unknown code sample and the tainted data. Argos [57] is another implementation of DFT logic using an modified emulator, QEMU. All these emulator-based approaches allow DFT tracking on ther entire system (including the OS-kernel) and are easily applicable to multi-threaded applications but they incur significantly higher overhead even from the DBI approaches for single applications.

For interpreted languages, e.g., JavaScript, Python, Perl, PHP, the instrumentation code can be added to the interpreter [77; 54] or the just-in-time compiler [70]. In particular, Perl has built-in support for taint, which allows the interpreter to prevent a variety of common security flaws from being exploited. A more recent effort, Resin [77], proposes a new application runtime that associates policies with application-level data objects and filters information transfer at system I/O boundaries. It operates in an interpreted programming environment (such as Python or PHP) and tracks the propagation of sensitive data at the level of program variables. Such implementations have been shown to detect high-level vulnerabilities, e.g., sensitive information leaking, instead of system-compromising security attacks, due to the lack of information only resolved at runtime. Their main disadvantage is that any code written in other languages require custom wrappers for safety. Moreover, this approach cannot track information flow across multiple processes or easily deal with multi-threaded executables.

Finally, software-based DFT can be implemented by instrumenting programs at the source code level [74]. Systems like these use source-to-source transformation and automatically insert the necessary instrumentation code to propagate the tags. Compile-time optimizations can significantly reduce the space and runtime overhead for these tools. But this approach is less practical as it requires the source code for the monitored applications, which is often unavailable for COTS software, and also it can not track information flow through third party programs, binary libraries, and system calls available only in binary form.

#### 2.3.1.2 OS-level DFT

Interception of system calls within the kernel coupled with the ability to add extension code that can be executed before and after the normal system call functionality is a common approach for OS-level DFT implementations. Asbestos [27] and HiStar [81] are representative frameworks that integrate the notion of data flow and taint tracking directly in the operating system. They use labels to indicate the taint level and protect sensitive data by restricting information flow from more sensitive objects to less sensitive objects without the use of a trusted agent. DStar [82] and Flume [40] are alternative OS-level DFT mechanisms used for distributed systems. Laminar [60], one of the more recent proposals, investigates a *hybrid* design that integrates language-level and dynamic OS-level DFT in an effort to combine their strengths. Laminar designs a new operating system, which mediates access to system resources, and a specialized VM, which enforces fine-grained DFT rules within the address space of an application.

In general, these DFT-enabled operating systems require lower system call interception overhead, and more flexibility in terms of what actions can be performed within the extension code. But they cannot track granularities smaller than coarse-grained high-level OS objects, i.e., files, processes and sockets, and are oblivious to fine-grained information transfers between variables or data structures within a process. Moreover, they require rewriting of the monitored applications for these experimental platforms to enable the tracking mechanism.

#### 2.3.1.3 Hardware-assisted DFT

Hardware support has been proposed in order to evade performance penalties imposed by software-only implementations and to accommodate multithreading. Broadly, hardware DFT architectures extend each register and memory location by one tag bit. Analogously, all machine instructions are extended with additional functionality to propagate and check tags in addition to their regular operation. Therefore, the hardware propagates and checks tags transparently as instructions execute without additional instrumentation or runtime overhead.

Minos [24] was one of the first systems to investigate hardware-assisted DFT. Its design addresses many basic issues pertaining to integration of tags in modern processors and management of tags in the OS. The architecture by Suh et al. [65] is also one of the first hardware implementations for DFT. Although these implementations achieve much lower overhead for the taint tracking and provide compatibility with multi-threading applications, they suffer from limited flexibility in specifying taint propagation rules, and in the number of taint bits associated with a value. Recognizing these limitations, there have been efforts to overcome the problems that limit their practicality. RIFLE [68] proposed a system solution that uses software binary rewriting to turn all implicit flows into explicit flows that can be tracked using DFT techniques. But the overall system combines this software infrastructure with a hardware DFT implementation to track the propagation of sensitive data and prevent leaks. Raksha[25], a more recent effort, investigates also *hybrid* DFT architecture that tries to combine the strengths of hardware- and software-based techniques in order to provide a flexible low-overhead solution.

One of the key concerns in designing a hardware-assisted scheme is that it requires non-standard commodity components and a redesign of the entire processor core, which deeply affects its practicality. Moreover, whereas a software solution can quickly be upgraded to guard against new attacks, hardware based tools can only be upgraded by replacing the processor or the entire system. Therefore, unfortunately hardware implementations of DFT had little appeal to the hardware vendors.

### 2.3.2 Scope of Taint Tracking

A different classification can be made based on the *scope* of the tracking. Dynamic DFT systems can operate on unmodified binaries of applications or entire systems. Currently, dynamic tracking approaches range from *per-process* taint tracking [19; 21; 36; 53; 59; 72; 86], to *whole-system* tracking [24; 25; 57; 76] using emulation environments or hardware extensions.

#### 2.3.2.1 Single-process Taint Tracking

Most application-level taint tracking tools, like TaintCheck [53], TaintTrace [19], Libdft [36], Dytan [21], and LIFT [59] use dynamic binary instrumentation (DBI) frameworks, like PIN [42], StarDBT [15] and Valgrind [52]. While quite effective and useful, as they do not require any modifications to source code or customized hardware, they impose significant impact on the performance, as every instruction needs to be instrumented, and additional storage, usually called *shadow memory* [51], is required for storing the tags. As a result, there has been great interest in optimization techniques in order to improve their performance. TaintTrace achieved significantly faster taint-tracking by using more efficient instrumentation based on DynamoRIO, combined with simple static analysis to speed up the taint-tag access. LIFT also achieved significant additional performance benefits by using better static analysis and faster instrumentation techniques.

One important disadvantage of single-process level taint tracking is that the taint analysis loses track of the information flow once the application moves data out of the process-level, e.g., through network output channels. Because most modern Web services include multiple applications single-process DFT systems result in false positives and/or negatives because the lose track of the taint status of the data exchanges between them and therefore must assume that all values of the cooperating processes are either tainted or untainted. Note that this limitation was the motivation for building the TaintExchange mechanism (see Chapter 3).

#### 2.3.2.2 Cross-process and Cross-host Taint Tracking

A large body of research has also focused on cross-process or system-wide taint tracking, leading to the creation of many tools [24; 25; 76; 83], mostly based on emulators and hardware extensions to efficiently handle data tracking for an entire operating system. For instance, the whole system emulator QEMU [10] is employed by various solutions that implement DFT [31; 57; 76], while TaintBochs [20] builds on the Bochs IA-32 emulator [14]. The architecture community attempted to integrate or assist dynamic taint tracking with hardware extensions [24; 25; 65; 68], to alleviate the significant performance impact due to extra tag processing from DBI frameworks and emulators.

While there is much research aiming at intra-process and system-wide DFT implementations, it was not until very recently that interest has risen for efficient cross-host taint propagation systems [8; 26; 83]. Most of these techniques are more problem-specific, and therefore it would be difficult to adapt the techniques and tools developed for use in other contexts. For instance, DBTaint [26] is targeting taint information flow tracking specifically for databases, whereas ConfAid [8] tackles the problem of discovering a set of possible root causes in configuration files that may be responsible for software misconfigurations. System tomography [49], which also looks into the concept of propagating taint information remotely, builds on the QEMU emulator so it cannot be applied on already deployed software and incurs large slowdowns. Finally, Neon [83] also requires modifications in the underlying system to perform dynamic taint tracking. It uses a modified NFS server for handling the initial tainting, and utilizes a network-filter for monitoring the tainted packets arriving/leaving the server.

In general, these systems require substantial changes to the underlying hardware or must emulate the entire system with significant performance penalty. The other category of system-wide DFT systems, like HiStar [81] and Asbestos [27], which are basically DFT-equipped OSes are also inappropriate for use with COTS software and Web services.

### 2.3.3 Dynamic Binary Instrumentation with PIN

In general, dynamic binary instrumentation (DBI) frameworks, analyze programs at runtime at the level of machine code, whereby the *analysis code* is added to the original code of the monitored program at run-time. They consist of a frond-end and a back-end. The front-end is an API allowing to specify the instrumentation code and the points at which it should be introduced at runtime. The back-end introduces the instrumentation an provides all necessary information to the front-end.

There are two main approaches for controlling the monitored application: *emulation* and *just-in-time* (JIT) instrumentation. The emulation approach consists of executing the application on a *virtual machine* while in the JIT approach the instrumentation is performed after a program has been loaded into memory and immediately prior to execution. This instrumentation has the advantage that the functionality can be selectively added or removed from the program without the need to recompile, i.e., the trace functionality is only present when needed.

Pin [42] is a representative framework of the JIT approach. Pin and the monitored application are loaded together and Pin is responsible for intercepting the application's instructions and analysing or modifying them as described by the analysis code written in the so-called *pintools*. Briefly, Pin consists of a virtual machine (VM) library, and an injector that attaches the VM in already running processes or new processes that launched itself. Pintools are shared libraries that employ Pin's API to inspect and modify the binary at the instruction level. The original application code and the analysis routines are translated by Pin's JIT compiler for generating the code that will actually run.

Note that although the *design* of our mechanisms is independent of the underlying DFT system for the *implementation* of our prototypes we chose the Pin framework. The main reasons for our choice were its availability [1], well-defined programming interface, efficient instrumentation, and its support for additional operating systems (should we choose to support them in the future).

**Libdft library** Libdft [36], the dynamic single-process DFT framework we used for our implementation, is also a library, which can be used by pintools to transparently apply *fine-grained* DFT for the binaries running over Pin. Libdft's tag propagation is accomplished

[1]Although the optimizations implemented by LIFT[59] on StarDBT [15] made it an attractive choice, StarDBT is not publicly available and lacks a programming interface.

using Pin to both instrument and analyze the target process. In particular, we build our mechanisms as Libdft-enabled pintools to specify the taint sources and sinks as well as build our cross-process taint propagation mechanism.

## 2.4 Cloud-related Concepts

Cloud computing is not a genuinely new concept in the IT world, but more of a buzzword for the long-held dream of using computing as a utility, or else, the latest extension in distributed computing that takes advantage of the latest technology advances in networking, higher throughputs and storage. The concept of the "*cloud*" dates back to early mainframe processing, when users connected to a shared computing resource through terminals to solve their computing needs. The advent of faster and cheaper microprocessors, RAM and storage brought computing into the client-server model, which grouped sets of users into networks sharing computing power on decentralized commodity servers. As increased bandwidth became more available and less costly, these networks interconnected and formed the Internet, which in its current form made enterprises re-examine their current onsite implementations and switch to the contemporary *cloud services.*

### 2.4.1 Term Definitions

There are a few topics in the IT community that are as promising and as confusing as is *cloud computing.* In the simplest of terms, cloud computing is Internet-based computing, done by a group of third-party shared servers that can provide on-demand hardware resources and/or software capabilities to users connected to the Internet. A clear and widely accepted definition for cloud computing has yet been agreed upon, especially since it encompasses so many different models and potential markets, depending on vendors and services. In fact, even the National Institute of Standards and Technology (NIST) presented 15 draft versions of the term before coming up with the current working definition [46]. In the same spirit as the NIST definition follows my working definition, which will be used for the reminder of this document.

**My working definition:** Cloud computing refers to the model of *on-demand*, *pay-per-use* access to a shared pool of off-premises, third-party, configurable computing resources, (e.g., machines, network, storage, operating systems, application development environments, and application programs) via the Internet or local area network. The details of the hardware or software infrastructures that support the computations are hidden from the users of the cloud.

In short, when a client of the cloud is granted the requested access, a fraction of the resources in the pool is dedicated to the requesting user until he or she releases them. One of the main characteristics of the "cloud" is that the user cannot actually see or specify the physical location or organization of the equipment hosting the resources he is ultimately allowed to use. That is, the resources are drawn from a pool of resources when they are granted to a user and returned to the pool when they are released. *Note that with the term "cloud" we will be referring to both the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those services.*

To avoid confusion regarding certain basic concepts used in this thesis, I will explicitly define the following core terms:

- **sensitive data**: any data of which the compromise with respect to confidentiality, integrity and/or availability could have an adverse effect to business interests or to the privacy to which individuals are entitled. This term is used for both individual users' *personally identifiable data* (PII) and for enterprises' *mission-critical* and private data. Synonym terms used: *privacy-sensitive data*, *important data.*[2]

- **data privacy**: keeping data safeguarded from disclosure to unauthorized parties. In this document, it is interchangeably used with *data confidentiality.*

- **privacy restrictions**: regulations on the storage, dissemination and, processing of sensitive data. Synonym term: *privacy preferences.*

- **data breach**: disclosure of data to unauthorized parties. Synonym terms used: *data leak*, *data exfiltration*, *privacy abuse.*

[2]A formal description of sensitive data can be found, among others, in the HIPAA and PCI-DSS regulations.

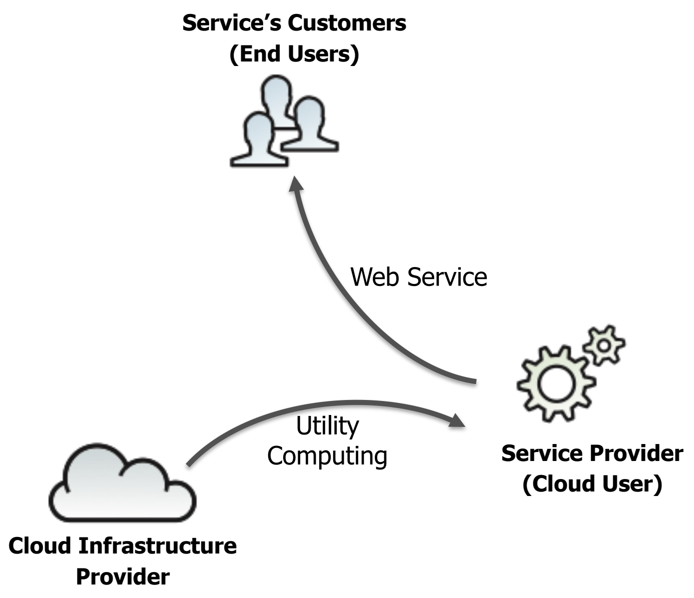

Service's Customers (End Users)
Web Service
Utility Computing
Service Provider (Cloud User)
Cloud Infrastructure Provider

**Figure 2.2: Cloud entities.** The service provider uses the cloud infrastructure to deploy and host its service. End users access the service through its web interface.

- **assurance**: refers to the need for a system to behave as expected. In this context, it is important that the cloud provider provides what the client has specified. This is not simply a matter of the software and hardware behaving as the client expects but that the needs of the organization are understood, and that these needs are accurately translated into information architecture requirements, which are then faithfully implemented in the cloud system.

- **cloud provider**: the provider of cloud resources, that offers its customers storage or software services available via a private (private cloud) or public network (cloud). It also refers to the storage and software that is provided to customers. Synonym term: *cloud infrastructure provider*.

- **service provider**: the provider of applications on top of a cloud platform. He directly utilizes the Infrastructure-as-service (IaaS) model provided by the cloud provider. Synonym terms used: *cloud tenant*, *cloud user*.

- **end-user**: usually refers to the individuals using the applications of the cloud tenants or software-as-a-service (SaaS) instances. Synonym term: *SaaS users*.

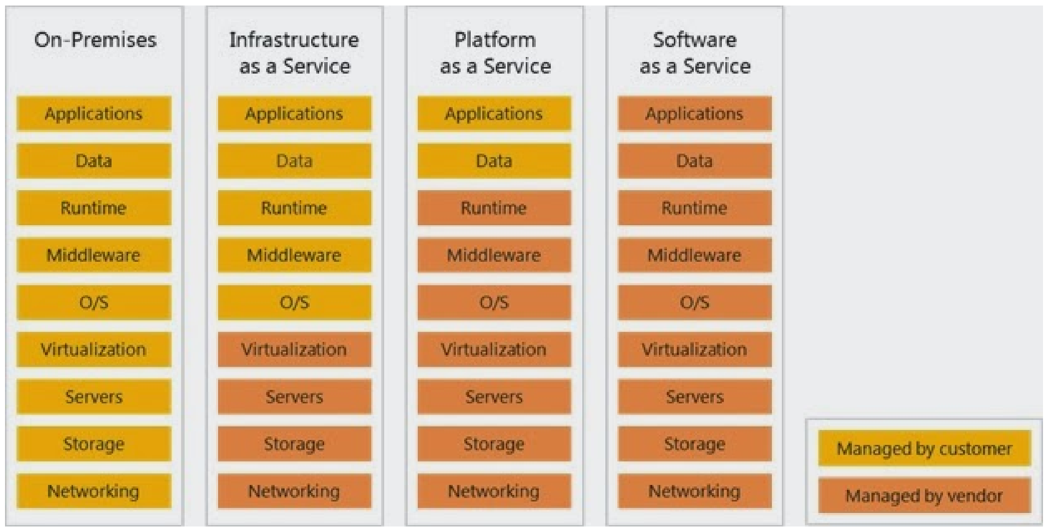


**Figure 2.3: Cloud service models.** Comparison of traditional (on-premises) software solutions and the cloud computing service models (IaaS, PaaS, SaaS).

- **cloud client**: refers to the *service provider* (as the client of the cloud provider), or the *end-user* (as the client of the service provider) or both of them.

Figure 2.2 clarifies the main entities in the cloud. Note that in this work the focus is mostly on the cloud-hosted web services model. The service provided by the cloud infrastructure providers is *utility computing*, and the end users are provided a *web service*, which is hosted on the cloud infrastructure.

### 2.4.2 Cloud Service Models

The type of computing resource that is offered in a cloud defines the cloud's *service model*. The three common service models that are based on what cloud services are provided are: *applications* (SaaS), *platform* (PaaS) and/or *infrastructure* (IaaS).

**Software-as-a-Service (SaaS).** Sometimes referred to as "on-demand software", SaaS is a software delivery model in which software and its associated data are hosted in the cloud and are typically accessed by *end-users* using a thin client, normally a Web browser over the Internet. SaaS' clients use the software applications from the *cloud provider* (or the *service provider* if they are different entities). The *end-user* uses the software as an application, while the underlying platform software and infrastructure hardware is abstracted to the end-user. SaaS provides the most integrated functionality built directly into the offering, with the least consumer extensibility, and a relatively high level of integrated security (at least the provider bears a responsibility for security). Examples of SaaS include Microsoft Office 365, Google Apps, and Salesforce.com applications.

**Platform-as-a-Service (PaaS).** In the PaaS model, the *cloud client* (a developer) uses specialized APIs for programming languages, tools, and the runtime platform provided by the *cloud provider* for application development and deployment. The platform may include databases and middleware in addition to application development tools. PaaS abstracts the underlying hardware cloud infrastructure. Popular PaaS platforms include the Google AppEngine (GAE) and Microsoft Windows Azure.

**Infrastructure-as-a-Service (IaaS).** As it is implied by Figure 2.3, in the IaaS model the *cloud provider* manages the underlying physical cloud infrastructure (servers, storage, network, and the associated virtualization), while the *cloud client* is provided with one (or more) virtual machine with the required specifications and has control over the OS, storage and deploys and runs his or her own application and platform software. Amazon's Elastic Compute Cloud (EC2) is a good example of the IaaS model.

**Note:** As we move from SaaS to PaaS to IaaS, cloud providers gradually release controls over security to the cloud clients, i.e., the service providers. In the SaaS model the burden of security relies with the the cloud provider (and the service provider, if they are different) to perform all security functions, whereas at the other extreme, at the IaaS model the security functions are the cloud client's (service provider) responsibility. SaaS users have the least control over security among the three fundamental delivery models in the cloud.

### 2.4.3 Trust Relationships

In the conventional company dedicated IT environments model, where a provider is offering an online service to its clients - the service is hosted on the provider's dedicated IT infrastructure - the explicit relationship among the participating entities is a simple **provider-user**

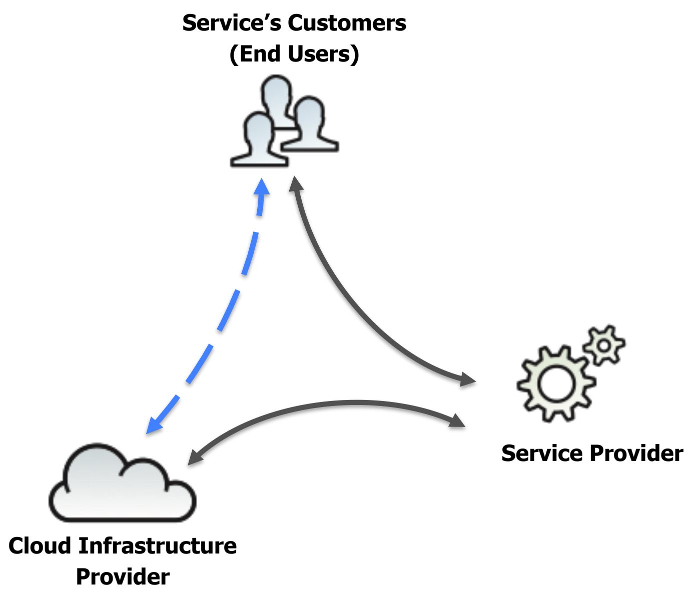

Service's Customers (End Users)
Service Provider
Cloud Infrastructure Provider

**Figure 2.4:** Users explicitly trust their data to service providers, but also implicitly trust the cloud provider that hosts these services.

**relationship**. This relationship involves to some extent an implicit level of trust, that the hardware and software provided by the technology vendors and used inside these companies are safe and secure. In this setting it is the provider's responsibility to make sure that the service was conforming to the security requirements of the relevant service level agreement (SLA). Where operational requirements dictate for more demonstrable levels of trust, they can be achieved through implementation of high assurance technologies and testing.

On the contrary, on-site testing and getting a clear insight of cloud services internals is not an option, at least to the extent that would alleviate user's concerns. Cloud systems execute in an abstract environment, located in physical data centers whose precise location may be unknown or obscured. Gaining access to those data centers to evaluate their integrity would invalidate the very controls established to ensure they deliver the security and integrity necessary for multiple other customers.

For cloud services a longer **trust chain** must be accommodated than the traditional provider-user relationship. For example, the application end-user could potentially use an application built by an SaaS provider, with the application running on a platform offered by a PaaS provider, which in turn runs on the infrastructure of an IaaS provider. While to our knowledge this extreme example cannot occur in practice today due to a lack of sufficient APIs, it illustrates that with any model of cloud computing, the traditional provider-user trust relationship is transformed into a **multi-party** system [17], as shown in Figure 2.4, in which users are often not aware of the existence of the cloud infrastructure provider at all (unless it is the same entity that also offers the service, as for example is the case with many of the services offered by Google or Amazon). The solid lines in Figure 2.4 represent the interactions between cloud service customer's and their providers as well as between service provider's and the cloud infrastructure provider hosting them. The **unrecognized** relation, denoted in the dashed line, occurs when the cloud service provider and the cloud infrastructure provider are different administrative entities and represents the **implicit trust** relation between the end users and the cloud infrastructure provider.

Users of online cloud services trust the providers of those services to securely handle and protect their data. In turn, service providers place their trust in the cloud infrastructure that hosts their services. From the users' perspective, there is an inherent shared responsibility between the cloud and the service providers regarding the security guarantees of the provided service. Although end users do not interact directly with cloud providers, they implicitly trust their infrastructure—the systems in which their data are kept. Consequently, cloud-hosted applications depend on the *trustworthiness* of both their providers and the cloud infrastructure providers. *This thesis proposes to exploit this implicit trust for the benefit of all parties by introducing a* ***direct relationship*** *between end users and cloud providers.*

Understanding the implicit and explicit relationships and dependencies between cloud computing entities is critical to understanding and evaluating cloud computing security risks.

# Chapter 3

# Generic Cross-Host Taint Propagation Mechanism

In this Chapter we discuss the design and implementation of TaintExchange [79], a *generic* mechanism for efficiently performing *cross-process* and *cross-host* data flow tracking. As suggested by the name of our tool, our goal was to create a mechanism that allowed collaborating processes (on the same or different hosts) to *transfer* taint information together with the data exchanged between them. With TaintExchange, fine-grained taint information is transparently *multiplexed* with user data through the existing communication channels (i.e., sockets or pipes), allowing us to mark individual bytes of the transferred data as tainted.

## 3.1 Introduction

Most real-world services are quite complex, and are usually built by "glueing" together a multitude of components, e.g., Web services. These feature-rich services consist of multiple applications exchanging data, that in many cases run on different hosts. Bugs and vulnerabilities in existing code, misconfigurations and incorrect assumptions about the interaction between different components can leave these services vulnerable.

Dynamic data flow tracking has been a prominent technique in the computer security domain, used for various tasks, including detecting software vulnerabilities, analyzing malware, preventing information leaks, while researchers seem to continuously find new applications for it. Originally, taint tracking systems enabled the tracking of tainted data throughout the execution of a single process [59; 53; 21; 19], or an entire host in the case of virtual machine (VM)- and emulator-based systems [31; 57; 76]. The latter enabled researchers to track the interactions between processes running within a virtual machine. Existing cross-application and cross-host taint propagation systems frequently make use of VMs and emulators [49; 83] incurring *unnecessary overhead* and requiring *extensive maintenance and setup.* Other implementations of cross-process DFT are very problem-specific, requiring extensive modifications for reuse by the research community to solve new problems. Therefore, the *motivation* behind TaintExchange was to create a valuable tool for rapidly developing prototypes that utilize cross-process information flow tracking, without severely impacting application performance.

This chapter presents TaintExchange, a *generic* cross-process and cross-host taint tracking framework. TaintExchange is build on the Libdft open source data flow tracking framework [36], which performs taint tracking on unmodified binary processes using Intel's Pin dynamic binary instrumentation framework [42]. We have extended Libdft to enable the **transfer** of taint information for data exchanged between hosts through network sockets, and between processes using pipes and unix sockets. Taint information is transparently **multiplexed** with user data through the same channel (i.e., socket or pipe), allowing us to mark individual bytes of the communicating data as tainted. Additionally, users have the flexibility to specify which communication channels will propagate or receive taint information. For instance, a socket from *HOST A* can contain **fine-grained** taint information, while a socket from *HOST B* may not contain detailed taint transfer information, and all data arriving can be considered as tainted. Similarly, users can also configure TaintExchange to treat certain files as tainted. Currently, entire files can be identified as a source of tainted data.

TaintExchange can be a valuable asset in the multi-component Web services setting, providing **transparent** propagation of taint information, along with the actual data, and establishing accurate cross-system data flow monitoring for *data of interest.* TaintExchange could find many applications in the system security field. For example, in tracking and protecting privacy-sensitive information as it flows throughout a multi-application environment (e.g., from a database to a web server, and even to a browser). In such a scenario, the data marked with a "sensitive" tag, will maintain their taint status throughout their lifetime, and depending on the policies of the system, TaintExchange can be configured to raise an alert or even restrict their use on a security-sensitive operation, e.g., their transfer to another host. In a different scenario, a TaintExchange-enabled system could also help improve the security of Web applications by tracking unsafe user data, and limiting their use in JavaScript and SQL scripts to protect buggy applications from XSS and SQL-injection attacks.

## 3.2 Design

We designed and implemented a prototype for TaintExchange based on the Libdft data flow tracking framework [36], to produce a *generic* system for efficiently performing cross-process and cross-host taint tracking. Nevertheless, TaintExchange could be easily retargeted to numerous other taint tracking systems similar to Libdft, e.g., TaintCheck [53], TaintTrace [19], LIFT [59], or Dytan [21], as TaintExchange is a broadly applicable design. Libdft was chosen because it is one of the fastest process-wide taint-tracking frameworks, which performs at least as fast, if not faster than similar systems such as LIFT and Dytan.

We will present the main aspects of TaintExchange, following the three-dimensions described in Chapter 2. Figure 3.1 shows an overview of TaintExchange, the various sources of tainted data that can be configured, and the mechanisms for exchanging taint information between processes and hosts.

### 3.2.1 Taint Sources

The taint sources are the "starting points" of the system, where taint is first assigned to the *data of interest*. Our current framework supports configurable taint sources from the two most common input channels, the file-system and the network. In the current implementation, the user of TaintExchange directly interacts with the underlying framework (i.e., both TaintExchange and Libdft) to define the taint sources, as they each time depend on the problem being tackled. Although the configuration of the taint sources is straightforward,

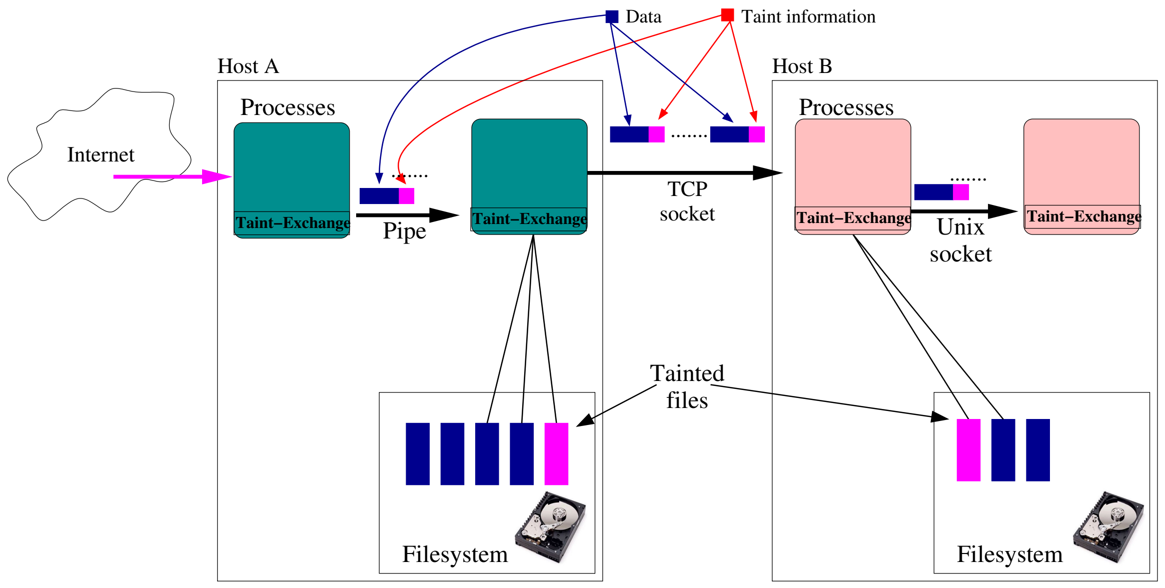


**Figure 3.1:** TaintExchange overview. Taint information can be exchanged using network sockets and pipes. Sockets and files can also be configured as taint sources.

we stress that a better user interface for this purpose would further improve its usability. But this is beyond the scope of our work for now.

Configuring taint sources from the filesystem is a straightforward process. In particular, a shadow file `taint_config` is maintained for listing all tainted files in the system. The designer has to update this file with the tainted files, using full-path format, and then all data originating from files listed in `taint_config` will be marked as tainted. Network sockets and pipes can also be among the taint sources, so data arriving from them can be also tagged as tainted. Sockets and pipes can be also configured to receive and transmit detailed taint information regarding the data being exchanged (described later in this section).

A global array, `state_sfd`, is used for keeping track of the important "channels" that comprise TaintExchange's taint sources. The `open()`, `socket()`, `accept()`, `dup()`, and `close()` system calls are intercepted and augmented with the necessary code for updating the `state_sfd` array accordingly. Once the tainted channels are identified, they start being monitored for tainted data. Briefly, once `read-like` calls, such as `read()`, `readv()`, `recv()` etc., read data from one of the identified tainted sources, the taint-header processing mechanism (described later in the chapter) should be initiated. This includes the extraction of a taint-header from the received data stream, including information for the taint status of the data received, and then the tainting of the received data accordingly.

#### 3.2.1.1 Data Tags

In addition to specifying the data to be tainted it is important also to define what taint tags should be associated with the selected data. The first prototype of our system supports only binary mode taint status, i.e., tainted or clean data, but this is only a limitation because of the chosen underlying taint tracking system Libdft [36] version. In fact, for the later version of TaintExchange, that is incorporated in the CloudFence framework (see Chapter 4), we use a later future version of Libdft that supports multiple labels/colors for tracked data. Therefore, the latest version of our prototype takes advantage of the richer data tags. Of course, the larger tags have a larger impact on the performance of data transfers, but with several other optimizations we managed to reduce the performance penalty from the larger data tags.

### 3.2.2 Taint Propagation

We examined three different *scopes* for the propagation of the tainted data. Firstly, the *intra-propagation* of taint values during the execution of a single process. As we discussed in Chapter 2, this has been thoroughly explored by past work, and there exist many tools [19; 21; 36; 53] for efficiently handling this issue. In general, all these tools allocate a separate *shadow memory* for every process to keep track of the taint status of data. In our design we refer to this shadow memory as a `tagmap`. The second case of taint propagation we consider is the *cross-process* propagation of taint tags for the data exchanged between processes. Note that previous research has mostly addressed this topic by performing system-wide taint tracking using modified VMs and specialized hardware [24; 25; 57; 76], mostly based on emulators and hardware extensions for efficiently handling system-wide tracking of tainted information. The last case we examine is the *cross-host* transfer of tainted tags. Relatively little research has explored this path [8; 26; 49; 83]. For the last two scopes, we build a separate *taint transfer* mechanism.

**Taint transfer mechanism.** For TaintExchange, *taint transfer* refers to the propagation of taint information along with the data, when processes on the same host or on different hosts communicate. Our mechanism supports processes that communicate using sockets and pipes. Briefly, the main idea is to monitor the information flow between the taint sinks and taint sources, and intercept the system calls from the `read/receive` and `write/send` families that are used to read from and write to these tainted channels. For data leaving the current process (or host), a **taint-header** is composed and attached to the data, indicating which bytes of the data payload, if any, are tainted. We will describe the taint-header in detail in Section 3.2.4. On the receiving side, as the "extended by the taint-header" data enters the process (or host), and assuming that the source descriptor is among the taint sources described earlier, the taint header will be extracted from the received data, and the relevant taint tags storage structure of the process will be updated accordingly.

Cross-process taint transfer is handled the same way as cross-host taint transfer. The only difference is the source descriptor, which in the case of pipes is the `pid` or name of the application we are communicating with. IPC through UNIX sockets is very similar to TCP sockets, so the mechanism remains almost entirely the same.

### 3.2.3 Taint Sinks

Taint sinks refer to the locations in the system, where the user needs to perform some assertions on the data. For example, tainted data may not be allowed to be transmitted over a certain socket, or used as program control data (e.g., a function return address). Taint sinks are problem-specific, and can be configured by the user. Libdft offers an extensive API for the users to check for the presence of tainted data on instructions and on system or function calls.

### 3.2.4 Taint Headers

#### 3.2.4.1 Taint Header Structure

To multiplex data and taint information, TaintExchange prefixes each data transfer with a taint header, which essentially encapsulates the transferred data into a new "packet"

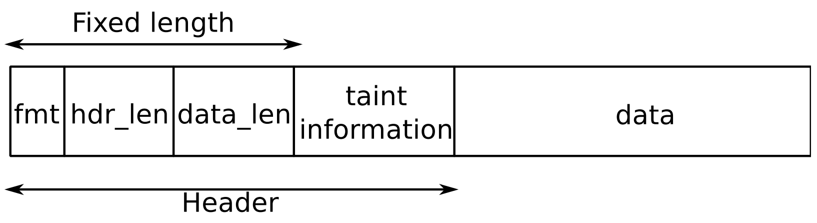


**Figure 3.2:** TaintExchange encapsulates data using a header to transparently inject taint information in data transfers.

protocol, following the format shown in Figure 3.2. The fields of the taint header are described in Table 3.1.

| **Taint Header Field** | **Description** |
|---|---|
| `fmt` | the format version of that taint information |
| `hdr_len` | the length of the header including that taint information |
| `data_len` | the length of the data payload (i.e., the data the application is actually transferring) |
| `taint information` | fine-grained taint information regarding the payload, and following the format specified by `fmt` |
| `data` | the data payload |

**Table 3.1:** Taint-header's fields. They include information regarding the taint status of the data, that is being transferred. The taint-header is transferred along with the real data.

#### 3.2.4.2 Composing the Taint Header

A taint header is created when `write-like` system calls, such as `write()`, `writev()`, `send()` etc., are executed and the destination descriptor of the system call is among the ones configured to transfer taint. The process' `tagmap` is referenced to determine which parts of the outgoing data message is tainted. Depending on the number of tainted bytes and their distribution, TaintExchange determines which format to use to encode the taint information for the data. We support two formats for encoding taint information. The first,

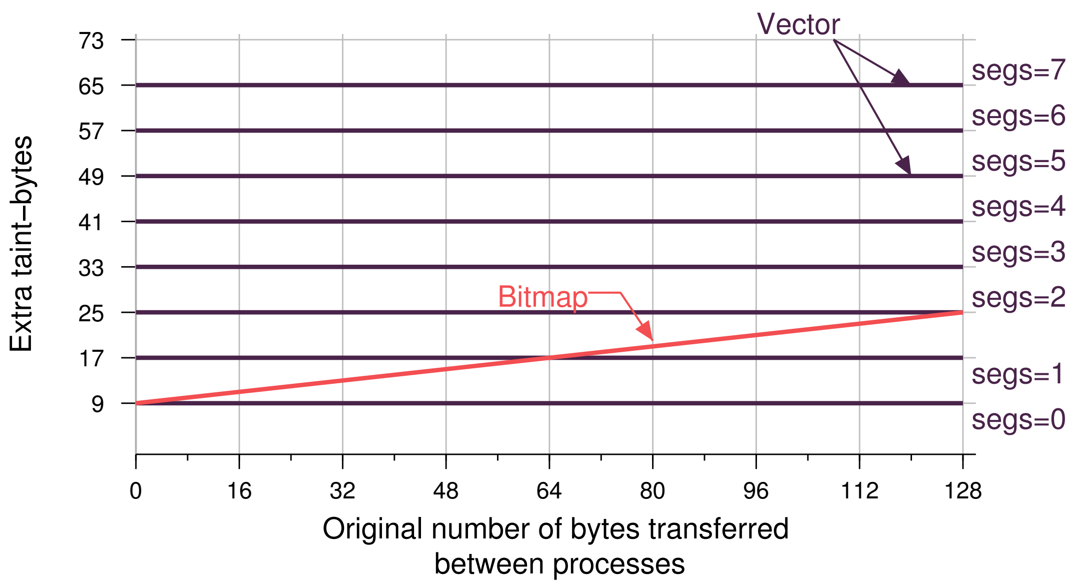


**Figure 3.3:** Space overhead of the different TaintExchange taint information transfer formats. For both formats, the fixed header size is 9 bytes. The overhead of the bitmap format is linear with the number of bytes transferred, while with the vector protocol it depends solely on the number of tainted data segments.

is a **bitmap** which contains one bit for every byte of data being transferred, and the second using a **vector** for each segment of data that is tainted. For example, if bytes 5 through 15 in the buffer to be transferred are tainted, the vector describing this segment will be `[5,10]`. That is, there is a segment of tainted data starting at offset 5 of the data and lasting for 10 bytes. The space overhead of these two formats is drawn in Figure 3.3. We see that depending on the number of tainted segments included the data payload a different format is preferable. Usually, for large continuous areas of tainted bytes, a vector proves to be a more efficient choice, whereas for sparsely tainted bytes the bitmap is preferable.

## 3.3 Implementation

### 3.3.1 The Libdft Data Flow Tracking Framework

TaintExchange operates by intercepting and instrumenting the system calls used for inter-process as well as for cross-host communication, while it relies on an already available tool, Libdft [36], to perform the taint tracking within each process. For this purpose, we instrumented the `socketcall` family of system calls (i.e., `socket()`, and `accept()`), the `dup()` system call, and the `read/receive`-like and `write/send`-like system calls. We also intercept the `open()`, `close()` and `mmap()` system calls for handling files.

For the *intra-process* dynamic taint tracking we chose Libdft [36], a dynamic data flow tracking (DFT) framework, designed to transparently perform DFT on binaries. Although our design is *independent* of the underlying data flow tracking system, for our implementation we chose Libdft because of its availability, well-defined API, and efficient instrumentation, which makes it one of the fastest process-wide taint-tracking frameworks. Libdft is used as a shared library offering a user-friendly API for customizing *intra-process taint propagation*, and can be used on unmodified multi-threaded and multiprocess applications. It relies on PIN [42], a dynamic binary instrumentation framework (DBI) from Intel, widely used in the implementation of other DFT tools [21; 86].

Similarly to PIN's, the Libdft API allows both instrumentation and analysis of the target process. In particular, Libdft's *I/O interface* component, offers an extensive *system call* level API, enabling instrumentation hooks before (`pre_syscall`) and after (`post_syscall`) every system call, while making use of the underlying DFT services. We have registered analysis callbacks for the "interesting" system calls, which get invoked when these system calls are encountered, to observe the process' communication channels, and to inject taint information along with the native data (i.e., the data the application is communicating). For the system calls that are not explicitly handled by TaintExchange, the default behavior is to clear the tags of the data being read (i.e., the data written in the process' memory). This way, **over-tainting** is avoided since "uninteresting" system calls, that overwrite program memory with new inputs read from the kernel, are not ignored. It is worth mentioning that Libdft does not suffer from taint-explosion (i.e., over-tainting data that should not be tagged), because it does not consider *control-flow* dependencies and it operates in user-space. Control-flow dependencies, kernel data structures, and pointer tainting have been identified as the prominent causes of taint-explosion [61].

### 3.3.2 Taint-Exchange Data Structures

**Tagmap structure.** The `tagmap` is the taint-tag storage, or else shadow memory, maintained by Libdft, reflecting the taint status of the running process' memory and CPU registers for each running thread. Its implementation plays a crucial role in performance and memory overhead. Libdft supports byte-level memory tagging, which is mapped to a single-bit tag in the process' tagmap, and four 1-bit tags for every 32-bit GPR. Libdft offers an extensive API for the update of the taint-tags in tagmap (e.g., `tagmap_setb`, `tagmap_getb`, `tagmap_clrb` are handling the taint-tag per byte of addressable memory).

**State_sfd global array.** The `state_sfd` is the global array of tainted descriptors, that designates the data streams, being monitored for tainted data. It is updated by the system call subset used for handling files and creating/closing sockets, and is indexed by the file (or socket) descriptor. Initially, all elements of the array are empty. The array is updated by the instrumented versions of `open()`, `socket()`, `accept()`, `dup()` and `close()` system calls, and variations of them.

**Taint_config configuration file.** `taint_config` is the configuration file for filesystem taint sources. The files listed in it should be written in full-path format.

### 3.3.3 Filesystem Taint Sources

Currently, our framework supports configurable taint sources from the file-system and the network. The `taint_config` file, lists the files that contain *data of interest*, which should be tainted and tracked throughout the monitored system. This is implemented, by the instrumented `open()` system call, which marks as tainted the `state_sfd` elements, that correspond to the files listed in `taint_config`. The descriptors of these files are considered input channels for tainted data. Therefore, whenever a system call, like `read()`, tries to read data from them the corresponding taint tags in the `tagmap` structure are updated accordingly.

### 3.3.4 Taint Propagation Over the Network

The main purpose of TaintExchange is the delivery of taint-tags along with the transferred data in all the tainted channels. To establish information about the TCP channels the `socket()` and `accept()` system calls are instrumented. For simplicity, in the current implementation, every network connection is considered *capable* of propagating tainted data, but this could be easily limited to work only on specific IPs. When a TCP connection is established, `state_sfd` structure is updated accordingly to add the new socket descriptor to the monitored channels.

The *cross-host* taint propagation mechanism is handled by the instrumented versions of `write/send`-like system calls on the sending side, and `read/receive`-like system call on the receiving side. When the sender transmits data by invoking a `write()` (or an equivalent) system call, TaintExchange constructs the corresponding taint-header according to the relevant taint-tags as reflected in the `tagmap` of the sending process and attaches it to the original data. The receiving side, will invoke an instrumented `read()` call (or an equivalent) to process the *extended data*, and update the process' `tagmap` with the taint-tags corresponding to the received data.

### 3.3.5 Cross-process Taint Propagation

Interprocess communication can happen through unix sockets, TCP/UDP sockets, pipes, and shared memory. If the processes are communicating via sockets or pipes, taint tags can propagate between communicating processes in the same manner we described in the previous section. The main difference is the source descriptor, which in the case of pipes is linked to the pid or name of the application the process is communicating with. IPC through UNIX sockets is very similar to TCP sockets, so the mechanism remains almost entirely the same. We are currently not handling data exchanged through shared memory segments.

## 3.4 Evaluation

Our aim is to demonstrate the communication overhead induced by the TaintExchange mechanism, when transparently passing taint information, along with real data, across the network. To assess the impact imposed by TaintExchange, we performed several micro benchmarks using utilities from the *lmbench3* [45] Linux performance analysis suite. During the tests we only used the bitmap format to represent taint information, as the overhead of the vector format can vary significantly depending on the application scenario.

Our testbed consisted of two identical virtual machines located on a server equiped with 3.50GHz eight core Intel Xeon E3-1270 V2 CPUs and 32GB RAM, running Xen 4.1.4. The virtual machines are assigned 1 virtual CPU and 1 GB RAM each and are running Linux (Debian "squeeze", with kernel version 2.6.32). The version of Pin used during the evaluation was 2.10 (build 45367). When conducting our experiments, the hosts were idle with no other user processes running under taint-tracking apart from the evaluation suites.

### 3.4.1 Bandwidth

Since TaintExchange intercepts socket connection calls to inject the additional taint information, we used lmbench's bandwidth benchmark *bw_tcp* to measure the impact of our approach when moving the "extended" data over the network. bw_tcp measures TCP bandwidth by creating two processes, a server and a client, that are moving data over a TCP/IP connection. We repeated our tests with data of different sizes and against four different scenarios:

(a) native execution of the benchmark.

(b) using *Pin*'s runtime environment alone, loading no pintool. We chose to use this setting, to establish the overhead imposed by the Pin runtime environment alone as lower bound for the overhead from the DBI.

(c) using *Libdft-dta*, a tool based on Libdft to employ basic dynamic taint analysis. We used this tool to achieve an estimation of the overhead imposed by Libdft and the data flow tracking tool.

(d) using *TaintExchange* with the bitmap format for the taint information.

We repeated the measurements 100 times for each of the different message sizes, i.e., 128, 256, 512, 1024, 2048 and 4096 bytes, sending a total of 50 MB (lmbench $M$ parameter) in each iteration, while we left all other options to their defaults settings. The results presented in Table 3.2 are mean values (MB/sec) and the standard deviation for the different message sizes, as shown at the leftmost column, and for the three different settings mentioned in the first line of the table. Note that the numbers in the parentheses at the leftmost column represent the additional bytes, i.e., the taint-header, that are tranferred along with the application data when employing the TaintExchange mechanism. When using Pin and Libdft-dta bytes transferred are the original application data. For instance, when `bw_tcp` client is running over Pin alone or over Pin and Libdft-dta it is reading 256 bytes of data. But in the setting employing TaintExchange the client will receive 41 more bytes along with the original 256 bytes.

| | **Pin** | | **Libdft-dta** | | **TaintExchange** | |
|---|---|---|---|---|---|---|
| **Total Bytes** | **Mean (MB/s)** | **St.Dev.** | **Mean(MB/s)** | **St.Dev.** | **Mean(MB/s)** | **St.Dev.** |
| **128 (+25)** | 20.41 | 0.20 | 13.18 | 0.19 | 10.79 | 0.40 |
| **256 (+41)** | 36.58 | 1.00 | 24.21 | 0.38 | 22.09 | 0.96 |
| **512 (+73)** | 66.95 | 4.13 | 45.03 | 0.97 | 37.21 | 0.21 |
| **1024 (+137)** | 124.89 | 17.47 | 84.91 | 4.96 | 57.99 | 0.43 |
| **2048 (+264)** | 255.31 | 36.84 | 163.28 | 15.41 | 89.71 | 0.94 |
| **4096 (+521)** | 524.76 | 56.39 | 333.49 | 9.18 | 136.87 | 2.95 |

**Table 3.2: Throughput (MB/sec) per message size when running *bw_tcp*.** The numbers in the parentheses represent the additional bytes sent along with the real data, when using TaintExchange. There is an obvious impact on the throughput of the TCP sockets in the case of TaintExchange justified by the additional data sent along with the application data, the overhead of intercepting the system calls and finally by the additional `read()` calls inserted by our mechanism, for every TCP socket `read()`.

The results from the throughput measurements are also illustrated graphically in Figure 3.4 for better interpretation. The right subfigure of Figure 3.4 represents the throughput of `bw_tcp` when running natively. Note that the effect of Pin is quite pronounced espe-

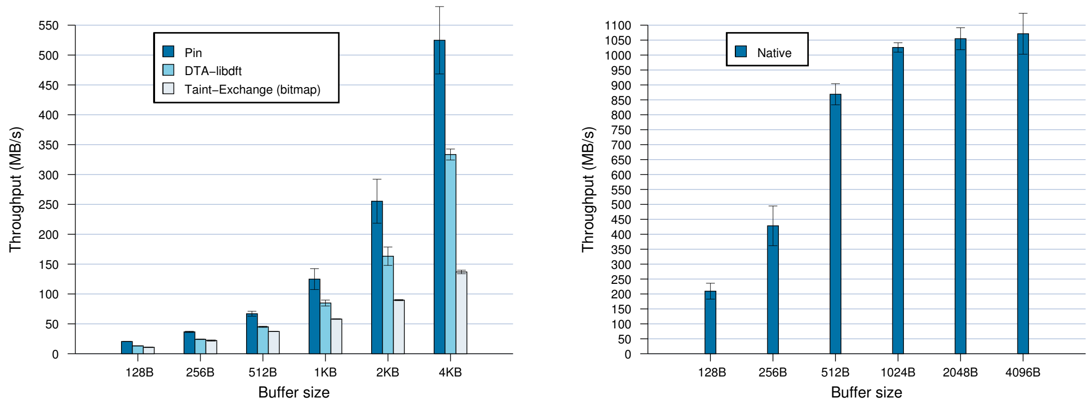


**Figure 3.4: TCP socket throughput measured with *lmbench* for various buffer sizes.** The numbers in the parentheses represent the additional bytes sent along with the real data, when using TaintExchange. We draw the mean and standard deviation. Note that the effect of Pin with the buffers drawn here is quite pronounced compared with the native execution of `bw_tcp`. Note though that the overhead diminishes as the size of the data read increases.

cially for the smaller-sized buffers but it diminishes as the size of the data read increases. Therefore, we chose to draw the results of the native execution separately for the chosen buffer-sizes as our goal is to illustrate the TaintExchange overhead over the one imposed by the underlying taint tracking system and it would not be as clear if we included all four scenarios in the same graph.

There is an obvious impact on the throughput of TCP sockets for the three settings shown at the left subfigure, in comparison to the native execution, which becomes more severe as the size of data transferred increases. As expected, TaintExchange has the largest impact of the three scenarios. This is justified by the number of additional data sent every time in comparison to the other settings but also by the overhead caused by the interception of the system calls as well as the two additional `read()` system calls that are performed for every application's `read()`.

When running our tests we also noticed that the standard deviation was smaller in the case of small-sized buffers in all three settings and increases as the number of data transferred grows. The total system call cost in the case of a `read()` system call is affected primarily by three factors:

- the mode switch cost, which is independent of the amount of data read,
- the time to transfer data from the network driver to the kernel and
- the time to copy the data to user space for the user process.

The last two factors are affected by the number of bytes read each time. Therefore, in the case of small-buffers the mode switch cost essentially *hides* the cost for the copying of the bytes and the oscillation in the `read()` system call cost seems to be quite small in comparison to larger read buffers. In conclusion, the bandwidth is smaller when reading small data buffers but the variation in the bandwidth results is also quite small.

Another observation that comes out from Figure 3.4 is that althought the bandwidth is degrading when applying the TaintExchange mechanism, there is smaller variation in the bandwidth results than in the other two scenarios for the same data sizes. Regarding the bandwidth the degrading was quite expected since we are sending more data, i.e., the taint-header, along with the application data.

As described in Section 3.3.4 every `read()` system call of the application (in this case `bw_tcp`) is invoking the relevant instrumented version of the system call, which introduces two additional `read()` system calls for every application `read()` from the TCP socket. These additional `read()` calls are used for reading the taint-header in two pieces. The first one reads the fixed part of the taint-header, i.e., the `fmt`, the `hdr_len` and the `data_len` which add up to 9 bytes. The second one reads the bitmap with the taint information for the corresponding application data. Therefore, the `read()` calls of the small-sized buffers are also responsible for lowering the bandwidth more thanwould be expected based on the additional bytes transferred.

The dominant instrumentation and processing inserted by Pin and Libdft-dta have an effect on the standard deviation of the results. The respective wait time hides the translation, tag propagation and other processing required by the three settings on top of the native version of the application. Also, in TaintExchange the two additional `read()` calls of small-sized parts of the taint header are also contributing to the reduction of the variation of the results in comparison to the other two scenarios.

We draw also the average slowdown observed when running the bandwidth benchmark using Pin alone, Libdft-dta and our TaintExchange, compared with native execution. We obtained the average slowdown $S_{slow}$ using the following formula:

$$S_{slow} = \frac{Throughput_{TaintExchange(avg)}}{Throughput_{Native(avg)}}$$

for the slowdown factor imposed by TaintExchange.

| **Total Bytes** | **Pin** | **Libdft-dta** | **TaintExchange** |
|---|---|---|---|
| **128 (+25)** | 10.2x | 15.9x | 19.4x |
| **256 (+41)** | 11.7x | 17.7x | 20.6x |
| **512 (+73)** | 13x | 19.3x | 23.7x |
| **1024 (+137)** | 8.2x | 12.1x | 17.7x |
| **2048 (+264)** | 4.1x | 6.9x | 11.7x |
| **4096 (+521)** | 2x | 3.5x | 7.8x |

**Table 3.3: Slowdown factor over native execution of `bw_tcp`.** The numbers in the parentheses represent the additional bytes sent along with the application data, when using TaintExchange. The columns represent the slowdown factor for each of the three settings.

Table 3.3 shows the slowdown factor for the three settings in comparison to the native execution of `bw_tcp`. Figure 3.5 illustrates the same results graphically. There is an obvious slowdown on the performance of the bandwidth benchmark when executed over each one of the three settings. Pin alone imposes a slowdown ranging from 2x to 13x. As expected, the slowdown is larger in the case of TaintExchange, although large part of this is attibuted to Pin and Libdft-dta, as it is built on top of these two and inherits their slowdown. It is worth noticing though that the relative slowdown rises in all three settings for the small packets (<1KB), but it shows the exact opposite trend for all the larger files. The throughput measurements for the native execution illustrated in Figure 3.4 can explain to some extent the slowdown trends. As it is obvious from the relevant subfigure, the throughput is rising as the size of the data-buffers becomes larger, but for larger buffers (1KB, 2KB and 4KB)

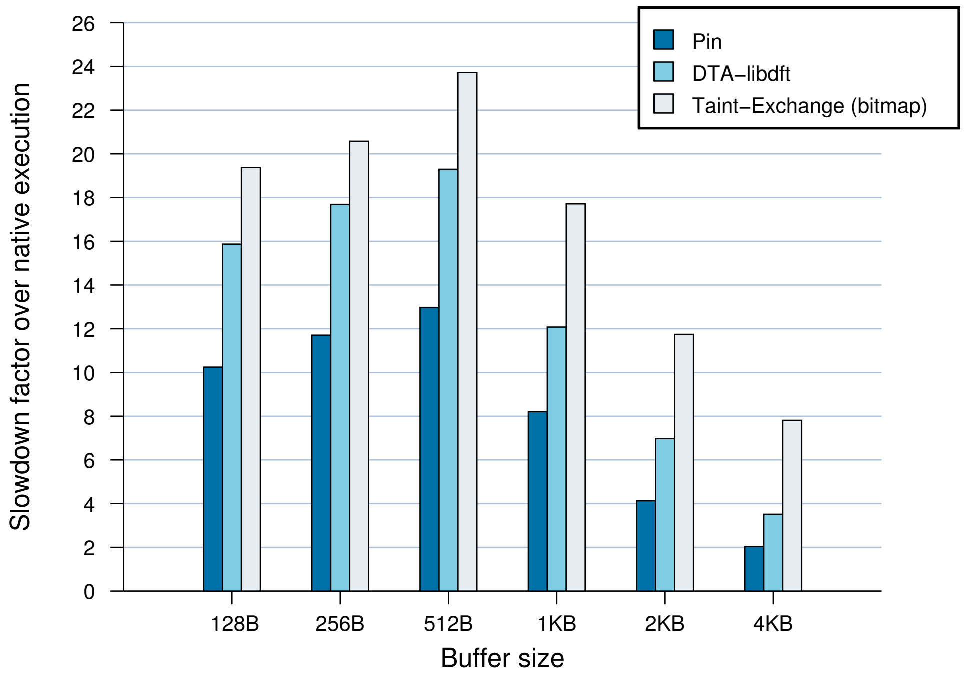


**Figure 3.5: Slowdown factor in comparison with the native execution of `bw_tcp`.** Note that the native execution reaches a lower bound for buffers larger than 1KB (as seen in Figure 3.4). Therefore, the relative overhead for the three settings diminishes when reading these larger buffers.

the throughput does not differ significantly among them. This obeservation implies that the benchmark seems as if it reaches an upper bound of how many bytes it can read per second. Therefore, to read more bytes the whole wait time becomes larger, which works in favor of the other three settings, since their processing time is affecting less the total slowdown. Another reason for this drop on the slowdown for buffer sizes larger than 1KB, is that each benchmark's native `read()` system call implies two additional `read()` calls. The first one is reading always 9 bytes, but the second one is reading equivalent number of data as large as 1/8 of the size of data that the application's `read()` is going to receive. Therefore, in the case of 4KB of application data, the second `read()` call will read 512 bytes, which causes a significant overhead as depicted in Table 3.3. Therefore, as the size of the application data to be read becomes larger, the relative slowdown caused by TaintExchange will become less observable.

| Operations | Native | Pin | Libdft-dta | TaintExchange |
|---|---|---|---|---|
| **Upload a 4-min video (30MB)** | 0.028 sec | 0.057 sec | 0.09 sec | 0.219 sec |
| **Upload 100 Photos (200MB)** | 0.187 sec | 0.381 sec | 0.6 sec | 1.461 sec |
| **Upload CD 80min (700MB)** | 0.652 sec | 1.334 sec | 2.099 sec | 5.114 sec |
| **Upload 1-hour HD movie (3GB)** | 2.867 sec | 5.854 sec | 9.212 sec | 22.445 sec |
| **Upload files to the cloud (10GB)** | 9.558 sec | 19.514 sec | 30.706 sec | 74.816 sec |

**Table 3.4: Operational overhead**. The numbers in the parentheses represent the total size of bytes read for the specific activity. Assuming that the files are read in 4KB chunks, the expected time overhead for the four settings is presented in the relevant columns.

Finally, another parameter worth exploring for the evaluation of TaintExchange is the **operational overhead**, i.e., the perceived impact for the user when employing TaintExchange for his operations (e.g., uploading files on a remote server). Table 3.4 shows some user operations and how they would be impacted if TaintExchange mechanism was in use.

### 3.4.2 Latency

Since the implementation of TaintExchange is mostly based on the instrumentation of system calls, we also employed the *lat_syscall* benchmark to measure the latency impact of the three implementations. We used *lat_syscall* with the `read()` and `write()` system calls, to show how these operations are affected. We chose these system calls since they are the ones handling the transfer of the tainted information and therefore they were affected the most by TaintExchange. In the performed tests, "`lat_syscall read`" measures how long it takes to read one byte from */dev/zero*, whereas "`lat_syscall write`" measures how long it takes to write one byte to */dev/null*. The results are presented in Figure 3.6. The conditions during these measurements were the same as with the *bw_tcp* benchmark, regarding repetitions and the calculation of the mean and standard deviation from the original measurements.

Note that Pin, as probably most DBI frameworks, greatly affects system call latency (approximately 20x slower than native) because it receives control before and after every call. The observed overhead is attributed to the overhead of Pin for the dynamic instrumentation analysis of the process, as well as the overhead inserted by Libdft performing the

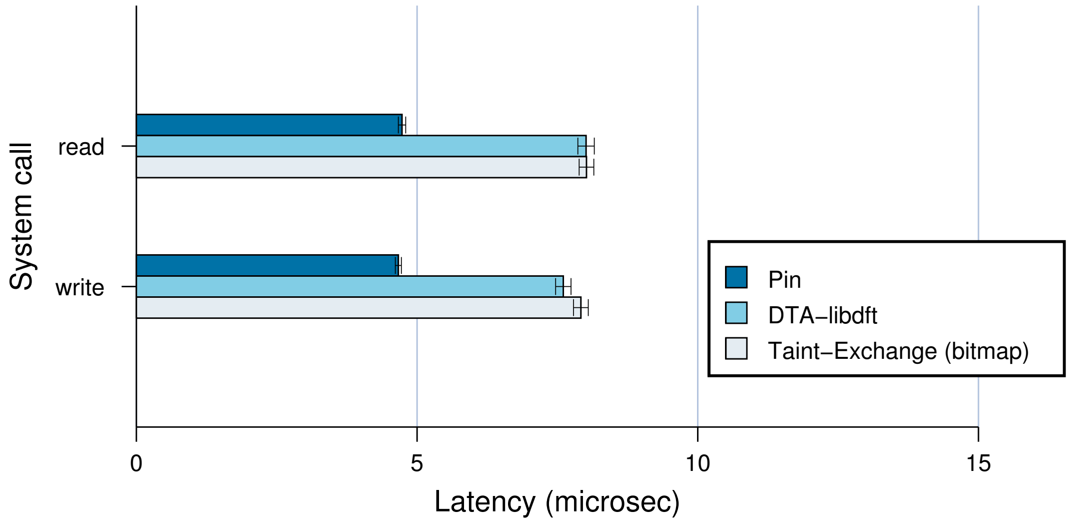


**Figure 3.6:** System call latency measured with *lmbench's* benchmark `lat_syscall`. The buffer read and written is 1 byte. Note that the additional latency imposed by TaintExchange for `write()` system calls is due to the extra analysis that TaintExchange employs to determine whether the output stream is among the taint sinks.

taint-tracking. The additional overhead imposed by TaintExchange, apart from the obvious reasons, such as the instrumentation of these system calls for handling the taint-headers and the continuous update of the taint data structures, is also implementation-specific. Specifically for the `write()` call used by `lat_syscall`, the overhead imposed by TaintExchange in Figure 3.6 is due to the extra analysis that TaintExchange has to do to determine whether the output stream is among the taint sinks. No additional taint-headers are involved in this case. The oscillation in the results presented is small as expected, since `lat_syscall` uses small sized buffers (1 byte) for the measurements, but this attributes to the general increase in the system call overhead.

## 3.5 Discussion

We presented a preliminary implementation of TaintExchange, our approach for handling *cross-application* and *cross-host* transfer of tainted information. There are some extensions to this work presented in this chapter, which we addressed in the next version of TaintExchange (see Chapter 4). In the current implementation, every TCP socket is by default considered among the monitored "channels", that participate in the taint-propagation process. We build a more fine-grained configuration procedure in the next version, so that certain IPs can be included (or excluded) from participating into the propagation of the taint-tags over the network.

In addition, we support persistent taint information storage for files, to be able to handle both tainted and untainted data stored in a file. Briefly, an auxiliary file per original file is used to maintain the information on the tainted bytes of the original file. A similar scheme with the one used for passing taint information over the network is also explored (e.g., bitmap for files with interleaved tainted and clean data, and vectors for files that store tainted data in large blocks). Coarser-grained tracking can already be performed. For instance, when tainted data are written to a file, taint-exchange can consider the entire file as tainted.

Finally, in order to reduce the overhead of the inserted taint header we explore the use of TCP optional headers to pass the taint information. This option would not only help us trivially implement communication between a native and a taint-exchange application, but it could also potentially improve the overall performance of our proposed mechanism. Unfortunately, we found certain limitations in this approach when trying to implement it for the CloudFence framework, but we did manage to employ it in the Cascading Rescue Points mechanism (see Appendix I), that also incorporates the logic of the *taint transfer mechanism* described in this chapter.

## 3.6 Conclusions

We presented TaintExchange, a *generic* cross-host and cross-process taint tracking framework. TaintExchange enables the transfer of fine-grained taint information across processes and the network. It does so by intercepting I/O system calls to transparently inject and extract information regarding the *taintness* of every byte transferred between processes running under TaintExchange. It also provides a flexible mechanism for easily customizing the sources of tainted data, be it a network socket, a file, or an IPC mechanism like a pipe or UNIX socket. Our evaluation of TaintExchange shows, as expected, that I/O is affected because of the additional data being sent, and the utilization of the same channel to do so. Nonetheless, we believe that the overhead is small, specially when compared with the high overheads imposed by the various dynamic taint tracking systems.

# Chapter 4

# Cloud-wide auditing mechanism

In this Chapter we discuss the design and implementation of CloudFence, a cloud-wide fine-grained data flow tracking framework for cloud-hosted services. CloudFence is a generic facility offered by the cloud hosting provider enabling the service providers to easily integrate data flow tracking techniques in their applications and thus enhance the security of their services.

## 4.1 Introduction

The motivation behind CloudFence was to create a data tracking service offered by the cloud hosting providers in addition to their existing hosting environment, to both service developers as well as the users of their services, the data owners. We envision that this service offered by a large, reputable cloud provider could help build users' confidence much more quickly and effectively than small individual service companies otherwise might.

As suggested by the name of our framework, the goal was to create a mechanism that allowed data flow tracking within the boundaries of a well defined *data flow domain*, i.e., the component applications of a *composite* cloud-hosted service (e.g., web server, back-end database etc.) or inside the domain of many services offered by the same cloud provider (e.g., Google, Amazon etc.).

Our goal is two-fold. We first want to support benign service providers, who are willing to integrate CloudFence in their applications to enhance the security of their provided services and build the *security reputation* of their services. Although we trust the intentions of these service providers to protect sensitive user data, we don't trust that their applications are free from software vulnerabilities and misconfigurations. Therefore, our approach offers protection against many classes of attacks that can lead to unauthorized data access (but which do not allow arbitrary code execution), such as SQL injection, command substitution, parameter tampering, directory traversal, and other prevalent web attacks that are seen in the wild. Note that in case of attackers who gain arbitrary code execution, we can no longer guarantee accurate data tracking, since they can not only compromise our framework, but can also exfiltrate data through covert channels.

But besides protecting against application-level vulnerabilities, an equally important goal is to bring into users' and service providers' attention any unintended data exposure that may lead to unauthorized access. In particular, we seek to reinforce the confidence of end users for the safety of their data, beyond any assurances offered by the online service, by giving users the ability to *independently audit* their cloud-resident data through a different—and potentially more trustful—entity than the actual provider of the service. Therefore, we envision that DFT will be offered as a service by the cloud infrastructure provider not only to their tenants but also to their tenants' customers, i.e., the data owners.

This can be achieved by taking advantage of the *multi-party* trust relationships (see Chapter 2) that exist in typical cloud environments [17], in which the service provider is different than the provider of the infrastructure on which the service is hosted. From the users' perspective, there is an inherent shared responsibility between the cloud and the service providers regarding the security guarantees of the provided service. Although end users do not interact directly with cloud providers, they implicitly trust their infrastructure—the systems in which their data are kept. *We aim to exploit this implicit trust for the benefit of all parties by introducing a* direct *relationship between end users and cloud providers*, as shown in Figure 2.4. With data flow tracking as the basic underlying mechanism, the cloud provider enables users to directly inspect the audit trail of sensitive data that was handled by services hosted on the cloud provider's infrastructure.

### 4.1.1 Incentives

While the trust relationship between users and service providers is not altered, CloudFence gives users an elevated degree of confidence by allowing them to independently monitor their private data as it propagates through the cloud. In fact, users are more likely to trust a large, well known, and highly reputable cloud provider, as opposed to a lesser-known developer or company (among the thousands that offer applications and services through online application distribution platforms).

CloudFence offers service providers two main benefits. First, with minimal effort, it allows them to provide an extra feature that reinforces the trust relationship with their users. This can also be considered as a competitive advantage: among two competing services, privacy-conscious users may prefer the CloudFence-enabled one, knowing that they will have an additional way of monitoring their data. Second, it empowers service providers with the ability to confine the use of sensitive user data in well-defined network and file system domains, and thus prevent inadvertent leaks or unauthorized data access. Besides guarding user data, service providers can also take advantage of CloudFence to implement an additional level of protection for their own digital assets, such as back-end credentials, source code, or configuration files.

Finally, by integrating CloudFence in its infrastructure, a cloud provider offers added value to both its tenants and their users, potentially leading to a larger customer base. Given the shared responsibility between cloud and service providers regarding the safety of user data, both have an incentive to adopt a system like CloudFence as a means of providing an additional level of assurance to their customers

### 4.1.2 Challenges

While a promising approach to security problems in the cloud setting, integrating data flow tracking mechanisms in this setting requires the consideration of several issues. First, the on-demand consolidation of computing elements in cloud settings allows service providers to easily "glue" together functionality and content from third-party sources, and build feature-rich applications. For instance, the term *mashup* is colloquially used to refer to web application hybrids that combine services—from potentially untrusted principals—to offer "rich web experience." As the benefits of this approach are numerous, it is critical not to interfere with that paradigm while enabling data tracking. We consider this as the **transparent tracking** requirement. The applied DFT method should support incremental deployment by not requiring intrusive changes, such as manually annotating source code [58], custom OSs [81], or modified hypervisors [83].

Second, tracking granularity plays a crucial role in the effectiveness of DFT. A service provider can choose between tracking data as small as a single byte [51], which enables robust protection against extreme cases of data leakage, or employing a more coarse-grained (and hence error-prone) approach [48]. However, fine-grained DFT has a significant performance cost, as tracking logic becomes more intricate (e.g., consider the case of two 32-bit numbers with only some of their bits marked as sensitive). We consider this as the **fine-grained tracking** requirement, which suggests performing DFT at the appropriate granularity for balancing overhead and accuracy.

Third, given the range of cloud delivery mechanisms with different compositional characteristics (e.g., IaaS, PaaS), it is important to ensure that dynamic collaboration is taken into consideration when performing DFT. The *domain-wide tracking* requirement refers to the precise monitoring of data flows beyond the process boundary. Examples include intra-host application elements that communicate through the file system or OS-level IPC, or consolidated application components running on remote endpoints.

Finally, a major challenge in supporting data auditing for these online services, that tend to have a very large number of users, is to allow for concurrent propagation of tagged data that carry different tags for each user. Therefore, since the main concept behind CloudFence requires that personal data is marked with a respective user ID, the goal is to support applications with a **practically unlimited number of users**, and thus the DFT component should be able to handle a respectively large number of tags. This requirement is highly challenging, as most DFT frameworks provide either a single tag [59; 16; 19] or just a few—usually eight [36; 85; 84].

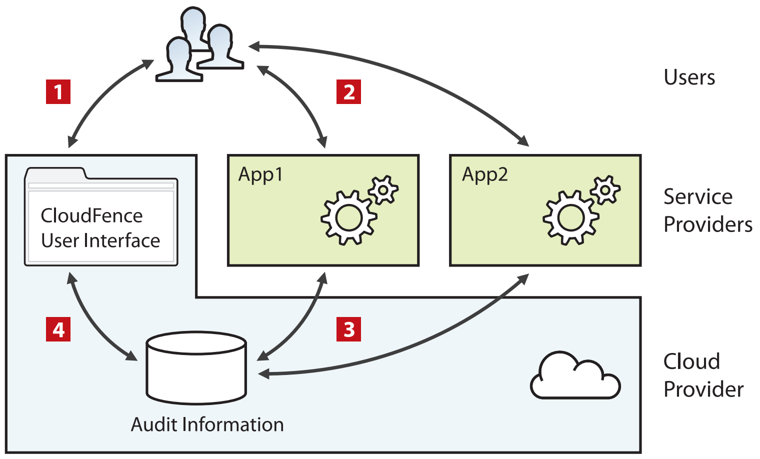


**Figure 4.1:** Main interactions between the different parties involved in CloudFence-enabled applications. Users register with the cloud provider (1), and then use the services offered by various service providers using the same set of credentials (2). Sensitive data is tagged and tracked transparently throughout the cloud infrastructure (3). Users can audit their data through a web interface exposed directly by the cloud provider (4).

## 4.2 System Overview

Figure 4.1 shows the main interactions among the different parties that are involved in CloudFence-enabled services. Initially, users register with the cloud provider (1) and acquire a universally unique ID, distinctive within the vicinity of the cloud provider's infrastructure. Then, they use the online services offered by various service providers by providing the ID acquired from the previous step (2).

The actual mechanism used for conveying user IDs to CloudFence is not addressed in this work. As possible solutions, the service provider can either request from users to provide their ID during the sign up process on the corresponding application, or in case a cloud-wide identity management system is in place, the application can access the respective ID transparently by requesting it directly from the cloud provider (after the user has successfully authenticated). Such functionality is gaining traction among cloud providers. Indicatively, Amazon recently launched the "Login with Amazon" feature [3], which allows users to login to Amazon-hosted services with a single account, while it also supports federated login using Google and Facebook identities [2].

Sensitive data is tagged by the service provider with the supplied user ID, and is tracked throughout the cloud infrastructure, while audit information is gathered and stored at the cloud provider (3). At any time, users can monitor the audit trails of data directly through the cloud provider using a user-friendly web interface (4). Service providers also have read access to the collected audit data through a specialized API. Besides user data, CloudFence can be used to protect the service providers' own assets, such as back-end credentials, configuration files, and source code. This can be achieved by tagging them as sensitive, tracking their propagation through the cloud infrastructure, and enforcing fine-grained perimetric access control based on the applied tags.

## 4.3 Design

Our design takes into consideration all the challenges presented in section 4.1.2, while providing fine-grained auditing capabilities for the cloud-hosted services. It consists of three main components: the **data flow tracking (DFT) subsystem**, the **API stub**, and the **audit trails generation component**.

**DFT subsystem.** The data flow tracking component is the nucleus of CloudFence and an essential part of our architecture. It performs fine-grained, byte-level explicit data flow tracking without requiring any modification to applications or the underlying OS, while at the same time handles $2^{32}$ different tags. Briefly, our DFT component is application *agnostic*; it uses dynamic binary instrumentation for retrofitting the DFT logic into unmodified binaries dynamically, at runtime, and supports tracking across processes running on the same or remote hosts by utilizing the TaintExchange mechanism (see Chapter 3). Specifically, it *piggybacks* tags on the data exchanged through IPC mechanisms or network I/O channels, keeps persistent tag information for marked data written to files, and handles (un)marshalling transparently.

**API stub.** The CloudFence API allows service providers to *tag*, i.e., attach metadata information, on sensitive user data that enters their applications. Note that CloudFence does not require application modifications regarding data tracking (e.g., extensive annotations). However, it requires small changes to application code for marking sensitive information.

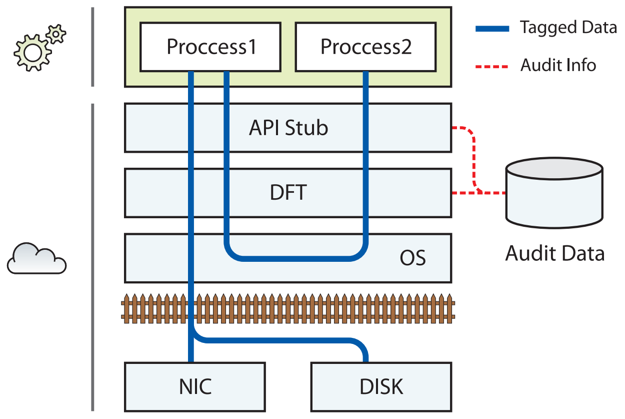


**Figure 4.2: The CloudFence architecture**. `Process1` and `Process2` are components of a consolidated application, hosted on the CloudFence enabled cloud.

**Audit Trails Generation.** The main purpose of CloudFence's auditing mechanism is to generate *detailed audit trails* for tagged data. Therefore, we implemented a generic "verbose" logging mechanism that collects information for tagged data accesses and generates audit logs. The generated trails are stored in a database outside the vicinity of the service provider in an "append-only" fashion to prevent tampering of archived audit trails. The DFT component pushes audit information to the audit component whenever tagged data is written to a cloud storage device or pass through I/O channels to endpoints inside or outside the cloud.

Figure 4.2 illustrates the overall architecture of CloudFence. The two processes in the upper part of the figure represent components of a consolidated application, while the rest of the components are part of the cloud provider's infrastructure. Note that for the rest of our discussion, we assume that the service provider relies on an IaaS delivery mechanism, and in this example both processes run on the same (virtual) host. However, CloudFence can be seamlessly employed in PaaS and SaaS setups. Data that are tagged as sensitive (denoted by the solid line in the figure) is tracked across all local files, host-wide IPC mechanisms, and selected network sockets. Tagged bytes that are written to storage devices, or transmitted to remote hosts, result in an audit message.

### 4.3.1 Data Flow Tracking

Although our DFT component is inspired by previously proposed DFT tools [36; 16], for reasons that are explained in detail in Section 4.4, we built it from scratch to provide a transparent, fine-grained, and domain-wide tracking framework suitable for the target cloud environment. We employ Intel's Pin [42], a dynamic binary instrumentation toolkit in order to retrofit the DFT logic into unmodified binaries at runtime. Pin injects a tiny user-level virtual machine monitor (VMM) in the address space of a running process, or in a program that launches itself, and provides an extensive API that CloudFence uses for inspecting and modifying (dynamically at run-time) the process' code at the instruction level. In particular, CloudFence uses Pin to analyze all instructions that move or combine data to determine data dependencies. Then, based on the discovered dependencies, it instruments program code by injecting the respective tag propagation logic *before* the corresponding instructions. Both the original and the additional instrumentation code, i.e., the data tracking logic, are re-translated using Pin's just-in-time compiler. However, this process is performed only once, right before executing a previously unseen sequence of instructions, and the instrumented code is placed into a code cache to avoid paying the translation cost multiple times.

Note that the instrumentation happens at the user level, which offers not only performance benefits, but also alleviates any "semantic gap" issues due to VMM introspection [50]. We also decided to implement our DFT design at the user level to make it more easily deployable by a cloud provider, compared to a modified hypervisor-based solution, since it does not require intrusive changes in the underlying cloud infrastructure.

## 4.4 Implementation

From a high-level perspective, most of CloudFence's functionality is built around the DFT component, except the user interface, which is a user-accessible web application coupled with a back-end database. Our current prototype is implemented using Pin 2.10, and works with unmodified applications running on x86-64 Linux. The data auditing component is layered on top of CloudFence using system call interposition.

### 4.4.1 32-bit Wide Tags and 64-bit Support

The *shadow memory* used for keeping data tag information plays a crucial role in runtime performance. Previously proposed DFT systems mainly use two approaches for tagging memory: (i) bit-sized tags [59], whereby every byte of addressable memory is tagged using a single bit in the shadow memory, and (ii) byte-sized tags [36; 19; 16], whereby each byte of program memory has a sibling in the shadow arena. In between, systems like Umbra [85] and TEMU [75] allow for various byte-to-byte and byte-to-bit configurations, as well as for lossy encodings (e.g., four bytes of addressable memory can be tagged using one byte). TEMU, in particular, enables very flexible tagging, by supporting tag values of arbitrary size, at the expense of higher runtime performance overhead [76]. CloudFence trades some of this flexibility for a lower runtime slowdown.

Implementing 32-bit wide tags requires re-designing the shadow memory from scratch. Driven by the fact that data from different sources, carrying dissimilar tags, are rarely combined in our context (e.g., the memory bytes of two different credit card numbers are unlikely to be combined), we opted for a solution that greatly increases the number of tags stored per datum, but unavoidably also increases the overhead of tag combination operations. More precisely, each tag value is stored as a different number, and when two tags are combined, a *new* tag value is created. Still, incorporating this change alone in commodity DFT systems [36; 16] would only increase the number of tags from 8 to 256, using byte-size tagging. Hence, our next step was to expand the tag size from one to four bytes, allowing for $2^{32}$ tags.

The transition to 64-bit not only helps overcoming available memory limits, but also enables further optimizations. The relatively expensive translation that involves shadow page table lookups is replaced by a faster one. Taking advantage of the ample address space, we split it in two parts: the shadow memory and the actual process memory. This is achieved by reserving the shadow memory as soon as the process is started, forcing it to allocate memory only in its own part. Address translation then becomes as simple as scaling the virtual address and adding an offset. For example, the memory tags of address `vaddr` can be obtained as follows: `taddr = (vaddr << 2) + toff`, where `toff` corresponds to the offset of the shadow memory. CloudFence reserves 16TB of user space for the application and 64TB for the shadow memory, resulting in an offset value of `0x100000000000`. However, it allocates pages in the shadow region on demand, i.e., only when a page contains tag information. As every byte of tracked program data needs four more bytes for its tag, part of the physical memory footprint of a process increases by a factor of four.

### 4.4.2 Lazy Tag Propagation

Most x86-64 instructions fall into one of two major categories: arithmetic and data transfer. For the latter, tags are always propagated following the flow of data, i.e., we *always* copy the tags of the source operand over the tags of the destination operand. On the other hand, whenever the destination operand is derived from a combination of its own value and that of the source operand, there are three possible cases, each having a different impact in terms of performance:

```
/* arithmetic instructions */
if (shadow[src] != 0)
  if (shadow[dst] == 0)
    shadow[dst] = shadow[src];
  else if (shadow[dst] != shadow[src])
    shadow[dst] = combine(shadow[src], shadow[dst]);
```

Starting from the worst case, (`else if`), if both operands have different tags, a lookup is performed and a *new* tag is generated. If only the source operand is tagged, its tag is copied to the destination. If the source operand is not tagged, no action needs to be taken. Given that only a small amount of data is usually tagged in our scenarios (recall that we care for discrete pieces of sensitive information), we optimized our design for the last case using Pin's API for fast conditional instrumentation. Arithmetic instructions are instrumented with a lightweight check of whether the source operand is tagged (*fast path*). In case it is, the appropriate propagation actions are performed according to the code snippet above (*slow path*). This avoids in the common case the excessive register spilling that usually occurs by larger instrumentation code that needs more registers [42]. Finally, tag information is kept into an array-like data structure, indexed by tag value. For every tag, we store whether it is basic or compound, and in the latter case, the tag values it stems from. Compound tags can be traced back to the basic tags they are made of, by recursively querying this data structure.

### 4.4.3 Tag Persistence

Accurate data flow tracking throughout a cloud-based application requires persistent data tags and tag propagation across different processes, which may run on different (physical or virtual) hosts. To this end, we have built a layer on top of our prototype for supporting tag propagation across BSD sockets, Unix pipes, files, and shared memory.

**Sockets and Pipes.** Exchange of tag information over sockets and pipes is handled by embedding all relevant data tags along with the actual data that is being transferred. Maintaining the tag propagation logic completely transparent to existing applications, without modifying them or breaking the semantics of their communication, is the most challenging part of this effort. In our prototype, the exchanged tag information consists of a copy of the relevant area of the shadow memory that CloudFence maintains for the transmitted data, encoded in RLE (Run Length Encoding). Recall that only a very small part of data is usually tagged, so most of the time there will be minimal communication overhead—just a header field that contains the number of triplets.

**Synchronous I/O.** We hook the `write`, `send`, and `writev` system calls using Pin's hooking API, and transmit tag information before the actual data of the original system call. Similarly, we hook the `read`, `recv`, and `readv` system calls, and read the tag information before the actual data. Messages can be received (i) at once, (ii) split in multiple parts, or (iii) interleaved. In the first case, the tag data and the original data are received within the same receiving operation, so they are simply decoded and attached to the original data. For messages received through several read operations, the receiver initially buffers the tag information, and each time a new part is received, its corresponding tag information is appended until the whole message is received. The most difficult case is when the size of the send buffer does not match the size of the receive buffer. Such cases are handled by changing the return value of the read operation to match the end of the current message.

**Non-blocking I/O.** For non-blocking I/O, the above system calls may return a special error code as if the requested operation would block (`EAGAIN`). If such an error occurs when trying to read the embedded tag information, control returns immediately to the application, as if its read operation failed. If some, but not all, of the tag data is available, the available part is buffered and CloudFence emulates a "would block" error, as if the read operation would block. Similarly, for write operations, we keep accounting of the relevant encoded shadow memory data that is actually sent, and emulate `EAGAIN` errors until all relevant shadow data has been completely transmitted.

**Multiplexed I/O.** For `select`, `poll`, and `epoll`, we chose to trade a small performance overhead in favor of a safer hooking implementation. Before read or write operations, the system blocks until all tag information is received or sent, as in synchronous I/O. A more robust implementation would check if any of the ready-to-read file descriptors are waiting to receive a new message, and attempt to first retrieve its tag information. If only partial information is available, we can buffer it, and remove the file descriptor from the returned set of `select` or `poll`, as if it were not ready to be read. However, such an implementation could break application semantics, since the actual intention of the application after a `select` or `poll` invocation is not known in advance, e.g., the application could use `recvmsg`, or not read any data at all.

**Files.** Tag information should persist even when data is written into files, so that it can be later retrieved by the same or other processes. CloudFence supports persistent tagging of file data by employing shadow files. Whenever a file is opened using one of the `open` or `creat` system calls, CloudFence creates a second shadow file, which is mapped to memory and is associated with the original file descriptor. Whenever a process writes a file using `write`, `writev`, or `pwrite`, the tag information of the relevant buffer (or buffers, in case of `writev`) is also written in the appropriate offset of the mapped shadow file. Similarly, after a read operation using `read`, `readv`, or `pread`, the relevant tag information from the corresponding shadow file is represented at the destination buffer. To limit space requirements, we take advantage of sparse files, which are supported by most modern OSs. For the common case of a file with just a few tagged bytes, the shadow file will consume just 4× the size of only the tagged data, while shadow files that contain no tag information require no extra space at all.

**Shared Memory.** Our current implementation supports shared memory regions allocated with `mmap`, but it can be easily extended to cover POSIX API calls (e.g., `shm_open`) or SysV API calls (e.g., `shmget`). CloudFence hooks calls to `mmap`, and for each shared memory region, it creates a shadow copy to hold tag information.

### 4.4.4 Data Flow Domain

Data flow tracking is performed within the boundaries of a well-defined *data flow domain*, according to the components of the online service. Whenever some tagged data crosses through the defined boundary, e.g., when a destination file or host does not belong to the specified domain, CloudFence logs the action in the audit database, and, depending on the configuration, may block it.

To automate the configuration of tag propagation between processes that exchange data through the network, CloudFence maintains a global registry of active sockets for the domain by hooking the `connect` and `accept` system calls. For each connection attempt, the initiator's IP address and port are recorded in a list of endpoints that support tag propagation. At the same time, the other endpoint's address is queried in the list, and if it exists, this means that both endpoints support it, and consequently tag propagation is enabled for this connection. At the server side, upon a call to `accept`, and before the call actually returns, the server's address is inserted in the list of sockets that support tag propagation (if not already present). After `accept` returns, the client's address is queried in the list, and if it exists, then tag propagation is enabled. Note that service providers must only specify the programs that comprise the cloud application, and then the rest of the tag propagation logic is determined automatically.

### 4.4.5 User Interface

CloudFence's user interface leverages Cloudopsy [78], a web-based data auditing dashboard, which is described in details in Chapter 5. Cloudopsy uses visualization and automated audit log analysis to provide users who lack technical background with a more comprehensible view of audit information. For example, the event of a user's credit card number being sent to an external host other than those included in the trusted domain, which could be a data leak incident, would be clearly depicted in the audit graph presented to the user. In particular, this suspicious data flow would be presented in the audit graph by a directed link in a pre-defined color (e.g., red) indicating the possible data leak. Details regarding the different formats of the audit graphs presented to the end users and the service providers are out of the scope of this chapter but are discussed in our paper [78]. Although this service targets mostly end users, it also provides administrators with a graphical overlook of the overall application dependencies and data flows of the service. The visualization of audit events allows for the immediate verification of legitimate operations and the identification of unexpected transmissions, which otherwise might have remained hidden much longer in the reams of raw audit logs, thus reducing decision and reaction latency.

## 4.5 Evaluation

We evaluate CloudFence in terms of ease of deployment in existing applications, runtime performance, and effectiveness against data leakage attacks, using two real-world applications: an e-commerce framework and a bookmark synchronization service. Our experimental environment consists of three servers, each equipped with two 2.66GHz quad core Intel Xeon X5500 CPUs and 24GB of RAM, interconnected through a Gigabit switch. To better match a cloud infrastructure environment, two of the servers run VMWare ESXi v4.1, and all CloudFence-enabled applications were installed in virtual machines. The third server was used to simulate clients and drive the experiments. In all cases, the operating system was 64-bit Debian 6.

### 4.5.1 Deploying CloudFence

#### 4.5.1.1 Online Store

The first scenario we consider is an online store hosted on a cloud-based infrastructure. Typically, during a purchase transaction, sensitive information, such as the credit card number and the recipient's postal and email address, is transmitted to the online store, and from there, usually to third-party payment processors. The service provider can incorporate CloudFence in the e-store application to allow users to monitor their data, as well as to restrict the use of sensitive data within the application's domain. The developers of the e-store know in advance the entry points of sensitive user data, as well as which processes and hosts should be allowed to access this data. For instance, after users input their credit card information through the e-store front end, it should only be accessed by the e-store's processes, e.g., its web and database servers. The only external channel through which it can be legitimately transmitted is a connection to the third-party payment processor, i.e., a well-known remote server address.

The application we chose for this scenario, called VirtueMart [69], is an open source e-commerce framework developed as a Joomla component, and is typically used in PHP/MySQL environments. We configured VirtueMart to accept payments only through credit card, and set up actual electronic payments through the Authorize.Net payment gateway service using a test account. To incorporate CloudFence, we had to add just a few lines of code at the user registration and order checkout modules. Specifically, we added a new input field in the registration form for the user's unique ID, a new column in the user's database table, and a few lines of code for storing the ID in the database along with the user's info. For the checkout phase, we added a few lines of code in the script that processes the payment information. First, the user ID for the current session is queried from the database. Then, the HTTP POST variable that holds the credit card number is tagged by calling the `add_tag` API function through a PHP wrapper. Finally, the data flow domain of the application comprises the web server, the database server, and any other processes these two may spawn.

#### 4.5.1.2 Bookmark Synchronization

This use case stems from the increased demand for data synchronization services, as users typically have many internet-connected devices. The scenario in this case is to host a bookmark synchronization service on the cloud based on SiteBar, an online bookmark manager written in PHP. When adding a link to SiteBar, users have the option to set it as public or private, and may change it later. From the side of the service provider, we would like to tag any private links as sensitive.

Incorporating CloudFence in SiteBar was very similar to the previous case, as both applications are written in PHP and use MySQL as a database back-end. On the other hand, changing the source code to tag the sensitive data (user links marked as private) was slightly more elaborate, as the sensitivity level of data can change dynamically. Thus, we had to change the code that adds a link so as to tag it in case it is marked as private, as well as the code for editing a link. It is essential to update the copy in the database on edit, in order for the change to be persistent.

### 4.5.2 Effectiveness

To evaluate the effectiveness of CloudFence, we tested whether it can identify illegitimate data accesses performed as a result of attacks. We used exploits against two publicly disclosed vulnerabilities in the studied applications. The first allows authenticated users of SiteBar versions earlier than v3.3.8 to read arbitrary files [5]. This is the result of insufficiently checking a user-supplied value through the `dir` argument, which was used as the base directory for reading language specific files, as shown below:

```
sprintf($dir.'/locale/%s/%s',$var1,$var2);
```

Passing a file name that ends with the URL-encoded value for the zero byte (`%00`) causes the `open` system call to ignore any characters after it and read the supplied file:

```
http://SB_APP/translator.php?download&dir=/var/lib/mysql/SCHEMA/TABLE.MYD%00
```

Using SiteBar v3.3.8 on top of PHP v5.2.3, we repeatedly read files by exploiting this bug through a browser on a remote machine. CloudFence reported successfully all accesses to data with persistent tags in the file system, which in our case corresponded to files belonging to MySQL.

Another type of attack that usually leads to information leakage is SQL injection. The main cause, again, is the insufficient user input validation. To demonstrate the effectiveness of CloudFence on preventing this type of attacks, we used another real-world vulnerability in VirtueMart version 1.1.4 [6]. The value of the HTTP GET parameter `order_status_id` is not properly sanitized, allowing malicious users to change the SQL SELECT query by using a URL like the following:

```
http://VM_APP/index.php?option=com_virtuemart&page=order.order_status_form
&order_status_id=-1' UNION ALL SELECT ... where order_id='5
```

which results in the execution of the following query:

```
SELECT * FROM jos_vm_order WHERE order_status_id=-1' UNION ALL SELECT ...
FROM jos_vm_order_payment where order_id='5';
```

The above query returns a row from the `jos_vm_order_payment` table, which holds the credit card numbers, instead of the table `jos_vm_order`. As in the previous case, we installed the vulnerable version of VirtueMart on top of PHP v5.3.3, and tried to access the credit card numbers by exploiting this bug. In all cases, CloudFence identified the exfiltration attempt, as the relevant data had been tagged as sensitive upon entry.

### 4.5.3 Performance

To assess the runtime overhead of CloudFence we compare it against Libdft [36], a data flow tracking framework for commodity systems, as well as the unmodified application in each case. We chose Libdft because it is publicly available, and it also uses Pin for runtime binary instrumentation. Libdft maintains a shadow byte for each byte of data, and thus supports only eight tags per byte, represented by individual bits. Compared to CloudFence, which uses four shadow bytes per actual byte of data, Libdft has thus significantly lower shadow memory requirements. Furthermore, representing each tag using a single bit allows Libdft to implement aggressive optimizations for tag propagation using bitwise OR operations. In contrast, CloudFence has to synthesize a new tag whenever two existing tags must be combined, and then maintain their association. As we show in this section, despite the increased requirements of CloudFence in terms of memory consumption and computation for supporting 32-bit tag propagation, its runtime overhead is comparable to Libdft for the cloud-based applications we consider.

#### 4.5.3.1 Microbenchmark

We begin by focusing on the overhead of tag propagation, and specifically exploring tag *generation*, which is the worst case scenario for CloudFence. The test program we used

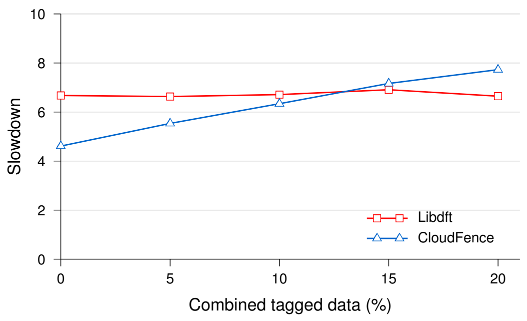


**Figure 4.3:** Slowdown as a function of the percentage of data with different tags that must be combined (worst case). CloudFence not only supports $2^{32}$ tags (instead of just eight for Libdft), but also is faster for the cases we consider in our setting ($< 10\%$).

allocates two buffers, `buf_a` and `buf_b`, of the same size. The bytes of `buf_a` are tagged with the value `1`. Each byte of a specified part of `buf_b` is tagged with a different value, starting from `2`. Then, each byte of `buf_a` is added to the corresponding byte in `buf_b`, and the process repeats for a number of times. For each add operation, if the current byte in `buf_b` is not tagged, then `buf_a`'s tag is copied over, otherwise, their tags are combined and a new one is generated.

Figure 4.3 shows the slowdown imposed by data flow tracking for CloudFence and Libdft. CloudFence not only provides extra functionality that is crucial for cloud environments, but at the same time it is even faster than Libdft for the cases we consider, i.e., minimal combination of data marked with different tags, as the personal data of different users are not likely to be intermixed. The extreme case in which each add operation generates a new tag results in a 20× slowdown (upper bound).

#### 4.5.3.2 Real-world Applications

We decided to focus our experiments on VirtueMart, as it represents the most complicated scenario among the chosen applications. VirtueMart stresses a larger part of CloudFence's functionality, and therefore results in a larger but more representative performance impact in comparison to SiteBar. In our experiment, we measure the sustained throughput of user requests that VirtueMart can handle when processing concurrent purchase transactions from multiple users. We installed two instances of VirtueMart on virtual machines in our

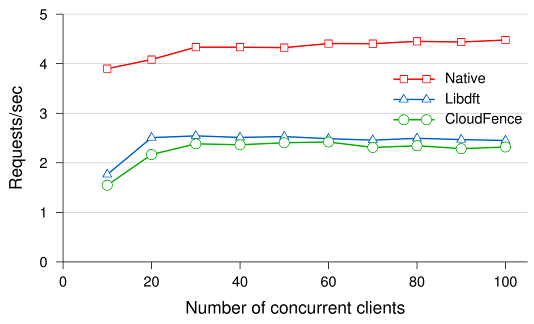


**Figure 4.4:** Request throughput for VirtueMart using the default web server configuration.

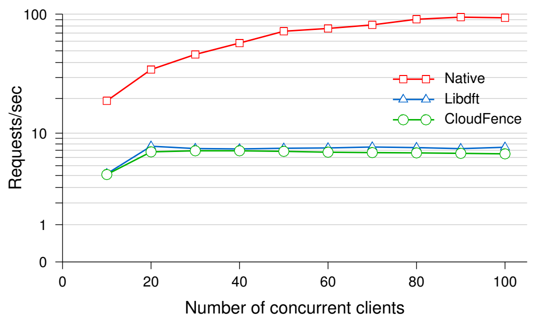


**Figure 4.5:** Request throughput for VirtueMart using Facebook's HipHop.

testbed. One runs on top of Apache using the PHP module, and the other was compiled after transforming the PHP to C++ using Facebook's HipHop. In both cases, MySQL was the database back-end. To generate a realistic and intensive workload, we used a second host connected through a Gigabit switch that emulated typical client requests for placing product purchases. The Gigabit network connection minimizes network latency, increasing this way the imposed stress on the server when emulating multiple concurrent user transactions.

Instead of performing the same request over and over, we simulated more realistic conditions by replaying complete purchase transactions. Each transaction consists of nine requests: retrieve the front page, login, navigate to the product page for a specific item, add that item in the shopping cart, verify the contents of the shopping cart, checkout, enter payment info, confirm the purchase, and logout. For each of these requests, the web clients also download any external resources, such as images, scripts, and style files, emulating the behavior of a real browser, without performing any client-side caching. We should stress that VirtueMart was fully configured as in a real production setting, including properly working integration with Authorize.Net for processing credit card payments using a test account.

Figure 4.4 shows the sustained request throughput for a varying number of concurrent web clients, when VirtueMart is running i) natively, ii) on top of Libdft, and iii) on top of CloudFence. The request throughput was calculated by dividing the number of requests by the total duration of each experiment. In all runs, each client was configured to perform three end-to-end transactions, so that the number of requests per client remains consistent across all experiments. We see that although CloudFence reduces the throughput in half, its performance is comparable to Libdft despite its much more CPU and memory intensive tag propagation logic. A significant fraction of the slowdown for both systems can be attributed to Pin's overhead for runtime binary instrumentation. We should note that the server throughput in the native case is not bounded due to limited computational resources, but rather due to the default configuration of Apache, which uses a pool of 10 processes for serving concurrent clients. Thus, to be more precise, CloudFence took advantage of the available cycles and imposed additional overhead.

Figure 4.5 shows the results of the same experiment using the compiled version of VirtueMart and the built-in multi-threaded web server that comes with the HipHop code transformer. This time, the native throughput is bound due to CPU saturation. In the worst case, the request throughput is roughly 13 times slower when CloudFence is enabled. Another contributing factor to performance degradation as concurrency increases lies in the underlining binary instrumentation framework. To provide thread-safe execution of system call hooks, Pin serializes their execution using a process-level global lock. This kind of hooks are used by both CloudFence and Libdft, which again achieve comparable performance.

## 4.6 Discussion

**Over-tagging.** We opted for a design that does not suffer from over-tagging or tag pollution. Specifically, CloudFence does not tag pointers nor it propagates tags due to implicit flow, which prior work has shown to produce over-tagging [21; 61]. Moreover, it takes into consideration that certain system calls write specific data to user-provided buffers. For instance, consider `gettimeofday`, which upon every call overwrites the user space memory that corresponds to one, or two, `struct timeval` data structures. Such system calls always result in sanitizing (untagging) the data being returned, unless CloudFence has installed a callback that selectively tags returned data.

**Under-tagging.** CloudFence only supports explicit data flows, which can lead to under-tagging whenever the service provider uses a code construct that copies sensitive data using branch statements. As an example, consider the code snippet `if (in == 1) out = 1;`. Although the value of `in` is copied to `out`, any tags associated with it are not. DTA++ [34] addresses this issue by identifying implicit flows within information-preserving transformations and generating rules to add additional tags only for a certain subset of control-flow dependencies. During our evaluation, we identified a couple of such cases, in AES encryption (used in SSL, MySQL, and the Suhosin PHP hardening extension) and Base64 encoding. Such cases should be handled manually by the service provider, by hooking the corresponding functions and copying the tag information from their source to the target operand using the `copy_tag` function.

**Binary Instrumentation.** The choice of a DFT framework based on binary instrumentation unavoidably comes with an increased runtime penalty. However, we have managed to support 32-bit wide tags per byte while maintaining a similar, or even lower, overhead compared to existing systems, allowing the practical use of CloudFence in real settings. Alternative implementations of this functionality within language runtimes [11; 13], or even at hardware, have been shown to decrease the imposed overhead.

**Fine-grained Tracking.** CloudFence is a general framework designed for use with all the components of a cloud-based service without modifications. To achieve this, we chose fine-grained over coarse-grained (process-level) tags, although this comes with an increased overhead. Other implementations [48] that tried to avoid this overhead by coloring each time the entire process serving the HTTP request for user data with the tag or color representing this specific user, ended keeping extra information in the application level, when its processes where handling data from multiple users at the same time. As expected, in this case the process would be assigned a tag representing both users, but if there was no merge of the data, this data would still carry the new tag instead of the initial unique user tag. Therefore, the audit capability provided to the end users for their cloud-based data would not be as precise in the case of process-level tags as it is in our fine-grained implementation.

**Alternative DFT Tools.** CloudFence has been influenced by previous DFT proposals, with the closest being Libdft [36], but none of them would suffice for our goal. In particular, although CloudFence and Libdft share the same underlying DBI framework (Pin), they differ completely in (i) shadow memory design, (ii) tag propagation logic, and (iii) I/O interface. Libdft uses dynamically allocated shadow memory (tracks memory allocations) and a page-table-like structure for performing virtual-to-shadow memory translations. CloudFence, on the other hand, reserves part of the abundant 64-bit address space for storing the 4-byte wide tags (per byte of program memory), thus making memory-to-tag translation without a lookup. Regarding the low-level optimization that Libdft uses, we retained what it considers as `fast_vcpu` and `huge_tlb`. Finally, the system call interface of 64-bit Linux is slightly different from the 32-bit version and the system call numbers are shuffled. Hence, the I/O system call descriptors that CloudFence uses had to be adapted.

**Universally-unique User IDs.** The use of the same ID across all services may raise privacy concerns, as this allows the cloud provider to track user activity within its premises. Although cloud providers could track users even if a cloud-wide user ID was not used, e.g., by combining user-identifying features such as browser fingerprints and HTTP cookies [39], a unique ID per user certainly makes tracking easier. Cloud providers, however, have already started offering access to hosted services through in-house [3] or third-party web identity providers [2], and this trend is expected to continue, as it improves user experience by having to manage fewer accounts.

## 4.7 Conclusions

One of the most highly cited concerns regarding cloud-hosted services is the fear of unauthorized exposure of sensitive user data. Users have to trust the efforts of both the third-party service provider and the cloud infrastructure provider for properly handling their private data as intended. In this work, we take a step towards increasing the confidence of users for the safety of their cloud-resident data by introducing a new direct relationship between end users and the cloud infrastructure provider. CloudFence is a service provided by the cloud infrastructure, that offers data flow tracking abilities to both service providers and their users for user data collected in the realm of cloud-based services. In particular, CloudFence allows users to independently audit their data by the cloud-based services, and additionally enables service providers to confine data propagation and protect their digital assets within well-defined domains.

# Chapter 5

# Cloudopsy

In this Chapter we discuss the design and implementation of Cloudopsy, a generic facility offered by the cloud hosting provider directly to the customers of the cloud-hosted services with the goal of providing them with *secure* and *comprehensible* auditing capabilities with respect to the handling of their sensitive user data collected in the realm of these third-party online services. While the framework is targeted mostly towards the end users, Cloudopsy provides also the service providers with an additional layer of protection against illegitimate data flows, e.g., inadvertent data leaks, by offering a graphical more meaningful representation of the overall service dependencies and the relationships with third-parties outside the cloud premises, as they derive from the collected audit logs. The novelty of Cloudopsy lies in the fact that it leverages the power of visualization when presenting the final audit information to the end users (and the service providers), which adds significant benefits to the understanding of rich but ever-increasing audit trails created by our auditing framework in chapter 4.

## 5.1 Introduction

As businesses and individuals rely on the cloud for most of their everyday tasks, it is inevitable that *personally identifiable information* (PII) are also stored and processed on third parties premises, like the cloud, outside the administrative control of its owners. From the data owner's perspective, users are usually not interested in, or are simply lacking the technical background to understand the specifics of the security mechanisms employed to ensure that their data is not being abused or leaked. Users though wish to have a better understanding on how data is being handled by the cloud-hosted online services, or at least have an indication in case a misuse event occurs.

To address this privacy challenge, we take a step towards increasing cloud *transparency* with respect to the handling of sensitive user data, by providing *secure* and *comprehensible* data auditing capabilities to the clients of the cloud-hosted services. Auditing has been long employed with success as an added security measure in alleviating user security concerns in domains like banking, where information security is mission critical. Unlike other privacy protection technologies that are built on enforcing policies and preventing data flows, auditing focuses on keeping data usage *transparent* and *trackable.* Even though information flow policies and enforcement techniques have the same goals and use similar mechanisms with our approach, we chose to focus our efforts in *auditing*, due to the concerns that enforcement could have adverse effects, like breaking parts of services, which is a highly probable case for the feature-rich, constantly changing and multi-component cloud-hosted services. Thorough, efficient auditing in cloud computing remains a challenge even for straightforward web services, but more research is directed towards this goal [71; 38; 43; 18]. Regular auditing and compliance with security frameworks will not only insure safety of the cloud systems but also be a point of differentiation for marketing of cloud services.

In Chapter 4 we discussed the design and implementation of our cloud-wide data flow tracking framework, CloudFence, which apart from sensitive data flow tracking it is also enabling logging capabilities of these flows among the services running within the cloud infrastructure provider's premises. CloudFence is a great tool for creating detailed and verbose audit logs for the cloud-resident sensitive user data collected in the realm of the cloud-hosted services. However, since the audited online services continuously increase in size (i.e., number of users, components, etc.), our mechanism has the potential to create massive numbers of audit logs and trails of information. Therefore, although a successful auditing mechanism its usefulness is limited because the task of interpreting the raw logs and identifying interesting details would be extremely challenging for end users, especially for the ones that are lacking the technical background. In consequence, in order to increase the effectiveness of the audit information collected by CloudFence, we decided to structure and present our results in a way that would be comprehensible by end users, which led to the introduction of Cloudopsy, an enhanced auditing cloud service.

## 5.2 Approach

To address the limitations of CloudFence and increase cloud data owners' understanding with respect to the handling of their sensitive data, we worked on a more meaningful representation of the sensitive data flows, as they derive from the events captured in the audit logs. The novelty of our approach lies in the fact that our mechanism leverages the power of *visualization* and automatic analysis when presenting the final audit information to the data owners. This novel feature can significantly reinforce end-user awareness regarding the use and exposure of their sensitive data by the cloud-hosted services. In a few words, we claim that audit visualization is a necessary feature in the analysis of audit logs, as it helps in recognizing events and associations recorded in the logs that might have otherwise remained unnoticed or would have taken longer to realize.

As was the case with CloudFence, to truly support our vision, we envision cloud providers offering the Cloudopsy service in addition to their existing hosting environment, to both service providers and data owners. A large, reputable cloud provider can help leverage user confidence much more effectively than a smaller lesser-known developer or company (among the thousands that offer applications and services through online application distribution platforms). In addition, the Cloudopsy platform architecture could dramatically reduce the per-service development effort required to offer data auditing, while still allowing rapid development and maintenance.

### 5.2.1 System Overview

A high-level overview of Cloudopsy's mechanism is depicted in Figure 5.1. The participating entities are the same as with CloudFence: the *cloud provider*, who offers the cloud infrastructure or platform, the *service providers* deploying the online services, and the *end users* or *data owners*, who are the clients of the online services. The main interactions between

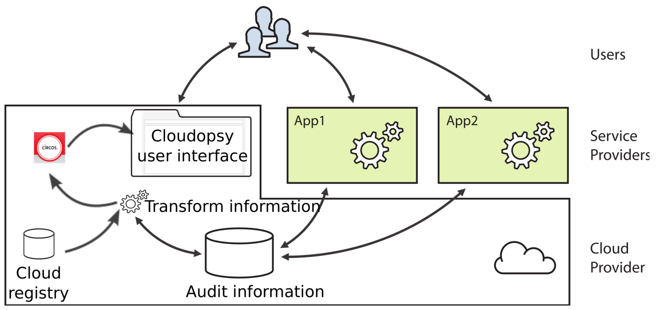


**Figure 5.1:** Cloudopsy Architecture. Sensitive data is tagged and tracked transparently throughout the cloud infrastructure, while audit information is gathered and stored at the audit database. The audit logs are analysed and transformed to meaningful graphs depicting the sensitive data flows for all the services hosted on the same cloud infrastructure. Users can see these audit graphs through the Cloudopsy web interface exposed directly by the cloud provider.

the different parties involved remain the same. Initially, users of the cloud-hosted services are assigned a *unique ID*, which will be their unique cloud-wide identification. Each end user has to register/sign in with this ID every time he uses one of the services offered by one or more service providers and are hosted on the same cloud provider's premises, if he wishes to have his sensitive data audited for the these services (e.g., Google Apps). On the other hand, the providers of the cloud-hosted services need to define the boundaries of their *data flow domain* (described in Section 4.4.4), i.e., the component applications of their *composite* service (e.g., web server, back-end database etc.). and also tag the sensitive data of each user with the supplied user ID the first time it enters their applications. Sensitive data is then tracked transparently throughout the cloud infrastructure, while audit information is gathered and stored in audit databases hosted on the cloud provider's infrastructure. At any time, users can monitor the audit trails of their data directly through the cloud provider using Cloudopsy's web interface. Although end-users are the main focus of this approach, mostly because they are usually the ones that lack the technical background to understand the collected raw logs, service providers can also benefit from Cloudopsy, for quickly identifying unexpected behaviour of their service, e.g., "suspicious" flows of user data outside of the service's data-flow domain.

From a high-level perspective, Cloudopsy utilizes the underlying CloudFence tracking mechanism (described in Chapter 4), for the domain-wide tracking as well as for the creation of the audit trails, but extends it with analysis and transformation components with the purpose of effectively converting the raw textual audit logs to more meaningful graphical outputs before presenting them on the Cloudopsy user interface.

## 5.3 Design

Cloudopsy's mechanism can be divided in three main components: (a) the **generation** of the audit trails, (b) the **transformation** of the audit logs for efficient processing, and (c) the **visualization** of the resulting audit trails.

### 5.3.1 Audit Logs Generation

To generate audit logs, Cloudopsy relies on information flow tracking techniques to track the entire flow of information in the realm of participating services hosted on the same cloud provider [55]. Separate audit logs are created for each cloud-hosted service and they are later correlated and filtered for creating the different audit logs that will be presented to data owner or the service provider requesting the audit. From a high level view, after one user's data enter the service, they are colored with a unique ID bound to that particular user, and audit information is generated every time it crosses the defined boundary of the data flow domain of the service, e.g., when colored data is about to be send to a cloud storage device or host inside or outside of the cloud, that does not belong to the defined components of the service. These events are recorded in an audit back-end database in an "append-only" fashion. Each component of the online service hosted on the cloud provider premises pushes audit messages to the audit database. Note that the auditing mechanism of Cloudopsy is designed to generate "verbose" and detailed audit logs. Although the same audit trails will be used to provide audit information to both service providers and end users, the final information displayed to them will be different. The end user sees information regarding his own data, whereas the service provider has access to the audit logs of all user data handled by his service.

In addition to identifying individual user's data, the colors that are assigned to user data can also reflect their type. That can be accomplished by assigning them a second ID or sacrificing some bits of information, previously part of the user ID, and using them for indicating their type or class. For instance, possible classes of data could refer to credit card numbers, e-mails, social security numbers (SSN), etc. The class of user data can be assigned by the service provider, or otherwise would be automatically determined by content scanners [23] that identify known patterns of PII like SSNs.

### 5.3.2 Audit Trails Processing

The audit logs produced from the auditing component include raw log data in a textual form and need further processing before reaching the visualization component.

```
16/Feb/2013:12:56:25 send 192.168.1.42:443 [DATA-uid1]
```

The necessary transformations include: (a) mapping of the applications that comprise the cloud-hosted services, (b) filtering out irrelevant information from the "verbose" audit logs according to the request and transforming the result to a format understood by the visualization component, and (c) correlating relevant information stored in several log files that were generated for the different cloud applications exchanging user data.

More specifically, Cloudopsy maintains a *global registry* of the active connections for the cloud-wide domain that it monitors. In order to transform the audit logs in an appropriate format for the next phase, technical details are used from this registry, and through automated procedures audited events are combined to create a more appropriate presentation of the collected audit information for input to the visualization component. Several other mechanisms are applied to the "verbose" audit logs to extract only the necessary segments for the final log information that will be presented to the end users (or the service providers). For example, information unrelated to "suspicious" flows of user data will not be extracted for the next phase.

### 5.3.3 Audit Trails Visualization

The visualization mechanism of Cloudopsy is a significant enhancement to previous auditing mechanisms as the power of images can offer to both service providers and end users, a better understanding as well as deeper insights in the real flows of user data. It was a challenge though to choose the appropriate and meaningful graphical representation of the collected audit data, which we discuss in more details in Section 5.4.3. In a few words, the visualization component receives the transformed audit logs and processes them according to a set of different parameters and represents the graphic results in a circular layout representing the cloud on Cloudopsy's web interface. The services monitored by Cloudopsy are included in the final graphical representation and the recorded transfers of the colored user data will be represented by links between them as well, colored and directed according to the information from the audit logs. As expected, the generated output from the visualization component differs not only between users but also between the data owners and the providers of the online service.

## 5.4 Implementation

### 5.4.1 Cloud-wide auditing mechanism

In previous work we implemented a cloud-wide fine-grained data flow tracking framework [55] for cloud-hosted services. This DFT framework is used for the auditing of the data flows and the generation of the raw audit logs that will be analyzed by Cloudopsy. We assume that the service providers have integrated this mechanism in their applications to enhance the security of the provided services and leverage the generic facility offered by the cloud provider.

From a high-level perspective, the auditing mechanism is built on top of a user-level data flow tracking framework libdft [36], based on runtime instrumentation and is integrated into the components of the service and works as follows: the data that is "tagged" as *sensitive* is tracked across all local files, host-wide IPC mechanisms, and selected network sockets. Audit information selected by the auditing component is kept in a back-end database located outside the vicinity of the service providers, and more importantly operates in an append-only fashion for preventing tampering of the archived audit trails. The audit information collected by the DFT component captures leakage events that result from writing the marked data into files or remote endpoints. Each entry of the audit log will include information about the time, the running application, the action, the destination stream (a network address or a file path) and the tagged data.

Our first prototype of the auditing component is implemented using Intel's binary instrumentation tool Pin 2.10, and works with unmodified applications running on x86-64 Linux. It performs byte-level data flow tracking and supports 32-bits tags, allowing support for $2^{32}$ different tags values (colors) per byte – consequently, the same number of different users. In a newest version of the auditing mechanism we used the last two digits of the tag to represent different classes of user's sensitive data. The auditing component provides a transparent, fine-grained and domain-wide data flow tracking mechanism suitable for our target cloud environment.

### 5.4.2 Visualization component

Our decision on enhancing our audit log analysis mechanism with visualization techniques was influenced by user studies comparing the effectiveness of visual presentations, in comparison to linear textual presentations of similar results to the ones we are handling in our audit logs analysis. In particular, in our case, since we wanted to represent the relationships and flow patterns of user data within the services in and outside of the cloud, Circos [41] seemed as a good candidate. Although Circos was originally developed to provide a better display of the chromosomal relationship between various species, it has been successfully used also for the graphical representations of other types of data, i.e., network traffic. We chose Circos also due to its support for a circular data domain layout, which resembles the cloud, and its multiple customizations opportunities, and finally due to its competence of smoothly partitioning the final image into any number of disjoint regions, drawing attention to multiple different events in a single display. Circos uses a circular ideogram layout to facilitate the display of relationships between pairs of services running on the cloud by the use of links (straight lines or Bezier curves), which can visually encode the position, size, and orientation of interacting elements.

Circos requires a specific format for the input data, which we generate through the transformations described in Section 5.3. In addition to our transformed audit log files, we also created configuration files including the necessary parameters for correctly generating the output required. The goal was to effectively present, in the final output, the monitored data flows within the cloud, as well as the ones "leaking" information out of the cloud premises, faithfully matching what is described in the initial audit files.

We provide two types of visual representations of data flows. For end users, we present a graphical display of the flow of their sensitive data that the cloud-hosted online service obtained as the result of their interactions with this service, e.g., a purchase from an online store would result into the storage of a user's credit card number in the service's back-end database. Cloudopsy in this case summarizes the transfers of this data in a circular graph, representing each data exchange as a link between the involved services. This can help users gain a deeper understanding of how their data are handled in a facile way.

On the other hand, Cloudopsy's output for the provider of the service is a much "richer" graph of the audited transmissions entailing information for *all* clients' sensitive data collected. In particular, since the service provider needs to analyze the very large and usually complex audit log files, the visualization of the audited events is extremely beneficial for him as he can immediately verify legitimate operations or identify unexpected "suspicious" transmission patterns that might have otherwise remained opaque in the reams of the audit logs. Furthermore, the service provider can also see the flow of data between components of his service that could help him identify internal errors.

Sample results of this process can be seen in figures 5.2 and 5.3. Figure 5.2 displays the movement of a single user's "marked" sensitive data, i.e., credit card number and email address, as they move in time between the audit-enabled applications. On the other hand, Figure 5.3, which is aimed for the provider of the service, is a much "richer" image showing at a first glance patterns in the data flows. In general, the final audit images presented on Cloudopsy's web interface include the communicating applications of the composite cloud-hosted services running over the Cloudopsy mechanism, as well as hosts outside the cloud which according to the logs participated in sensitive data transmissions. In general, the final circular graphs consist of an outer ring representing the cloud-hosted applications (colored blocks) running over the auditing infrastructure, whereas the colored arcs depict the recorded transmissions of monitored user data. More specifically, in these two figures the blue and orange colored directed links represent different classes of sensitive data. In particular, in Figure 5.3, the different colors could also represent the different users of the service. It is on the provider's discretion to customize the parameters on Cloudopsy's web

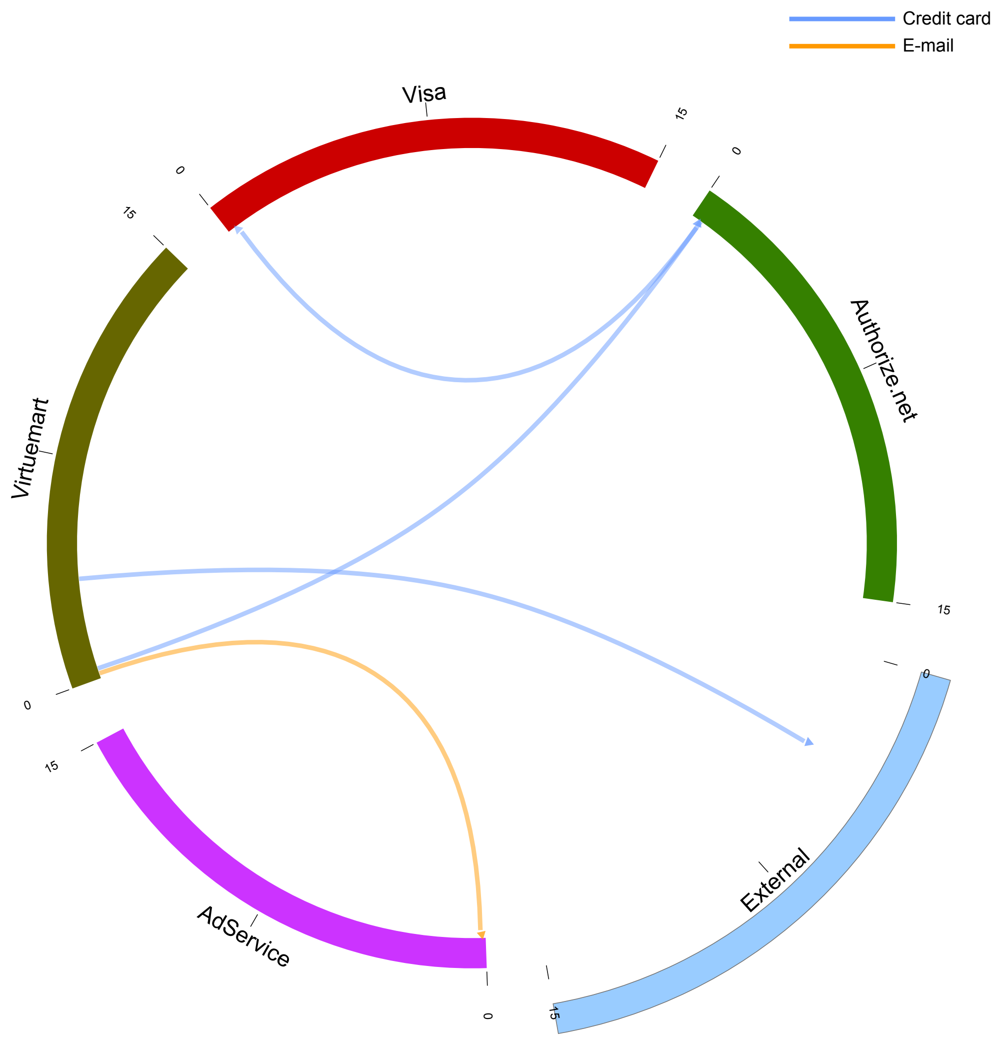


**Figure 5.2:** A user's view of a transaction.

interface, in respect to the information he is interested in on different occasions.

### 5.4.3 Illustrative Scenario: Testing Cloudopsy with an E-store

To test Cloudopsy we chose an e-store application, namely VirtueMart [69], hosted on a cloud-based infrastructure. We configured VirtueMart to accept only payments with credit card, and set up actual electronic payments through the Authorize.Net service using test accounts. Typically during a purchase transaction, users enter their personal data, credit card, email etc., through the web front end of the application, which then is transmitted to the back-end database of the e-store, and to the payment processor. After their credit card

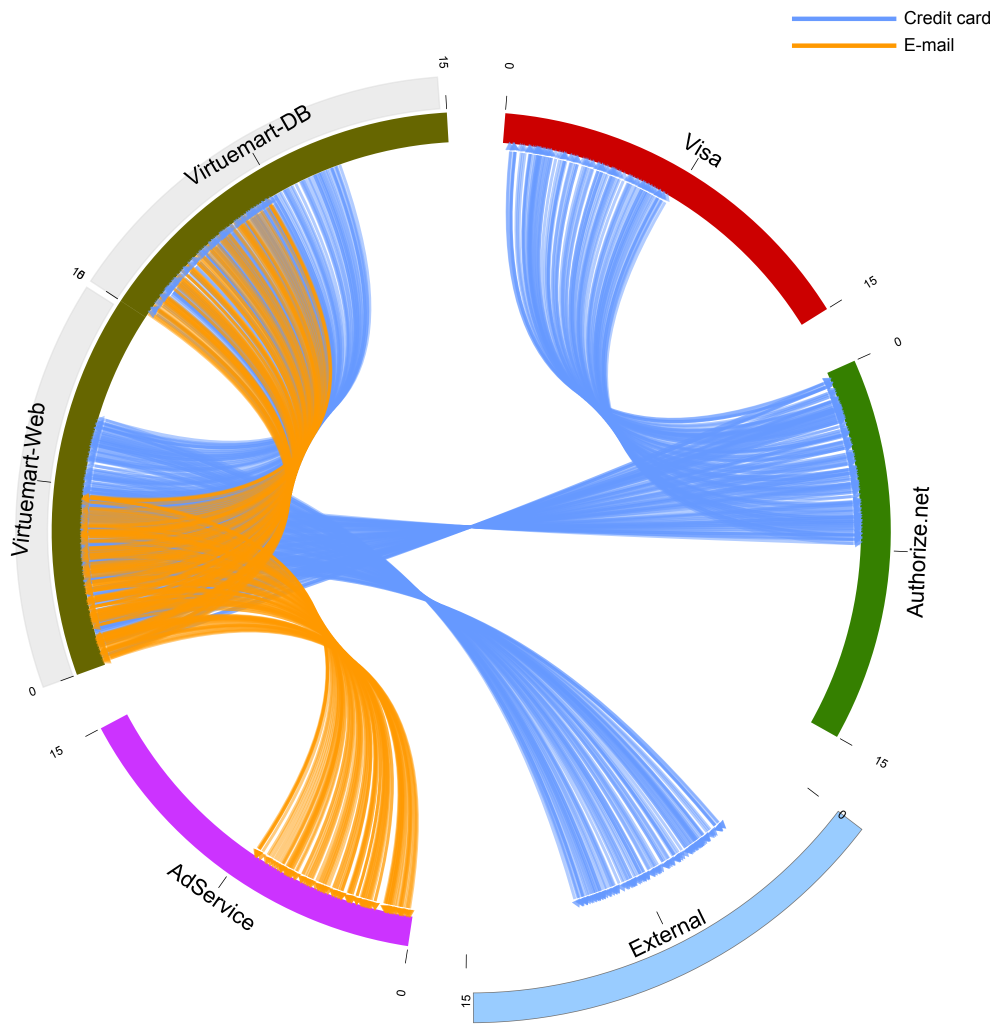


**Figure 5.3:** A service provider's view of user transactions.

is verified, the e-store approves the purchase and the transaction completes successfully. The personal data that the user enters remain stored (as is frequently the case) in the database of the e-store.

Figure 5.2 depicts the movement of the credit card number for one user that completed a successful purchase through VirtueMart. It also represents the email of this user also stored in the backend database of the e-store. Apart from the exchanged data though, this figure shows also the time that each event took place. (For each segment time progresses clockwise). Therefore, we can immediately identify two events. First, that the email is sent to the AdService. This might be a legitimate action in case the client did not opt out of sending his information for advertising reasons while using the online service. But the most interesting event presented in this figure is that credit card number seems to transfer to an unknown IP outside the cloud premises at a later point in time, which definitely looks suspicious and should be further investigated as a possible inadvertent data leak. Note that this cannot be the legitimate channel of the user, as this was known from the start of the transaction and should be whitelisted by the auditing mechanism as a legitimate flow and therefore would not be part of the audit logs. Even with this simple figure this unexpected event was very easily identified.

## 5.5 Conclusions

We argue that the concerns of the end users of cloud-hosted services about the usage of their data can be a major deterrent for users looking to embrace cloud-hosted services. We focused our efforts on enhancing the effectiveness of the information collected in the audit log files and presented Cloudopsy, a framework that through visualization and automated analysis of the results and based on the graphs produced with the Circos visualization tool, remediates cloud users security concerns and enables even users without any particular technical background to get a better understanding of the treatment of their data by third-part cloud-hosted services.

# Chapter 6

# Conclusions

Cloud Computing appears as a computational paradigm as well as a distribution architecture with its main objective being to provide quick, convenient data storage and computing services, with all computing resources visualized as services and delivered over the Internet. Many of the most successful and most visible services in the cloud today are consumer services, such as social networks, e-stores, and business solutions. The companies providing these services end up collecting terabytes of data, much of it sensitive, personal information, which are being stored in cloud-hosted databases. Because of its popularity and its trumpeted benefits, the adoption of cloud computing came before the appropriate technologies were integrated in the cloud to tackle the accompanying security challenges. Therefore, moving critical applications and sensitive data beyond the control of the enterprise's network to public cloud environments is of great concern for them.

*Data security* remains the overriding barrier against the wide-spread adoption of cloud technologies, especially for security-sensitive operations, followed by issues regarding privacy, compliance and availability. More specifically, these security concerns are caused by the *lack of control over data*, the *limited transparency* in the cloud and finally the *risk of unauthorized data access*, which is inherited by the traditional online services. The data leaks can be either inadvertent (code bugs, bad configuration) or advertent (SQL injections, XSS attacks). Data breaches are not at all uncommon incidents even for major cloud services, and apart from the direct financial repercussions for the companies involved, they can have severe impact on the reputation of the involved companies as well as affect the privacy that individuals are entitled to.

To address this very important problem, we propose a *data-centric* security approach for the cloud setting, as the sensitive data is the valuable aspect that needs to be protected. Our approach was built upon the observation that people tend to trust a system less, when it gives insufficient information about its internal procedures. Therefore, in this dissertation we argue that providing *information flow auditing capabilities* in the cloud helps *reveal data leakage* incidents and can also serve as the *basis* for increasing the *transparency* on the use of sensitive data by the cloud-hosted services. Unlike other privacy protection technologies that are built on enforcing policies and preventing data flows, auditing focuses on keeping data usage *transparent* and *trackable*. Even though information flow policies and enforcement techniques have the same goals and use similar mechanisms with our approach, we chose to focus our efforts in auditing, due to the concerns that enforcement could have adverse effects, like breaking parts of services, which is a highly probable case for the feature-rich, constantly changing and multi-component cloudhosted services. Thorough, efficient auditing in cloud computing remains a challenge even for straightforward web services. Therefore, to evaluate the efficacy and applicability of our proposed information flow auditing mechanism we developed three prototype systems: TaintExchange, CloudFence and Cloudopsy.

TaintExchange is a *generic* mechanism for efficiently performing *cross-process* and *cross-host* data flow tracking. As suggested by its name, its goal is to enable the *transfer* of taint information between collaborating processes (on the same or different hosts) along with the data exchanged between them. It does so by intercepting I/O system calls to *transparently* inject and extract information regarding the *taintness* of every byte transferred between these processes running under TaintExchange. Note that, unlike previous works, in TaintExchange the taint information is transferred through the existing communication channels (i.e., sockets or pipes) between the collaborating processes. To the best of our knowledge, our system is the *first* that allows *cross-host taint tracking without requiring full-system emulation.*

CloudFence is a novel data tracking framework for cloud hosting environments, implemented with the vision of supplying cloud providers with an auditing service in addition to their existing hosting environment. As suggested by its name, CloudFence offers data flow tracking abilities, *transparent* and *fine-grained*, within the boundaries of a well-defined *data flow domain*. This domain consists of the component applications of a single *composite* cloud-hosted service or many separate services offered by the same cloud provider (e.g., Amazon services). CloudFence effectively offers protection against many classes of web attacks that can lead to unauthorized data access. But besides protecting against these application-level vulnerabilities, it also provides an additional functionality; the logging of tainted data transfers out of the vicinity of the defined data flow domain. More specifically, the DFT component pushes audit information to relevant CloudFence component whenever tagged data is written to a cloud storage device or pass through I/O channels to endpoints inside or outside the cloud. CloudFence is the nucleus of our proposed solution and apart from extending the original TaintExchange mechanism with optimizations, it supports 32-bit wide tags as well as persistent tagging on disk and across the network.

Finally, we designed and implemented Cloudopsy with the goal of enhancing the presentation and the effectiveness of the information collected in the CloudFence-generated audit logs. Cloudopsy is a generic facility offered by the cloud hosting providers, that through visualization techniques and automated analysis of the rich but textual audit logs offers the service providers and their customers more comprehensible auditing capabilities with respect to the handling of the sensitive user data collected in the realm of these online services. More specifically, it utilizes the visualization tool Circos to transform CloudFence's textual audit logs to circular graphs depicting the sensitive data flows among the collaborating components of compound cloud-hosted services as well as flows outside of the authorized data flow domain. Cloudopsy serves as an additional layer of protection against illegitimate data flows, which can be very useful to both the service providers and the end-users, acting as a fast and effective mechanism for indicating "supsicious" data flows, that could potentially be inadvertent data leaks.

We have implemented all of these systems, and extensively measured their efficacy and performance. Our examination with real applications and known bugs and vulnerabilities shows that our technique can be used to detect data leakage incidents for all examined cases. In case of attackers who do arbitrary code execution, we can no longer guarantee accurate data tracking, since they can not only compromise our framework, but can also exfiltrate data through covert channels.

To conclude, we envision cloud providers offering our information flow auditing service in addition to their existing hosting environment, to both service providers and data owners, either as a *by default* mechanism, or as a *security-as-a-service* option. A large, reputable cloud provider could help leverage user confidence much more effectively than a lesser known service provider. Nevertheless, our approach allows service providers to provide, with minimal effort, an extra feature that reinforces the trust relationship with their users, knowing that they have an additional way to monitor what happens with their data. In addition, we empower service providers with the ability to build an additional level of protection and get a better understanding of how their services really work as they are given the tools to quickly identify inadvertent leaks or suspicious data flows. The work proposed in this thesis is not expected to provide the ultimate solution, but we consider it to be a good starting point for establishing the trust foundation for large enterprises and security-conscious users to adopt cloud services for privacy-sensitive operations.

# Bibliography

[1] Apparmor. `http://wiki.apparmor.net`.

[2] AWS taps social networks for identity verification. `http://www.theregister.co.uk/2013/05/29/aws_social_identity_verification`.

[3] Login with Amazon. `http://login.amazon.com/`.

[4] Security enhanced linux (selinux). `http://selinuxproject.org/`.

[5] SiteBar: Multiple issues. `http://www.securityfocus.com/archive/1/483364`.

[6] VirtueMart Multiple SQL Injection Vulnerabilities. `http://www.securityfocus.com/bid/37963`.

[7] Cloud Security Alliance. The notorious nine: cloud computing top threats in 2013, February 2013. `https://cloudsecurityalliance.org/download/the-notorious-nine-cloud-computing-top-threats-in-2013/`.

[8] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the* 11th *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11, 2010.

[9] E. D. Bell and J. L. La Padula. Secure computer system: Unified exposition and multics interpretation. Technical Report ESD-TR-75306, Mitre Corporation, 1976.

[10] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 41–46, 2005.

[11] Luciano Bello and Alejandro Russo. Towards a Taint Mode for Cloud Computing Web Applications. In *Proceedings of the* 7th *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 1–12, 2012.

[12] Hal Berghel. Identity Theft and Financial Fraud: Some Strangeness in the Proportions. *Computer*, 45(1):86–89, January 2012.

[13] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, and V. N. Venkatakrishnan. WAPTEC: Whitebox Analysis of Web Applications for Parameter Tampering Exploit Construction. In *Proceedings of the* 18th *ACM Conference on Computer and Communications Security (CCS)*, pages 575–586, 2011.

[14] Bochs. The cross platform IA-32 emulator, 2001. `http://bochs.sourceforge.net`.

[15] Edson Borin, Cheng Wang, Youfeng Wu, and Guido Araujo. Software-based transparent and comprehensive control-flow error detection. In *Proceedings of the* 4th *Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 333–345, 2006.

[16] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The World's Fastest Taint Tracker. In *Proceedings of the* 14th *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, pages 1–20, 2011.

[17] Yanpei Chen, Vern Paxson, and Randy H. Katz. What's New About Cloud Computing Security? Technical Report UCB/EECS-2010-5, EECS Department, University of California, Berkeley, Jan 2010.

[18] Yu-Yuan Chen, Pramod A. Jamkhedkar, and Ruby B. Lee. A software-hardware architecture for self-protecting data. In *Proceedings of the* 19th *ACM Conference on Computer and Communications Security (CCS)*, pages 14–27, 2012.

[19] W. Cheng, Qin Zhao, Bei Yu, and S. Hiroshige. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *Proceedings of the* 11th *IEEE Symposium on Computers and Communications (ISCC)*, pages 749–754, 2006.

[20] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Proceedings of the* 13th *USENIX Security Symposium*, pages 321–336, 2004.

[21] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA)*, pages 196–206, 2007.

[22] Computerworld. Microsoft BPOS cloud service hit with data breach, December 2010. `http://www.computerworld.com/s/article/9202078/Microsoft_BPOS_cloud_service_hit_with_data_breach`.

[23] Cornell University. Open-source Forensics Tools for Network and System Administrators – Spider, February 2010. `http://www2.cit.cornell.edu/security/tools/`.

[24] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the* 37th *Annual International Symposium on Microarchitecture (MICRO)*, pages 221–232, 2004.

[25] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Real-world buffer overflow protection for userspace & kernelspace. In *Proceedings of the* 17th *USENIX Security Symposium*, pages 395–410, 2008.

[26] Benjamin Davis and Hao Chen. DBTaint: cross-application information flow tracking via databases. In *Proceedings of the 2010 USENIX Conference on Web Application Development (WebApps)*, pages 135–146, 2010.

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

[28] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 1–14, 2007.

[29] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the* 11$^{th}$ *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 393–407, 2010.

[30] The Guardian. PlayStation Network hack: why it took Sony seven days to tell the world, April 2011. `http://www.theguardian.com/technology/gamesblog/2011/apr/27/playstation-network-hack-sony`.

[31] Alex Ho, Michael Fetterman, Christopher Clark Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. In *Proceedings of the* 1$^{st}$ *European Conference on Computer Systems (EuroSys)*, pages 29–41, 2006.

[32] BearingPoint Institute. In cloud we trust?, September 2012. `http://http://www.bearingpoint.com/en-uk/7633-6211/in-cloud-we-trust/`.

[33] Fujitsu Research Institute. Personal data in the cloud: The importance of trust, November 2010. `http://www.fujitsu.com/downloads/WWW2/news/publications/FSL-0016_A4_TrustReport_online-final.pdf`.

[34] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the* 18$^{th}$ *Symposium on Network and Distributed System Security (NDSS)*, 2011.

[35] Lori M. Kaufman. Data security in the world of cloud computing. *IEEE Security and Privacy*, 7(4):61–64, July 2009.

[36] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Proceedings of the* 8$^{th}$ *Annual International Conference on Virtual Execution Environments (VEE)*, pages 121–132, 2012.

[37] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification

of an OS Kernel. In *Proceedings of the* 22nd *ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, 2009.

[38] Ryan K. L. Ko, Peter Jagadpramana, Miranda Mowbray, Siani Pearson, Markus Kirchberg, Qianhui Liang, and Bu-Sung Lee. TrustCloud: A framework for accountability and trust in cloud computing. In *2011 IEEE World Congress on Services (SERVICES)*, pages 584–588, 2011.

[39] Georgios Kontaxis, Michalis Polychronakis, Angelos D. Keromytis, and Evangelos P. Markatos. Privacy-preserving social plugins. In *Proceedings of the* 21st *USENIX Security Symposium*, pages 631 – 646, 2012.

[40] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans, Kaashoek Eddie, and Kohler Robert Morris. Information Flow Control for Standard OS Abstractions. In *Proceedings of the* 21st *ACM Symposium on Operating Systems Principles (SOSP)*, pages 321–334, 2007.

[41] Martin I. Krzywinski, Jacqueline E. Schein, Inanc Birol, Joseph Connors, Randy Gascoyne, Doug Horsman, Steven J. Jones, and Marco A. Marra. Circos: An information aesthetic for comparative genomics. *Genome Research*, 19:1639–1645, June 2009.

[42] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, 2005.

[43] Philippe Massonet, Syed Naqvi, Christophe Ponsard, Joseph Latanicki, Benny Rochwerger, and Massimo Villari. A monitoring and audit logging architecture for data location compliance in federated cloud infrastructures. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 1510–1517, 2011.

[44] M. Douglas McIlroy and James. A. Reeds. Multilevel security in the unix tradition. *Software - Practise and Experience*, 22(8):673–694, August 1992.

[45] Larry McVoy and Carl Staelin. lmbench, 2005. `http://lmbench.sourceforge.net/`.

[46] Peter Mell and Timothy Grance. The NIST definition of cloud computing, January 2011. `http://csrc.nist.gov/publications/drafts/800-145/Draft-SP-800-145_cloud-definition.pdf`.

[47] David Molnar and Stuart Schechter. Self Hosting vs. Cloud Hosting: Accounting for the security impact of hosting in the cloud. In *Proceedings of the* 9th *Workshop on the Economics of Information Security (WEIS)*, pages 1–18, 2010.

[48] Yogesh Mundada, Anirudh Ramachandran, and Nick Feamster. SilverLine: Data and Network Isolation for Cloud Services. In *Proceedings of the* 3rd *USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2011.

[49] Shashidhar Mysore, Bita Mazloom, Banit Agrawal, and Timothy Sherwood. Understanding and visualizing full systems with data flow tomography. In *Proceedings of the* 13th *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 211–221, 2008.

[50] Kara Nance, Matt Bishop, and Brian Hay. Virtual Machine Introspection: Observation or Interference? *IEEE Security and Privacy*, 6(5):32–37, September 2008.

[51] Nicholas Nethercote and Julian Seward. How to Shadow Every Byte of Memory Used by a Program. In *Proceedings of the* 3rd *Annual International Conference on Virtual Execution Environments (VEE)*, pages 65–74, 2007.

[52] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 89–100, 2007.

[53] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the* 12th *Symposium on Network and Distributed System Security (NDSS)*, 2005.

[54] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the 20th International Information Security Conference (IFIP)*, pages 372–382, 2005.

[55] Vasilis Pappas, Vasileios P. Kemerlis, Angeliki Zavou, Michalis Polychronakis, and Angelos D. Keromytis. CloudFence: Data Flow Tracking as a Cloud Service. In *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, pages 411–431, 2013.

[56] Georgios Portokalidis and Angelos D. Keromytis. REASSURE: A self-contained mechanism for healing software using rescue points. In *Proceedings of the 6th International Workshop on Security (IWSEC)*, pages 16–32, 2011.

[57] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys)*, pages 15–27, 2006.

[58] Sören Preibusch. Information Flow Control for Static Enforcement of User-Defined Privacy Policies. In *Proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*, pages 157–160, 2011.

[59] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of the 39th Annual International Symposium on Microarchitecture (MICRO)*, pages 135–148, 2006.

[60] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: practical fine-grained decentralized information flow control. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 63–74, 2009.

[61] Asia Slowinska and Herbert Bos. Pointless tainting? Evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM SIGOPS/EuroSys European Conference on Computer Systems 2009 (EuroSys)*, pages 61–74, 2009.

[62] Naked Security (Sophos). Adobe customer data breached - login and credit card data probably stolen, all passwords reset, October 2013. `http://nakedsecurity.sophos.com/2013/10/04/adobe-owns-up-to-getting-pwned-login-and-credit-card-data-probably-stolen-all-passwords-reset/`.

[63] Naked Security (Sophos). Groupon subsidiary leaks 300k logins, fixes fail, fails again, 2011 June. `http://nakedsecurity.sophos.com/2011/06/30/groupon-subsidiary-leaks-300k-logins-fixes- fail-fails-again/`.

[64] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. Chapter 24. Out-of-Band Data. In *UNIX Network Programming Volume 1, Third Edition: The Sockets Networking API*. Addison Wesley, 2003.

[65] G. Edward Suh, Jaewook Lee, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the* 11$^{th}$ *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–96, 2004.

[66] TechCrunch. Google Privacy Blunder Shares Your Docs Without Permission, 2009. `http://techcrunch.com/2009/03/07/huge-google-privacy-blunder-shares-your-docs-without-permission/`.

[67] The Wall Street Journal. Google Discloses Privacy Glitch, March 2009. `http://blogs.wsj.com/digits/2009/03/08/1214/`.

[68] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the* 37$^{th}$ *Annual International Symposium on Microarchitecture (MICRO)*, pages 243–254, 2004.

[69] VirtueMart eCommerce Solution. VirtueMart shopping cart software. `http://virtuemart.net`.

[70] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the* 14th *Symposium on Network and Distributed System Security (NDSS)*, 2007.

[71] Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *Proceedings of the* 29th *Conference on Information Communications (INFOCOM)*, pages 525–533, 2010.

[72] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 497–512, 2010.

[73] Wired. Dropbox left user accounts unlocked for 4 hours, June 2011. `http://www.wired.com/threatlevel/2011/06/dropbox/`.

[74] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the* 15th *USENIX Security Symposium*, pages 121–136, 2006.

[75] Heng Yin and Dawn Song. TEMU: Binary code analysis via whole-system layered annotative execution. Technical Report UCB/EECS-2010-3, EECS Department, University of California, Berkeley, 2010.

[76] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the* 14th *ACM Conference on Computer and Communications Security (CCS)*, pages 116–127, 2007.

[77] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving Application Security with Data Flow Assertions. In *Proceedings of the* 22nd *ACM Symposium on Operating Systems Principles (SOSP)*, pages 291–304, 2009.

[78] Angeliki Zavou, Vasilis Pappas, Vasileios P. Kemerlis, Michalis Polychronakis, Georgios Portokalidis, and Angelos D. Keromytis. Cloudopsy: an Autopsy of Data Flows in

the Cloud. In *Proceedings of the* 15th *International Conference on Human-Computer Interaction (HCI)*, pages 366–375, 2013.

[79] Angeliki Zavou, Georgios Portokalidis, and Angelos D. Keromytis. Taint-Exchange: a Generic System for Cross-process and Cross-host Taint Tracking. In *Proceedings of the* 6th *International Workshop on Security (IWSEC)*, pages 113–128, 2011.

[80] Angeliki Zavou, Georgios Portokalidis, and Angelos D. Keromytis. Self-healing multitier architectures using cascading rescue points. In *Proceedings of the* 28th *Annual Computer Security Applications Conference (ACSAC)*, pages 379–388, 2012.

[81] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making Information Flow Explicit in HiStar. In *Proceedings of the* 7th *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 263–278, 2006.

[82] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing Distributed Systems with Information Flow Control. In *Proceedings of the* 5th *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 293–308, 2008.

[83] Qing Zhang, John McCullough, Justin Ma, Nabil Schear, Michael Vrable, Amin Vahdat, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Neon: System Support for Derived Data Management. In *Proceedings of the* 6th *Annual International Conference on Virtual Execution Environments (VEE)*, pages 63–74, 2010.

[84] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Efficient memory shadowing for 64-bit architectures. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM)*, pages 93–102, 2010.

[85] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Umbra: efficient and scalable memory shadowing. In *Proceedings of the* 8th *Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 22–31, 2010.

[86] David Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. TaintEraser: Protecting Sensitive Data Leaks Using Application-Level Taint Tracking. *ACM Operating Systems Review*, 45(1):142–154, 2011.

# Part I

# Appendices

# Appendix A

# Uses of the TaintExchange Mechanism

TaintExchange is a *generic* cross-process and cross-host dynamic data flow tracking, and therefore with few changes and by combining it with REASSURE [56], a tool for deploying rescue points on binaries, we used it as a building block for the *Cascading Rescue Points* (CRPs) mechanism, which is presented in details in our paper [80].

The *Cascading Rescue Points* protocol is a mechanism we proposed to address the state inconsistency issues that can arise when using software self-healing techniques using rescue points (RPs) to recover from errors in interconnected applications. Briefly, with CRPs, when an application is executing within a RP and transmits data, the remote peer is notified to also perform a checkpoint, so the communicating entities checkpoint in a coordinated, but loosely coupled way. Notifications are also sent when RPs successfully complete execution, and when recovery is initiated, so that the appropriate action is performed by remote parties. We developed a tool that implements CRPs by dynamically instrumenting binaries and transparently injecting notifications in the already established TCP channels between applications. With the aid of TaintExchange, the proposed checkpointing protocol is piggybacked on the existing communication channels. More specifically, TaintExchange is used to convey the state-related signals between the applications of the multitier architecture, which are necessary to retain a consistent state while employing the self-healing techniques provided by REASSURE.

## A.1 The Problem

Previous software self-healing approaches cannot apply rescue points on functions that have side effects, such as transmitting data to other entities on the network. Doing so can result in inconsistent states between the communicating parties, as shown in Figure A.1, because the effects of process $p1$ sending a message to $p2$ cannot be undone. The problem with this scenario is that the client's state has been rolled back and the client believes that an error, such as being unable to communicate with the server, occurred. However, data has been exchanged with the server, which is oblivious of the error that occurred in the client. Depending on the nature of the communicating applications this can lead to various problems and can require additional mechanisms, like transactions employed by database (DB) servers, for restoring them to a consistent state. By *consistent state*, we refer to every party having a correct view of what is the state of their peer. For example, if $p1$ tries to issue a command to $p2$ to switch it to state $state'$ and it fails, $p1$ can still think that $p2$ is in state $state$.

Previous designs simply ignore data exchanges and rely on the protocols implemented by applications to discover and correct such inconsistencies (e.g., they use transactions). Moreover, the problem cannot be trivially addressed by simply delaying the transmission of messages. Figure A.2 depicts an example where the application expects to receive a response to a message send from within a RP. Buffering the transmitted message would break function `bug()`, causing it to fail or wait forever because no data is sent as a response from $p2$.

## A.2 Protocol Overview

The CRP protocol is transparently implemented over the application's TCP connections. The protocol encapsulates application data, and serves the sole purpose of allowing us to convey signals between applications of the architecture.

Consider process *p1* shown in Figure A.3. All of its communications with other processes

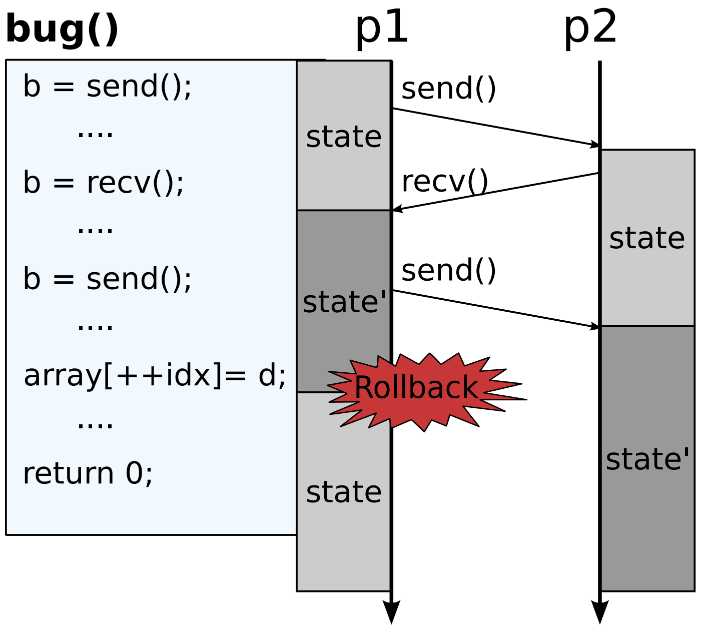


**Figure A.1:** A rescue point deployed on function `bug()` of process $p1$ needs to both send and receive data to and from $p2$. When an error triggers a roll back, $p1$ can end up in an inconsistent state with $p2$. Deploying rescue points in routines that communicate with other parties over the network can be problematic because their effects cannot be reversed.

in the architecture are modified to implement our CRP protocol. When *p1* executes within a RP, it is essentially checkpointing, indicated by the highlighted areas in Figure A.3. This means that a fault will cause all the changes performed within the RP to be undone, and we will simulate the return of an error code from the RP routine. When *p1* transmits data to another process (while in a RP), we use our protocol to instruct the remote peer to also begin checkpointing. Later on, if an errors occurs in *p1*, the RP will recover the process. Since we piggyback our protocol on existing communications, *p1* does not immediately notify *p2* that it discarded the state generated in the RP, and *p2* will continue checkpointing until the next message is received by *p1*. Figure A.3(a) depicts this process, which is propagating in time to the other processes. If *p2* sends any data to another process (e.g., *p3*), that process also begins checkpointing, and so forth

If no fault occurs, the process is almost entirely the same, and it is shown in Figure A.3(b). Like before, the RP of *p1* causes the checkpointing to cascade to *p2* and *p3*. However, in this case no error occurs and *p1* successfully exits the RP. When this happens,

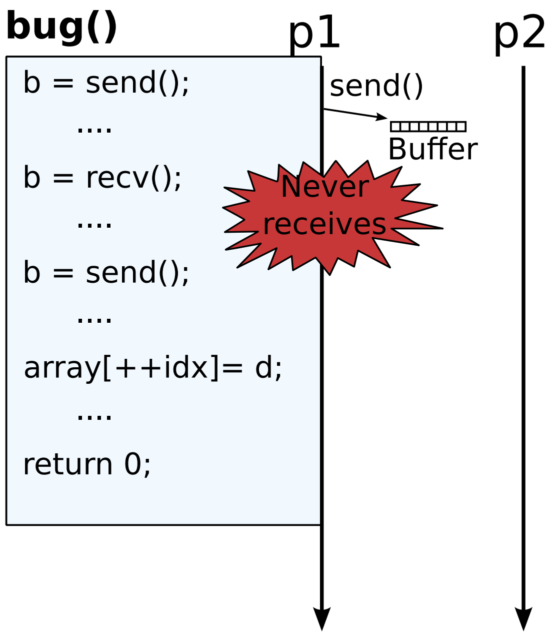
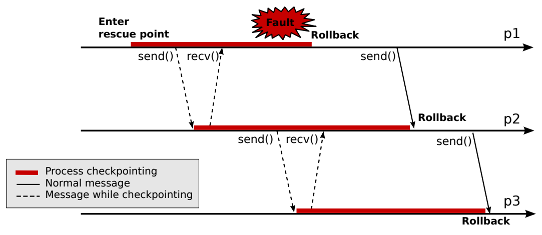


**Figure A.2:** Adopting a naive approach to address the issue in Figure A.1 will not work. For example, buffering the data being send from a rescue point, and only transmitting them after determining that an error did not occur, can break applications. In this case, $p2$ never receives the data that will cause it to respond to $p1$.

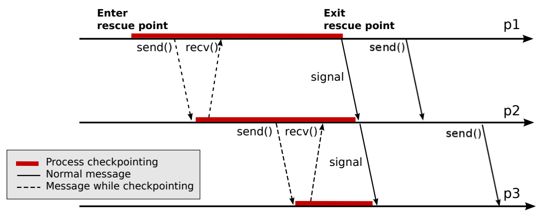
we immediately notify the other processes by utilizing TCP's out-of-band (OOB) data [64]. OOB data are not part of the regular data stream, so we can signal *p2* and *p3* without corrupting the application data stream and without requiring data to be read by the processes. Instead, we can rely on the OS to notify the process when such a packet is received (e.g., by raising a signal or exception). TCP does not support multiple OOB signals on a particular stream (i.e., a second OOB would overwrite the first and would be the only one to raise a signal on the receiver). For this reason, we can only use it to signal successful exits from RPs. Our approach is an optimistic one, assuming that errors will be rare.

## A.3 Implementation

We built our tool using Pin [42], a framework that we also used for the original implementation of TaintExchange, which enables the development of tools that can at runtime augment or modify a binary's execution at the instruction level through an extensive API. The tar-



(a) Fault occurs. The next transmission from *p1* to *p2* will notify the latter to also roll back. *p2* will eventually notify *p3*.



(b) No fault. When *p1* exits the rescue point, it immediately notifies *p2*, which also exits checkpointing and notifies *p3*, and so forth.

**Figure A.3:** Cascading rescue points overview. When process *p2* receives a message from process *p1*, which executes within a rescue point, it also begins checkpointing. Other processes, like *p3*, that receive messages from a checkpointing process also begin checkpointing. This way the original rescue point *cascades* to the communicating processes.

get binary executes on top of Pin's virtual machine (VM), which essentially consists of a just-in-time (JIT) compiler that combines the binary's original code with the code inserted by the tool and places the produced code blocks into a code cache, where the application executes from. Pin facilitates the instrumentation of binaries by enabling developers to inspect and modify a program's instructions, as well as intercept system calls and signals. It is actively developed and supports multiple hardware architectures and OSs. Pintools can be applied on binaries by either launching them through Pin or by attaching on already running binaries. The latter behavior is highly desirable, as it allows us to deploy RPs without interrupting an already executing application. Our tool currently runs on x86 Linux systems, however there are no significant challenges in porting it to other OSs and architectures supported by Pin.

**Note.** For the purpose of this thesis, we will focus on the part of the CRP implementation that is incorporating the TaintExchange mechanism. The specifics of the CRP protocol is out of the scope of this thesis, but are described in much detail in our paper [80].

### A.3.1 I/O Interception

The cascading rescue point protocol is used to communicate events between peers exchanging data over TCP sockets. The protocol is implemented transparently over the sockets used by the application. This is done by intercepting system calls used with TCP sockets (as described in detail in the relevant section). We can classify these system calls into two groups. The first group consists of calls handling socket creation and termination, and the second group is dealing with data transmission and reception. We intercept system calls `socket()`, `close()`, `shutdown()`, `connect()`, `accept()`, `socketpair()`, and the `dup()` family of calls to track the state of descriptors used by the application (i.e., distinguish TCP socket descriptors from others, like files). For this reason, we maintain a global array to store information on active descriptors, like their type and protocol state data. We also intercept the `read()`, `write()`, `recv()`, and `send()` family of system calls that are used to transmit and receive data from sockets to implement our protocol.

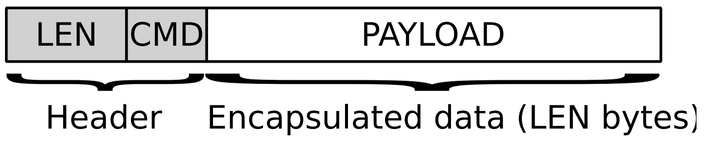


**Figure A.4:** The cascading rescue points protocol encapsulates user data using a small header prepended to every data write made by the user.

The protocol consists of variable length messages that encapsulate user data as shown in Figure A.4. In particular, we use a small header that comprises of a 4-byte field specifying the length of user data, and a single-byte `CMD` field used to communicate events to remote peers.

The header is inserted into existing TCP streams using Pin to replace system calls used to write data, like `write()` and `send()`, with `writev()` which allows us to transmit data from multiple buffers by performing a single call. This minimizes the number of operations (data copies and system calls) required to transparently inject the header into the stream. If the message cannot be written in its entirety, for instance because non-blocking I/O is performed and the kernel buffers are full, we keep trying until we are successful.

To extract the header from the stream, the reverse procedure is followed. Initially, we replace calls used to receive data with `readv()` to read into multiple buffers. If necessary, we repeat the process until the whole header is received. User data is placed directly in the buffer supplied by the application. However, we can read into the next message, which will be placed into the application's buffer. When this happens, we move the data belonging to subsequent messages into a buffer associated with the socket descriptor. Consequent reads will read data from this buffer instead of performing a system call. Reading one message at a time may be suboptimal performance-wise, but allows us to pair read system calls with particular events received on a socket (e.g., a request to begin checkpointing), which is necessary for rolling back.

### A.3.2 Protocol Commands

The `CMD` field in the protocol is used to inform remote peers of changes in the state of the running thread. For instance, when data are written to a socket while in a rescue point, `CMD` changes to indicate that the destination should also begin checkpointing, the socket is marked as having been signaled to checkpoint, and is placed in a list containing other similar sockets (`fd_checkpointed`). If an error occurs and the thread rolls back, sockets in `fd_checkpointed` are marked accordingly, so that the next write will convey the status change. *If the next write occurs within a RP, the fact is also passed to the remote process, so that it first rolls back memory changes and then enters a new checkpoint.*

On the receiving end, if a thread receives a command to checkpoint, it begins checkpointing similarly to entering a RP. The socket descriptor number where the command was received is saved, so that a consequent request to roll back is only honored, if it was received on the same socket. On rollback, execution resumes right before the system call that caused the thread to checkpoint. Note that receiving requests to begin checkpointing from other sockets, while already checkpointing or executing in a RP are ignored.

**Checkpoint Commits Through Out-of-band Signaling** To notify remote peers of a successful exit from a RP, we utilize out-of-band (OOB) signaling, as provided by the TCP protocol and the OS. In particular, we make use of TCP's OOB data to notify a remote application that it should also commit changes performed within a checkpoint. We send OOB data by using the `send()` system call and supplying the `MSG_OOB` flag for every descriptor in `fd_checkpointed`.

On the receiver, the reception of an OOB signal by the OS, causes the signal `SIGURG` to be delivered to the thread, which previously took ownership of the socket descriptor that triggered the checkpointing by calling `fcntl()`.[1] The signal is intercepted, and execution is switched from checkpointing to normal execution. If a RP is entered very frequently, multiple OOB signals can be transmitted in succession. On account of TCP's limitations, only a single OOB byte can be pending at any time, so previous OOB signals are essentially overwritten. This does not affect the correct operation of our system, but unfortunately implies that we cannot also use OOB signaling to notify remote peers of roll backs.

[1]In Linux a thread can take ownership of a descriptor, causing the OS to deliver all descriptor related asynchronous events to the specific thread, instead of a randomly selected thread of the process.