Practical Improvements to User Privacy in Cloud Applications

Raymond Cheng

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

Doctor of Philosophy

University of Washington

2017

Reading Committee:

Thomas Anderson, Chair

Arvind Krishnamurthy, Chair

Tadayoshi Kohno

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science and Engineering

University of Washington

Abstract

Practical Improvements to User Privacy
in Cloud Applications

Raymond Cheng

Co-Chairs of the Supervisory Committee:
Professor Thomas Anderson
Paul G. Allen School of Computer Science and Engineering

Professor Arvind Krishnamurthy
Paul G. Allen School of Computer Science and Engineering

As the cloud handles more user data, users need better techniques to protect their privacy from adversaries looking to gain unauthorized access to sensitive data. Today's cloud services offer weak assurances with respect to user privacy, as most data is processed unencrypted in a centralized location by systems with a large trusted computing base. While current architectures enable application development speed, this comes at the cost of susceptibility to large-scale data breaches.

In this thesis, I argue that we can make significant improvements to user privacy from both external attackers and insider threats. In the first part of the thesis, I develop the Radiatus architecture for securing fully-featured cloud applications from external attacks. Radiatus secures private data stored by web applications by isolating server-side code execution into per-user sandboxes, limiting the scope of successful attacks. In the second part of the thesis, I focus on a simpler messaging application, Talek, securing it from both external and insider threats. Talek is a group private messaging system that hides both message contents as well as communication patterns from an adversary in partial control of the cloud.

Both of these systems are designed to provide better security and privacy guarantees for users under realistic threat models, while offering practical performance and development costs. This thesis presents an implementation and evaluation of both systems, showing that improved user privacy can come at acceptable costs.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

First, I would like to thank my advisers, Thomas Anderson and Arvind Krishnamurthy, for their sage advice, continued encouragement, and support of my wide interests. Tom's dedication to his students, enthusiasm to learn, and insightful perspective is a source of daily inspiration. Arvind has an electric positive energy and masterful approach to research that has taught me a great deal. Both continued to support me, even as my interests and pursuits took unexpected turns. I am forever indebted to them for this deeply impactful graduate school experience. I am lucky to have worked with Will Scott, a trusted collaborator and good friend. In addition to drawing from his can-do attitude and ninja coding skills, I cherish the many adventures we had together. I would also like to thank Franzi Roesner, Yoshi Kohno, Xi Wang, Hank Levy, Dan Ports, and all the students in the Networks Lab, Systems Lab, and Security Lab, who introduced me to new topics and different ways to thinking. Our lab could not function as well as it does without Melody Kadenko.

I am fortunate to have found fantastic advising and mentorship from Bryan Parno and Jon Howell. I am indebted to their generous and valuable feedback. I also want to thank Lucas Dixon and the incredible team at Jigsaw. Working with them was one of my most memorable experiences in graduate school, where I applied myself towards a worthy mission with a team who would turn out to be great friends. Thank you to Lidong Zhou, Fan Yang, Michael Schueppert, Hila Becker, and Mayur Thakur for wonderful research experiences.

In college, I was lucky to take classes and begin research with Frans Kaashoek, Chris Lesniewski-Laas, Patrick Winston, and Richard Fletcher. Their classes inspired me to pursue graduate school and my work continues to be influenced by their wisdom.

Many friends contributed to a memorable time in Seattle. Thanks for all the fun memories and joining me in many adventures. I thank my family for their unconditional love and support. Finally, my heartfelt thanks to Clarice. You always believed in me, even when I doubted myself. I hope you know I will always believe in you too.

Chapter 1
INTRODUCTION

The cloud provides a highly-available platform for shared computation and storage that powers a wide spectrum of applications for users around the world. As a consequence, people are increasingly relying on the cloud, both in in their professional and personal lives. The biggest cloud applications see billions of monthly users [105, 36]. The cloud's role as a platform for processing and storing data between devices and users will only grow in importance as users become more connected, devices gain network access, and applications build multi-user features.

As the cloud plays an increasingly important role in our lives, users are looking for stronger privacy assurances that their data is only accessible by parties expected by its users. This expectation becomes even more important as more cloud-supported digital devices augment our experience in traditionally private spaces, such as our homes. Privacy and security are equally important for organizations that require secrecy, including companies, governments, and militaries.

Unfortunately, most cloud-based applications provide weak assurances with respect to user privacy. Developers often design for functionality and user experience, rather than privacy. Numerous studies have shown that people often expect stronger privacy guarantees from applications than they actually provide in practice [155, 240, 158, 196].

Privacy is violated when adversaries gain unauthorized access to user data. Because the cloud stores valuable data generated by users, usually in unencrypted form, adversaries have an incentive to exploit the data for their own purposes. Worse, privacy violations often occur in ways that are invisible to the end user.

Both criminals and nation-state actors are increasingly adept at breaking into cloud-hosted services. Because modern cloud software is complex, vulnerabilities allow hackers to launch remote attacks to gain access and steal data from the cloud. These episodes are frequent and often occur for long periods of time before being disclosed to the public. In 2017, studies have shown that the likelihood of an organization facing at least one data breach incident within the next 24 months is

27.7% [191]. Current security practices have dire consequences for both cloud applications and end users. For companies operating cloud applications, the average cost of a data breach is approaching $4M [191]. These figures can grow based on the user's location and the types of data stored. Just in 2014, 17.6 million Americans, 7% of US residents over the age of 16, were the victims of identity theft [228].

Another type of powerful threat and a significant source of privacy violations are governments. Cloud-based businesses are legally bound by the laws and regulation of the countries in which they operate and serve. For example, governments can request user data and block disclosure of the request [229, 230, 231]. The cloud's concentration of information about users has an adverse effect on the greater society, as governments increasingly request cloud data to target individuals and quash dissent. Out of the total world population, an estimated 67% live in a country where criticism of the government, military, or ruling family is subject to censorship, and 27% live in a country where people have been arrested for publishing, sharing, or liking content on Facebook [17]. In these countries, governments data requests about users have grown dramatically [20, 14, 2].

In this thesis, I assume the developer of the application to be trusted and have a mutual interest with their users in preventing third parties from accessing users' data. In the case where the application developer cannot be trusted, I refer to related work on restricting applications [117, 247, 151].

1.1 User Privacy in the Cloud

Average users want to be able to easily and privately communicate sensitive information through the public cloud with their friends, family, and associates. Users should be able to leverage the cloud for its highly available storage and networking, while being assured that no parties other than participating users and the applications they trust will gain any knowledge of the nature of a user's actions, including data contents, time of delivery, and participants. Assuming that a user's device has not been compromised, the cloud should provide a safe space for people to freely communicate, associate, and organize, without fear of surveillance and repercussion.

In order to achieve this goal, we must formally understand how user privacy is compromised in a modern cloud system and design new systems that reduce or eliminate the attack surface.

Thus, I consider the roles of all of the actors in the system as part of the threat model. *Users* interact with their personal computing devices, each running software written by a *client developer*. These devices communicate over the *Internet* to the *cloud*, running software written by the *server developer*. I characterize an adversary in the threat model as having full control over some subset of these entities, which are called *untrusted resources*.

In this thesis, I consider two major classes of adversaries to user privacy. *External threats* include malicious users on the Internet that try to remotely attack the cloud. They do not have any visibility or control over the cloud or honest users, beyond what is publicly accessible on the Internet. These malicious users do not necessarily run the official client software and they can craft arbitrary network packets to attack the cloud service. *Insider threats* involve malicious actors operating with the privileges of the cloud operator. These threats can monitor all communication and actively manipulate the cloud's behavior in order to attack honest users of the system. Insider threats do not necessarily need to come in the form of malicious employees within the cloud company. They can also come in the form of government mandates, as well as malware running on the system. In these cases, the cloud takes an active role in violating user privacy.

A sufficiently powerful adversary could attack users in more fundamental ways. For example, an adversary could compel monitoring of an end-user's device. An adversary could also control the physical equipment and deny a user's access to the cloud by turning off the network. In this thesis, I make the assumption that devices of honest users are trusted and that the cloud is always reachable. I focus on external and insider threats to user data stored in the cloud.

Previous efforts to offer strong cloud privacy from both external and insider threats, while offering practical performance and supporting a wide range of applications, have seen limited success. Techniques to offer strong security guarantees for arbitrary applications have been shown to have prohibitive server and network costs [114, 48, 119, 121, 181]. Other efforts to improve performance come at the cost of relaxed security goals [193, 194, 41, 43, 210, 232, 84, 85, 86, 149, 245]. Weaker security can put users at risk, such as if the system's activity can be visible to the adversary over time [91, 141, 163]. In Chapter 2, I expand on existing techniques in the context of our goals of security, performance, and application support.

In this thesis, I focus on two separate cases of cloud security and user privacy. In the first, I consider an arbitrary cloud application and focus on securing it from external threats. In the second, I consider a simpler messaging application where security from both external and insider threats can be formally proved. In both cases, I show that we can improve security and achieve practical performance, validated by experiments on real implementations. As an additional constraint, I focus only on building systems for average users running commodity computers on the Internet.

1.2 Thesis Overview

The thesis of this work is that *for realistic threat models, we can significantly improve user privacy for fully-featured cloud applications as well as for messaging, while offering reasonable performance.*

Described in greater detail below, I first consider the case of securing arbitrarily complex cloud applications from external threats. In practice, these systems involve millions of lines of potentially buggy code, which when exploited, can allow external intruders to quickly amplify their attacks into large-scale data breaches. If we can assume that users trust the cloud and server-side software with their data, then better isolation techniques can be applied to limit the scope and impact of external attacks.

In the second half of the thesis, I consider how to design a simpler messaging application that is robust to both external and insider threats. In other words, even if an adversary has control of the cloud, passively monitoring communication and actively manipulating its behavior, we want a solution that can robustly provide private messaging, where it can be formally proved that the adversary learns nothing about the messages traversing their portion of the cloud. From the perspective of the adversary, all it should see is useless noise. In this case, both the cloud service and remote attackers are trying to learn both the contents of messages, as well as communication patterns to see which users are communicating. While these techniques do not generalize to arbitrary applications, it allows us to design a protocol that can offer both practical performance as well as strong security guarantees.

1.3 Securing Existing Cloud Applications from External Threats

Initially, I focus on the problem of securing existing cloud applications from external intrusion. Cloud applications are a frequent target of successful attacks. In most programming frameworks, the damage is amplified by the fact that application code is responsible for security enforcement. Code runs on behalf of the cloud service, rather than with the limited privileges of a user. Thus, vulnerabilities in the complex application code can lead uninhibited access to code, data, and other service resources.

In Chapter 3, I design and evaluate Radiatus, a *shared-nothing* programming framework where application-specific computation and storage on the server is contained within a sandbox with the privileges of the end-user. By strongly isolating users, user data and service availability can be protected from application vulnerabilities. Attacks that exploit application vulnerabilities are isolated to the sandbox assigned to the attacker.

To make Radiatus practical at the scale of modern applications, I introduce a distributed capabilities system to allow fine-grained secure resource sharing across the many distributed services that compose an application. I further analyze the strengths and weaknesses of a shared-nothing architecture. This architecture protects applications from a large class of vulnerabilities, but it comes at the cost of added overhead of 60.7% per server and an additional 31MB of memory per active user. I demonstrate that the system can scale to 20K operations per second on a 500-node AWS cluster.

1.4 Designing an Oblivious Messaging Service

With Radiatus, I assume that the cloud is trusted. To generalize to a stronger threat model to include insiders in the cloud, I restrict the scenario to a narrower programming model, where the cloud only serves as a message broker. This requires techniques to privately communicate, even when the cloud service is untrustworthy. Designing systems that are robust to this threat model is increasingly compelling as privacy-violating adversaries gain more sophisticated capabilities.

While existing end-to-end encryption techniques can protect the contents of a message, it is also important to hide the communication patterns, such as who is communicating with whom. In Chapter 4, I describe Talek, a private group messaging system that sends messages through potentially untrustworthy servers, while hiding both data content and the communication patterns among its users. Talek is designed with two goals that distinguish it from the prior work in private messaging. First, Talek is designed with the strong security goal of *access sequence indistinguishability*, where clients leak no information to adversarial servers that might help an adversary distinguish between two arbitrary-length client access sequences. Second, Talek aims to be practical, offering 3–4 orders of magnitude greater performance over related work with similar security goals. To achieve these properties, I introduce two novel techniques. *Oblivious logging* is a mechanism for supporting private reads and writes to shared logs stored on servers without coordination between clients. *Private notifications* provide a private and efficient mechanism for users to learn which logs have new messages without polling.

I demonstrate a 3-server Talek cluster that achieves throughput of 566K messages per minute with 5.57-second end-to-end latency on commodity servers.

1.5 Summary of Contributions

In the course of investigating techniques to secure and improve privacy in cloud-based communication systems, this dissertation makes the following high-level contributions:

1. A set of techniques for securing server-side application code in the cloud. By leveraging lightweight isolation mechanisms, I design a framework that offers competitive performance compared to existing application frameworks, while limiting the potential adverse impact of remote attackers.

2. A new approach for private messaging that offers strong privacy in the face of a persistent active adversary with control of parts of the cloud. I detail a set of formal security definitions for private messaging and prove that our protocol satisfies these definitions.

3. Implementations that validate our hypothesis and that demonstrate practical performance for reasonable workloads.

Chapter 2

BACKGROUND AND RELATED WORK

How can developers improve the security and privacy of their cloud applications, while still offering practical performance for users? In this chapter, we survey existing techniques for securing cloud-based applications against external and insider threats. These techniques help developers provide stronger assurances that user data is only accessible by intended parties, and not by hackers and criminals.

The chapter begins by describing the typical architecture of existing cloud applications (Section 2.1). We then describe external threats to cloud applications, how information is leaked in external attacks, and previous defenses, such as monitoring and isolation mechanisms (Section 2.2).

Next, we characterize insider threats and techniques for protecting user data from adversaries in control of the cloud. We describe encryption techniques for ensuring confidentiality of data contents. However, encrypting data is not sufficient for ensuring user privacy, because a user's pattern of usage can also leak information. We then describe a set of techniques for hiding communication patterns (Section 2.3).

2.1 Existing Cloud Applications

In order to understand how information is leaked, it is important to first understand the typical architecture of a modern cloud application. While there are many ways that cloud applications can be constructed, this section defines common design patterns and entities. We begin by describing the clients and usage model (Section 2.1.1). Then, we discuss the programming model of server-side logic (Section 2.1.2), the execution environment, and how these systems scale to support millions of users (Section 2.1.3). This architecture offers many avenues for surveillance, as we discuss in Sections 2.2 and 2.3.

2.1.1 Clients

Consider a user who installs a new cloud-based application onto his/her device. This application will consist of a user interface, some background processes (e.g. to retransmit lost messages), and a number of third-party libraries (e.g. advertising). While the operating system may limit the application's access to data from other applications [19, 21, 132, 169, 131, 236] and physical resources, such as the camera, applications are typically unrestricted in terms of the computations that can run on the client device (up to the battery life of the device). Furthermore, most modern operating systems do not limit the application from sending network requests to arbitrary destinations. This is because a typical application will communicate with a variety of cloud services to support application functions, such as to synchronize data, display ads, and report telemetrics.

Client-side applications come in a number of forms, including applications on a mobile phone, webpages loaded in a browser, and those that run in Internet-of-Things (IoT) devices. In each case, developers write code that runs on a user's end-device. In practice, cloud applications need to address several questions:

Where is data stored? Depending on the application, user data can be either stored on a remote server, on the local device, or both. For applications that focus on rich features, such as personal assistants, the application may store raw data on the server to be processed, providing augmented services. On the other end of the spectrum, privacy-conscious applications may store data only on the local device, using the server only as a temporary message buffer.

How do users know that client applications do not leak information? In practice, users often do not have any technical means to ensure that data is not leaked from their devices in unexpected ways. Instead, users rely on the reputations of the operating system, application store, and cloud application to protect their security and privacy.

Securing clients and user devices is an active area of research. Some approaches include auditing applications to ensure they behave correctly, using verification tools to prove correctness [242, 128], and monitoring for proper behavior [200]. Because this thesis focuses on server-side security and privacy, client-side security solutions are complementary to this work. In this thesis, we assume that honest clients can be trusted and that they do not leak user data.

2.1.2 Cloud Servers

Developers program cloud services to service incoming requests from clients. This Internet-facing interface typically intermingles authentication, user actions, and content fetches. The server developer must properly handle requests, administer access control and prevent leakage of information. Storage, caching, and user authentication are typically implemented as libraries. Developer speed is a critical issue, but this expediency comes at the cost of an increased number and severity of bugs. For example, in the case of a social network, one may store a list of users and their permissions in a relational database. When a user requests a feed of recent content, the server assembles a response by querying the database for recent content, filtering the content with access control policies in another table, and populating a template. Furthermore, the application must be written in a way such that server-side capacity can be added easily.

When the cloud service responds with webpages using HTTP, the service can also be called a web application. Web frameworks describe the programming framework used to build web applications, which span both server and client. HTTP-based web frameworks have become a widely prevalent programming model for cloud-based applications [192, 180].

2.1.3 Scaling a Cloud Application

In the simplest cloud service, a developer can deploy a cloud service with just a single machine. However, single machines are limited in their processing capability and availability.

Horizontal scaling refers to the practice of deploying additional machines to accommodate larger numbers of users as an application grows in popularity. Figure 2.1 illustrates the architecture of a typical medium-sized cloud service. When a client makes a request to the service, a load balancer distributes incoming requests across servers running identical copies of the application. Physical resources can be dynamically scaled up or down to meet demand.

In large-scale cloud services, a developer may also deploy to multiple data centers around the world. By geo-replicating, cloud services benefit from better resilience to failures and disasters. Furthermore, clients can be routed to the nearest data center, resulting in better performance for the user.

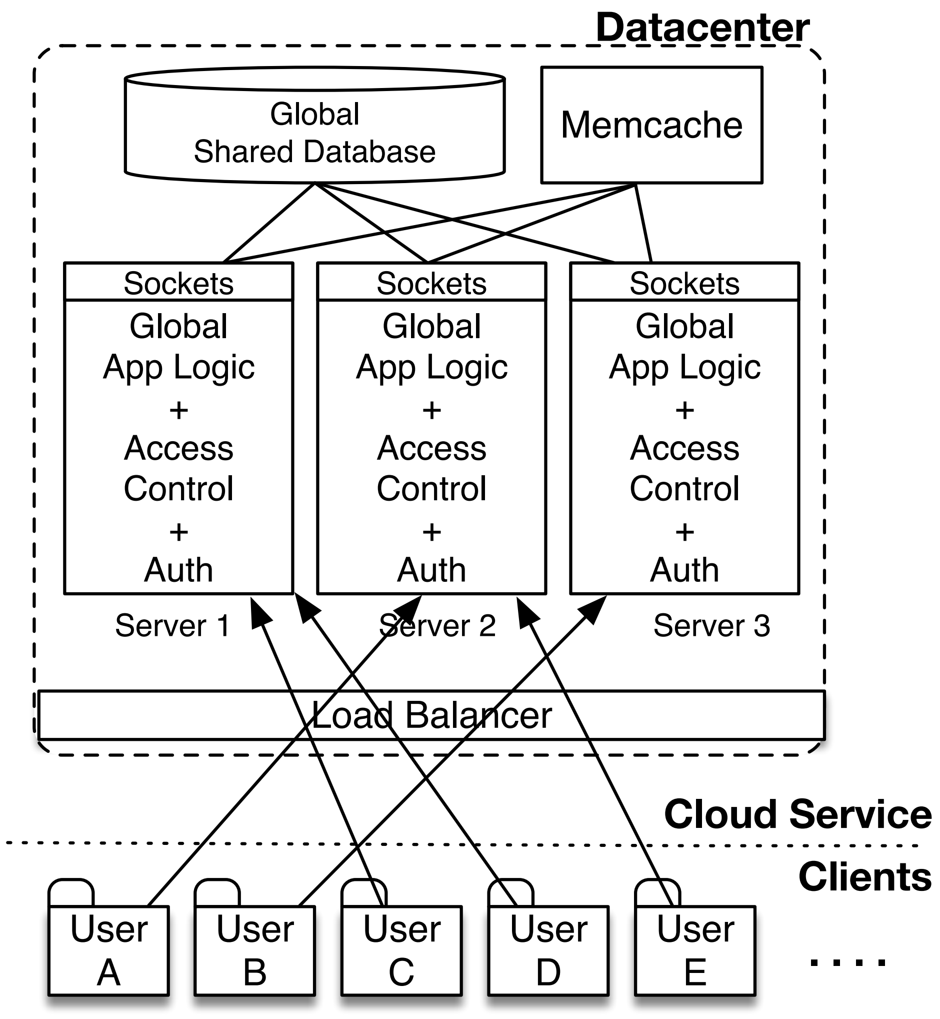

Figure 2.1: Layout of a traditional cloud service. Typically, application logic is treated as part of the trusted computing base with access to global state.

2.1.4 Composing Cloud Services

As cloud applications grow in code complexity, large monolithic applications may split functionality into multiple smaller services in a service-oriented architecture (SOA). For example, site search may be written and maintained by a different product group from the shopping cart. In turn, these services may be supported by a separate database service. In this example, each individual service is typically written in the same model as above.

Modern cloud applications integrate many internal services with a front-end web service to service user requests. In practice, all of these individual cloud services can be developed and operated by different companies. Modern cloud services exist for a variety of functions, such as storage, caching, search, video processing, analytics, and machine learning.

2.2 External Threats

In the following section, we focus on external threats to user data. We characterize the threat model and outline prior approaches to providing privacy despite these threats.

2.2.1 Threat Model

External threats can be modeled as malicious clients on the network. Like honest clients, they send requests to the cloud service, but they do not necessarily conform to specified protocols. The attacker's goal is to craft packets to trigger unexpected code paths in the cloud service. For example, malicious clients can send random requests to try to trigger unexpected behavior or they can craft software exploits to exploit software vulnerabilities in the cloud service.

Understanding Remotely Exploitable Cloud Vulnerabilities

Because cloud applications are under active development and the server software is part of the trusted computing base, developers often inadvertently introduce vulnerabilities that can be remote exploited. For example in 2013, Snapchat introduced a vulnerability in their "Find My Friends" feature that allowed attackers to enumerate the entire set of Snapchat users and their phone numbers [34]. The vulnerability allowed any user to request the username corresponding to an input phone number using the `/ph/find_friends` HTTP endpoint without limit. Attackers enumerated

```
if (isset($_COOKIE['ari_auth'])) {
  $buf = unserialize(stripslashes($_COOKIE['ari_auth']));
  list($data,$chksum) = $buf;
}
```

Figure 2.2: FreePBX (versions ≤2.9.0.9), a VoIP server, improperly sanitizes the `ari_auth` cookie before calling `unserialize` in htdocs_ari/includes/login.php. Because `unserialize` can import arbitrary PHP objects, this vulnerability can be exploited to execute arbitrary code (Sept. 2014).

all possible phone numbers in a brute force attack that revealed the entire list of usernames and their corresponding phone numbers.

Because request handler code is executed on behalf of the service, rather than as the user, remote code execution vulnerabilities are particularly devastating. These vulnerabilities allow attackers to upload and run arbitrary code, giving adversaries full control of the cloud service. For example, attackers could upload code to exfiltrate data from the database, take the service down, change the cloud service's behavior, or plant malware.

In order to understand how external attackers can gain remote code execution, Figure 2.2 highlights an example of a subtle code injection vulnerability in FreePBX, an open source VoIP server, as exploited on Sept. 2014 [174]. An improperly sanitized HTTP cookie, `ari_auth`, passed into `unserialize()` allows an attacker to import arbitrary PHP objects into the context. Code injection vulnerabilities have led to numerous data leaks and service disruptions in web applications [38, 26, 88, 47, 239]. For example, attackers in 2014 were able to write files and execute arbitrary code on Flickr servers by exploiting an injection vulnerability in a new photo books feature [239].

Survey of the National Vulnerability Database

In order to understand the relative frequency of different types of web-related vulnerabilities, we catalogued the 31,380 vulnerabilities in the National Vulnerability Database [26] that are related

CWE	**Description**	**Percent**
CWE-20	Improper Input Validation	6.7%
CWE-22	Path Traversal	6.8%
CWE-79	Cross-site Scripting*	25.9%
CWE-89	SQL Injection	22.0%
CWE-94	Code Injection	6.8%
CWE-119	Buffer Overflow	6.9%
CWE-189	Numeric Errors	1.7%
CWE-200	Information Exposure	3.9%
CWE-264	Improper Access Controls	7.6%
CWE-287	Improper Authentication	2.4%
CWE-352	Cross-Site Request Forgery*	2.2%
CWE-399	Resource Management Errors	3.5%
	Other (Server-Side)	3.6%

Figure 2.3: Most common vulnerabilities related to web technology as reported by the National Vulnerability Database [26]. Each is labeled using the standard Common Weakness Enumeration (CWE). 71.9% involve server-side attacks, resulting in remote code execution or data leakage.

*Client-side attacks that coerce browsers into performing unauthorized actions.

Attack Outcome	Percentage
Leakage of Information	27.6%
Downtime	20.6%
Defacement	16.3%
Planting of Malware	8%
Stolen Property	6.2%
Planting False Information	4.8%

Figure 2.4: Data reported by the Web Hacking Incident Database [38]. Most attacks on web applications lead to loss of information and service disruption to the users.

to web technologies or the systems that power them, such as SQL databases. The results are presented in Figure 2.3. Each vulnerability comes categorized with a Common Weakness Enumeration (CWE) [10] label. The methodology likely under-reports the frequency of server-side problems; for most web applications, the server-side code is not public, limiting the ability for outside groups to diagnose precisely why compromises occur.

In this data set, 28.1% are client-side attacks that coerce a web browser client into performing unauthorized actions, such as cross-site scripting and cross-site request forgery. The rest involve different types of server-side vulnerabilities. The most prevalent server-side vulnerabilities all involve some form of remote code execution (e.g. “SQL injection”, “code injection”, “buffer overflow”) or data leakage (e.g. “improper access controls”, “path traversal”).

Characterizing Hacking Incidents

Hacking-related incidents occur frequently. Due to the public nature of cloud applications, anyone with a computer can launch a remote attack. The Web Hacking Incident Database (WHID) [38] maintains a database of publicly reported hacking incidents. Figure 2.4 shows the outcomes of the 1377 publicly reported hacking incidents between 1999 and 2014. Of these, information leakage is the most common outcome of attack, reported in over a quarter of cases. We note that a non-trivial number of attacks also lead to disruption of service or information integrity. Many attacks directly lead to downtime or defacement. Other attacks use the hacked website as a vector of malware distribution or stealing property such as cash.

2.2.2 Defenses

We categorize defenses to external threats into two main categories. *Analysis* systems examine source code, network activity, computation, storage, and interprocess data flows, alerting the developer to vulnerabilities and anomalous behavior. Some systems augment monitoring with policy enforcement, allowing the developer to globally specify policies of allowed behavior. For example, information flow control systems track the flow of information across a distributed system, limiting flows based on policies allowed by the developer.

Complementary to analysis, *isolation* techniques are used to split a complex monolithic application into independent units based on the principle of least-privilege [206]. By limiting the code and data available to any particular unit, we can reduce the attack surface and impact of a successful attack.

In Figure 2.5, we summarize server-side web security solutions. All solutions focus on either preventing or reducing the impact of external intrusions. By imposing code structure constraints, applications can achieve resilience to different categories of vulnerabilities.

Monitoring, Analysis, and Policy Enforcement

Black Box Network Testing: A variety of black box techniques have been proposed to detect attack signatures [188, 204, 233] and block known attack vectors [39, 171, 211]. These defenses are most effective against attacks where there is a clearly identifiable signature of network traffic when the attack is being launched. Other systems can replay attacks [142, 58] to test an application's resilience. Black box techniques are desirable because they require no code changes from the developer. However, recent studies show that black box testing often misses many important vulnerabilities in the wild [53]. The study highlights the limitations in detecting complex attacks that leverage multiple vulnerabilities in a specific way or in detecting unknown vulnerabilities.

Source Code Analysis: While black box testing treats the code as an opaque block, source code analysis tools enable developers to analyze their source code for vulnerabilities. Some tools

Technique	Cross-site Scripting	SQL Injection	Improper Access Controls	Buffer Overflow	Code Injection	Path Traversal	Improper Input Validation	Information Exposure	Resource Management	Improper Authentication
Process Isolation [145, 60, 151, 187]	✗	✓	✓^{1}	✓^{1}	✓^{1}	✓^{1}	✓^{1}	✗	✓^{1}	✗
Information Flow Control [117, 186, 93, 146, 247]	✗	✗^{2}	✓	✗^{2}	✗^{2}	✗	✗^{2}	✓	✗	✗^{2}
Monitoring / Analysis / Firewall [233, 171, 211, 142, 58]	✗	✓^{3}	✗	✓^{3}	✓^{3}	✗	✓^{3}	✗	✓^{3}	✗

Figure 2.5: Categories of vulnerabilities mitigated by web application security techniques. ^{1}This thesis expands on these works to make per-user isolation practical at scale. ^{2}Invalid flows can be blocked, but it can be difficult to specify correct policies. ^{3}Intrusion detection systems use heuristics to deny requests, but can miss unknown exploits.

specifically search for known vulnerabilities [222, 248, 207, 59], such as parameter tampering [59]. In other systems, developers specify expected behavior and use symbolic execution to detect possible code paths that can lead to violations [110, 69]. Verification tools can also be used to prove that a particular implementation satisfies the specification [242, 128]. Code analysis and proofs provide assurances, but writing specifications and proofs can impose a heavy burden on developers and are not practical for rapid application development.

Global Data Security Policies: While the previous two sections focused on finding code vulnerabilities, other techniques monitor data accesses in a live system. Most commercial systems allow sysadmins to specify access control policies, typically at the granularity of a database table or collection. Some systems allow access control on each item or row of data in a table for more precise control [187, 16]. The typical development pattern for access control involves providing global access and iteratively restricting access to define a proper security policy. Access control can be enforced at the database interface. Other frameworks bind policies to the data [209, 65] which follows the data as it propagates through the system. Data policies are explicitly defined by the developer, or inferred through monitoring real access patterns at runtime [60]. Data security policies can prevent initial unauthorized access to a data source, but cannot restrict data propagation once access is granted.

Information Flow Control: While access control policies specify who has access to data, information flow control (IFC) allows the developer to track entire data flows from beginning to end [247, 146]. For example, IFC could allow developers to specify that billing information should never end up transmitted to a customer. Thus, IFC is a powerful method to limit data flows and prevent data exfiltration, even if the application was compromised. Hails [117] uses IFC to track privacy violations when untrusted third-party applications run on private data provided by a web service. PHP Aspis [186] uses IFC to guard against injection attacks and DBTaint [93] tracks IFC across different applications. While IFC systems can block invalid data flows, it does not prevent service disruption if the server is compromised. Furthermore, specifying correct policies for a complex cloud application can be difficult across many internal services.

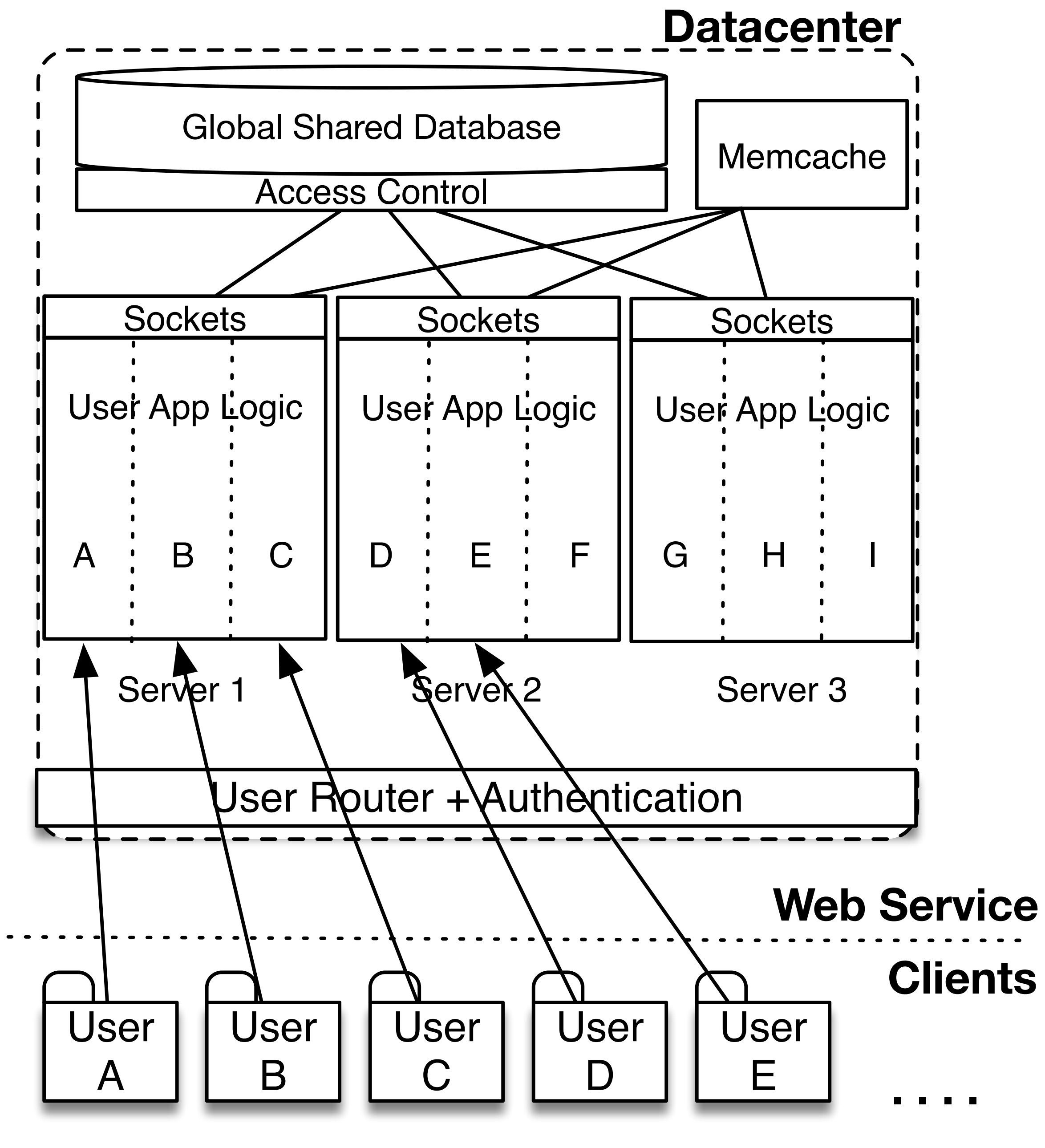

Figure 2.6: Layout of a service decomposed into per-user sandboxes, such as CLAMP [187].

Isolation

An alternative approach to limiting the effects of an attack is to run different components of a service in its own sandbox. By separating the code and data into different sandboxes, compromise of a single sandbox does not affect other sandboxes. Each sandbox can be hardened to minimize their respective attack surfaces.

Service Isolation: Service-oriented architecture provides a natural way to isolate components of a cloud application. OKWS [145] and Passe [60] introduce process isolation within an individual web application. These frameworks provide protection boundaries between naturally isolated components of the application (e.g. search and storage). Passe also introduces a mechanism for automatically generating a security policy by monitoring accesses during normal operation.

User Isolation: Per-user isolation takes this a step further and isolates users on the server, such that compromising the cloud service as a single user does not affect other users. πBox [151] introduces a per-user sandbox that spans a mobile app and web server; it interposes on all communication between users, with the goal of providing an end-to-end privacy-preserving mobile-cloud platform. CLAMP [187] introduced per-user sandboxes for server-side code execution, spawning a new virtual machine for each user session. To port existing web applications written in a shared-everything model, developers in CLAMP specify an access control policy to limit each user's view of the database. Specifying a correct data policy can become intractable if policies must be cross-table or cross-database.

2.2.3 Summary

Many previous efforts to defend from external threats involve a top-down approach. In these systems, developers need to be able to properly specify allowed behavior globally. However, as applications become more complex, often with interoperating internal cloud services, it becomes intractable to do this correctly. In Chapter 3, we investigate the costs and effectiveness of a bottom-up approach, where all users are completely isolated to start and developers use an explicit interface to share data between users.

2.3 Insider Threats

2.3.1 Threat Model

When considering insider threats, we treat the cloud service as part of the adversary. We assume all servers are collecting information about all client network requests, such as the source, operation type, parameters, timing, and size of requests. Servers can exhibit arbitrarily malicious behavior, such as send faulty responses to clients that deviate from the expected protocol.

When considering insider threats, we must consider all ways in which user information is leaked to the adversary. For example, an adversary can monitor *how* a client uses the cloud service, in addition to *what* a client sends. Metadata, such as with whom clients communicate or the schedule of network requests, is often valuable to an adversary (e.g. a government tracking dissident groups). As a consequence, merely encrypting content, such as images and message data, is insufficient for protecting user privacy. Recent studies have shown that an attacker could learn 80% of a user's search keywords just by monitoring access patterns, even when messages were encrypted [136].

When a persistent adversary can monitor the system over time, protecting against insider threats prove even more challenging. Many systems that are designed for user privacy are weak against persistent adversaries [41, 43, 210, 232, 84, 85, 86, 149, 245]. For example, in an intersection attack [91, 141, 163], the adversary tracks when users send requests in order to measure the statistical likelihood that two users are communicating together.

In some related work, there is a separate notion of a network adversary that can either passively monitor or actively manipulate packets on the network. Because we consider cloud applications where users send all data through a cloud service, we can treat a network adversary as a type of insider threat.

In practice, we do not need to assume the developer is intentionally malicious. Insider threats can manifest in a cloud application in many other ways. For example, a developer's credentials may be stolen in a phishing attack. A government could compel the cloud service to install surveillance tools. A malicious attacker could have successfully installed malware on cloud servers. In these cases, the attacker has as much power as the developer to mount attacks on users.

2.3.2 Security Goals:

The prior work in this space includes a wide range of security goals for user privacy from insider threats. We summarize common security goals in this section.

k-anonymity: Systems based on k-anonymity focus on privacy in the context of a single round of communication. Suppose a subset k out of n total users each send a message in a round. These systems ensure that the adversary cannot learn from which of k users any particular message was sent. These systems do not provide guarantees when the system is observed across many rounds of communication.

Differential Privacy: Differential privacy [101, 102, 167] provides a theoretical framework for quantifying information leakage. By introducing randomness into responses, the system can limit information disclosure. For example, differential privacy has been applied to databases [168], allowing approximately accurate statistical queries over sensitive data, while using randomness to protect individual records. Differential privacy has also been applied to aggregate analytics [107], where users introduce noise into their sensitive data before uploading it to a database. In private communication systems where an adversary is trying to determine which users are communicating, differential privacy can also be used to quantify information leakage as a function of the number of fake requests [232]. In each of these systems, the developer must tune the noise, at the cost of accuracy and performance, in order to reduce information leakage to an acceptable level.

Indistinguishability: A stronger form of privacy used in Chapter 4 of this thesis is indistinguishability, where an adversary cannot distinguish between any two access patterns. For example if a communication system consists of requests to a cloud service, the adversary should not be able to distinguish between a particular sequence of requests, a random set of requests, or an idle user.

Anytrust: A common assumption made in private messaging systems and in Chapter 4 of this thesis is the anytrust model. In these systems, we assume l independent servers and at least one of l servers to be honest. This honest server adheres to the specified protocol and does not collude with other servers to de-anonymize users. For example anytrust may be achieved if each server was managed by a different company in different countries, provided those companies do not collude. The anytrust assumption can be combined with k-anonymity, differential privacy, and indistinguishability.

2.3.3 Defenses

When defending from insider threats, we consider adversaries with control over the cloud. These adversaries can install their own operating system, monitor and response to network traffic, and change any state on the system (e.g. on disk). For these adversaries, we need to consider both the contents of user data, as well as metadata describing how a user interacts with the system. In this section, we categorize defenses between *data encryption* and *metadata protection*. With data encryption, we focus on the privacy of individual data items. Users can encrypt messages with a key, preventing anyone without the key from reading the message. With metadata protection, we focus on hiding metadata such as the source, destination, or pattern of messages. There exist a large range of security goals, such as indistinguishability and differential privacy. In this section, we discuss the performance and security tradeoffs between various systems.

Storing Encrypted Data:

Encryption allows users to encode a *plaintext* message into *ciphertext* by using a secret key. Only users with the secret key can decrypt the ciphertext to read the original message. Some applications today offer end-to-end encryption, where clients encrypt all data before sending it to the cloud. A user can then give other users access by sharing the secret key. When all data on the cloud is encrypted with a sufficiently random key and a strong algorithm, users can gain confidence that data is likely to be safe, even if a service gets compromised.

A number of companies now provide end-to-end encryption for a variety of applications, such as messaging [33] and password storage [35]. However in general, encrypting data can compromise service functionality. For example, an email application needs to train on user data to detect spam, which becomes impossible if all data is encrypted. In research, Mylar [194] proposed a general way to transparently incorporate end-to-end encryption for certain applications, supporting a limited form of search over encrypted data.

Computing on Encrypted Data:

Property-preserving Encrypted Databases: In property-preserving encrypted databases, first proposed by CryptDB [193], data is encrypted with specialized encryption schemes that reveal certain properties about the underlying data. For example, with order-preserving encryption, ciphertexts can be sorted in the same order as the corresponding plaintext. By definition, property-preserving encryption reveals information about the underlying data. Recent studies have shown that the underlying data can be inferred in more than 80% of records when applying order-preserving encryption to medical databases [175].

Homomorphic Encryption: Homomorphic encryption describes a set of techniques for performing computations on encrypted data. For example, Pallier [185] is a partially homomorphic cryptosystem, where a cloud service could compute the sum of two values by multiplying their respective ciphertexts. If $Enc(x)$ is an encryption function and x is a plaintext message, then $Enc(x_1) \cdot Enc(x_2) = Enc(x_1 + x_2)$. Fully homomorphic encryption [114] schemes support arbitrary computations on encrypted data. These systems hold the promise of being able to support arbitrary cloud applications where the cloud service is completely blind to user data. Unfortunately, current implementations are impractically expensive. Improving the performance of homomorphic encryption is an active area of research.

Trusted Computing: Trusted computing provides an alternative model for building secure applications. The Trusted Platform Module (TPM) is a dedicated cryptoprocessor that embeds a secret key, generated randomly at manufacture time and stored in tamper-resistant hardware. The TPM can perform cryptographic functions, like encryption and signatures, with this secret key. Because the secret key is signed by the hardware manufacturer, the processor can also prove to remote users that it has a valid key by signing a random number.

Intel SGX [22] introduced *enclaves*, protected areas of execution in memory. Modern Intel processors can sign a hash of the code and data loaded in an enclave, forming a remote attestation. Remote attestations prove to a remote user that the processor is in fact running what the user expects, and not malicious surveillance software. Intel SGX also provides regions of encrypted memory. Assuming that the user trusts the hardware manufacturer, these hardware extensions allow a user to run arbitrary applications efficiently on remote hardware, such as the cloud, with hardware-based assurances that the software on the system is configured as expected and data outside of the processor is encrypted. An application loaded into an SGX enclave can then perform key exchange to set up a secure channel with a remote client, allowing the client to send encrypted data for processing.

Recent work has explored ways to protect the integrity and confidentiality of such applications, even when the underlying operating system, virtual machine monitor, and firmware are not trusted. Haven uses Intel SGX enclaves to protect unmodified applications from a malicious operating system. [54]. Ryoan leverages both Intel SGX and compiler techniques to restrict unauthorized data flows when running untrusted applications in enclaves [134].

Limitations: While the ability to compute over encrypted data is a powerful primitive, it is not sufficient to provide strong user privacy in cloud applications. Attackers can leverage side channels by observing timing of instructions, power consumption, cache behavior, and memory/network/disk usage to gain more information. For example if the same location in memory is accessed by the application only when Alice and Bob make requests, an observer can infer that Alice and Bob are communicating. Other attacks have shown that secret encryption keys can be leaked by observing cache timing [213] and power consumption [161]. Furthermore, all trusted computing schemes rely on strong assumptions on the integrity of hardware secrets.

Even if individual messages are encrypted, an insider can still observe how users interact with the system. Examples of such metadata observations:

- Time that data is sent or received
- Message sizes
- Patterns of communication (e.g. which users communicate together)
- Time that users are online

In order to remove metadata from an insider threat, a number of systems have been proposed with varying security goals. In Figure 2.7, we summarize recent systems, their security goals, techniques used, and intended application.

System	Security Goal	Threat Model	Technique	Application
Talek	indisting.	≥ 1	IT-PIR	group messaging
Pynchon [210]	k-anon.	≥ 1	mixnet/IT-PIR	email
Riffle [149]	k-anon.	≥ 1	mixnet/IT-PIR	file-sharing
Riposte [84]	k-anon.	≥ 1	IT-PIR	broadcast
Dissent [85]	k-anon.	≥ 1	DC-nets	broadcast
Vuvuzela [232]	diff. privacy	≥ 1	mixnet	1–1 messaging
DP5 [63]	indisting.	≥ 1	IT-PIR	chat presence
Popcorn [126]	indisting.	≥ 1	C-PIR/IT-PIR	video stream
Pung [48]	indisting.	0	C-PIR	key-value store
ORAM [217, 218, 219]	indisting.	0	ORAM	storage

Figure 2.7: Comparison of privacy-related techniques. Indistinguishability offers the strongest level of privacy. Prior work with indistinguishability are either expensive or tailored to a single application. Under repeated use, k-anonymity systems leak which of k clients originated a particular message, opening the systems to intersection attacks [91, 141, 163]. Differential privacy provides a formal framework to quantify information leakage and privacy bounds, but does not inherently guarantee that no information is leaked. The threat model column denotes the number of servers that must be honest for security properties to hold.

Private Information Retrieval:

Private information retrieval (PIR) allows a client to retrieve a block from an untrusted server, or set of servers, without revealing to any server the blocks of interest to the client. In the naive approach, a client downloads the entire database to read a single block. An adversary would be unable to determine in which block a client is interested. The challenge of PIR is to design more efficient techniques.

There exist two major categories of PIR techniques, information-theoretic PIR (IT-PIR) [78, 118, 95] and computational PIR (C-PIR) [148]. Of the best known techniques and implementations, IT-PIR has been shown to be orders of magnitude less expensive in computation and network usage than C-PIR. This trade-off comes at the cost of supporting a weaker threat model. With C-PIR, privacy is preserved even with one untrusted server, whereas IT-PIR requires one honest server in an anytrust threat model.

Information-theoretic PIR: In order to gain an intuition for the performance and cost of IT-PIR, we illustrate the protocol with an example (Figure 2.8). Let l represent the number of servers in the system, each storing a full copy of the database, partitioned into equal sized blocks. While IT-PIR generalizes to arbitrary numbers of servers and blocks, the example in Figure 2.8 contains $l = 3$ servers and $n = 3$ blocks ($\{B_1, B_2, B_3\}$).

1. Suppose a client wants to read the second block, $\beta = 2$. It encodes that with the bit vector, $q' = [0, 1, 0]$, which consists of zeros and a one in position β.

2. The client generates $l - 1$ random n-bit request vectors, q_1 and q_2, for each of the l servers except one.

3. The request vector for the remaining server is computed by taking the XOR of the vectors from (1) and (2), $q_l = q' \oplus q_1 \oplus \ldots \oplus q_{l-1}$.

4. The client then sends q_i, to server i for $1 \leq i \leq l$. Because request vectors are generated randomly, this reveals no information to any collection of $< l$ colluding servers.

5. Suppose the server receives $q_i = [b_1, \ldots, b_n]$ and B_j represents the j^{th} block of the database. Each server computes R_i, the XOR of all B_j for which $b_j == 1$ and returns R_i to the client.

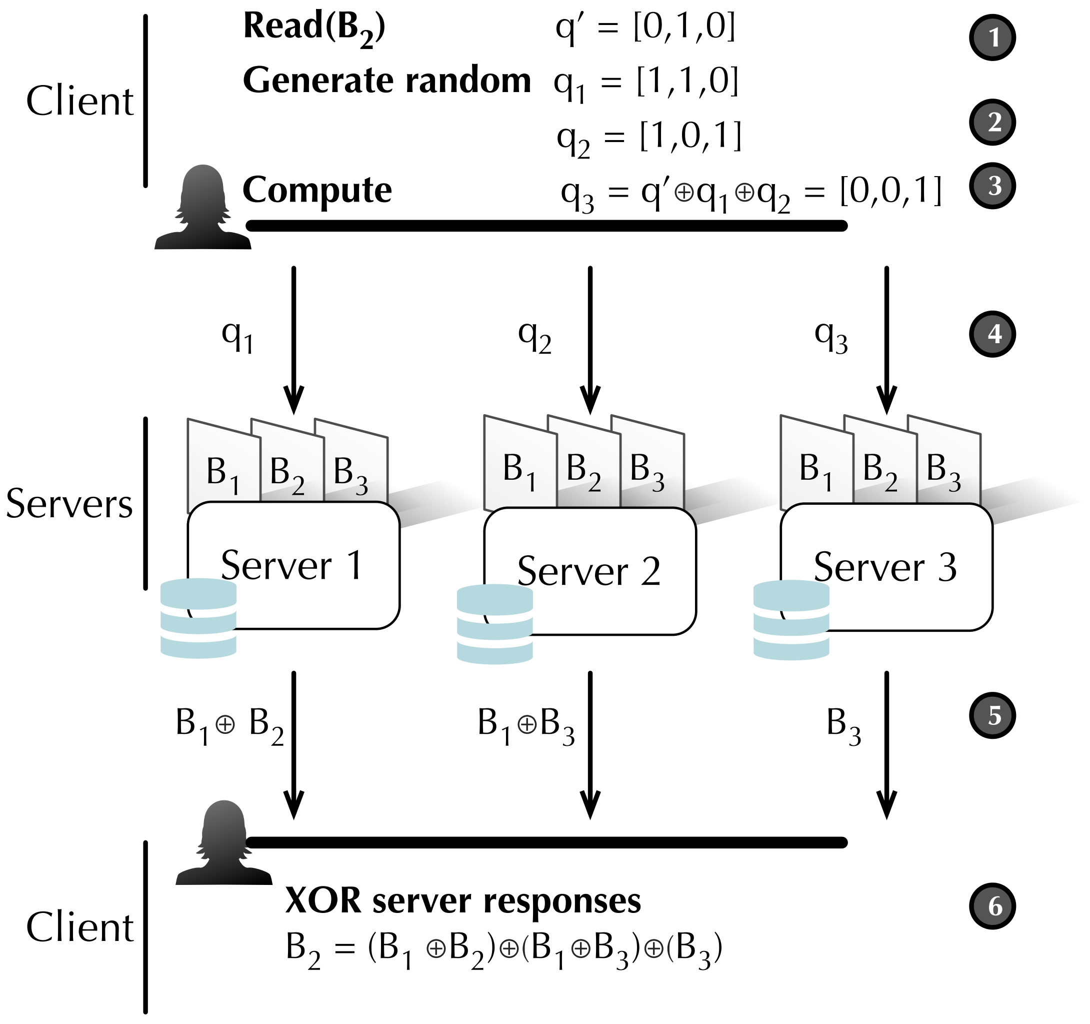

Figure 2.8: Information-theoretic PIR example. Each client sends a random request vector to each server, except for one, which receives the XOR of the other random requests plus the true request. Each server responds with data equal in size to a single data block. As long as some server does not collude, the remaining servers cannot determine q', which of n blocks the client is retrieving.

6. The client retrieves the desired block, B_β, by taking the XOR of all R_i replies, $B_\beta = R_1 \oplus \ldots \oplus R_l$.

Computational PIR: C-PIR is conceptually similar to IT-PIR, except partially homomorphic encryption is used to hide queries in place of secret sharing across non-colluding servers. We illustrate C-PIR with an example with one untrusted server and $n = 3$ blocks ($\{B_1, B_2, B_3\}$). Let $Enc(pk, x)$ and $Dec(sk, c)$ represent the encryption and decryption functions of an additive homomorphic cryptosystem, such as Pallier [185], where pk is the public key, sk is the secret key, x is the plaintext message, and c is the resulting ciphertext. Recall that in additive homomorphic encryption, $Dec(sk, Enc(pk, x_1) \cdot Enc(pk, x_2)) = x_1 + x_2$. Enc is also a randomized function, which means repeated calls to encrypt the same message yields different ciphertexts. Specifically for Pallier, let the public key, $pk = (g, m)$, consist of a base g and modulus m and $r \in \{0, \ldots, (m-1)\}$ be a random number. Then,

$$\begin{aligned} Enc(pk, x_1) \cdot Enc(pk, m_2) &= (g^{x_1} r_1^m)(g^{x_2} r_2^m) \mod m^2 \\ &= g^{x_1+x_2}(r_1 r_2)^m \mod m^2 \\ &= Enc(pk, x_1 + x_2) \end{aligned}$$

1. Suppose a client wanted to read the second block, $\beta = 2$, encoded by the bit vector, $q' = [0, 1, 0]$, which consists zeros and a one in position β.

2. The client generates a request by encrypting every bit with the public key, $q = [Enc(pk, 0), Enc(pk, 1), Enc(pk, 0)] = [c_1, c_2, c_3]$.

3. The client then sends the request, q, to the server. Because each bit is encrypted, this reveals no information to the server.

4. The server receives $q = [c_1, \ldots, c_n]$. Let B_i represent the i^{th} block of the database. The server computes $R = \prod_{i=1}^{n} c_i^{B_i}$ and returns R to the client.

5. The client restores the desired block, B_β, by decrypting the result R. $B_\beta = Dec(sk, R)$.

Recall for Pallier,

$$\begin{aligned}
B_\beta &= Dec(sk, R) \\
&= Dec(sk, \prod_{i=1}^{n} c_i^{B_i}) \\
&= \sum_{i=1}^{n} Dec(sk, c_i) \cdot B_i \\
&= Dec(sk, c_1) \cdot B_1 + Dec(sk, c_2) \cdot B_2 + Dec(sk, c_3) \cdot B_3 \\
&= 0 \cdot B_1 + 1 \cdot B_2 + 0 \cdot B_3 = B_2
\end{aligned}$$

Benefits and Limitations of PIR: PIR has desirable network properties: a client sends one request vector to and receives one block from each server. These requests and responses appear random to the network and the servers. PIR is computationally expensive, with read cost that scales with the size and number of blocks in the system. C-PIR uses more expensive cryptographic operations compared to XOR in IT-PIR. The size of a client request scales with total number of blocks and the client work scales with the number of servers. IT-PIR also requires consistent snapshots across servers, with equal sized blocks in the data structure.

On its own, PIR is only a protocol for private reads. How writes are performed can also affect user privacy. For example, two users writing to the same location leaks information, even if all reads are serviced using PIR.

PIR-based systems: DP5 [63] is a chat presence system to allow users to discover their friends' online presence. Users periodically send their online status (e.g. online/offline/away) to a set of servers. Users then poll the servers using PIR to retrieve their friends' statuses, without revealing which users they are following. Popcorn [126] uses both C-PIR and IT-PIR to construct a private read-only video streaming system over a static video database. Both DP5 and Popcorn leverage assumptions about application workloads to make PIR practical. Pung [48] is a PIR-based key-value store with user request indistinguishability. Pung assumes C-PIR, and thus works with fully untrusted infrastructure. However, their overhead is high. For example, key lookup requires $O(log(n))$ round trip PIR requests.

Private Information Storage:

A related concept to PIR is private information storage (PIS) [182]. PIS allows a client to write to a row to a database of n rows without revealing which row was updated. Consider an example with 2 non-colluding servers storing a L-bit replica of the database, and a client that wants to write a "1" into bit β of the database.

1. The client composes e_β, a L-bit string of zeros, except with a 1 in the β-th bit.
2. The client then generates random L-bit string, w_1, and sends w_1 to the first server.
3. The client computes $w_2 = e_\beta \oplus w_1$, and sends w_2 to the second server.

Each server collects write requests from all clients and XORs them together. After processing n writes, the database at the first and second servers will be D_1 and D_2 respectively. At the end of the round, servers combine their respective databases to retrieve the plaintext database, D.

$$\begin{aligned}
D_1 &= w_{1,1} \oplus \ldots \oplus w_{1,n} \\
D_2 &= w_{2,1} \oplus \ldots \oplus w_{2,n} \\
&= e_{\beta_1} \oplus w_{1,1} \oplus \ldots \oplus e_{\beta_n} \oplus w_{1,n} \\
&= (e_{\beta_1} \oplus \ldots \oplus e_{\beta_n}) \oplus D_1 \\
D &= D_1 \oplus D_2 \\
D &= (e_{\beta_1} \oplus \ldots \oplus e_{\beta_n})
\end{aligned}$$

PIS schemes provide k-anonymity across a single round of communication. As long as one server does not collude, the adversary cannot determine which bit was written by which user. However, the above scheme incurs $O(n)$ communication complexity and is vulnerable to write conflicts between users. Ostrovsky and Shoup [182] introduced a PIS scheme that incurred poly-logarithmic communication complexity. Riposte [84] further improves on this work, applying distributed point functions for $O(\sqrt{n})$ writes and addressing write conflicts with a coding technique. Riposte is a scalable broadcast messaging system and does not specify how users should read from the system. Riposte is vulnerable to intersection attacks when the system is observed for more than a single round of communication.

Mixnets:

Chaum mixnets [71, 72, 125, 137] and verifiable cryptographic shuffles [64, 113, 176] are a set to techniques to obfuscate the source of a message. All mixnet systems rely on the anytrust assumption to offer k-anonymity in a round of communication.

In order to explain how mixnets work, we illustrate an example with l servers, where one is assumed to be honest and non-colluding. We arbitrarily assign an order to the servers, each with public keys $pk_1 \ldots pk_l$ respectively. Clients prepare messages by iteratively encrypting the message with server public keys, in a process known as onion encryption. The plaintext, x is first encrypted with the last server's key. Then, the i-th layer is computed by encrypting the $(i+1)^{th}$ layer with the i-th server's public key. As a consequence, messages must be passed in sequence from s_1 to s_l in order to decrypt the subsequent layers.

$$C_l = Enc(pk_l, x)$$
$$C_i = Enc(pk_i, C_{i+1})$$

Mixnet systems operate at the granularity of rounds of communication. In a round, k clients submit their messages to the first server. Each server reorders the messages in a random permutation before decrypting a layer off the onion encryption and passing the messages to the next server. As long as one of the servers remains honest, plaintext messages should arrive randomly ordered at the final server. Thus, mixnets provide k-anonymity, where the adversary cannot tell which of k users is the source of any particular message in a round. Mixnets do not offer security guarantees between rounds and require a set honest users in every round. If only one honest user participates in a round and all other messages come from the adversary, the adversary can determine the source.

Mixnet-based systems: Mixnets are applied to provide anonymity in a variety of applications, such as email [72, 90, 125] ISDN networks [137], voting [139, 176], data collection [64], MapReduce computations [98], social applications [71], and web proxies [111]. When a mixnet is used to access an encrypted database, unlinkability can be difficult to guarantee when the database is also untrusted. Network-level onion routing [197, 97, 138] systems can also be used to access an encrypted database with similar limitations. Using differential privacy analysis, Vuvuzela [232] formalizes the amount of noise that honest shufflers would need to inject in order to bound information leakage at the database.

Combining Mixnets with PIR: Pynchon Gate [210] is an email system where messages are sent to email servers through mixnets. Emails are dumped daily to distributor servers, where clients use IT-PIR to privately retrieve messages. While PIR hides which messages clients are interested in, the email server stores the communication patterns between email addresses. Riffle [149] follows a similar design, using mixnets to send messages and IT-PIR to retrieve messages to provide k-anonymity in a round. Riffle and other systems based on k-anonymity are susceptible to intersection attacks [91, 141, 163], where communication patterns can be deanonymized over time with statistical analysis if users are not always online.

DC-nets:

DC-nets are a technique for broadcasting anonymous messages based on the dining cryptographer's problem. In order to explain DC-nets, we illustrate an example where Alice would like to broadcast a single bit, b, to Bob and Charlie. Bob, Charlie, and any adversary listening on their communications should not know who sent the bit.

First every pair of participants establishes a 1-bit random shared secret by flipping a coin. Alice and Bob share s_{ab}, Bob and Charlie share s_{bc}, and Alice and Charlie share s_{ac}. In the second stage, every participant broadcasts the XOR of all the shared secrets they possess and the sender also includes the bit b. Thus, Alice broadcasts $t_a = s_{ab} \oplus s_{ac} \oplus b$, Bob broadcasts $t_b = s_{ab} \oplus s_{bc}$ and Charlie broadcasts $t_c = s_{ac} \oplus s_{bc}$. Finally, all participants take the XOR of all broadcasted bits to read b from the system.

$$\begin{aligned} t_a \oplus t_b \oplus t_c &= s_{ab} \oplus s_{ac} \oplus b \oplus s_{ab} \oplus s_{bc} \oplus s_{ac} \oplus s_{bc} \\ &= b \end{aligned}$$

DC-nets provide k-anonymity. At the end of a communication round, nobody aside from the sender knows which of the users sent the message. In a system with n users, DC-nets require random bits be shared between every pair of users for every message sent, yielding $O(n^2)$ communication complexity. In the protocol described, DC-nets are also weak to collisions; only one user can send a bit at a time.

DC-net systems: Herbivore [216] applies DC-nets to building file-sharing networks over small cliques. Dissent [85, 86, 245] extends traditional DC-nets to provide transmission schedules and accountability mechanisms for detecting misbehaving participants. DC-nets enable effective broadcast messaging, but they are not a good fit for cloud application workloads because of the high network costs.

Oblivious RAM (ORAM):

ORAM [119, 121, 181] is a set of protocols that allow a single trusted client to access an untrusted storage without revealing its access pattern, even to a strong adversary who controls the storage. ORAM offers a strong security goal based on indistinguishability of access patterns. Most ORAM schemes arrange blocks in a hierarchical scheme, where each node in the tree stores a bucket of blocks. In order to read a block, the client must read $\Omega(\log N)$ nodes on a path from the root to a leaf. Once a block is read, it must be re-encrypted with new randomness and moved to a new random location in the database. While recent work improve the cost of ORAM [79, 115, 147, 201, 219, 237], and offload some work onto dedicated servers [159, 217], the cryptographic security of the system depends on large reads and constant data shuffling. ORAM has a theoretical lower bound of $\Omega(logN)$ overhead [121] with respect to the total number of blocks in the system.

While ORAM protocols are inefficient when conducted over a wide-area network, it can be paired with a trusted processor in the cloud. The trusted processor enables the application to conduct the large ORAM reads and data reshuffling locally and securely on the cloud, only sending the final result to remote clients. ORAM and trusted processors present an opportunity to build arbitrary privacy-preserving applications subject to limits on side channels. Recent systems have applied trusted processors and ORAM to build virtual disks [159], filesystems [243], cloud storage [92, 218], and data analytics [249, 212].

2.4 Summary

This chapter summarizes research on securing user privacy in cloud applications. We identify both external and insider threats and describe existing attempts to hide data and metadata from these adversaries. Existing systems either suffer from poor performance or offer security goals that are weak to a persistent active adversary, suggesting that users can benefit from improving the performance of privacy techniques with strong security goals.

Chapter 3

SECURING CLOUD APPLICATIONS FROM EXTERNAL INTRUSION

I first focus on securing arbitrary cloud applications from external attacks. Web sites are routinely broken into, resulting in frequent service disruptions and massive leakage of private information. With the current architecture of most web services, wide-scale compromise is all too easy because the server-side application logic is part of the trusted computing base (TCB). Existing web applications are structured as monolithic controllers with access to all user data, interpreting user permissions in order to dynamically assemble pages for a user. Thus, compromises allow attackers nearly unimpeded access to all of the information available to the service. Data compromises of this nature have remained the largest class of web application vulnerabilities for the full decade of OWASP (Open Web Application Security Project) vulnerability reports [29].

One approach to securing web applications is to de-privilege the code into sandboxed processes for individual services (e.g. search and newsfeed) [145, 60] or for individual users [187, 151]. Sandboxing users of a modern web application with reasonable performance is challenging. Generating a single page can span many layers of web servers, caches, storage systems, and coordinators, across multiple machines and data centers. A user container must isolate users at every layer of the stack, while supporting cross-user data sharing and application flexibility. Even if the code execution environment is isolated, existing frameworks assume a global data model. Instead of cross-layer isolation, applications are written assuming full access to a single shared database across all users, requiring that the developer iteratively restrict global data policies [187, 65, 209], e.g. by potentially using information flow control [186, 117].

In this chapter, we propose an alternative data security model in a *shared-nothing* web architecture. The web platform already treats the browser as a per-user isolated sandbox running untrusted code. We extend this into the server and database, where application code is run in a strongly isolated sandbox containing its own logical data partition with the privileges of the logged-

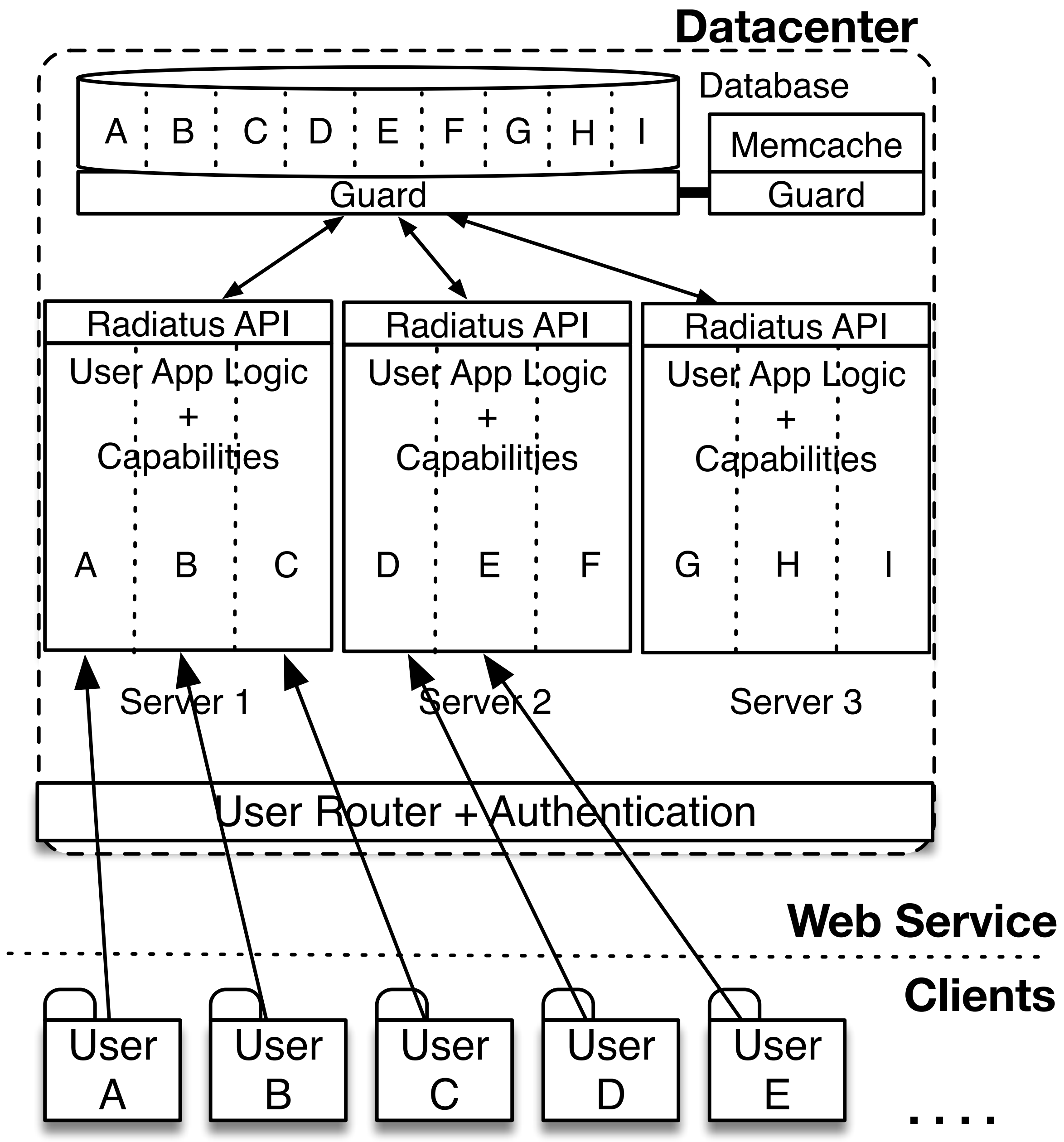

Figure 3.1: Layout of a Radiatus web service. In Radiatus, we extend isolation into the storage layer in an end-to-end shared-nothing architecture. Both application logic and storage for each user runs in sandboxes with de-escalated privileges, which communicate through a restricted message passing interface.

in user. By default, no state is shared between users. Developers write their applications in terms of mutually distrusting users, who can only communicate through messages. Radiatus's distributed runtime uses a capability-based security system to protect access to private data, while being both storage space-efficient and horizontally scalable. Capability-based security can be a more tractable approach for the data sharing patterns of web applications across internal services compared to group-based data policies. Barring compromise of the user authentication mechanism, intrusions are contained to the subset of data already available to the malicious user.

Our goal with Radiatus is to show that we can implement a shared-nothing web architecture in a way that is practical today with scale, cost, and performance within a factor of existing shared-everything web frameworks. The changes to the server are completely *transparent to the user*, who continues to access the site through an unmodified web browser. Similarly, developer should be able to continue using existing programming languages, distributed databases, distributed caches, content delivery networks, and infrastructure-as-a-service cloud providers.

In order to evaluate the strengths, weaknesses, and performance implications of our architecture, we have implemented a Node.js-based web framework in 8764 lines of code, called Radiatus, and three applications: an academic social network, a file sharing tool, and a messaging service. The difficulty of porting applications can vary depending on the application workload. For example, we found it much easier to port applications that were written to be self-deployed or federated. We describe our experiences porting Arc Forum, the engine behind Hacker News, to run on our system.

While our framework can contain the damage caused by many external intrusions and exploits, we do not protect against insider threats with administrative access to site infrastructure. The framework also does not attempt to protect clients. I discuss ways to address insider threats in Chapter 4. Radiatus is complementary to other web security-related work, including encryption [193, 194, 114], language-based security [69, 164, 81], and bug-finding [222, 110, 248, 207, 59] as discussed in Chapter 2.

The rest of the chapter elaborates on our contributions:

- I describe the Radiatus shared-nothing architecture for strongly isolating users in web applications and describe how existing web applications can be written in this model (§ 3.1).

- I have built the Radiatus platform and illustrate the Radiatus API with three applications written in the framework (§ 3.2).

- I show that sandboxing users prevents large-scale exploitation of the most severe web-related vulnerabilities of 2014. We quantify the performance impact of Radiatus, including using it on a 500-node deployment on Amazon AWS (§ 3.3).

3.1 Radiatus Design

While introducing per-user isolation seems like an intuitively simple idea, a number of challenges make it uniquely difficult for web applications. Previous work [145, 60, 151] has proposed process isolation in a single web server, but without addressing the practical demands of how to accomplish per-user isolation in the context of a scalable web service using a variety of distributed storage systems, caches, and content distribution networks.

For example, how do you support per-user database security? Different storage backends have different user models, a problem sidestepped when application code is trusted. How do you manage memory consumption and storage costs? We can give each user their own cache and storage silo, but many objects in modern web applications are shared across users, sometimes across millions of users. How do you efficiently support one-to-many communication patterns? Copying data between users may not be feasible, and certainly adds overhead. How do you perform distributed container management? User containers need to be placed to minimize communication cost and maximize load balancing.

3.1.1 Threat Model

We focus on preventing attacks aimed at compromising a web server from an external vantage point. We assume a malicious user can craft arbitrary network packets and send arbitrary requests to the server. This includes URL interpretation attacks, server-side includes, code injection attacks, SQL injection, malicious file executions, and buffer overflows.

We do not address server misconfiguration, insider attacks, social engineering, or weak cryptographic primitives. Each of these are better addressed by other, complementary techniques [193, 114, 117, 67, 73, 99].

3.1.2 Goals

In this section, we describe the user container model and the techniques that we use to make per-user isolation practical.

- **Strong Isolation:** Radiatus should provide a general framework for isolating users, such that server-side application vulnerabilities do not compromise data integrity or service availability for other users.
- **Minimal Overhead:** We should support each additional user with minimal overhead in performance and cost compared to existing web frameworks.
- **Scalable:** The scale and performance of applications written and deployed on Radiatus should be comparable to that of existing web frameworks.
- **Interoperability:** The system should interoperate with existing cloud infrastructure, storage systems, and tools.

3.1.3 Approach

Figure 3.1 shows the high-level model of a Radiatus application. We move developer's code into a protection domain that runs on behalf of the user. Attackers that exploit a vulnerability in that code are limited to the containers they have credentials to access.

Sandbox Users: In Radiatus, we spawn a sandboxed process, which we call a *user container*, for each active user. All code written by the developer runs inside this protection domain with the privileges of the given user. As such, the user container can only read and modify data that is owned by the user. In practice, we use Linux containers, which provides memory, filesystem, and fault isolation between containers. We leverage existing techniques to apply resource limits to user processes.

Limited Interfaces: Existing web services expose a single large HTTP interface for authentication, user actions, and content fetches. Radiatus splits this interface into three with access restricted by least privilege. Any user can authenticate with the *user authenticator*. Once equipped with an authorization token, the *user router* forwards a user's request to their own de-privileged containers;

requests are never directly processed by privileged code. We expose a *cross-container message* interface between user containers to facilitate data sharing. User containers can only communicate with mutual consent and access to this interface is blocked by default.

Passive Containers: To minimize memory overhead, containers are offline by default. Applications are written using an event-based programming model. A distributed container manager determines placement, suspends, and resumes user containers as necessary to process incoming requests.

Distributed Capabilities: Logically, each user has a storage partition, but physically the underlying data is shared and stored on commodity databases. A *storage guard*, intermediates access to the database and enforces access control for user data. We provide capabilities for scalable fine-grained access control across disparate database systems, while deduplicating common data between users.

Minimize trusted computing base: When processing requests from the browser or between containers, we expect developers to employ *defensive programming*, treating communicating parties as untrusted entities. In addition to these message checks, our trusted computing base consists of a user authenticator, user router, storage guard, Radiatus runtime, and container mechanism (e.g. OS processes/hypervisor). These components are written once and shared across all Radiatus web applications.

3.1.4 Example Application

In this section, we walk through the typical lifecycle of Radiatus applications. A user container acts as the server-side agent for each user, in a shared-nothing architecture. The container manages the user's private data and capabilities to access data that has been shared with that user. When a user visits the site, the application code running in the container retrieves the data necessary to assemble the desired page. Because user containers run identical application logic, our system maintains a pool of instantiated but unconfigured containers, which are lazily bound when users log in and subsequently destroyed when they log out. This technique reduces the latency of the first request by a new user.

Figure 3.2 shows the workflow of sharing a file in a Radiatus application. Consider the scenario

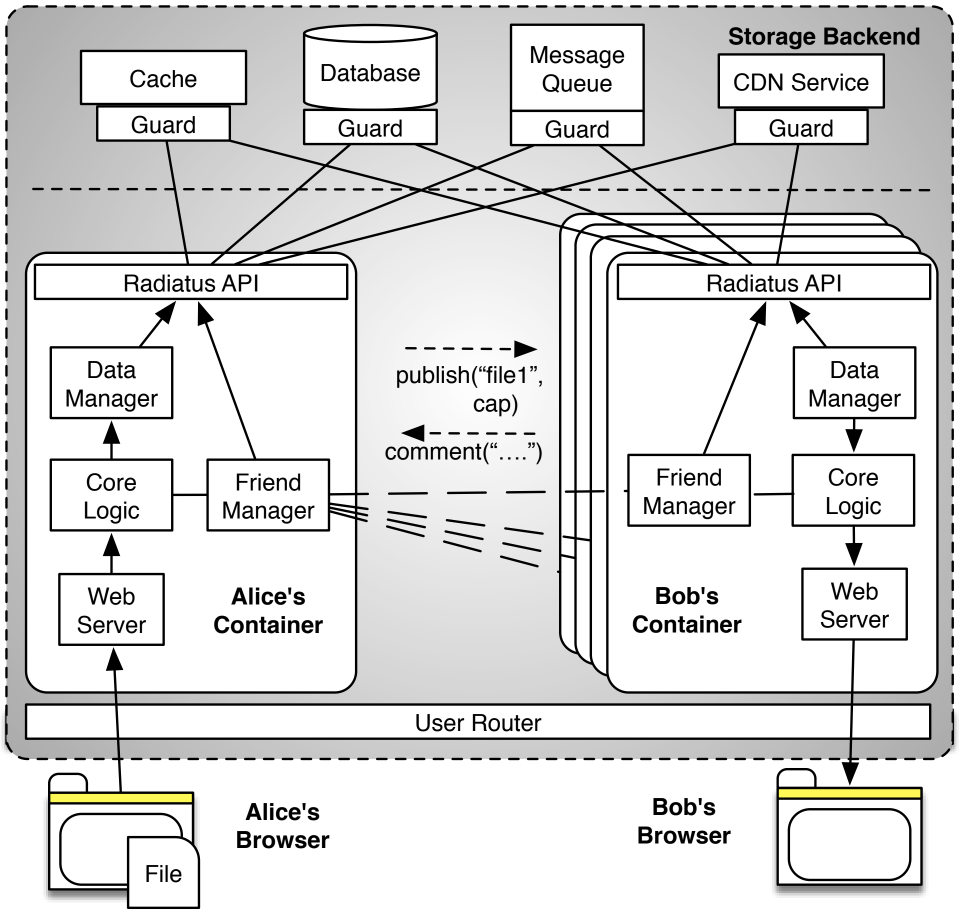

Figure 3.2: Workflow of uploading and sharing files in the user container model. Each user container acts in isolation and stores data in a private location. Alice and Bob communicate using a typed message passing interface, through which they share files and messages.

where Alice shares a file with Bob. When Alice logs in, a user container is assigned to her. Alice uploads the file to her user container and the application uses the storage interface to store the file and metadata. The storage request returns a capability giving Alice (and only Alice) the ability to retrieve the paper. Using the cross-container communication interface, Alice's container can send the capability in a message to Bob's container. If Bob is currently offline, this message is stored in a persistent queue until Bob logs in.

Capabilities are transferable and provide read-only access to an immutable snapshot of data. If Alice makes changes to the file, she would need to send another capability to Bob for him to see the revision. As we will see later, the fact that capabilities refer to immutable data is important for system scalability and capability revocation.

3.1.5 Container Management and User Routing

A container manager keeps track of the web server on which each user container is running. The container manager suspends containers when they become inactive. When a user container needs to be initiated, the manager chooses an appropriate server. The container manager also attempts to co-locate user containers that frequently communicate with each other.

Like a traditional load balancer, the Radiatus user router proxies connections to servers, optionally terminating the TLS connection. The user router looks for a session cookie in the HTTP request, uniquely identifying the user. If the router instance has never seen the user before, it will query the container manager for the host server of the user container and caches this information. Subsequent requests from this user are then forwarded to the proper user container. Even if an application contains an exploitable vulnerability, network requests from the attacker will only be routed to the attacker's container. User routers are horizontally scaled as necessary to meet traffic demands.

3.1.6 Cross-Container Communications

In order to support offline users, messages between user containers are persisted in a distributed message queue, like Apache Kafka [5] and Amazon SQS [3], until the recipient's user container wakes up. Developers can also specify *wake-on message* policies for high priority messages (e.g.

```
type: {
  title: ‘string’,
  author: ‘string’,
  timestamp: ‘number’,
  pdf: ‘buffer’
},
rate: 100,
priority: ‘wake’
```

Figure 3.3: Example of a declared message type. Developers must specify the type, rate limit, and priority of all messages across the cross-container message interface.

ones that impact the user interface).

By default, containers are disconnected. Developers use `addPeer/removePeer` calls in the API to specify permitted communication channels. Depending on the application, developers may require bilateral consent before accepting messages (e.g. friend requests), or they may allow one-way consent (e.g. chat messages). Limiting the connectedness of the container graph makes it more difficult for the attacker to crawl the site by slowing virus propagation.

We introduce two optimizations to relieve stress on our message queue. First, messages between two user containers on the same machine are directly routed to each other, bypassing the network. Second, we batch messages to different user containers on the same node to reduce overhead.

As with incoming handling web requests, developers are expected to employ defensive programming when writing message interfaces, treating communicating parties as untrusted entities. While it is possible for vulnerabilities to exist in this code (cf. CWE-20), Radiatus employs a defense-in-depth strategy to mitigate the risk of widespread exploitation. Attackers must first compromise a user container to even have access to the cross-container interface. They must then find an exploit that provides control over the neighbor container in a way that can be propagated.

Because containers run on server hardware, we can use three additional techniques to limit attacks:

Typed interfaces: Developers must declare the messages and protocol for cross-container communication [133]. The system checks all messages at runtime, denying non-conforming messages. Figure 3.3 shows an example of a declared message type.

Resource limits: Each container is subject to strict limits on resources (e.g. CPU, memory, network), to prevent attacks from launching denial-of-service attacks.

Anomaly detection and eviction: While we have not yet implemented this feature, we could use machine learning to build a steady state model of expected container behavior, because application communication patterns are hardcoded into the application logic. Anomalous resource usage or communication patterns can trigger an operator alert for manual review. A dashboard gives the operator controls to pause containers, partition the network and evict users as necessary to preserve the health of the system. Many of the techniques, such as real-time notifications, high-speed event monitoring, and policy scripts, are borrowed from decades of research in intrusion detection systems [188].

3.1.7 Storage Access

The storage guard layer provides access control to protect back-end storage systems. In our shared-nothing architecture, each user reads and writes into their own logical partition. A storage guard instance is co-located with each database entry point, intercepts all requests, and tags each record with the owner. Storage guard implementations must be adapted to communicate with each type of database (e.g. SQL, NoSQL). For example in our MongoDB deployment, an instance of the storage guard is run in front of each `mongos` query router. As the MongoDB cluster grows, there can be many storage guards and query routers independently coordinating distributed operations over the database, itself partitioned over many `mongod` database shards. We assume the developer will use a heterogeneous set of diverse databases. While we could have used the built-in access control provided by the database, synchronizing fine-grained access control policies across databases could easily become a bottleneck.

Key-Value Storage	
Name	**Description**
get(key)	Get a key
set(key, value)	Set a key/value
remove(key)	Remove a key
enumerate()	Return all keys
clear()	Clear partition

Cross-Container Communication	
Name	**Description**
send(userId, msg, msgType)	Send a message
registerHandler(handler)	Handles incoming messages
addPeer(userId)	Add a peer
removePeer(userId)	Remove a peer

Figure 3.4: Radiatus APIs for interacting with storage and other containers. The storage system exposes a logically isolated user partition.

Distributed Capabilities

Radiatus uses distributed capabilities to encode access control in the existing communication patterns of the application. For example, when Alice notifies Bob about a new photo, Alice can directly pass the capability that gives Bob access to the photo. The capability is a cryptographic hash of the content plus a random nonce $H(data, nonce)$, which acts as a self-certifying name proving read-only access to immutable data [112].

When a user stores a value, $set(k, v)$, the user container adds a random nonce, n to the value and computes the hash, $H(v, n)$. The container then sends a request to the storage guard to persist the ownership metadata, $(user, k) \rightarrow H(v, n)$, the capability, as well as the content, $H(v, n) \rightarrow v$.

Capabilities are then self-certifying and transferable. Containers can send the capability, $H(v, n)$, to other containers, which can use it to retrieve the data from the database. The storage guard will only return content if the capability has been registered by the owner. Radiatus also uses Memcached to cache metadata, such as which keys a user owns, to accelerate data fetches.

This mechanism helps satisfy our original goals of scalable isolation with minimal overhead, with these properties:

- Because the capability provides proof of access, any storage guard can independently verify a capability, allowing the database and application to scale independently.

- Regardless of the number of users that persist the same content, or share the content with their friends, the database only needs to store one copy of every unique data value, deduplicated by content hash.

- Transferring capabilities is cheap, regardless of the content size or number of users with access.

- If a capability is granted to a malicious user, they cannot destroy the owner's data.

To support revocation, e.g., to remove an ill-advised tweet, the owner's container can delete both the capability and the data value from the store. This invalidates any outstanding capabilities to the data and prevents future retrieval. To revoke a capability from a specific user, e.g., on a change to a friend list, the owner's container picks a new nonce m, computes the new capability

$H(v, m)$, installs it, distributes the capability to the new set of friends, and then deletes the old capability mapping. Of course, a corrupted friend's account could have already retrieved and stored a copy of the data or leaked it to the tabloids. Revocation only prevents later access.

We have implemented storage guards for MongoDB and Memcached, which have been sufficient for the applications that we have built to date. Other NoSQL and key-value systems can be supported similarly. We next describe how capabilities would interact with other types of storage systems; these are not part of our current implementation.

Object-Relational Mapping (ORM) Object-relational mapping (ORM) [52] is a common programming model that allows developers to persist objects in relational databases. For example, it is natural to write an object-oriented program where an instance of an AddressBook class stores an array of Record instances. ORM libraries provide synchronization primitives to convert these objects into representations which are compatible with a relational database. ORM is the default programming model for many popular web frameworks including Django, Ruby on Rails, and PHP. In this case, the Radiatus storage guard functions identically as when in front of an SQL database, described next. Objects are serialized and hashed before persisted to the database.

Relational Databases We want to give a user container access for any table (or object-relational) data for which the user holds the matching capability. To do this, we configure every table in the database with two extra columns to store the owner of the row and a hash of its contents (the capability). On an `INSERT` operation, the storage guard automatically populates the *owner* and *capability* columns. Subsequent requests to `UPDATE` a row are allowed if the user is the owner; this also modifies the hash value, ensuring that each capability is valid only for a particular data snapshot.

For queries, the user container sends the storage guard a list of its capabilities; these lists can be cached for efficiency. The results of simple `SELECT` queries can be post-processed to ensure only rows that the user has permission to access are returned, with the owner and hash value stripped off. More complex queries involving `JOIN` need to be prepended with a `SELECT` operation to check and strip off the capability.

Content Distribution Networks (CDN) Many modern CDNs provide a programming interface for adding and removing content from the network. As such, we can create a storage guard that uses similar techniques. We treat the CDN as a blob store, which stores a single copy of every published piece of content. A NoSQL database is used to store user ownership metadata. Capabilities can then be embedded in a unique URL to be linked from HTML pages.

3.1.8 Internal Services

To simplify development and improve modularity, a number of web applications are designed with internal structure: a stateless frontend web server that coordinates calls to a mixture of backend application, caching, and storage services. For example, an e-commerce site may operate different internal web services for their shopping cart, user recommendations, personalized search, notifications, and customer support. As with frontend servers, these internal services often mix application logic with access control and privacy enforcement, raising the vulnerability of user data to compromise.

With Radiatus, internal services can exist within containers, with user credentials transparently propagating from the frontend to the backend services. User containers are not always appropriate or necessary. Some backend services are general purpose, such as a distributed configuration or lock manager. Others require access to site-wide data, such as computing trending topics in Twitter. By sandboxing frontend code, Radiatus provides defense in depth protection for these internal services. More fundamentally, these services need to be treated as part of the trusted computing base: carefully designed with a narrow interface that is hardened against attack.

In a few cases, internal services can be treated by the rest of the system as an untrusted user. An example is a service to aggregate content for a public newsfeed. For this, Radiatus supports the notion of a *service user* for shared computation. A service user encompasses a unit of aggregate computation on behalf of the service. Service users are addressable like normal users, but their containers run code on behalf of the service. Since service users communicate with normal user containers, the service developer must apply defensive programming with the assumption that user containers can be compromised, and vice versa. For example, one could apply differential privacy libraries for privacy-preserving data collection.

3.2 Implementation

3.2.1 Radiatus Framework

We have implemented the Radiatus web framework as a collection of various software components. A *container runtime* hosts a number of user containers on a server, each isolated in a unique sandbox. A *user router* routes incoming requests to user containers. A *storage guard* mediates calls to the storage systems by checking capabilities and translates the request to the database-specific interface. Lastly, *cross-container messaging* is supported by a message queuing system and a distributed container manager.

We have implemented Radiatus as a web framework in 8764 lines of code on the Node.js runtime [27], where each user is allocated a Docker container [12] running a separate Node.js process. We inject stubs for each of the Radiatus APIs and block any other interfaces normally provided by Node.js. The Storage Guard interposes on user storage requests to expose a partitioned NoSQL database, but internally uses MongoDB and memcached.

While our implementation uses Docker, the Radiatus design is compatible with other virtualization technologies. Depending on the operating environment, performance, and security requirements, developers can choose a virtualization technique that works for them, from OS-level virtualization [12, 28, 24, 25] to full virtual machines [51, 23]

Our message-passing system fits the growing use of event-driven programming for web development, similar to channels in Go [18], event emitters in Node.js [27], and Scala's actor model [1]. As with these systems, event-driven programming in Radiatus comes with a cost: added complexity in managing long chains of actions. We describe developer experiences more fully next.

3.2.2 Applications

In order to explore the expressiveness of our Radiatus framework, we used it to build a number of collaborative applications. Radiatus fits well with the wide range of web applications that involve interacting users, including productivity software, games, social networking, e-commerce, and media. Because Radiatus is a server-side web framework, developers are unrestricted in how they design client-side user interfaces. Figure 3.5 shows the number of lines of code for each

Application	**Blizi**	**FileDrop**	**Chat**
Total LOC	2958	614	285
Server-side LOC	870	219	133
User Interface LOC	2088	395	152

Figure 3.5: Lines of code to implement each application.

application we developed using Radiatus.

Academic Social Network: Blizi is an academic social network that allows authors to post papers and solicit reviews from other users. The application also allows an author to privately share paper drafts and reviews with certain individuals. The intent is to allow limited dissemination without violating anonymous conference reviewing, as might occur when papers are posted to Facebook or the Web. We have started to organize one of our seminars around this tool.

File Sharing: FileDrop allows a user to upload files to their user container. When a friend is granted access to a file, the friend's container can retrieve it from the storage service using the file's capability. The application can then serve the file to the friend's browser.

Chat Messaging: The chat application uses the cross-container messaging system to relay chat messages between people. In this particular example, we wrote a custom authentication manager that automatically assigns everyone a pseudonym and registers them on a global buddy list.

3.2.3 Porting Existing Applications

We provide a simple tool for bundling existing Node.js libraries in Radiatus user containers. However, not all applications can be easily ported, such as those that use direct filesystem access. While individual components of an existing Node.js web application can be ported using the same tool, any application logic that requires global access to state must be rewritten to exist within a restricted user container.

Arc Forum: Because the Radiatus container manager works with operating systems processes, we can port applications written in other languages, subject to the same limitations above. We ported the Arc Language Forum [123], the application behind the popular Hacker News web application, to the user container model. The forum is written in Arc, a dialect of the Lisp programming language that includes a built-in web server and libraries for generating HTML. The forum application provides a social news web application using these language primitives. Because data is persisted to files, rather than a database, the port required no changes to Arc Forum, and 188 lines of changes to our user router and container runtime.

3.3 Evaluation

Our evaluation asks the following questions to understand the security and performance of Radiatus. How do user containers prevent existing classes of attacks (§ 3.3.1)? Does the Radiatus implementation provide acceptable performance given the added overhead of user containers (§ 3.3.2)? What is the incremental cost per user (§ 3.3.3)?

3.3.1 Security Analysis

Radiatus is designed to reduce vulnerability of user data to exploits that take advantage of bugs in web server application code. To evaluate this, we analyzed all of the vulnerabilities in the National Vulnerability Database (NVD) with a maximum severity score of 10.0 from 2014. The Common Vulnerability Scoring System (CVSS) is an open industry standard, which reserves the 10.0 score for the most severe vulnerabilities that fit 6 criteria: (1) remotely exploitable, (2) low barrier to access, (3) requires no authentication, (4) total information disclosure, (5) complete loss of system integrity, and (6) leads to total loss of availability of the attacked resource. Out of all 7316 software vulnerabilities reported in 2014, only 233 received this score. Of these 233, we analyzed the 40 that involved server-side web software. Many web applications are proprietary software running on managed infrastructure, and bugs in that software are likely to be under-reported. As a consequence, we expect Radiatus to help more cases than those reported in this section.

For each reported bug, we attempted to understand how the vulnerability affects the web application if the software with the vulnerability was translated into the Radiatus model. Figure 3.6 lists each vulnerability, its original impact, and its impact in Radiatus.

CVE ID	Short Description	Original Impact	Impact in Radiatus
	Code Injection and Buffer Overflow		
CVE-2014-0294$^{\spadesuit}$	MS Forefront 2010 improper email parsing	Arbitrary	Sandbox
CVE-2014-0474$^{\spadesuit}$	Django improper type conversion	Data leak	Sandbox
CVE-2014-0650$^{\spadesuit}$	Cisco Secure ACS allows arbitrary shell commands	Arbitrary	Sandbox
CVE-2014-0787$^{\bigstar}$	WellinTech KingSCADA buffer overflow	Arbitrary	Sandbox
CVE-2014-2866$^{\clubsuit}$	PaperThin CommonSpot uses client-side JavaScript for access restrictions	Arbitrary	Sandbox
CVE-2014-3496$^{\blacklozenge}$	OpenShift Origin executes arbitrary shell commands in URL	Arbitrary	Sandbox
CVE-2014-3791$^{\bigstar}$	Easy File Sharing Web Server buffer overflow in cookie parsing of vfolder.php	Arbitrary	Sandbox
CVE-2014-3804 (+5)$^{\blacklozenge}$	AlienVault OSSIM executes arbitrary commands with crafted requests	Arbitrary	Sandbox
CVE-2014-3829$^{\blacklozenge}$	Centreon Enterprise Server executes arbitrary commands from command_line variable	Arbitrary	Sandbox
CVE-2014-3913$^{\bigstar}$	Ericom AccessNow Server buffer overflow in AccessServer32.exe	Arbitrary	Sandbox
CVE-2014-3915$^{\blacklozenge}$	Tivoli Storage Manager executes arbitrary commands	Arbitrary	Sandbox
CVE-2014-4121$^{\spadesuit}$	Microsoft .NET improperly parses internationalized resource identifiers	Arbitrary	Sandbox
CVE-2014-6321$^{\bigstar}$	Schannel in Microsoft Windows Server executes arbitrary code via crafted packets	Arbitrary	Sandbox
CVE-2014-7192$^{\clubsuit}$	Node.js eval injection in syntax-error package	Arbitrary	Sandbox
CVE-2014-7205$^{\clubsuit}$	Node.js eval injection in internals.batch() of lib/batch.js	Arbitrary	Sandbox
CVE-2014-7235$^{\spadesuit}$	FreePBX executes arbitrary code via ari_auth cookie in htdocs_ari/includes/login.php	Arbitrary	Sandbox
CVE-2014-7249$^{\bigstar}$	Allied Telesis buffer overflow via crafted HTTP POST request	Arbitrary	Sandbox
CVE-2014-8361$^{\spadesuit}$	miniigd SOAP service executes arbitrary code via crafted NewInternalClient request	Arbitrary	Sandbox
CVE-2014-8661 (+1)$^{\blacktriangle}$	SAP CRM executes arbitrary commands via unspecified vectors	Arbitrary	Unclear
CVE-2014-9190$^{\bigstar}$	Schneider Electric Wonderware InTouch Access Anywhere Server buffer overflow	Arbitrary	Sandbox
CVE-2014-9371$^{\spadesuit}$	ManageEngine Desktop Central MSP executes arbitrary code via crafted JSON object	Arbitrary	Sandbox
	Path Traversal		
CVE-2014-0598$^{\sharp}$	Novell Open Enterprise Server allows directory traversal	Unclear	Prevented
CVE-2014-0754$^{\blacksquare}$	SchneiderWEB allows directory traversal	Data leak	Prevented
CVE-2014-2863 (+1)$^{\sharp}$	PaperThin CommonSpot allows absolute path traversal	Unclear	Prevented
CVE-2014-3914$^{\flat}$	Tivoli Storage Manager allows arbitrary filesystem access	Arbitrary	Sandbox
CVE-2014-7985$^{\flat}$	EspoCRM allows remote include/execute via install/index.php	Arbitrary	Sandbox
CVE-2014-9373$^{\flat}$	ManageEngine NetFlow Analyzer executes arbitrary code via .. in the filename	Arbitrary	Sandbox
	Improper Authentication		
CVE-2014-0648$^{\diamondsuit}$	Cisco Secure ACS improperly enforces admin access	Arbitrary	Auth
CVE-2014-2075§	TIBCO Enterprise Administrator improperly enforces admin access	Arbitrary	Auth
CVE-2014-2609$^{\diamondsuit}$	Java Glassfish Admin Console in HP Executive Scorecard doesn't check authentication	Arbitrary	Auth
CVE-2014-8329$^{\heartsuit}$	Schrack Technik microControl stores sensitive information publicly in ZTPUsrDtls.txt	Arbitrary	Auth
	SQL Injection		
CVE-2014-3828$^{\infty}$	Centreon Enterprise Server SQL injection	Data leak	Sandbox
CVE-2014-5503$^{\infty}$	Sophos CyberoamOS SQL injection in Guest Login Portal	Data leak	Sandbox

Figure 3.6: Security analysis of all 40 web-related vulnerabilities in the National Vulnerability Database from 2014 with the highest severity score. In most cases, an attacker would be able to arbitrarily affect the service or access data. In Radiatus, we either prevent these attacks entirely, restrict compromise within the user container sandboxes, or are mitigated by the delegation of authentication logic to Radiatus, which can be shared and independently audited.

- Arbitrary: exploitation leads to arbitrary code execution, service disruption, and information disclosure;
- Data leak: bug can be leveraged to leak arbitrary information, but not run code;
- Prevented: bug cannot manifest in the target system;
- Sandbox: exploitation is limited to a single user's sandboxed container;
- Auth: app developers delegate the responsibility of implementing authentication to Radiatus;
- Unclear: bug report did not specify enough details to make a determination.

In most cases, the original impact allowed an attacker to arbitrarily affect the service or access data. In Radiatus, we either prevent the attack entirely, restrict compromise to the user containers for which the attacker has user credentials, or require applications to delegate authentication logic to Radiatus, which can be shared and independently audited.

Code Injection

The majority of high severity vulnerabilities involved code injection, which allows an attacker to execute arbitrary code with the privileges of the server process and affect the state of any user. In Radiatus, packets are only routed to a user container if the attacker has proper credentials, limiting the scope of the attack.

- 6 ($^\bigstar$) consist of stack-based buffer overflow vulnerabilities, which can be remotely exploited by sending crafted network requests.
- 7 ($^\spadesuit$) allow arbitrary code execution due to improper sanitization of inputs to components, such as JSON parsers.
- 3 ($^\clubsuit$) involve code injection vulnerabilities in JavaScript (e.g. calling *eval(. . .)* in Node.js).
- 9 ($^\blacklozenge$) are vulnerabilities that allow a remote attacker to run shell commands with the privileges of the server process.
- 2 ($^\blacktriangle$) involve code injection via unspecified vectors.

Path Traversal

Similar in nature to shellcode injection, 7 vulnerabilities involved path traversal bugs (e.g. by adding ../ in the requested resource). When exploited these bugs can allow an attacker to read, write, and execute files from the filesystem with the privileges of the server process. Radiatus confines all application code in per-user sandboxes.

- 1 ($^{\blacksquare}$) allows remote attackers to read arbitrary files via crafted HTTP packets.
- 3 ($^{\flat}$) consist of variations where an attacker can execute arbitrary commands by using '../' to navigate to shell commands in input parameters.
- 3 ($^{\sharp}$) are known path traversal vulnerabilities with unknown impact.

Improper Authentication

4 vulnerabilities involved improper authentication checks. Most web frameworks require developers to write custom authorization logic. Flaws in this logic can easily lead to privilege escalation. In Radiatus, developers delegate authentication checks and enforcement to Radiatus, which can be independently audited and shared across all applications.

- 1 ($^{\heartsuit}$) stored sensitive credentials in publicly accessible resources, which attackers can use to obtain privileged access.
- 2 ($^{\diamondsuit}$) do not require authentication to access a privileged interface.
- 1 (§) does not require authentication to issue arbitrary commands as an administrator.

SQL Injection

2 vulnerabilities ($^{\infty}$) involve SQL injection, allowing an attacker to exfiltrate data. With Radiatus, the storage guard filters JavaScript from all commands going into the database and similarly filters all outgoing data for values not belonging to the authenticated user. SQL injection has become less common with the prevalence of parameterized client libraries. The number of SQL injection vulnerabilities in the National Vulnerability Database has fallen from its peak of 1476 in 2008 to 261 in 2014.

Other Vulnerabilities

Cross-site scripting: The most common web vulnerability is cross-site scripting. No cross-site scripting vulnerability in 2014 had a severity score exceeding 8.0. Radiatus is not aimed at these vulnerabilities, but our implementation addresses them by using industry-standard CSP Policies [9].

Cross-container communication: While Radiatus makes it harder for wide-scale compromise of an application, it is still possible to use cross-container messaging as a new attack vector if the attacker can gain access to it (e.g. by exploiting a code injection vulnerability in the user container). We restrict which containers can communicate, enforce typed interfaces, detect anomalous behavior, restrict resource consumption, and evict bad actors from the system. Browsers [19, 21], operating systems [133], and cloud providers [3] use similar techniques to protect communications between mutually distrusting isolated processes.

3.3.2 Performance

Radiatus is designed to achieve better security at modest overhead. We evaluated the performance of user containers on Amazon Web Services using r3.large EC2 instances (2 CPU, 15GB memory, 32GB SSD, $0.175/hr in 2015). We stress test the performance of a single web server and the overall throughput of a web service over 500 servers.

Memory Overhead: In Radiatus, each user container is a sandboxed process, running a separate copy of Node.js, the Radiatus runtime library, and the web application. When scaling the number of active users on a single machine, memory quickly becomes a bottleneck in the context of a single server. We measured the memory consumption across 100 user containers running our benchmark suite and found the average container to consume 30.5MB. As such, each memory-optimized r3.large EC2 instance was able to support around 490 processes before swapping.

Throughput Microbenchmark: Figure 3.7 shows the serving performance of a number of web frameworks for generating simple dynamic web pages. The serving performance data was collected using the Siege load testing tool, which simulates 100 users making HTTP requests in parallel. The page response was an HTML page displaying a simple counter of how many replies had been served so far. This experiment disables caching while stress testing the HTTP request handler.

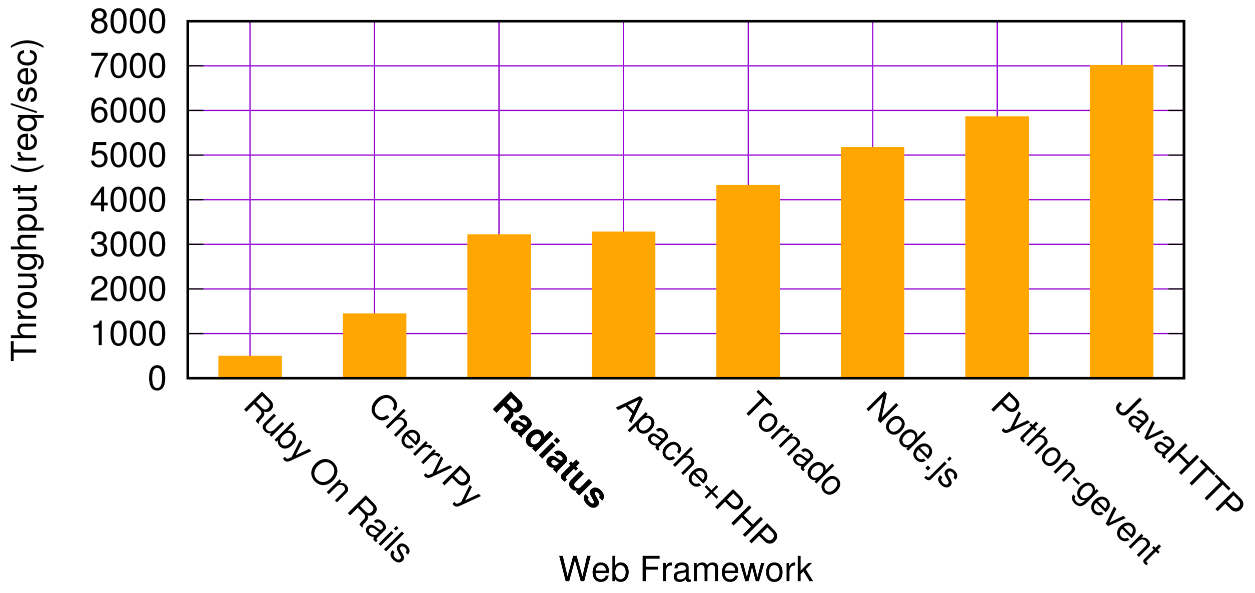

Figure 3.7: Comparison of single web server performance using the Siege Benchmark to make 1000 parallel connections at a time. Radiatus remains competitive with other frameworks despite handling requests in isolated user containers.

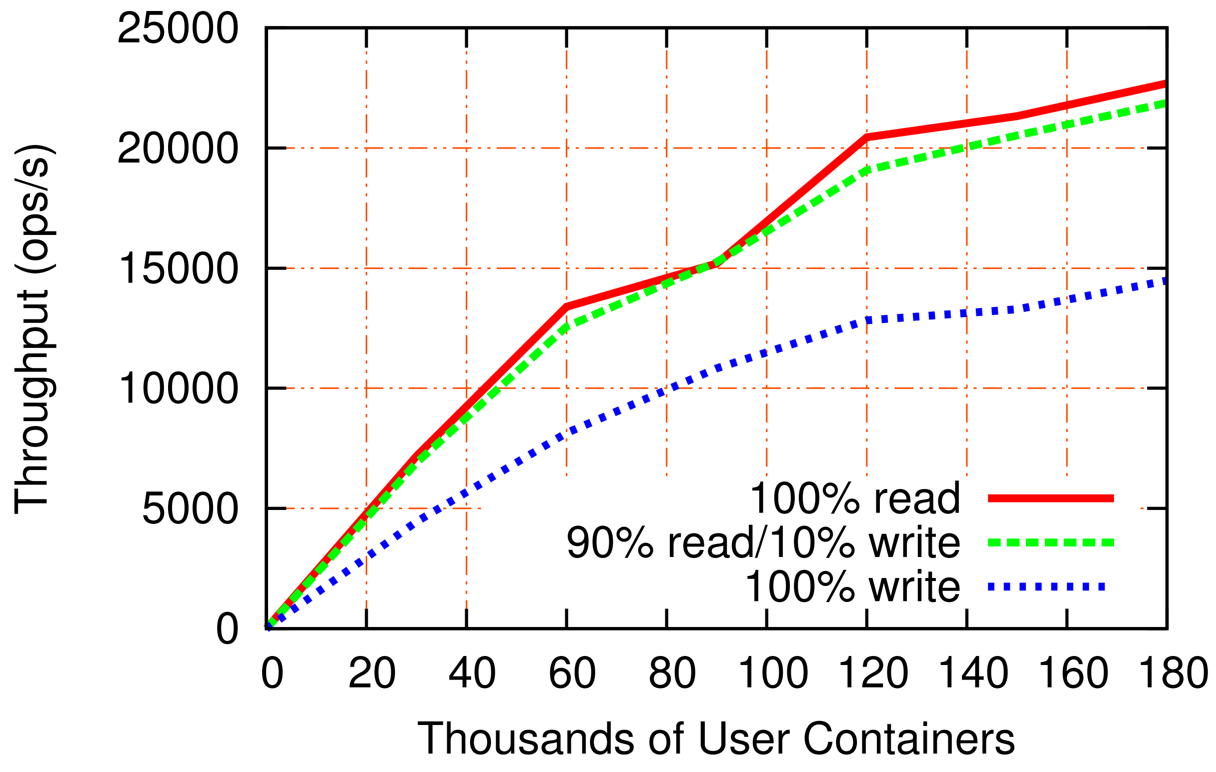

Figure 3.8: Aggregate throughput with different workloads across a 500-node cluster. We scale the number of user containers across our cluster, sending messages to each other via AWS Simple Queuing Service.

While Radiatus performs additional routing to send requests to the proper container, our system performs comparably to existing frameworks and better than some popular frameworks, such as Ruby on Rails. Because the Radiatus router was written in Node.js, the microbenchmark shows a maximum overhead of 60.7% over our baseline. We expect the relative overhead to be less in a real web application, but we did not directly test that effect.

Macrobenchmarks: In order to stress test the system at scale and evaluate the performance of the cross-container messaging system, we set up a cluster of 500 virtual machines (VMs) supporting 180,000 user containers, communicating through the AWS Simple Queuing Service. As a point of comparison, Wikipedia in 2010 had 205 Apache web servers to support 414M readers and 100K active editors per month, with 2000 HTTP requests per second [40]. In our benchmark, we stress tested our file-sharing application under varying workloads. A read request consisted of a request to the storage guard and a response containing an item from the user's personal storage. A write request consisted of sending a message to a peer container through the message router. Each user container performs either a read or write request every 4 to 6 seconds. Figure 3.8 shows the aggregate throughput of the system with various workloads. In Radiatus, the global performance can be bottlenecked by both the database and the message queue. Developers will need to properly balance these resources to fit the application workload.

3.3.3 Cost Estimation

The largest incremental cost of scaling a web service in Radiatus is the memory overhead of running user containers for each active user. As reported in § 3.3.2, a single user container consumes 30.5MB of memory when running our benchmark suite. Using average DRAM prices in 2015 [7] and an average server life of 3 years, we can estimate the incremental memory cost of an active user to be \$0.007/year. To include the additional cost of CPU overhead, power, and space, we use current EC2 pricing (2 cores @\$0.175/hour). Assuming 1000 requests/day for each user and using the differential CPU cost of handling requests in Radiatus versus Node.js from Figure 3.7, brings the total cost to \$0.008/user/year.

In order to estimate the potential cost of scaling such a system up to traffic levels seen by some of the biggest web apps today, we can use Little's Law and publicly reported numbers from the Facebook Newsroom [15]. On average, Facebook supports 26 million concurrent users on the site, for an estimated additional annual cost of \$200K. Note that this cost will multiply in a service-oriented architecture, depending on the number of internal services using Radiatus. Compared to the average cost of a data breach recovery in the U.S. (\$5.4M [190]), Radiatus may be appropriate for security-conscious web applications with sensitive high-value data.

3.4 Summary

Modern trends in OS-level containers, cost of memory, and elastic cloud computing make it an opportune time to revisit per-user isolation and study the costs at scale. Radiatus provides an alternative model for web application design offering increased security over existing frameworks. User containers are a lightweight mechanism to strongly isolate users within a web application. We show that it is practical to provide per-user isolation, while offering performance competitive with existing web frameworks, at modest cost per user. While application design with Radiatus is different from traditional frameworks, we show that our APIs are expressive enough to support many of the web applications today.

The web platform already treats the browser as a per-user isolated container running potentially untrusted code. Leveraging this design pattern on the server provides a structured approach to isolation, offering the same containment we expect from our own machines, mobile applications, and multi-tenant data centers.

Chapter 4

AN EFFICIENT AND SCALABLE OBLIVIOUS MESSAGING SERVICE

While the techniques in Chapter 3 limit the damage when cloud applications are remotely exploited, it does not address insider attacks as a means to violate user privacy. Even expert employees can be tricked into disclosing login credentials in phishing attacks, giving the attacker internal privileges. Malware can be inadvertently installed on corporate networks when employees download malicious files. Governments around the world are increasingly compelling hosting companies to disclose user data. To address these attacks on user privacy, we must treat the cloud as an untrusted insider threat. In this chapter, we focus on addressing insider threats for a single application, group messaging.

Messaging applications depend on cloud servers to send data between users, giving server operators full insight into the communication patterns of the application's users. Even if the communication contents are encrypted, network metadata, such as HTTP headers, can be used to infer which users share messages, when traffic is sent, where data is sent, and how much is transferred, allowing the network and provider to guess the contents of the communication [136]. When remote hacking, insider threats, and government requests are common, protecting the privacy of communications requires that we guarantee security against a stronger threat model. For some users, such as journalists and activists, protecting communication patterns is critical to their job function and safety [165, 166].

In this chapter, we present Talek, a private group messaging system. Talek provides an efficient single-writer several-reader log abstraction, storing asynchronous messages on untrusted servers without revealing metadata. Users create logs for each message thread, which small groups of trusted friends can read at a later time. As long as clients within a trust group and at least one server are uncompromised and running authentic versions of the software, Talek prevents a cloud operator from learning anything about the communication patterns of the users. Combined with encryption, we conceal both the *contents* and *metadata* of users' application usage without losing the reliability and availability of the cloud.

Recent research has advanced both one-to-one private messaging [41, 43, 210, 232] and anonymous broadcasting [84, 85, 86, 149, 245]. These systems offer security guarantees rooted in k-anonymity [223], plausible deniability [130] or differential privacy [101, 102]. Talek focuses on a stronger security goal based on *access sequence indistinguishability*, where two arbitrary-length client access sequences are indistinguishable to the server, and thus the server learns no information about which users may be communicating. Existing systems guaranteeing indistinguishability are either impractical due to prohibitive network costs [48, 119, 121, 181], or are custom-tailored for specific applications [63, 126], limiting their applicability.

Talek provides a practical design for a private group messaging system with strong security goals based on indistinguishability of access patterns. It is designed to be network bandwidth efficient — usable with mobile clients reading and writing asynchronously to many message threads, each modeled by a log.

Talek is based on private information retrieval (PIR) [78, 95, 118], but PIR by itself is not enough to support a private group messaging system. We combine GPU-based performance improvements with two novel techniques:

- *Oblivious logging* describes a new way to construct a real-time message broker that can deliver messages with provable unlinkability between users.

- *Private notifications* allow users to determine which logs have new messages without polling or revealing anything about their usage.

With oblivious logging, all clients issue identically sized random-looking read and write requests to servers at an independent rate. Within a group of clients reading and writing to a message log, a shared log secret determines the pseudorandom and deterministic sequence of locations for messages. The writer places new messages in locations that appear random to the adversary. Any user with the log secret can follow the pseudorandom sequence, reading new messages without coordination with other users. Users are granted access to logs by receiving a secret from the log's owner. These secrets are shared using an in-band mechanism called *control logs.* Although Talek only supports single-writer several-reader logs, we can emulate several-to-several communication by creating a log for each writer. Users append to their own log and each member of the group subscribes to all logs of the group.

Talek relies on private information retrieval (PIR) to read the message stored at a location without disclosing to the server which location is being read. We apply updates and reads consistently across PIR servers using timestamp ordering [57]. To support message asynchrony, servers store a window of the latest n messages, purging older messages. Choosing a larger value for n means data is stored on the database for longer, at the cost of more expensive reads. Messages are stored in a cuckoo hash table to achieve efficient time and space usage without disclosing information to the adversary.

With private notifications, users periodically retrieve a *global interest vector*, which privately and efficiently encodes the set of all logs with new messages. Users apply the global interest vector to locally prioritize reads. Servers maintain the global interest vector without leaking any information about its contents.

In our system, the developer chooses l independent servers to host replicas of the data in an anytrust model. The security model assumes at least one of the servers is honest. Our guarantees hold for arbitrary behavior by the other servers, who may collude, share secrets, and send faulty responses to clients. An adversary could control the network and $l-1$ servers without impacting the security of the system.

Talek does not guarantee liveness; a single faulty server can deny all use of the system in a way that is detectable to all clients. Service providers are chosen with reputations for high availability. Because clients connect directly to Talek servers, we also do not hide when users are online. We expect the system to be used for communication among small groups of trusted users. If a log secret is shared with the adversary, writer anonymity for that log is compromised, but readers' anonymity and writer anonymity for other logs are preserved. Talek is best suited for applications where users communicate with groups of trusted friends; any user in the group can block writes to the log. Applications that require wide broadcasts to many untrusted users (e.g. a public blog), are better served by anonymous broadcast [84, 85, 86, 245].

We have implemented Talek in Go and evaluated the system on a 3-server deployment using Amazon EC2. Our source code is public. Our evaluation shows that for a messaging workload where users send and receive 1KB messages every 5 seconds, we can support 32,000 concurrent users sustaining a total throughput of 566,000 messages per minute with an average end-to-end latency of 5.57 seconds. Like other PIR-based systems, PIR reads are the primary computational bottleneck. Because clients send read and write requests at a regular rate, the amount of requests the servers handle increases linearly with the number of clients. At the same time, the size of the data stored in the server increases linearly with the number of clients. Finally, performing PIR requires work linear with the size of the data. Consequently, the amount of work servers perform grows with the square of the number of clients, or alternatively, the performance for clients drops with the square of the number of clients. In all, we show that we can achieve 3–4 orders of magnitude better performance than comparable systems with the same security goals. We also show that Talek is practical for mobile applications, as clients only send requests when the device screen is on, amounting to 9.2MB/day for our workloads. Further, our design is compatible with horizontal scale-out to support higher message rates and/or more users, although this is left for future work as discussed in Chapter 5.

This chapter highlights the following contributions:

- Oblivious logging is a new approach to achieving indistinguishability of access patterns, by efficiently storing logs of messages on the server in a way that looks random to an adversary. (Sections 4.3 and 4.4)

- Private notifications privately encode the set of new messages, helping clients prioritize reads. (Section 4.5)

- Implementation and evaluation of Talek, which applies these two techniques in an end-to-end messaging system with practical performance. (Sections 4.6 and 4.7)

4.1 Background

4.1.1 Threat Model

Figure 4.1 illustrates a system with mutually distrusting clients located across a wide-area network, sharing data through Talek services, each hosted in a unique data center. We use the term 'server' to

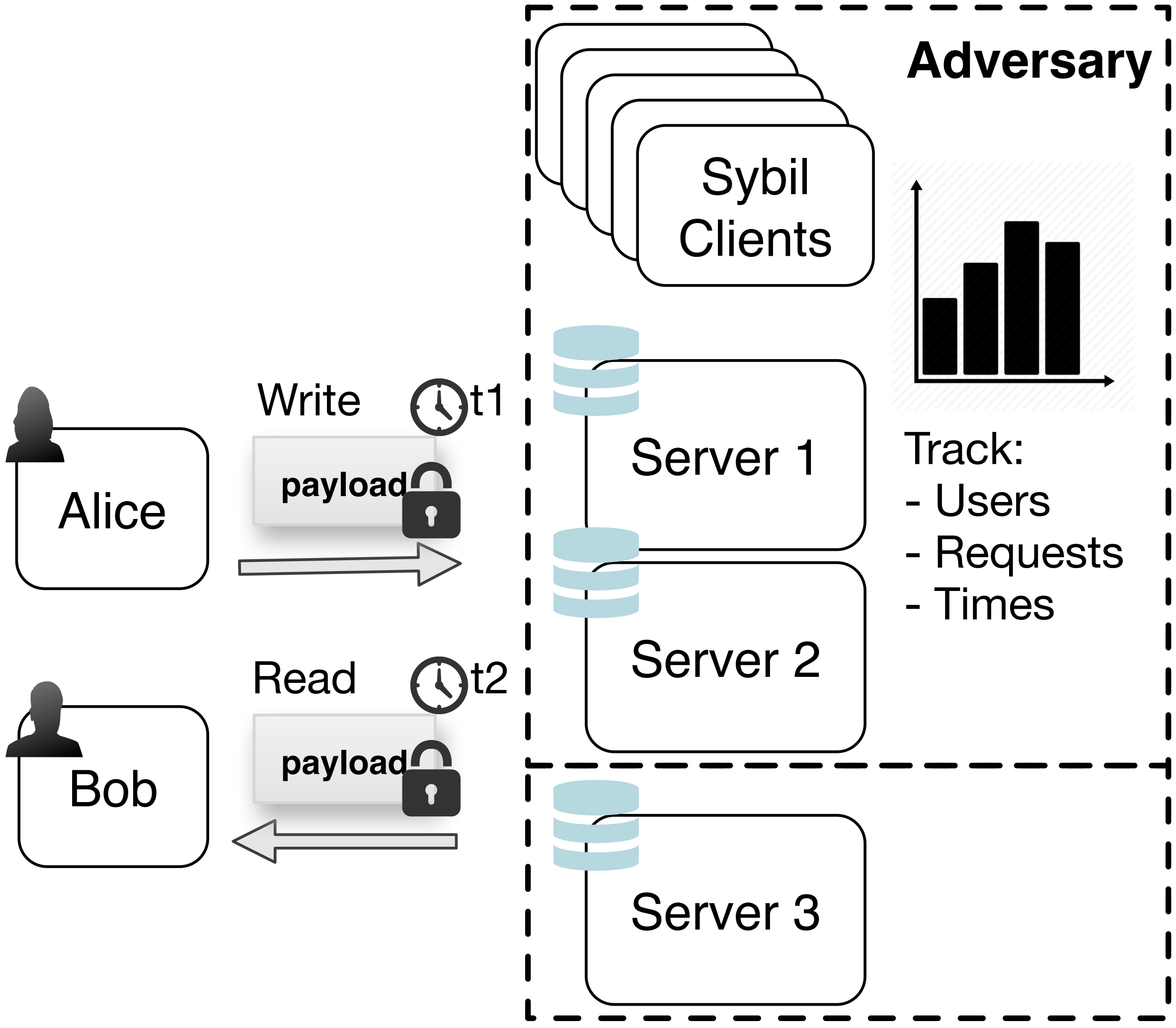

Figure 4.1: System and threat model in Talek. We assume the adversary can control all but one of l servers in the system ($l = 3$ in figure). Clients send network requests directly to the servers. Adversarial servers are free to record additional data, such as the source, type, parameters, timing, and size of all requests to link users who are likely to be communicating together. All servers must be available and reachable by all clients.

refer to a single Talek service controlled by an independent administrative domain.[1] The adversary's goal is to build a statistical model of users who are likely to be communicating.

Talek assumes the adversary controls all but one of a set of servers. Clients do not know which server is honest. The adversary can also control the network and generate an unbounded number of clients. We assume message storage capacity is scaled to the number of clients. We assume all servers are collecting information about all client network requests, such as the source, operation type, parameters, timing, and size of requests. The Talek protocol ensures correctness and unlinkability, even when adversarial servers and clients exhibit arbitrarily malicious behavior, such as if they collude, share secrets, and send faulty responses to clients. Our security guarantees must hold even as clients are observed over long periods of time, such as in an intersection attack.

While session keys are exchanged in-band, we assume communicating clients already know each other's long-term public keys. Talek is compatible with bootstrapping keys from existing applications [42], using identity-based encryption [62, 61, 150], or through an out-of-band channel.

Talek is designed for groups of mutually trusting users. We assume that users communicating together trust each other not to disclose shared secrets. Generally, clients colluding with servers cannot expose arbitrary users. However, malicious users can collude with any server to expose only the writer of a log for which it has the log secret.

During normal operation, all servers must be available and reachable by all clients. Any single server can deny all use of the entire system by refusing to respond or by responding with faulty information. However, this behavior is detectable to the developer and clients. We assume developers will choose services with high reputations for availability. While we do not discuss it in this thesis, Byzantine fault tolerant variations of private information retrieval [95, 118] can be used for better liveness guarantees at the cost of higher overhead. Adversarial clients can degrade service in denial of service attacks.

While not described in this thesis, other end-to-end guarantees, such as message integrity, authentication [55, 203], forward secrecy [33], and fork consistency [154, 162], can be layered on top of Talek by including additional data in the message payload. There exists a wide body of work in secure messaging, which is largely compatible with this work [227]. Talek focuses on privacy of

[1]Our design allows each independent server to be implemented across multiple machines for scalable performance and fault tolerance, but that is beyond the scope of this thesis.

access sequences.

We assume the existence of secure encryption, key-exchange protocols, signatures, hash functions, and random number generators. We also assume that server public keys are known to all users. These issues are orthogonal to the properties Talek is designed to provide.

4.1.2 Security Goals

We define access sequence indistinguishability using the following security game, played between the adversary, $\mathscr{A}$, and a challenger, $\mathscr{C}$. $\mathscr{A}$ is a probabilistic, polynomial-time adaptive adversary, who is in control of the network, all but one of the servers, and an unbounded number of clients. $\mathscr{A}$ can drop any message, send arbitrary messages from any of the adversarial clients to any server, respond arbitrarily to requests, and modify any server-side state for adversarial servers. Assume the presence of authenticated secure channels between each client-server pair (e.g. with TLS).

1. $\mathscr{A}$ chooses a non-negative integer, m, and submits this number to the challenger, who spawns m clients, $\mathscr{C}_0 \ldots \mathscr{C}_{m-1}$

2. The challenger flips a coin, $b \in \{0, 1\}$, uniformly at random, which is fixed for the duration of the game.

3. For each of the challenger's clients, $\mathscr{C}_j$, $\mathscr{A}$ maintains two unique data access sequences, seq_j^0 and seq_j^1.

4. Repeat the following until $\mathscr{A}$ chooses to end the game:

   - $\mathscr{A}$ chooses the i-th operation for both sequences for all challenger clients, $\{seq_0^0[i] \ldots seq_{m-1}^0[i]\}$ and $\{seq_0^1[i], \ldots seq_{m-1}^1[i]\}$. $\mathscr{A}$ submits the operations $seq_j^0[i]$ and $seq_j^1[i]$ to the respective client, $\mathscr{C}_j$. Chosen operations can be a *Read*, *Write*, or *NoOp* to logs for which the adversary does not hold the secret log handle, τ.

   - Each client, $\mathscr{C}_j$, plays one of the two operations, $seq_j^b[i]$, into the Talek client library.

   - Adversary-controlled clients can send arbitrary requests to any server. Adversary-controlled servers can also modify its own state and respond arbitrarily.

- $\mathscr{A}$ observes the network events, $events_j^{b'}[i]$ sent from $\mathscr{C}$'s clients to adversarial servers. These events include $Write$ and $Read$ network requests.

5. $\mathscr{A}$ outputs its guess for b'.

Definition 1. *(Access Sequence Indistinguishability) We say that the system provides access sequence indistinguishability if for any polynomial-time probabilistic adversary, any challenger clients, and any data access sequences,*

$$|Pr(b = b') - 1/2| \leq negl(\lambda)$$

in the security game, where λ is a security parameter and $negl$ is a negligible function.

Practically, this definition means that an adversary would not be able to distinguish between a real user's access patterns from random access patterns of arbitrary length or an idle user. It follows from this definition that the adversary should also not be able to determine which users access the same logs, because the adversary could have chosen *seq* with overlapping logs across users. These properties must hold regardless of how long a client is observed. We do not hide IP addresses or when a user is online or offline from the system; instead users directly interact with the servers. However, this must not undermine our security goal.

Note that $\mathscr{A}$ only specifies the actions of correct users and does not specify access sequences between correct and adversarial clients. As described in Section 4.4.4, malicious clients with a log secret could collude with an adversarial server to de-anonymize the writer to that log. While this weakens $\mathscr{A}$'s power in the game, it is consistent with our goal of providing privacy guarantees to groups of trusted users. Adversarial clients can still act arbitrarily against any server.

Access sequence indistinguishability provides one of the strongest definitions of privacy available. It is stronger than k-anonymity [223], where the adversary can narrow the user to one of k users. It is also stronger than plausible deniability [130], where information leakage is allowed up to a certain confidence bound.

Informally, we achieve our security goal by designing the system such that

1. The schedule of requests seen by the server is independent from the data access sequence, *seq*. Requests are made by all clients at an independent rate regardless of whether the client actually needed to perform a `Read` or `Write`.

2. All parameters look random from the perspective of any $l-1$ set of servers. Dummy request parameters look indistinguishable from parameters of legitimate requests.

4.1.3 Intersection attacks

Talek's goal of access sequence indistinguishability makes it less susceptible to intersection attacks [91, 141, 163] compared to systems based on k-anonymity. If the application allows the user to have periods of offline usage, then Talek is potentially susceptible to intersection attacks if two users go online and offline at the same time, or if external events (e.g., protests) correlate with the user's online/offline status. As a consequence, Talek is best used with asynchronous user interfaces, such as email.

In contrast, k-anonymity systems are vulnerable to intersection attacks even without these correlations; every network request leaks information. Because mobile devices frequently go offline to conserve energy, k-anonymity systems are weak under mobile workloads. Recent studies show that mobile devices only experience on-screen activity 8.6% of the time [74].

4.1.4 System Goals

In order to be practical for modern workloads, Talek must also satisfy the following goals:

Performant: The system should support large numbers of ephemeral clients over a wide-area network, comparable to the workloads supported by other privacy-preserving systems.

Low Latency: High-priority messages should be delivered in seconds, in order to support messaging.

We answer the following questions in designing our messaging system:

- *Random writes*: How can users write in a way that appears random to the server? (Section 4.3)
- *Consistent Snapshots*: How do we maintain consistent snapshots across servers despite updates, for PIR operations to work over? (Section 4.3)
- *Garbage collection*: How do we constrain database size, keeping PIR operations tractable? (Section 4.3)

- *Zero coordination*: How do readers leverage PIR without coordinating with the writer? (Section 4.4)

- *Notifications*: How do we minimize the need to poll for new data? (Section 4.5)

4.2 Talek Design

Client Overview: Our system achieves our security goal by *requiring all users to behave identically from the perspective of any colluding set of $l - 1$ servers.* Figure 4.2 illustrates how the system is organized; Figure 4.3 enumerates the interfaces and client/server state; Figure 4.4 lists constants that parameterize the design. We designed Talek to be easily integrated into existing messaging applications. Developers link their messaging application to the Talek client library, calling `Publish` and `Subscribe` on the *client developer interface (CDI)*. When a function is called on the CDI, Talek places it on an internal *request queue*, which gets translated into privacy-preserving `Read` and `Write` network requests by the *network protocol interface (NPI)*.

Every user issues equal-sized requests for each operation on the NPI (e.g. `Read` and `Write`) at an independent rate, potentially issuing a dummy request if the respective request queue is empty. The key constraint is that the distribution of network requests from a device is independent from that user's real usage. We describe the system using fixed rates of periodic requests for convenience. In practice, the developer should measure real global usage and sample randomly from this distribution. For example if the messaging user interface encouraged users to compose messages that were independent from each other, then Talek should sample from a Poisson distribution with a known average rate.

A dummy request, including its parameters and payload, must be indistinguishable from a legitimate request. Messages are encrypted with a CCA-secure encryption scheme [56] to provide confidentiality and authenticity. Thus, only the access pattern and not the contents of communication is disclosed when all servers collude. We define a globally-fixed message size, z, to which messages are split and padded to fit. In practice, an application might run two parallel instances of the Talek protocol, one for text-based data, and one with higher latency for images. Because we expect Talek to be used with mobile and web applications, we can take advantage of pre-existing data types specified in the application to facilitate such categorization.

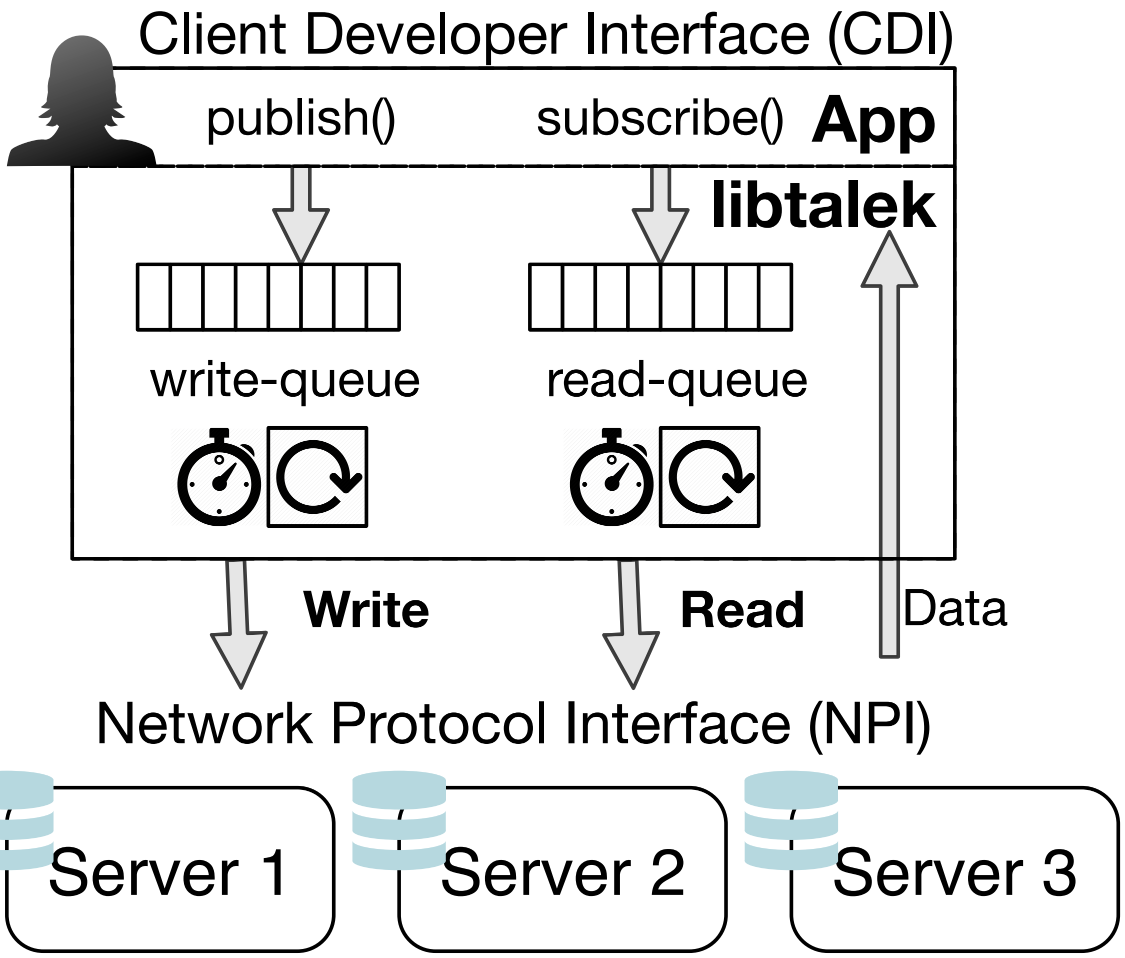

Figure 4.2: Overview of the Talek architecture. All clients must behave identically from the perspective of any $l - 1$ servers. Any calls by the messaging application to publish or subscribe are internally queued by the client library, which is then translated into a privacy-preserving network request. The client library independently issues requests with equal-sized parameters and messages that appear random to the adversary.

Client Developer Interface (CDI)

`Publish`(log, message)

`Subscribe`(log)

Network Protocol Interface (NPI)

`Write`(bucket1, bucket2, encryptedMsg, interestVector)

`Read`(requestVectors[]) → encryptedData

`GetUpdates`() → globalInterestVector

Client State

• *logs* - List of subscribed logs

• *writeQueue* - Queue of write operations

• *readQueue* - Queue of read operations

Server State

• *log* - Global log of write operations

• *table* - Blocked cuckoo hash table

Figure 4.3: Summary of Talek interfaces and client/server state

Server Overview: The server is designed to store a limited set of messages in order to allow asynchronous senders and receivers to be decoupled in time, rather than participating in synchronous rounds of communication. Because the cost of PIR operations scales linearly with the size of the database, for good performance we globally fix the number of messages stored on the server to n, garbage collecting the oldest. Thus, n is directly related to the time-to-live, TTL, for messages, which dictates how tightly synchronized senders and receivers need to be. As the number of clients in the systems grows, the system must use larger values of n to support the same TTL.

In order to efficiently pack these messages into a dense data structure that is compatible with PIR, we store messages in a blocked cuckoo hash table [96], where each of the b buckets stores a fixed number of messages, d. Client `Write` requests explicitly specify two pseudo-randomly chosen buckets in which messages can be inserted, potentially resulting in cuckoo evictions (table rearrangement) if both buckets are full. In a `Read` request, the hash table is treated as a PIR database with each hash bucket as an entry. The client uses PIR to retrieve an entire hash bucket without revealing to the server which bucket it retrieved. Each server stores a consistent replica of the hash table to participate in the PIR protocol.

Blocked cuckoo hashing has a number of desirable properties for our system. Compared to chained hash tables, buckets have equal fixed size, a necessary requirement for PIR. Each message is stored in one of two buckets. To handle collisions, the size of the table must be larger than n by a small overhead factor (generally less than 20% for reasonable values of the bucket size d).

A client issues at most two `Read` requests to check both buckets where a message could be stored. If the client finds the message it is looking for in the first bucket, then it can use its next `Read` request for another task, rather than querying the second cuckoo hash location. From the server's perspective, the client is simply issuing a stream of opaque PIR requests.

Private Log Overview: In order for users to write a series of messages without online coordination with other users, oblivious logging is used to hide messages within a stream of apparently random writes. This log is defined by a secret log handle. Exposure of the log handle (e.g., by an untrustworthy user) would expose the user's write pattern, but not reader consumption. The log handle is used with a pseudorandom function family, PRF, to generate a deterministic sequence of buckets, called a *log trail*. The log writer stores encrypted messages along the log trail. Readers use PIR to retrieve messages following the same sequence.

In order to avoid the need to poll for new messages, private notifications (§ 4.5) assist readers in knowing when to read. With each write, clients submit a Bloom-filter-based *interest vector*, which privately encodes the log ID and sequence number of the message. Servers combine the interest vectors of all messages currently in the database to form a *global interest vector*, which privately encodes which messages are currently stored on the server. Clients periodically retrieve this global interest vector, which let them skip reading buckets with no new messages. Clients read and write on their independent schedules, which is not changed by information from this vector.

The next few sections describe each aspect of the system more formally. We first consider how Talek works with m idle online clients and l servers, illustrating the data structures, network requests, and a framework for security (Section 4.3). Then, we expand on foundation to hide legitimate traffic among requests using oblivious logging (Section 4.4) and private notifications (Section 4.5).

	Globally Configured	
l	constant	Number of servers
n	constant	Number of messages stored on server
b	constant	Number of server-side buckets
d	constant	Depth of a bucket
z	constant	Size of a single message
w	constant	Per-user rate of writes
r	constant	Per-user rate of reads
	Dynamically Measured	
m	variable	Number of online clients
TTL	$n/(m * w)$	Lifetime of a message on the server
$load$	$n/(b * d)$	Load factor of the server hash table

Figure 4.4: Variables in the system, including those configured by the developer and dynamic behavior measured at run-time.

4.3 Talek with Idle Users

Online clients issue dummy `Read` and `Write` requests at fixed rates of r and w respectively. We choose an arbitrary server to be the *leader*, $\mathscr{S}_0$, with the rest of the servers forming the *follower* set, $[\mathscr{S}_1, \ldots, \mathscr{S}_{l-1}]$. All `Read` and `Write` requests are directed to the leader and forwarded down the chain of followers.

Talek is further configured with a window size, n, such that messages older than the most recent n are garbage collected and deleted. It is possible for clients to miss a message if they fall behind and it is garbage collected. In this case, readers can request retransmissions as described in Section 4.4.3. We show detailed pseudocode for the server in Figure 4.6.

4.3.1 Cryptographic Assumptions

Each server has a public-private key pair, pk, sk, generated using an algorithm $PKGen()$. We assume the public key of each server is known to all clients. We write $PKEnc_{pk}(text)$ for the encryption of *text* under pk, and $PKDec_{sk}(cipher)$ for the decryption of *cipher* under sk. Clients also have access to an efficient symmetric encryption scheme that provides *authenticated encryption with associated data* (AEAD). The associated data is authenticated, but not included in the ciphertext. We write $Enc_k(text, ad)$ for the encryption of *text* with key k and associated data *ad*, and $Dec_k(cipher, ad)$ for the decryption of *cipher*. Our implementation uses an IND-CCA2 [56] RSA encryption scheme and AES-GCM for symmetric encryption. Let $PRF(key, input)$ denote a pseudorandom function family and $PRNG(seed)$ denote a cryptographically secure pseudorandom number generator. For the purposes of this description, let $|$ denote tagged concatenation.

4.3.2 Strawman: Chained Hash Tables

We first consider a strawman approach to designing the server. Clients periodically write into pseudo-random positions on the server. Suppose we model the server's state as a table of b buckets, and clients explicitly place writes in a bucket. To make reading oblivious to the server, clients would use PIR to retrieve an entire bucket. To prevent collisions, the server must have a mechanism for handling collisions.

One way to deal with collisions is to use chaining, where each bucket is a linked list of values.

Because PIR requires elements of equal size, buckets would need to be padded to the length of the largest bucket. In the worst case scenario, one bucket could contain the entire database.

4.3.3 Write: Cuckoo Hashing

Talek organizes server-side state into a blocked cuckoo hash table [184, 96], where each server's storage is organized into b buckets, each bucket storing d messages, each of size z. PIR requests fetch an entire bucket of size $d \cdot z$. Figure 4.5 illustrates the server-side data structures. Cuckoo hashing has a number of desirable properties for PIR-based reads. In practice, the number of messages stored, n, is chosen as a fraction of the capacity of the cuckoo table, $b \cdot d$. This fraction is set to ensure with high probability, that a message will fit with minimal rearranging of the cuckoo table [96].

- PIR requires buckets be of equal size. Talek's blocked cuckoo hash table is configured with a fixed depth, d.
- Individual PIR operations are relatively expensive. Cuckoo hashing bounds the maximum number of client probes to 2.
- The cost of a PIR request scales linearly with the size of the database. Cuckoo hashing enables dense placement of messages in a pre-allocated data structure with minimal wasted space.

Because cuckoo hashing is a random algorithm and PIR requires consistent replicas across all servers, a shared random seed enables all servers to achieve identical state as long as items are inserted in the same order. The leader assigns each incoming request a global sequence number for consistent ordering.

Client `Write` requests are generated using the following protocol. A client, $\mathscr{C}$, periodically issues random `Write` requests to the server. $\mathscr{C}$ is preconfigured with a randomly chosen k_{idle}, which is used to generate the i-th random number by $PRF(k_{idle}, i)$, and an idle encryption key k_{enc}. Similarly, all servers share a key k_{cuckoo}, used to generate random values in the cuckoo algorithm below. The i-th client request is generated as follows:

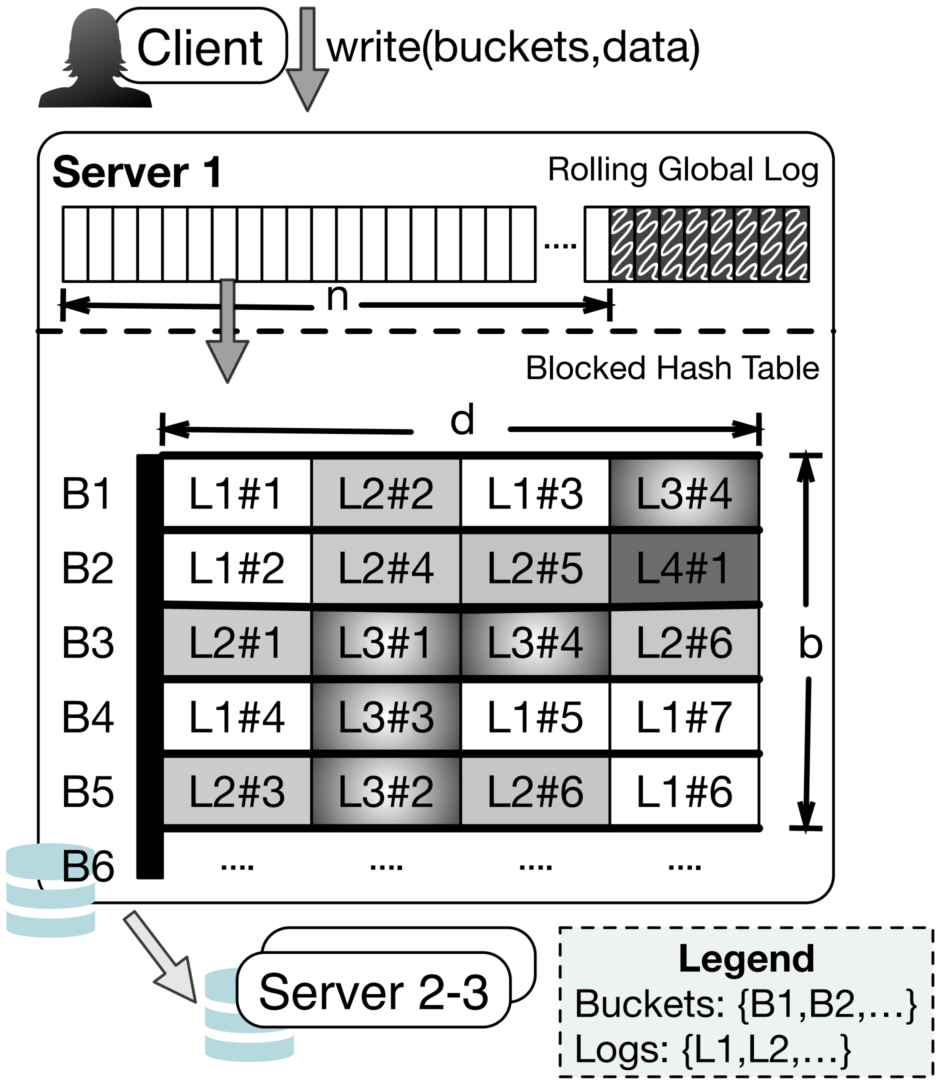

Figure 4.5: Oblivious logging data structures and workflow. The leader serializes all write operations into a global log, assigning each message a globally unique sequence number. Server-side state is replicated to all other servers from the leader. In order to garbage collect old messages, we keep only the latest n messages. Client writes specify two random buckets in which the message can be placed, forming a blocked cuckoo hashing scheme. Logs are spread across the hash table, which can be read by the readers using PIR. Messages in the same log are colored with the same shade in the diagram.

1. $\mathscr{C}$ chooses two random buckets,

$$\beta_1 = PRF(k_{idle}, i|1) \mod b$$
$$\beta_2 = PRF(k_{idle}, i|2) \mod b$$

   where b is the number of buckets.

2. $\mathscr{C}$ encrypts a random z-length bit-string,

$$data = Enc_{k_{enc}}(PRF(k_{idle}, i|3) \mod 2^z)$$

   and submits $\beta_1|\beta_2|data$ to the leader, $\mathscr{S}_0$

3. Upon receiving the request, $\mathscr{S}_0$ forwards the request to all other follower servers, $\mathscr{S}_1 \ldots \mathscr{S}_{l-1}$, each following the cuckoo algorithm in steps 4–7.

4. Each server deletes the n-th oldest element.

5. The server inserts $\beta_1|\beta_2|data$ into the bucket at either β_1 or β_2 if there is spare capacity in either bucket.

6. If both buckets are full, choose $\beta_e \in \{\beta_1, \beta_2\}$, using randomness from k_{cuckoo}. Let $\delta_e = \beta_1|\beta_2|data$.

7. Repeat the following until all values are inserted

   (a) Try to insert δ_e in β_e if the bucket has space.

   (b) If not, randomly evict an entry in β_e and insert δ_e in its place.

   (c) Let δ_e equal the evicted value and β_e equal its alternate bucket location.

Correctness: The leader is only responsible for assigning a global sequence number, which does not affect security or correctness. If the leader misrepresents the global sequence number of a message (e.g. by giving a different sequence number to different follower servers), it could cause those replicas to become inconsistent. Because any follower could also deny service by failing to respond or deviating from the protocol, the leader is in no more privileged a position to affect correctness or liveness of the system than any other server in the system. In Section 4.4.3 we describe how clients detect misbehavior.

Performance: Cuckoo tables have a maximum capacity that is lower than the size of the table, $b \cdot d$. The ratio of the maximum capacity of the cuckoo table to the allocated space is known as the *load factor*, a function of the bucket depth, d. For example, the load factor for $d = 1$ is less than 0.5, such that you must allocate twice as much memory as the number of items in the table. The load factor grows asymptotically towards 1 as d increases [96]. For values of $d > 3$, the load factor is over 95%. A high load factor translates to a more densely packed table and cheaper PIR operations for the same number of messages in the database, n.

In Talek, the number of buckets b and the depth of each bucket d are tuned to the client workload. Clients issue `Read`s with a random b-bit request vector and receive a $O(d)$-sized bucket in response. A smaller value for b and higher d enables cheaper PIR reads and smaller request vectors in PIR requests at the cost of larger network overhead for the response. This configuration lends itself to frequent writes, common in chat applications. Conversely, a high value of b and low value of d resembles a traditional cuckoo hash table, resulting in a lower load factor, but better bandwidth utilization. This configuration is appropriate for infrequent writes of large messages, such as for images.

4.3.4 Read: Serialized PIR

In order to reduce the network costs of `Read` requests to the client, we use a *serialized* variation of PIR, which offloads work from clients to the leader. The client sends a single request to the leader, containing PIRs for each follower, and receives a single response from the leader. We use one-time pads to preserve the confidentiality of each server's results, while allowing the leader to combine them on behalf of the client. In contrast, traditional PIR requires that the client receive messages from each server, and locally calculate the result.

A client, $\mathscr{C}$, periodically issues random PIR requests to the server as follows:

1. $\mathscr{C}$ chooses a random bucket to read and generates b-bit PIR requests for each server, $\{q_0, \ldots, q_{l-1}\}$, where b is the number of server buckets.

2. $\mathscr{C}$ generates a high-entropy random seed for each server,

$\{p_0, \ldots, p_{l-1}\}$

3. $\mathscr{C}$ encrypts each server's parameters with its respective public key and generates a `Read` request,
$PKEnc_{pk_0}(q_0|p_0), \ldots, PKEnc_{pk_{l-1}}(q_{l-1}|p_{l-1})$

4. $\mathscr{C}$ sends this request to the leader, $\mathscr{S}_0$, who forwards it to the remaining follower servers.

5. In parallel, each server, $\mathscr{S}_i$, decrypts its respective PIR request vector, q_i and computes its response, R_i.

6. Each server, $\mathscr{S}_i$, also computes a random one-time pad, $P_i = PRNG(p_i)$, from the seed parameter. This one-time pad should be of equal size to R_i.

7. Each server, $\mathscr{S}_i$, responds to the leader with $R_i \bigoplus P_i$.

8. $\mathscr{S}_0$ combines the server responses and responds to $\mathscr{C}$ with $R_0 \bigoplus P_0 \bigoplus \ldots R_{l-1} \bigoplus P_{l-1}$

9. $\mathscr{C}$ restores the bucket of interest by XOR'ing this response with each server's one-time pad, $P_0 \bigoplus \ldots \bigoplus P_{l-1}$

Security: This serialized variant of PIR is functionally equivalent to the traditional PIR scheme described in Section 2.3.3. As long as the adversary only has access to $l - 1$ servers' secret keys, it cannot decrypt the honest server's request vector and reconstitute the secret request. Similarly, each response is combined with a random one-time pad, which prevents the adversary from learning any information from any individual server's response. Because each server's one-time pad is computed from a shared secret with the client, the client can recover the underlying value.

Correctness and Liveness: As long as all servers are reachable and behave according to the protocol, serialized PIR is functionally equivalent to its parallel form described in Section 2.3.3. If any server deviates from the protocol or becomes unavailable, PIR operations would fail without affecting the security of the system. The correctness and liveness guarantees are equivalent to the parallel case.

Theorem 1. *Serialized PIR (Informally)*

1. *Security: As long as there exists at least one server's secret key that is unknown to the adversary, the adversary learns nothing of the user's secret request.*

2. *Correctness: The client receives the contents of the bucket corresponding to its request as long as every server is reachable and behaves according to the protocol.*

3. *Liveness: Misbehavior or unavailability of any server would prevent PIR operations from completing, but does not affect security.*

4.3.5 Security Analysis of Idle Sequences

By definition, the access sequence, seq_i, of each client is a null list. For each client, the adversary observes $events_i$, a log of randomly generated requests. In the protocol as described thus far, $events_i$ is completely independent from seq_i. While we use fixed rates for `Read` and `Write` for convenience, our security goals are met as long as the rates are independent. For example, if requests follow a Poisson distribution between the hours of 9am and 5pm for every user, our security properties still hold. In the next section, we discuss how legitimate accesses are hidden among this cover traffic in a way that is indistinguishable to the adversary.

4.4 Oblivious Logging

The goal of *oblivious logging* is to translate secret calls to `Publish` and `Subscribe` on the client developer interface (CDI), into random-looking `Write` and `Read` network requests on the network protocol interface (NPI). Critically, these `Write` and `Read` requests must look indistinguishable to the adversary from the cover traffic described in Section 4.3. In this section, we describe our log abstraction: single-writer, several-reader logs stored on Talek servers. A log is only ever written by its creator, but it may be read by many clients. We show detailed pseudocode for the client in Figure 4.7

4.4.1 Log Handles and Messages

When a user creates a new log, Talek generates a *log handle*, which contains a unique ID, *id*, encryption key, k_{enc}, and two seeds, k_{s1} and k_{s2}. The log handle is a shared secret between the writer and readers of a log. All messages are encrypted with k_{enc} using a CCA-secure symmetric encryption scheme, $Enc_{k_{enc}}(message)$. We further assign all messages in a log a sequence number, $seqNo$. The two seed values are used in conjunction with a pseudorandom function family, $PRF(seed, seqNo) \in \{0 \ldots (b-1)\}$, to produce two log trails, unique and deterministic sequences of bucket locations for writes. Similar in nature to frequency hopping [106, 99], log handles allow writers and readers to agree on a pseudorandom sequence of buckets without online coordination.

4.4.2 Scheduling Requests

When a writer wants to publish a message, M, with sequence number $seqNo$ to a log, the Talek client library does the following:

- On the next periodic random `Write` request, $\beta_1|\beta_2|data$, replace its parameters with the following:

$$\begin{aligned} \beta_1 &= PRF(k_{s1}, seqNo) \mod b \\ \beta_2 &= PRF(k_{s2}, seqNo) \mod b \\ data &= Enc_{k_{enc}}(M) \end{aligned}$$

When a reader polls for the next message in a log at sequence number, $seqNo$, they do the following:

1. On the next periodic random `Read` request, replace it with a PIR read to the first bucket,

$$PRF(k_{s1}, seqNo) \mod b$$

2. If the returned bucket is missing the message, on the following periodic random `Read` request, replace it with a PIR read to the second bucket,

$$PRF(k_{s2}, seqNo) \mod b$$

3. Attempt to decrypt every message in the bucket using $Dec_{k_{enc}}(M)$ and return the result if found

For both `Write` and `Read`, legitimate requests must follow the same periodic schedule as when idle in Section 4.3. To the adversary, writes look indistinguishable from the idle case. Log handles allow readers to find the latest content with zero coordination. Because reads are done with PIR, many readers can poll the same log or poll the same bucket repeatedly without revealing any information.

4.4.3 Control Logs

In order to facilitate control messages between users, we automatically generate a *control log* between every pair of users that share at least one log. We expect the log handle for the control log to be generated and exchanged out of band when public keys are exchanged and verified. The control log is used by the Talek system to send retransmission requests, bootstrap new logs by exchanging new handles, and other control messages to coordinate between users.

Clients also use a control log to periodically send low-priority messages to itself. If these messages are lost, it serves as a hint to the client of a denial of service attack. A limitation of our work is that it does not give clients the ability to determine which server is misbehaving.

4.4.4 Security Analysis

We more formally prove the security of oblivious logging in Section 4.7.7. Informally, we prove the security by reduction to cryptographic assumptions. For `Read`, we rely on the security properties offered by PIR. PIR queries that correspond to legitimate requests are indistinguishable from a PIR query for a random item [78]. For `Write`, we rely on the security properties of a PRF and our encryption algorithm. We use an IND-CCA secure encryption algorithm for message payloads. For any `Write`, the bucket locations are generated by a PRF, using either the log handle's seed values, (k_{s1}, k_{s2}), or the idle seed, k_{idle}. In both cases, the output is indistinguishable from a random function. [120].

A malicious client still has the ability to try to deny service by deviating from the protocol and choosing a fixed bucket to DoS. DoS is limited by the fixed `Write` rate per client, the number of Sybil clients, and the size of the database, n. We rely on the self-balancing nature of cuckoo tables, where legitimate messages can be evicted to their alternate locations.

We assume the writer trusts the readers of a log. Log handles can expose a writer if shared with adversarial servers, by observing the users that write to a known log trail. Even so, PIR protects the privacy of readers. Writers must only use each output from $PRF(seed, seqNo)$ for any sequence number once.

```
//GlobalState
$globalLog \leftarrow Array()$    ▷ Global log of write operations
$seqNo \leftarrow 0$    ▷ Global sequence number
$hashtable \leftarrow BlockedCuckoo(b, d)$    ▷ $b$ buckets of depth $d$

1: //Writes the data into one of two buckets
2: function WRITE($buckets$,$data$,$interestVec$)
3:     if $isLeader()$ then
4:         $seqNo \leftarrow seqNo + 1$
5:         Append operation to $globalLog$
6:         Forward operation to follower servers with $seqNo$
7:     else
8:         Insert operation into $globalLog$ at given $seqNo$
9:     end if
10:    Remove $n$-th oldest element from $hashtable$
11:    $hashtable.insert(buckets[0], buckets[1], data)$
12: end function
```

```
1: //Performs a PIR-based read
2: function READ($bucketVector$)
3:     return $bucketVector \cdot hashtable$
4: end function
```

```
1: //Returns the global interest vector
2: function GETUPDATES
3:     $v \leftarrow BloomFilter()$
4:     for all $e \in$ last $n$ elements of $globalLog$ do
5:         $v \leftarrow v \cup e.interestVec$
6:     end for
7:     return $v$
8: end function
```

Figure 4.6: Pseudocode for server-side RPC handlers (NPI). The NPI was designed such that all parameters for any operation reveal no information about the user's application usage. Writes are serialized by the leader and replicated in global order to the follower servers. When writing, clients explicitly specify the two potential hash table buckets into which data is inserted. When data is read using a PIR protocol, we expose a blocked cuckoo hash table with the n most recent messages in the log and return full buckets. For simplicity, we only describe the original IT-PIR algorithm, which we show is equivalent to the serialized PIR algorithm in Section 4.3.4

```
  $//GlobalState$
  $logs \leftarrow Map()$    ▷ Latest sequence numbers seen for each log
  $readQueue \leftarrow Queue()$
  $writeQueue \leftarrow Queue()$

 1: function PUBLISH($log$, $message$)
 2:     Enqueue operation to $writeQueue$
 3: end function

 1: function SUBSCRIBE($log$)
 2:     Add $log$ to $logs$
 3: end function

 1: function PERIODICWRITE
 2:     if $writeQueue.isEmpty()$ then
 3:         Send a random write to the leader
 4:     else
 5:         $log, data \leftarrow writeQueue.dequeue()$
 6:         $seqNo \leftarrow logs[t.id]{+}{+}$
 7:         $bucket1 \leftarrow PRF(t.seed1, seqNo)$
 8:         $bucket2 \leftarrow PRF(t.seed2, seqNo)$
 9:         $data' \leftarrow Enc_{log.key}(data)$
10:         $intVec \leftarrow BloomFilter()$
11:         $intVec.insert(log.id|seqNo)$
12:         $leader.Write([bucket1, bucket2], data', intVec)$
13:     end if
14: end function

 1: function PERIODICREAD
 2:     if $readQueue.isEmpty()$ then
 3:         Send a random read to each server
 4:     else
 5:         $log, seqNo, seedChoice \leftarrow readQueue.dequeue()$
 6:         $seed \leftarrow (seedChoice == 1)?log.seed1 : log.seed2$
 7:         $data, query_l, query' \leftarrow [0 \dots 0]$
 8:         $query'[PRF(seed, seqNo)] \leftarrow 1$    ▷ secret
 9:         for each server in followers do
10:             $query \leftarrow RandomBitString(numBuckets)$
11:             $data \leftarrow data \bigoplus server.Read(query)$
12:             $query_l \leftarrow query_l \bigoplus query$
13:         end for
14:         $query_l \leftarrow query_l \bigoplus query'$
15:         $data \leftarrow data \bigoplus leader.Read(query_l)$
16:         if $data$ contains $(log, seqNo)$ then
17:             return $data$
18:         else if $seedChoice == 1$ then
19:             Enqueue read $(log, seqNo, 2)$ to $readQueue$
20:         end if
21:     end if
22: end function

 1: function PERIODICUPDATES
 2:     $globalIntVec \leftarrow leader.GetUpdates()$
 3:     for all $log \in logs$ do
 4:         $seqNo \leftarrow logs[log.id]$
 5:         if $globalIntVec.contains(log.id, seqNo)$ then
 6:             Enqueue read $(log, seqNo, 1)$ to $readQueue$
 7:         end if
 8:     end for
 9: end function
```

Figure 4.7: Pseudocode for the Talek client library. Calls to publish and subscribe are queued in a global request queue. A periodic process either issues a random request or dequeues a legitimate operation to be translated into a privacy-preserving NPI request. Messages in a log are written to a deterministic random order of buckets. Subscribers then use PIR to retrieve these messages. For simplicity, we describe the protocol assuming an authenticated secure channel to each server. In practice, we use a functionally equivalent serialized version of PIR described in Section 4.3.4.

```
{
  logID: uint128,
  seed1: uint128,
  seed2: uint128,
  encKey: byte[]
}
```

Figure 4.8: Log Handle

```
Encrypt ({
  logId: uint128,
  seqNo: uint64,
  value: byte[],
  signature: byte[],
}, encKey)
```

Figure 4.9: Message Payload

Figure 4.10: Schema of the log handle and a message payload. The log handle is a shared secret between a trusted group of users, used to reconstitute a log from the servers. Each message payload in the log is encrypted with a shared encryption key.

4.5 Private Notifications

Regular polling in Talek presents two problems. First, because every user polls at the same rate to meet our security goal, message latency gets worse as the user subscribes to more logs. Second, it is hard to know which log should be polled at any given time.

We introduce a private notification system that allows users to efficiently determine when new messages have been published to a log without revealing their subscribed log list. By detaching reads from notifications, clients can prioritize reads and reduce how often they read buckets with no new messages.

4.5.1 Computing a Global Interest Vector

With every `Write` request, the client computes a Bloom filter, called an *interest vector*, encoding the log ID and sequence number of the message being written,

$$v.insert(logId|seqNo) \rightarrow [b_0 \ldots b_{|v|}]$$

If we use a large random log ID and h cryptographically-secure hash functions (modeled as random oracles) in the Bloom filter, then the adversary has a negligible advantage in learning the client's input given the interest vector. Because the server can expect every interest vector to contain only one element, it can also filter any malicious interest vectors where there are more ones than hash functions used, $\sum_{i=0}^{|v|} b_i > h$. If the `Write` request is a dummy request, we randomly choose h bits to set to 1, with the remaining bits set to 0.

Servers maintain a *global interest vector*, computed by taking the union of all *message interest vectors* of messages stored on the server. This data structure efficiently encodes the log ID and sequence numbers of every message on the server without revealing anything to the server. Clients periodically query for the global interest vector using `GetUpdates`, allowing each client to independently determine which logs have new messages.

4.5.2 Security Analysis

The security of private notifications relies on the cryptographic hash functions used in the Bloom filter. As long as we use a log ID with sufficient entropy, each interest vector provides a negligible advantage in the indistinguishability security game.

Private notifications are only used to prioritize reads on the internal request queue. As such, it has no impact on the security goals of oblivious logging. It simply reorders the schedule of private requests.

Because private notifications are a parallel mechanism to oblivious logging, servers could manipulate the vector to influence clients. In order to detect server misbehavior, clients can retrieve the global interest vector from every server, and ensure they are consistent.

4.6 Implementation

To demonstrate that Talek is practical, we implement a prototype in approximately 6,200 lines of code; the source code is online. We implement two versions. The first, written in Go, runs entirely on the CPU. The second offloads PIR operations to the GPU using a kernel written in C on OpenCL. We wrote Go language bindings to share memory between the CPU implementation and the GPU. The prototype uses SipHash [49] as the pseudorandom function, RSA for public-key encryption, and AES-GCM for symmetric encryption.

4.7 Evaluation

Our evaluation addresses the following questions:

- What is the cost of operations for clients and servers?
- How does system performance scale with more users?
- How does Talek compare with previous work?
- What is the end-to-end latency of messages?

4.7.1 Setup

All experiments are conducted on Amazon EC2 P2 instances. These virtual machines are allocated 4 cores on an Intel Xeon E5–2686v4 processor and 61 GB of RAM. They also include an NVIDIA K80 GPU with 2496 cores and 12 GB of memory. We use 3 servers; One is chosen as the leader and the others are followers. We allocated an additional two VMs to run user clients. Each user client issues periodic `Read` and `Write` requests to the server.

In order to evaluate a realistic workload, we chose system parameters to reflect public email datasets from a large EU research institution [152] and Enron [144, 153]. We varied the number of users, $m \in (0, 35K]$, and the number of messages in the database, $n \in \{10K, 32K, 100K, 500K, 1M\}$. We fix the message size to $z = 1K$ and the number of subscribed logs per user to 10. For comparison, the first month of the EU email dataset consisted of 254,080 emails across 38,090 different addresses with an average degree of 5.75. The Enron dataset consisted of 33,696 users, 200,399 messages, 2.87KB average message size, and an average degree of 10.73.

While our experiments are run in a single data center, we expect the performance to be similar for a more realistic cross-data center setting. This would incur higher network latency, both to reach the leader and in communicating between servers. However these latencies will not impact the results here which focus on the main bottleneck: the server-side computational cost of Talek.

4.7.2 Cost of Operations

To understand Talek's costs, we benchmark different components of the system. Each value is the average of 200 runs. We vary the number of messages on the server, $n \in \{10K, 100K, 1M\}$. We fix the bucket depth in the blocked cuckoo table to 4, such that clients retrieve 4 messages at a time. This depth allows the cuckoo table to support a load factor of 95%. The number of buckets is chosen to hold n messages at the maximum load factor for the table. Figure 4.11 highlights the results.

In general, client costs are low due to IT-PIR. Each write encrypts the message and uses a PRF to determine the bucket location. The cost of `Publish`ing rises with the database size due to the Bloom filter. The cost of generating a PIR query also increases with the database size. Larger values of n translate to more buckets and larger PIR request vectors.

For the server, we implement two versions of IT-PIR. Our CPU implementation is primarily bottlenecked by memory bandwidth to the CPU. The GPU implementation accelerates performance by 1–2 orders of magnitude by taking advantage of the inherent parallelism of PIR operations across many GPU cores and the optimized on-device memory hierarchy. Writes incur negligible

	Messages on Server (n)		
	10K	100K	1M
Client-side CPU costs			
Generate new log handle	7753 μs		
Publish (1 KB messages)	12.3 μs	70.5 μs	840.3 μs
PIR query	65.4 μs	574 μs	6888 μs
PIR response (1 KB messages)	146 μs		
Server-side CPU costs			
PIR Read: CPU (1 KB messages)	1.34 ms	11.1 ms	88.1 ms
PIR Read: GPU (1 KB messages)	0.073 ms	0.54 ms	4.36 ms
Write (1 KB messages)	0.0219 ms		
Server-side storage costs			
1 KB messages	24.1 MB	241 MB	2.41 GB
Network costs			
GetUpdates	6.01 KB	59.9 KB	599 KB
Read request	0.96 KB	9.39 KB	93.7 KB
Read response (1 KB messages)	4.16 KB		
Write request (1 KB messages)	1.08 KB		

Figure 4.11: Cost of individual operations in Talek. We vary the number of messages stored on the server, n. For values that do not depend on the number of messages on the server, we write a single value in the left column.

cost compared to the cost of reads. In our current implementation, we store all messages twice in memory. Writes are applied to the working copy stored in DRAM. Periodically, a snapshot of this state is copied into the GPU. Read requests are batched and forwarded to the GPU for processing. The leader is free to reorder reads without violating serializability.

Network costs between client and server are minimal. Clients must submit a read request containing a b-bit vector for each server. The size of `Read` responses and `Write` requests are within a small factor of the message size. The global interest vector returned from `GetUpdates` grows linearly with n in order to preserve a fixed false positive rate of 0.1. This cost is independent of the message size, such that its relative cost is lower for larger messages. In choosing a size for the Bloom filter, we trade off bandwidth with the false positive rate. The network costs per operation are identical between servers, as both `Read` and `Write` operations simply relay from the leader to followers.

4.7.3 Cost of Cover Traffic

Because each Talek client must generate network traffic on a regular schedule that is independent from the user's real usage, developers must choose a global schedule that is appropriate for their application. Naturally, there is a trade-off between efficiency and latency. At its most efficient, the schedule matches the average workload, such that occasional users generate more dummy requests to meet the average and prolific users are rate-limited. At the cost of higher overhead, application developers can choose to send requests more frequently to proportionally reduce the average latency of messages. This design decision will largely depend on application workloads.

As a concrete example, the first month of the EU email data set [152] contained 254,080 emails involving 1,258 researchers at the research institution, or on average 201.9 messages per month. If we were to configure each user to send and receive a message every 3.56 hours, the majority of network requests would constitute real work, but end-to-end latency for messages is 7 hours in the worst case. If Talek were configured to read and write four times as often, then most messages would be seen by subscribers within 2 hours.

For the rest of this evaluation, we fix both the read and write rates to `Publish` and `Poll` every 5 seconds in order to stress test the system's limits. Even at this rate, we believe the cover traffic

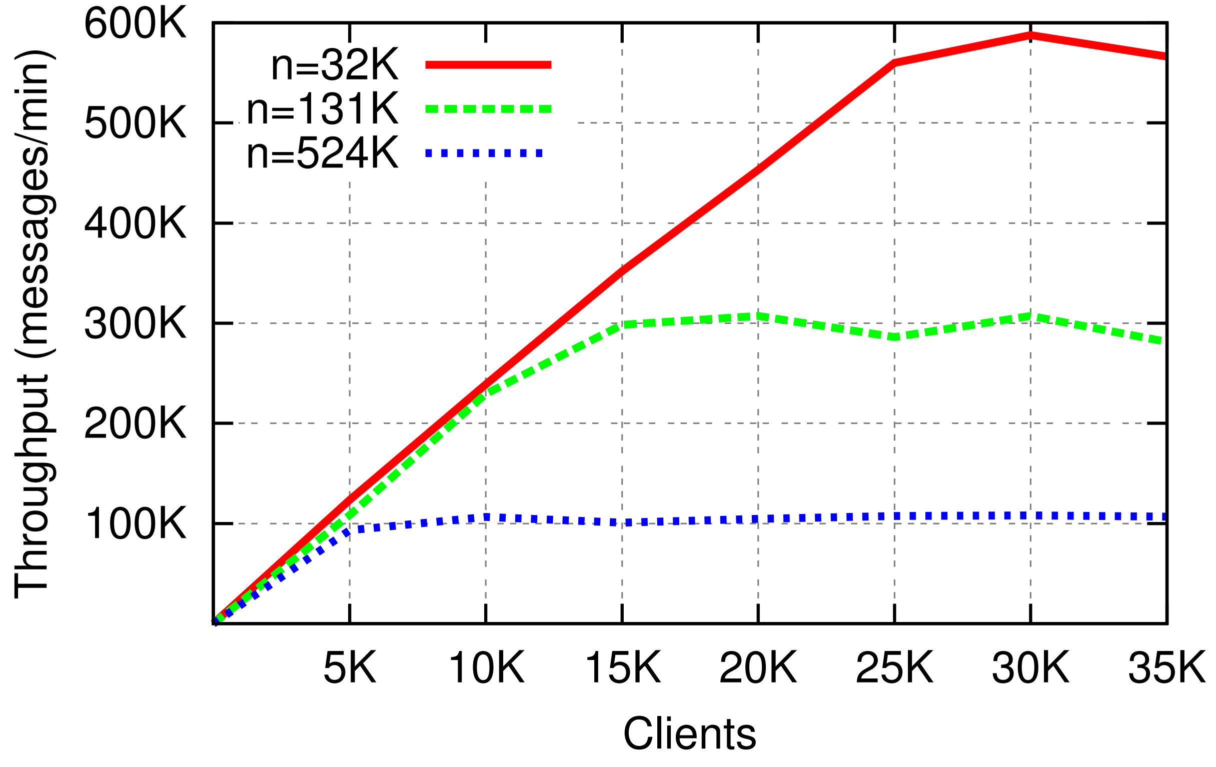

Figure 4.12: Throughput of the system when varying the number of real clients. Each client issues read and write requests every 5 seconds independently. Each line represents a different value for n, the number of messages stored on the server. For larger values of n, each read requires scanning a larger table, resulting in lower throughput.

for mobile devices is reasonable. In practice, users are engaged with their mobile devices only 8.6% of the time [74]. As described in Section 4.1.3, because clients only need to send requests when the device is online, we estimate this cover traffic to be about 9.2MB per day per client for our workloads. Assuming the server supports 32000 users, the server would use 294.4GB/day.

4.7.4 Throughput

To understand Talek's peak performance, we conducted an experiment with a simulated messaging workload. Each client sends a message every five seconds, and receives a message every five seconds. For each data point, we spawn a number of clients and measure the leader's response rate over 5 minutes, giving the system enough time to reach steady-state performance. Writing in Talek is cheap, so we limit our workload such that each written message must be read before being garbage collected. If writes were not throttled, servers could easily accommodate a higher write throughput, while reads are bottlenecked by PIR computation.

Figure 4.12 shows the results for three values of $n \in \{32K, 131K, 524K\}$, the number of messages stored on the server. For small numbers of clients, the server achieves linear growth in throughput, demonstrating that the PIR operations are keeping up with read requests. The throughput is bottlenecked by the GPU's PIR process. Smaller values of n correspond to a smaller cuckoo table, resulting in cheaper PIR operations and higher throughput. We only evaluate the system with numbers of clients, m, such that $m < n$, corresponding to a message lifetime of at least one round of reads.

4.7.5 Comparison with prior work

In order to understand the relative performance between Talek with prior work, we benchmarked the `Read` and `Write` mechanisms. Figure 4.13 shows the relative throughput of each system.

Pung [48] is a read/write key-value based on computational PIR (C-PIR). Pung has a stronger threat model than Talek, where all servers are assumed to be untrusted. Pung uses an implementation of C-PIR called XPIR [45] for reads, which we compare to our IT-PIR implementation. For 256 B messages, XPIR took $117ms$ and $11.55s$ per message for table sizes of $10K$ and $100K$ respectively. Because the memory on the test machine was exhausted at $n = 200K$, we estimate that XPIR would take at least $100s$ for $n = 1M$ if the cost were to increase linearly. XPIR leverages lattice-based homomorphic encryption and efforts to build special-purpose cryptoprocessors [208, 205] have yielded at most a 10x performance boost. This still leaves a computational cost that is 3–4 orders of magnitude higher than Talek, limiting its practicality for real-world use. In contrast, Talek is easily parallelizable and accelerated using commodity GPUs.

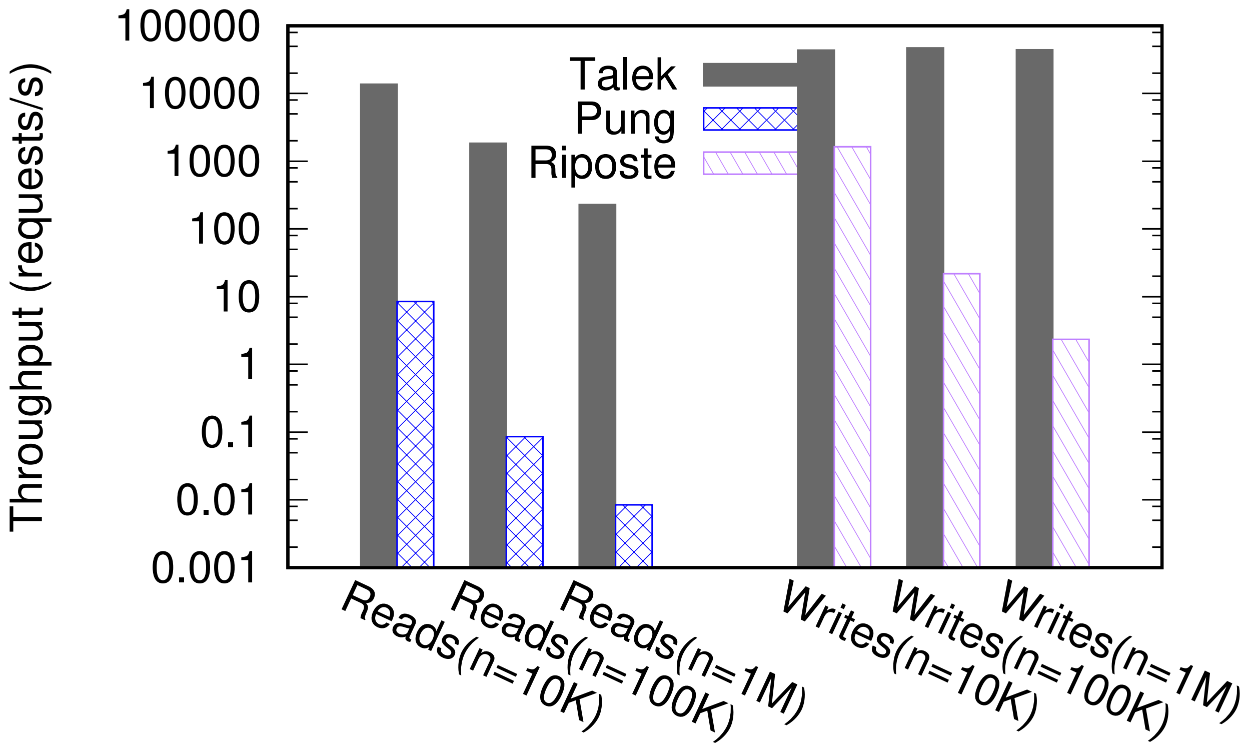

Figure 4.13: Performance comparison of `Read` and `Write` handlers of various systems. Pung uses C-PIR, which imposes a computational cost that is 3–4 orders of magnitude higher than our IT-PIR implementation. Riposte uses a write mechanism based on a reverse variant of PIR which incurs $O(\sqrt{n})$ cost compared to Talek's $O(1)$ writes.

	Talek	Pung	Riposte	Vuvuzela
Client-side CPU				
Read	$O(l)$	$O(log(n))$	✗	$O(l)$
Write	$O(1)$	$O(1)$	$O(\sqrt{n})$	$O(l)$
Server-side CPU				
Read	$O(ln)$	$O(n \cdot log(n))$	✗	$O(l)$
Write	$O(l)$	$O(log(n))$	$O(\sqrt{n})$	$O(l)$
Server-side storage	$O(ln)$	$O(n)$	$O(ln \cdot \sqrt{n})$	$O(n)$
Network costs				
Read request	$O(ln)$	$O(n \cdot log(n))$	✗	$O(1)$
Read response	$O(1)$	$O(log(n))$	✗	$O(1)$
Write request	$O(1)$	$O(1)$	$O(l \cdot \sqrt{n})$	$O(1)$
Write response	$O(1)$	$O(1)$	$O(1)$	$O(1)$

Figure 4.14: Comparison of the asymptotic complexity of computational, storage, and network costs between related work. Costs are parameterized on the number of servers, l, and the size of the database, n. Talek and Pung share a stronger security goal compared to Vuvuzela and Riposte. Riposte does not specify a read mechanism.

Pung has higher network overhead that Talek. Pung uses an interactive binary search algorithm for retrieval, requiring $O(log(n))$ round trips between client and server, compared to the $O(1)$ cuckoo table lookup in Talek. Thus, to retrieve a 1KB message from a database size of $n = 32K$ messages, Pung requires 15 rounds of PIR requests and $36.7MB$ of data, while Talek makes 2 requests and transfers $< 12KB$. Even if Pung used IT-PIR, their read/write protocol would be impractical for mobile workloads.

Riposte [84] is an anonymous broadcast system that uses PIR in reverse to anonymize writes to a database. Riposte has a weaker security goal based on anonymity within a round of communication and does not offer privacy over multiple rounds of communication. The Riposte implementation does not include an implementation for reads. Riposte writes incurs $O(\sqrt{n})$ cost compared to Talek's $O(1)$ writes.

Vuvuzela [232] has a weaker security goal based on differential privacy and noise but better performance. It scales to millions of users with a peak throughput of nearly 4M messages/min using the same number of servers as Talek. Although we have not implemented sharding, Talek is designed to be horizontally scalable to allow system throughput to increase by spreading buckets across servers and then combining the results. In this case, the leader and followers each consists of r replicas. Writes must be replicated to every replica server; however, this is acceptable because writes are inexpensive. Reads only require participation from one server in each replica group, which allows the system to scale for read-heavy workloads, similar to Vuvuzela.

4.7.6 End-to-End Latency with Notifications

In order to understand the latency of message delivery, we used the same messaging workload as in the throughput experiment, each user client sending and receiving messages every 5 seconds. Two additional clients are created, a sender and a receiver. We measure the end-to-end time for a message sent by the sender to be seen by the receiver, varying the number of logs to which the receiver client is subscribed. The spread of each value over 20 trials reflects the read and write rates.

Figure 4.15 shows the results with and without private notifications. When notifications are off, the client polls each log in a round robin fashion until it notices the new message. Because the

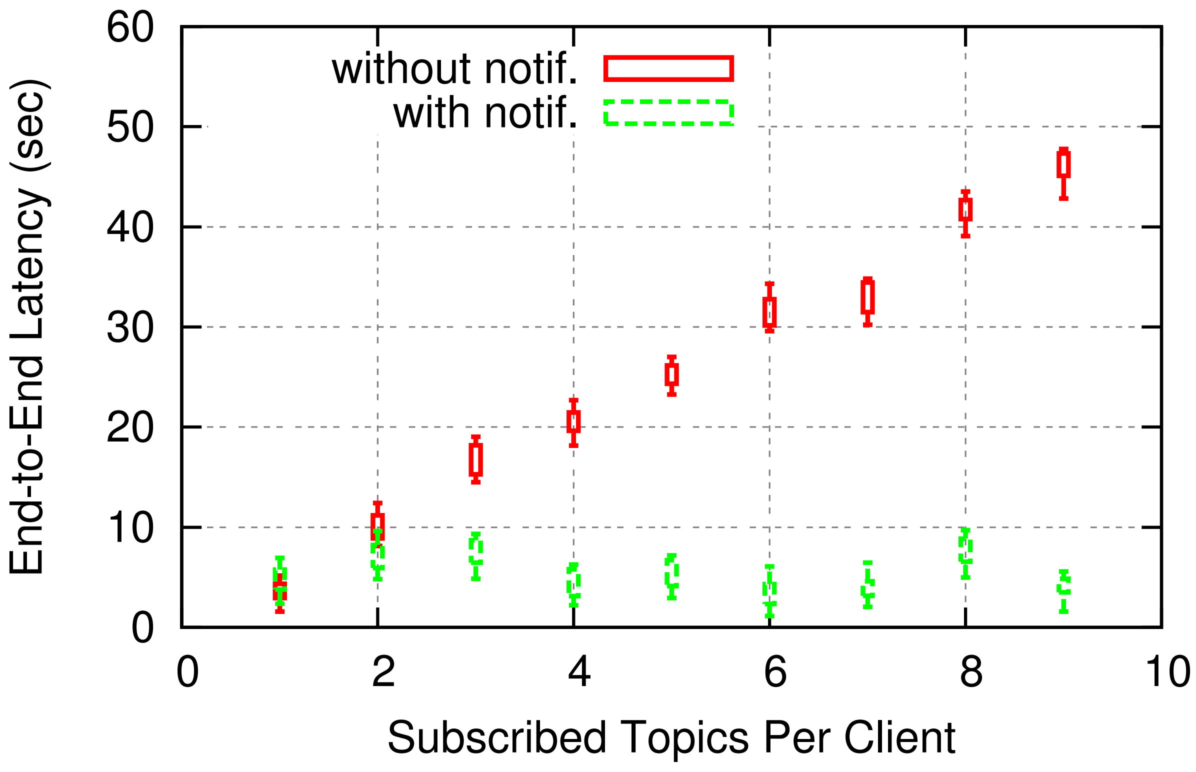

Figure 4.15: End-to-end latency of message delivery when a client is subscribed to multiple logs. Each data point represents 20 trials. With notifications, clients can prioritize logs with new messages.

read rate is fixed, the end-to-end latency grows linearly with the number of subscribed logs. With private notifications, the receiver periodically receives a global interest vector that encodes the log with the new message, allowing it to prioritize that read. As a result, the end-to-end latency for a single message is relatively fixed.

4.7.7 Proof of Access Sequence Indistinguishability

We provide a security proof for Talek's protocol by reduction to the cryptographic assumptions listed in Sections 4.1 and 4.3.

Theorem 2. *Talek security:*
Talek provides access sequence indistinguishability.

Proof:

We consider a series of games adapted from the game above, each defined from the previous one by idealizing some part of the protocol. For game i, we write p_i for the maximum advantage, $|Pr(b = b') - 1/2|$, that $\mathscr{A}$ holds in the security game. At each step, we bound the adversary's advantage between two successive games. Technically, each of the following games consists of a series of hybrid games, where we change each of the m clients one by one.

Game 0: Consider the game defined above with an adversary $\mathscr{A}$ that chooses m challenger clients, and submits sequences with α_0 calls to *Poll* and α_1 calls to $Write$.

Game 1: (PIR *Read*) This game is as above, except we replace the PIR request vectors, q_j, generated in *Poll*, with a bitstring $q_j \leftarrow \{0,1\}^b$ sampled at random. Let $\epsilon^{PIR}(\lambda_0, n)$ bound the advantage of an adversary breaking the PIR assumption in n calls to *Read* with security parameter λ_0. The adversary distinguishes between the request vectors in Game 0 and the randomly sampled requests in Game 1 with advantage $\epsilon^{PIR}(\lambda_0, \alpha_0)$.

$$p_0 \leq p_1 + m \cdot \epsilon^{PIR}(\lambda_0, \alpha_0)$$

Game 2: (IND-CCA with $Write$) This game is as above, except that for each client, we maintain a table T that maps ciphertexts under key k to plaintext messages δ. *Publish* is modified to encrypt a dummy message instead of δ and to record in T the resulting ciphertext and δ. Attempting to decrypt any ciphertext not in the table is rejected. *Poll* is modified to retrieve plaintext from T. We can apply our IND-CCA assumption for AEAD to each key k. Let $\epsilon^{AEAD}_{IND-CCA}(\lambda_1, n)$ be the advantage of an IND-CCA adversary that performs n oracle encryptions and decryptions with security parameter λ_1.

$$p_1 \leq p_2 + m \cdot \epsilon^{AEAD}_{IND-CCA}(\lambda_1, \alpha_1)$$

Game 3: (PRF with *Write*) This game is as above, except we replace the PRF used to generate the bucket locations of *Write*s with a perfect random function. Let $\epsilon^{PRF}_{distinguish}(\lambda_2, n)$ bound the advantage of an adversary breaking our PRF assumption in n calls to PRF with a security parameter, λ_2

$$p_2 \leq p_3 + 2 \cdot m \cdot \epsilon^{PRF}_{distinguish}(\lambda_2, \alpha_1)$$

Game 4: (Hash functions in interest vectors) This is game is as above, except we replace the h cryptographic hash functions used in the Bloom filter of the message interest vector with queries to a random oracle. Thus, the message interest vector will contain all 0's, except for 1's in h random positions. Let $\epsilon^{hash}(\lambda_3, n)$ bound the advantage of an adversary breaking our assumption of cryptographic hash functions in n calls with a security parameter, λ_3.

$$p_3 \leq p_4 + m \cdot \epsilon^{hash}(\lambda_3, \alpha_1)$$

From this final game, all of the parameters in any network request have been replaced with random values. Because Game 4 involves all clients issuing periodic requests with random parameters, by definition the adversary's advantage, p_4, must be zero.

Privacy: Collecting the probabilities from all games yields:

$$\begin{aligned} p_0 \leq & m \cdot \epsilon^{PIR}(\lambda_0, \alpha_0) + \\ & m \cdot \epsilon^{AEAD}_{IND-CCA}(\lambda_1, \alpha_1) + \\ & 2 \cdot m \cdot \epsilon^{PRF}_{distinguish}(\lambda_2, \alpha_1) + \\ & m \cdot \epsilon^{hash}(\lambda_3, \alpha_1) \end{aligned}$$

p_0 becomes negligible for large security parameters λ_0, λ_1, λ_2, and λ_3.

4.8 Summary

In this chapter, we present Talek, a general-purpose private group messaging system. Talek protects both the contents and metadata of users' application usage from untrusted servers. We show that strong security goals based on *access sequence indistinguishability*, where the adversary provably learns no information about which users may be communicating, is practical with two new techniques, *oblivious logging* and *private notifications.* Our evaluations confirm that our implementation achieves 3–4 orders of magnitude better performance than previous systems with similar security goals.

Chapter 5

CONCLUSION

The cloud will only grow in importance as a platform for processing and storing sensitive user data as more users become connected and devices gain network access. As the cloud plays an increasingly important role in our lives, we need to develop systems and techniques to safeguard user privacy in cloud applications. Users want strong guarantees that their data is only accessed by parties that the user expects. This thesis presents two novel systems to protect user privacy against realistic threats, while also providing good performance.

Radiatus secures full-featured applications from external intrusion, by isolating server-side code execution into per-user sandboxes and using a distributed capabilities system to efficient share data between users. This architecture protects applications from a large class of vulnerabilities, but it comes at the cost of added overhead of 60.7% per server and an additional 31MB of memory per active user. I demonstrate that the system can scale to 20K operations per second on a 500-node AWS cluster.

I then extend the threat model to include cloud insider attacks and design Talek, a private messaging system that is resilient to an untrusted cloud. Talek provides *access sequence indistinguishability*, preventing an adversary from learning anything about user communication patterns. Talek offers 3–4 orders of magnitude better performance compared to related work with similar security goals. Our implementation of a 3-server Talek cluster achieves a throughput of 566K messages per minute with 5.57-second end-to-end latency on commodity servers.

Future Work in Practical Privacy

In future work, I plan to continue working on designing practical systems with stronger security and privacy properties and deploying them in practice.

Scaling privacy services

Beyond designing provably secure systems, how can we engineer privacy services to support the traffic demands of popular web and mobile applications? Designing for scalability, fault tolerance, and consistency are critical to getting these systems used in practice. As an example, Talek is designed to be horizontally scalable to allow system throughput to increase by spreading data across servers and then combining the results. Co-designing scalable architectures with specialized hardware accelerators, such as GPUs and FPGAs, will also help improve performance.

Recovering from faulty servers

Privacy-based services, such as Talek, can be vulnerable to attacks on liveliness if servers decide to misbehave. While one can leverage multiple independent clouds, designing for fault tolerance under malicious participants requires new protocols that allow users to determine the source of the problem and use alternatives.

Integrating privacy into existing applications

Another area for further research is application integration. Privacy research often requires that clients adhere to multiple constraints, such as a fixed message length and a fixed rate of sending messages. In practice, these constraints impose usability costs to the user, resulting in message delays or unexpected behavior. I plan on studying real application workloads to better design mathematical models for privacy-preserving client behavior with usability in mind. I also plan to study better programming models for integrating privacy primitives into existing applications and embedded devices. For example, we can use programming languages to help developers reason about privacy in their code and to bound information leakage.

Computing over private data

Often, what the developer wants is the insight and value derived from the data. For example, an application might want a machine learning model trained from data across many users. How can we leverage advances in cryptographic primitives, to cooperatively train machine learning models without revealing user data to third parties? Similarly, can we redesign big data systems to privately compute results without access to unencrypted user data?

Leveraging hardware advances to accelerate private computation

As described in Chapter 2, trusted processors, PIR, and ORAM present an opportunity to build arbitrary applications with strong security goals with efficient performance. By relying on the hardware to protect confidentiality of computations within a CPU, we can design more performant systems by offloading sensitive computations to trusted processors. Trusted processors could also reduce the need to rely on expensive cryptographic primitives, like homomorphic encryption. How can we leverage these hardware advances to accelerate privacy-preserving applications and computations? Consequently, we may be able to provide practical performance for arbitrary cloud applications from both external and insider threats.

As Melvin Kranzberg put it, "Technology is neither good nor bad; nor is it neutral." The cloud has enabled countless new applications that have improved the lives of many people around the world. Along with every technological advancement comes with it, social, political, and human consequences that go beyond the immediate designed benefits. As we grow to depend on the cloud, we carry a responsibility to design new systems with a modern understanding of its consequences, systems that respect the dignity, freedom, and privacy of all people.

BIBLIOGRAPHY

[1] Akka Actor Model. `http://akka.io/`

[2] Amazon Information Request Report. `https://d0.awsstatic.com/certifications/Information_Request_Report_June_2016.pdf`

[3] Amazon Web Services. `https://aws.amazon.com`

[4] Apache HTTP Server Benchmarking Tool. `https://httpd.apache.org/docs/2.2/programs/ab.html`

[5] Apache Kafka. `https://kafka.apache.org/`

[6] Apache Virtual Host. `https://httpd.apache.org/docs/2.2/vhosts/`

[7] Average Selling Price of DRAM 1Gb Equivalent Units from 2009 to 2017. `http://www.statista.com/statistics/298821/dram-average-unit-price/`

[8] Benchmarking Go and Python Web servers. `http://ziutek.github.io/web_bench/`

[9] Content Security Policy 1.0. `http://www.w3.org/TR/CSP/`

[10] CWE/SANS Top 25 Most Dangerous Software Errors. `http://cwe.mitre.org/top25/`

[11] Django: How to Use Sessions. `https://docs.djangoproject.com/en/dev/topics/http/sessions/`

[12] Docker. `https://www.docker.com`

[13] eBay Digital Service Efficiency. `http://tech.ebay.com/dashboard`

[14] Facebook Government Requests Report. `https://govtrequests.facebook.com/`

[15] Facebook Newsroom. `https://newsroom.fb.com/company-info/`

[16] Firebase. `https://www.firebase.com`

[17] Freedom on the Net 2016. `https://freedomhouse.org/report/freedom-net/freedom-net-2016`

[18] Go Language Specification. `http://golang.org/ref/spec`

[19] Google Chrome Multi-process Architecture. `http://blog.chromium.org/2008/09/multi-process-architecture.html`

[20] Google Transparency Report. `https://www.google.com/transparencyreport`

[21] IE8 and Loosely-Coupled IE (LCIE). http://blogs.msdn.com/b/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx

[22] Intel Software Guard Extensions. https://software.intel.com/en-us/sgx

[23] Kernel Virtual Machine. http://www.linux-kvm.org/page/Main_Page

[24] Linux-VServer. http://linux-vserver.org/

[25] lmctfy. https://github.com/google/lmctfy

[26] National Vulnerability Database. https://nvd.nist.gov/

[27] Node.js. http://nodejs.org/

[28] Open Container Initiative. http://www.opencontainers.org/

[29] Open Web Application Security Project. https://www.owasp.org

[30] Passport.js. http://passportjs.org/

[31] RemoteStorage. http://remotestorage.io/

[32] Siege. http://www.joedog.org/siege-home/

[33] Signal. https://whispersystems.org/

[34] Snapchat Security Advisory. http://gibsonsec.org/snapchat/

[35] Spideroak. https://spideroak.com/

[36] Two Billion People Coming Together on Facebook. https://newsroom.fb.com/news/2017/06/two-billion-people-coming-together-on-facebook/

[37] Understanding Service-Oriented Architecture. http://msdn.microsoft.com/en-us/library/aa480021.aspx

[38] Web hacking incident database. http://projects.webappsec.org/w/page/13246995/Web-Hacking-Incident-Database

[39] GreenSQL. http://www.greensql.com/ 2009.

[40] Wikimedia Foundation Annual Report. http://upload.wikimedia.org/wikipedia/commons/4/48/WMF_AR11_SHIP_spreads_15dec11_72dpi.pdf 2011.

[41] Pond. https://github.com//agl/pond 2016.

[42] Keybase. https://keybase.io/ 2017.

[43] Ricochet: Anonymous peer-to-peer instant messaging. https://github.com/ricochet-im/ricochet 2017.

[44] Atul Adya, Gregory Cooper, Daniel Myers, and Michael Piatek. Thialfi: a Client Notification Service for Internet-Scale Applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 129–142. ACM, 2011.

[45] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR: Private Information Retrieval for Everyone. *Proceedings on Privacy Enhancing Technologies (PETS '16)*, 2016(2):155–174, 2015.

[46] Carlos Aguilar-Melchor and Philippe Gaborit. A Lattice-Based Computationally-Efficient Private Information Retrieval Protocol. *Cryptology ePrint Archive, Report*, 446, 2007.

[47] Fredrik Almroth and Mathias Karlsson. How We Got Read Access on Google Production Servers. `http://blog.detectify.com/post/82370846588/`

[48] Sebastian Angel and Srinath Setty. Unobservable Communication over Fully Untrusted Infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, 2016.

[49] Jean-Philippe Aumasson and Daniel J Bernstein. SipHash: a Fast Short-input PRF. In *International Conference on Cryptology in India*, pages 489–508. Springer, 2012.

[50] S. Balasubramaniam and Benjamin C. Pierce. What is a File Synchronizer? In *Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98)*, October 1998.

[51] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[52] Douglas Barry and Torsten Stanienda. Solving the Java Object Storage Problem. *Computer*, 31(11):33–40, 1998.

[53] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the Art: Automated Black-box Web Application Vulnerability Testing. In *IEEE Symposium on Security and Privacy (SP '10)*, pages 332–345. IEEE, 2010.

[54] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.

[55] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying Hash Functions for Message Authentication. In *Annual International Cryptology Conference*, pages 1–15. Springer, 1996.

[56] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations Among Notions of Security for Public-Key Encryption Schemes. In *Annual International Cryptology Conference*, pages 26–45. Springer, 1998.

[57] Philip A. Bernstein and Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[58] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, Radoslaw Bobrowicz, and V.N. Venkatakrishnan. NoTamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*, pages 607–618. ACM, 2010.

[59] 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 '11)*, pages 575–586. ACM, 2011.

[60] Aaron Blankstein and Michael J Freedman. Automating Isolation and Least Privilege in Web Services. In *IEEE Symposium on Security and Privacy (SP '14)*. IEEE, 2014.

[61] Alexandra Boldyreva, Vipul Goyal, and Virendra Kumar. Identity-based Encryption with Efficient Revocation. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*, pages 417–426. ACM, 2008.

[62] Dan Boneh and Matt Franklin. Identity-based Encryption from the Weil Pairing. In *Annual International Cryptology Conference*, pages 213–229. Springer, 2001.

[63] Nikita Borisov, George Danezis, and Ian Goldberg. DP5: A Private Presence Service. *Proceedings on Privacy Enhancing Technologies (PETS '15)*, 2015(2):4–24, 2015.

[64] Justin Brickell and Vitaly Shmatikov. Efficient Anonymity-Preserving Data Collection. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 76–85. ACM, 2006.

[65] Jonathan Burket, Patrick Mutchler, Michael Weaver, Muzzammil Zaveri, and David Evans. GuardRails: A Data-Centric Web Application Security Framework. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, 2011.

[66] Ramesh Chandra, Priya Gupta, and Nickolai Zeldovich. Separating Web Applications from User Data Storage with BSTORE. In *Proceedings of the USENIX Conference on Web Application Development*. USENIX Association, 2010.

[67] Ramesh Chandra, Taesoo Kim, Meelap Shah, Neha Narula, and Nickolai Zeldovich. Intrusion Recovery for Database-backed Web Applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 101–114. ACM, 2011.

[68] Ramesh Chandra, Taesoo Kim, and Nickolai Zeldovich. Asynchronous Intrusion Recovery for Interconnected Web Services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 213–227. ACM, 2013.

[69] Avik Chaudhuri and Jeffrey S Foster. Symbolic Security Analysis of Ruby-on-Rails Web Applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*, pages 585–594. ACM, 2010.

[70] David Chaum. The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. *Journal of Cryptology*, 1(1):65–75, 1988.

[71] David Chaum, Debajyoti Das, Farid Javani, Aniket Kate, Anna Krasnova, Joeri De Ruiter, and Alan T Sherman. cMix: Mixing with Minimal Real-Time Asymmetric Cryptographic Operations. In *International Conference on Applied Cryptography and Network Security*, pages 557–578. Springer, 2017.

[72] David L Chaum. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[73] Eric Yawei Chen, Jason Bau, Charles Reis, Adam Barth, and Collin Jackson. App Isolation: Get the Security of Multiple Browsers with Just One. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*, pages 227–238. ACM, 2011.

[74] Xiaomeng Chen, Ning Ding, Abhilash Jindal, Charlie Hu, Maruti Gupta, and Rath Vannithamby. Smartphone Energy Drain in the Wild: Analysis and Implications. *ACM SIGMETRICS Performance Evaluation Review*, 43(1):151–164, 2015.

[75] Raymond Cheng, Will Scott, Irene Zhang, Thomas Anderson, Arvind Krishnamurthy, and Bryan Parno. Talek: Private Group Messaging with Indistinguishable Access Patterns. Technical Report UW-CSE-16-11-01, University of Washington Computer Science and Engineering, Seattle, Washington, Nov 2016.

[76] Raymond Cheng, William Scott, Paul Ellenbogen, Jon Howell, Franziska Roesner, Arvind Krishnamurthy, and Thomas Anderson. Radiatus: A Shared-Nothing Server-Side Web Architecture. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 237–250, New York, NY, USA, 2016. ACM.

[77] Raymond Cheng, William. Scott, Arvind. Krishnamurthy, and Thomas Anderson. FreeDOM: a new baseline for the web. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 121–126. ACM, 2012.

[78] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private Information Retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

[79] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-Secure ORAM with $O(log^2 N)$ Overhead. In *20th Annual International Conference on the Theory and Applications of Cryptology and Information Security (ASIACRYPT 2014)*, pages 62–81. 2014.

[80] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Designing Privacy Enhancing Technologies (PETS '01)*, pages 46–66. Springer, 2001.

[81] William R Cook and Siddhartha Rai. Safe Query Objects: Statically Typed Objects as Remotely Executable Queries. In *27th International Conference on Software Engineering (ICSE '05)*, pages 97–106. IEEE, 2005.

[82] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC '10)*, pages 143–154, 2010.

[83] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Googles Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[84] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An Anonymous Messaging System Handling Millions of Users. In *2015 IEEE Symposium on Security and Privacy (SP '15)*, pages 321–338. IEEE, 2015.

[85] Henry Corrigan-Gibbs and Bryan Ford. Dissent: Accountable Anonymous Group Messaging. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*, pages 340–350. ACM, 2010.

[86] Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford. Proactively Accountable Anonymous Messaging in Verdict. In *22nd USENIX Security Symposium*, pages 147–162, 2013.

[87] Richard Cox, Jacob Gorm Hansen, Steven Gribble, and Henry Levy. A Safety-oriented Platform for Web Applications. In *2006 IEEE Symposium on Security and Privacy (SP '06)*, pages 15–pp. IEEE, 2006.

[88] Sophie Curtis. Barclays: 97 Percent of Data Breaches still due to SQL Injection. `http://news.techworld.com/security/3331283/`

[89] Wei Dai, Yarkın Doröz, and Berk Sunar. Accelerating NTRU-based Homomorphic Encryption using GPUs. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1–6. IEEE, 2014.

[90] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a Type III Anonymous Remailer Protocol. In *2003 IEEE Symposium on Security and Privacy (SP '03)*, pages 2–15. IEEE, 2003.

[91] George Danezis and Andrei Serjantov. Statistical Disclosure or Intersection Attacks on Anonymity Systems. In *International Workshop on Information Hiding*, pages 293–308, 2004.

[92] Jonathan Dautrich, Emil Stefanov, and Elaine Shi. Burst ORAM: Minimizing ORAM Response Times for Bursty Access Patterns. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 749–764, 2014.

[93] Benjamin Davis and Hao Chen. DBTaint: Cross-application Information Flow Tracking via Databases. In *2010 USENIX Conference on Web Application Development*, 2010.

[94] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[95] Casey Devet, Ian Goldberg, and Nadia Heninger. Optimally Robust Private Information Retrieval. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 269–283, 2012.

[96] Martin Dietzfelbinger and Christoph Weidling. Balanced Allocation and Dictionaries with Tightly Packed Constant Size Bins. *Theoretical Computer Science*, 380(1):47–68, 2007.

[97] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security*, 2004.

[98] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. M2R: Enabling Stronger Privacy in MapReduce Computation. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 447–462, 2015.

[99] Colin Dixon, Thomas E Anderson, and Arvind Krishnamurthy. Phalanx: Withstanding Multimillion-Node Botnets. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI '08)*, volume 8, pages 45–58, 2008.

[100] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software: Practice and Experience*, 21(8):757–785, 1991.

[101] Cynthia Dwork. Differential Privacy. In *Automata, languages and programming*, pages 1–12. 2006.

[102] Cynthia Dwork. Differential Privacy: A Survey of Results. In *Theory and applications of models of computation*, pages 1–19. 2008.

[103] Michael Egorov and MacLane Wilkison. ZeroDB Whitepaper, 2016.

[104] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266. ACM, December 1995.

[105] IDG Enterprise. Cloud Computing Survey. 2016.

[106] Anthony Ephremides, Jeffrey E Wieselthier, and Dennis J Baker. A Design Concept for Reliable Mobile Radio Networks with Frequency Hopping Signaling. *Proceedings of the IEEE*, 75(1):56–73, 1987.

[107] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*, pages 1054–1067. ACM, 2014.

[108] Joan Feigenbaum and Bryan Ford. Seeking Anonymity in an Internet Panopticon. *Communications of the ACM*, 58(10):58–69, 2015.

[109] Ariel J Feldman, Aaron Blankstein, Michael J Freedman, and Edward W Felten. Social Networking with Frientegrity: Privacy and Integrity with an Untrusted Provider. In *21st USENIX Security Symposium (USENIX Security '12)*, pages 647–662, 2012.

[110] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *USENIX Security Symposium*, pages 143–160, 2010.

[111] Michael J Freedman, Emil Sit, Josh Cates, and Robert Morris. Introducing Tarzan, a Peer-to-Peer Anonymizing Network Layer. In *Peer-to-Peer Systems*, pages 121–129. 2002.

[112] Kevin Fu, M Frans Kaashoek, and David Mazieres. Fast and Secure Distributed Read-Only File System. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '00)*, pages 13–13. USENIX Association, 2000.

[113] Jun Furukawa and Kazue Sako. An Efficient Scheme for Proving a Shuffle. In *Advances in Cryptology (CRYPTO 2001)*, pages 368–387, 2001.

[114] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, 2009. `crypto.stanford.edu/craig`.

[115] Craig Gentry, Kenny A Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and Using it Efficiently for Secure Computation. In *Proceedings on Privacy Enhancing Technologies (PETS '13)*, pages 1–18, 2013.

[116] Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic Evaluation of the AES Circuit. In *Advances in Cryptology–CRYPTO 2012*, pages 850–867. Springer, 2012.

[117] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. Hails: Protecting Data Privacy in Untrusted Web Applications. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI '12)*, pages 47–60, 2012.

[118] Ian Goldberg. Improving the Robustness of Private Information Retrieval. In *2007 IEEE Symposium on Security and Privacy (SP '07)*, pages 131–148. IEEE, 2007.

[119] Oded Goldreich. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 182–194. ACM, 1987.

[120] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. On the Cryptographic Applications of Random Functions. In *Workshop on the Theory and Application of Cryptographic Techniques*, pages 276–288. Springer, 1984.

[121] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[122] Philippe Golle and Ari Juels. Dining Cryptographers Revisited. In *Advances in Cryptology (Eurocrypt 2004)*, pages 456–473, 2004.

[123] Paul Graham and Robert Morris. Arc Forum. `http://arclanguage.org/forum` 2008.

[124] Saikat Guha, Bin Cheng, and Paul Francis. Privad: Practical Privacy in Online Advertising. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI '11)*, pages 13–13. USENIX Association, 2011.

[125] Ceki Gülcü and Gene Tsudik. Mixing E-mail with Babel. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS '96)*, pages 2–16. IEEE, 1996.

[126] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath TV Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and Private Media Consumption with Popcorn. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*, Santa Clara, CA, March 2016. USENIX Association.

[127] Erika Harrell and Lynn Langton. Victims of identity theft, 2012. `http://www.bjs.gov/content/pub/ascii/vit12.txt` 2013.

[128] Chris Hawblitzel, Jon Howell, Jacob R Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI '14)*, pages 165–181, 2014.

[129] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: a Highly Reliable, Self-Repairing Operating System. *SIGOPS Operating Systems Review*, 40(3):80–89, July 2006.

[130] Jason I Hong and James A Landay. An Architecture for Privacy-Sensitive Ubiquitous Computing. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services*, pages 177–189. ACM, 2004.

[131] Jon Howell, Collin Jackson, Helen J Wang, and Xiaofeng Fan. MashupOS: Operating System Abstractions for Client Mashups. In *Hot Topics in Operating Systems (HotOS '07)*, volume 7, pages 1–7, 2007.

[132] Jon Howell, Bryan Parno, and J Douceur. Embassies: Radically Refactoring the Web. *USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, 2013.

[133] Galen C Hunt and James R Larus. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, 2007.

[134] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 533–549, 2016.

[135] Lon Ingram and Michael Walfish. TreeHouse: Javascript Sandboxes to Help Web Developers Help Themselves. In *Proceedings of the USENIX Annual Technical Conference (ATC '12)*, 2012.

[136] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access Pattern Disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *Proceedings of the*

Symposium on Network and Distributed System Security (NDSS '12), volume 20, page 12, 2012.

[137] Anja Jerichow, Jan Muller, Andreas Pfitzmann, Birgit Pfitzmann, and Michael Waidner. Real-time Mixes: A Bandwidth-Efficient Anonymity Protocol. *IEEE Journal on Selected Areas in Communications*, 16(4):495–509, 1998.

[138] Nicholas Jones, Matvey Arye, Jacopo Cesareo, and Michael J Freedman. Hiding Amongst the Clouds: A Proposal for Cloud-based Onion Routing. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI '11)*, 2011.

[139] Chris Karlof, Naveen Sastry, and David Wagner. Cryptographic Voting Protocols: A Systems Perspective. In *USENIX Security*, volume 5, pages 33–50, 2005.

[140] Sachin Katti, Jeff Cohen, and Dina Katabi. Information Slicing - Anonymity Using Unreliable Overlays. In *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI '07)*, pages 43–56, 2007.

[141] Dogan Kedogan, Dakshi Agrawal, and Stefan Penz. Limits of Anonymity in Open Environments. In *International Workshop on Information Hiding*, pages 53–69, 2002.

[142] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. Efficient Patch-based Auditing for Web Application Vulnerabilities. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI '12)*, pages 193–206. USENIX Association, 2012.

[143] Rob King and TippingPoint DVLabs. Pixaxe: a Declarative, Client-focused Web Application Framework. In *Proceedings of the 2010 USENIX Conference on Web Application Development*, pages 10–10. USENIX Association, 2010.

[144] Bryan Klimt and Yiming Yang. Introducing the Enron Corpus. In *CEAS*, 2004.

[145] Maxwell Krohn. Building Secure High-Performance Web Services with OKWS. In *USENIX Annual Technical Conference (ATC '04)*, pages 185–198, 2004.

[146] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M Frans Kaashoek, Eddie Kohler, and Robert Morris. Information Flow Control for Standard OS Abstractions. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 321–334, 2007.

[147] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (In)security of Hash-Based Oblivious RAM and a New Balancing Scheme. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 143–156. SIAM, 2012.

[148] Eyal Kushilevitz and Rafail Ostrovsky. Replication is Not Needed: Single Database, Computationally-Private Information Retrieval. In *IEEE Symposium on Foundations of Computer Science (FOCS '97)*, page 364. IEEE, 1997.

[149] Young Hyun Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An Efficient Communication System with Strong Anonymity. In *Proceedings on Privacy Enhancing Technologies (PETS '16)*, 2016.

[150] David Lazar and Nickolai Zeldovich. Alpenhorn: Bootstrapping Secure Communication without Leaking Metadata. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, 2016.

[151] Sangmin Lee, Edmund L Wong, Deepak Goel, Mike Dahlin, and Vitaly Shmatikov. πBox: A Platform for Privacy-Preserving Apps. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, pages 501–514, 2013.

[152] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph Evolution: Densification and Shrinking Diameters. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):2, 2007.

[153] Jure Leskovec, Kevin Lang, Anirban Dasgupta, and Michael Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[154] Jinyuan Li, Maxwell Krohn, David Mazieres, and Dennis Shasha. SUNDR: Secure Untrusted Data Repository. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, 2004.

[155] Jialiu Lin, Shahriyar Amini, Jason I. Hong, Norman Sadeh, Janne Lindqvist, and Joy Zhang. Expectation and Purpose: Understanding Users' Mental Models of Mobile App Privacy Through Crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 501–510, New York, NY, USA, 2012. ACM.

[156] Kaisen Lin, David Chu James Mickens, Li Zhuang Feng Zhao, and Jian Qiu. Gibraltar: Exposing Hardware Devices to Web Pages using AJAX. In *Proceedings of the 3rd USENIX Conference on Web Application Development*, pages 7–7. USENIX Association, 2012.

[157] Derrell Lipman. *LIBERATED: a Fully In-browser Client and Server Web Application Debug and Test Environment.* PhD thesis, University of Massachusetts, 2011.

[158] Yabing Liu, Krishna P Gummadi, Balachander Krishnamurthy, and Alan Mislove. Analyzing Facebook Privacy Settings: User Expectations vs. Reality. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet Measurement Conference (IMC '11)*, pages 61–70. ACM, 2011.

[159] Jacob R Lorch, Bryan Parno, James Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: Ensuring Private Access to Large-Scale Data in the Data Center. In *11th USENIX Conference on File and Storage Technologies (FAST '13)*, pages 199–213, 2013.

[160] Wouter Lueks and Ian Goldberg. Sublinear Scaling for Multi-Client Private Information Retrieval. In *International Conference on Financial Cryptography and Data Security*, pages 168–186, 2015.

[161] Chao Luo, Yunsi Fei, Pei Luo, Saoni Mukherjee, and David Kaeli. Side-channel Power Analysis of a GPU AES Implementation. In *33rd IEEE International Conference on Computer Design (ICCD '15)*, pages 281–288. IEEE, 2015.

[162] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud Storage with Minimal Trust. *ACM Transactions on Computer Systems (TOCS)*, 29(4):12, 2011.

[163] Nick Mathewson and Roger Dingledine. Practical Traffic Analysis: Extending and Resisting Statistical Disclosure. In *Proceedings on Privacy Enhancing Technologies (PETS '04)*, pages 17–34, 2004.

[164] Russell A McClure and Ingolf H Krüger. SQL DOM: Compile Time Checking of Dynamic SQL Statements. In *27th International Conference on Software Engineering (ICSE '05)*, pages 88–96. IEEE, 2005.

[165] Susan E McGregor, Polina Charters, Tobin Holliday, and Franziska Roesner. Investigating the Computer Security Practices and Needs of Journalists. In *USENIX Security*, pages 399–414, 2015.

[166] Susan E McGregor, Franziska Roesner, and Kelly Caine. Individual versus Organizational Computer Security and Privacy Concerns in Journalism. *Proceedings on Privacy Enhancing Technologies (PETS '16)*, 2016(4):418–435, 2016.

[167] Frank McSherry and Kunal Talwar. Mechanism Design via Differential Privacy. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS '07)*, pages 94–103. IEEE, 2007.

[168] Frank D McSherry. Privacy Integrated Queries: an Extensible Platform for Privacy-Preserving Data Analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 19–30. ACM, 2009.

[169] James Mickens and Mohan Dhawan. Atlantis: Robust, Extensible Execution Environments for Web Applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 217–231. ACM, 2011.

[170] Microsoft Patterns and Practices. Code Review. http://msdn.microsoft.com/en-us/library/ff648637.aspx.

[171] Alexander Moshchuk, Tanya Bragin, Damien Deville, Steven Gribble, and Henry Levy. Spyproxy: Execution-based Detection of Malicious Web Content. In *16th USENIX Security Symposium*, number 3, pages 1–16. USENIX Association, 2007.

[172] Sape J Mullender, Guido Van Rossum, AS Tananbaum, Robbert Van Renesse, and Hans Van Staveren. Amoeba: A Distributed Operating System for the 1990s. *Computer*, 23(5):44–53, 1990.

[173] Daniel Myers, Jennifer Carlisle, James Cowling, and Barbara Liskov. MapJAX: Data Structure Abstractions for Asynchronous Web Applications. In *Proceedings of the 2007 USENIX Annual Technical Conference (ATC '07)*, 2007.

[174] National Vulnerability Database. Vulnerability Summary for CVE-2014-7235. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-7235` 2014.

[175] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference Attacks on Property-Preserving Encrypted Databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, pages 644–655. ACM, 2015.

[176] C Andrew Neff. Verifiable Mixing (Shuffling) of ElGamal Pairs. *VHTi Technical Document, VoteHere, Inc*, 2003.

[177] Edmund B Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, pages 221–234. ACM, 2009.

[178] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS '12)*, 2012.

[179] Femi Olumofin and Ian Goldberg. Revisiting the Computational Practicality of Private Information Retrieval. In *Financial Cryptography and Data Security*, pages 158–172. 2011.

[180] Opera Mediaworks. State of Mobile Advertising - 2015 Q2. `http://operamediaworks.com/innovation-and-insights/`

[181] Rafail Ostrovsky. Efficient Computation on Oblivious RAMs. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 514–523. ACM, 1990.

[182] Rafail Ostrovsky and Victor Shoup. Private Information Storage (Extended Abstract). In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 294–303, New York, NY, USA, 1997. ACM.

[183] John K Ousterhout, Andrew R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite Network Operating System. *Computer*, 21(2):23–36, 1988.

[184] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. In *European Symposium on Algorithms*, pages 121–133, 2001.

[185] Pascal Paillier et al. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Eurocrypt*, volume 99, pages 223–238. Springer, 1999.

[186] Ioannis Papagiannis, Matteo Migliavacca, and Peter Pietzuch. PHP Aspis: Using Partial Taint Tracking to Protect Against Injection Attacks. In *2nd USENIX Conference on Web Application Development*, page 13, 2011.

[187] Bryan Parno, Jonathan M McCune, Dan Wendlandt, David G Andersen, and Adrian Perrig. CLAMP: Practical Prevention of Large-Scale Data Leaks. In *30th IEEE Symposium on Security and Privacy (SP '09)*, pages 154–169. IEEE, 2009.

[188] Vern Paxson. Bro: a System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[189] Cam Pedersen and David Dahl. Crypton: Zero-Knowledge Application Framework, 2014.

[190] Ponemon Institute. 2013 Cost of a Data Breach Study: Global Analysis. `http://www.ponemon.org/`

[191] Ponemon Institute. 2017 Cost of a Data Breach Study. `http://www.ponemon.org/`

[192] Lucian Popa, Ali Ghodsi, and Ion Stoica. HTTP as the Narrow Waist of the Future Internet. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 6. ACM, 2010.

[193] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 85–100. ACM, 2011.

[194] Raluca Ada Popa, Emily Stark, Jonas Helfer, Steven Valdez, Nickolai Zeldovich, Frans Kaashoek, and Hari Balakrishnan. Building Web Applications on Top of Encrypted Data Using Mylar. In *USENIX Symposium of Networked Systems Design and Implementation (NSDI '14)*, 2014.

[195] K.S. Ramesh. Design and Development of MINIX Distributed Operating System. In *Proceedings of the ACM Sixteenth Annual Conference on Computer Science*, page 685. ACM, 1988.

[196] Ashwini Rao, Florian Schaub, Norman Sadeh, Alessandro Acquisti, and Ruogu Kang. Expecting the Unexpected: Understanding Mismatched Privacy Expectations Online. In *Symposium on Usable Privacy and Security (SOUPS)*, 2016.

[197] Michael G Reed, Paul F Syverson, and David M Goldschlag. Anonymous Connections and Onion Routing. *Selected Areas in Communications, IEEE Journal on*, 16(4):482–494, 1998.

[198] Charles Reis and Steven Gribble. Isolating Web Programs in Modern Browser Architectures. In *Proceedings of the 4th ACM European Conference on Computer Systems*, pages 219–232. ACM, 2009.

[199] Michael K Reiter and Aviel D Rubin. Crowds: Anonymity for Web Transactions. *ACM Transactions on Information and System Security (TISSEC)*, 1(1):66–92, 1998.

[200] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. Recon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 361–374. ACM, 2016.

[201] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. Constants Count: Practical Improvements to Oblivious RAM. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 415–430, 2015.

[202] David Richardson and Steven Gribble. Maverick: Providing Web Applications with Safe and Flexible Access to Local Devices. In *Proceedings of the 2011 USENIX Conference on Web Application Development (WebApps '11)*, volume 11, 2011.

[203] Ronald Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[204] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *LISA*, volume 99, pages 229–238, 1999.

[205] Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. Compact Ring-LWE Cryptoprocessor. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 371–391. Springer, 2014.

[206] Jerome Saltzer and Michael Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[207] Mike Samuel, Prateek Saxena, and Dawn Song. Context-sensitive Auto-Sanitization in Web Templating Languages using Type Qualifiers. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*, pages 587–600. ACM, 2011.

[208] Ismail San, Nuray At, Ibrahim Yakut, and Huseyin Polat. Efficient Paillier Cryptoprocessor for Privacy-Preserving Data Mining. *Security and Communication Networks*, 9(11):1535–1546, 2016.

[209] Nuno Santos, Rodrigo Rodrigues, Krishna P Gummadi, and Stefan Saroiu. Policy-sealed Data: A New Abstraction for Building Trusted Cloud Services. In *USENIX Security*, 2012.

[210] Len Sassaman, Bram Cohen, and Nick Mathewson. The Pynchon Gate: A Secure Method of Pseudonymous Mail Retrieval. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*, pages 1–9. ACM, 2005.

[211] Prateek Saxena, David Molnar, and Benjamin Livshits. SCRIPTGARD: Automatic Context-sensitive Sanitization for Large-scale Legacy Web Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*, pages 601–614. ACM, 2011.

[212] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy Data Analytics in the Cloud using SGX. In *IEEE Symposium on Security and Privacy (SP '15)*, pages 38–54. IEEE, 2015.

[213] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. *arXiv preprint arXiv:1702.08719*, 2017.

[214] Radu Sion and Bogdan Carbunar. On the Practicality of Private Information Retrieval. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS '07)*, 2007.

[215] Emin Gün Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B Schneider. Logical Attestation: an Authorization Architecture for Trustworthy Computing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 249–264. ACM, 2011.

[216] Emin Gün Sirer, Sharad Goel, Mark Robson, and Doan Engin. Eluding Carnivores: File Sharing with Strong Anonymity. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, page 19. ACM, 2004.

[217] Emil Stefanov and Elaine Shi. Multi-Cloud Oblivious Storage. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*, pages 247–258. ACM, 2013.

[218] Emil Stefanov and Elaine Shi. ObliviStore: High Performance Oblivious Cloud Storage. In *IEEE Symposium on Security and Privacy (SP '13)*, pages 253–267. IEEE, 2013.

[219] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an Extremely Simple Oblivious RAM Protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*, pages 299–310. ACM, 2013.

[220] Bryan Sullivan. Server-side JavaScript Injection. *Black Hat USA*, 2011.

[221] Chengzheng Sun and Clarence Ellis. Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, pages 59–68. ACM, 1998.

[222] Fangqi Sun, Liang Xu, and Zhendong Su. Static Detection of Access Control Vulnerabilities in Web Applications. In *USENIX Security Symposium*, 2011.

[223] Latanya Sweeney. k-anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, 2002.

[224] Jeff Terrace, Stephen Beard, and Naga Praveen Kumar Katta. JavaScript in JavaScript (js.js): Sandboxing Third-party Scripts. In *Proceedings of the USENIX Conference on Web Application Development*. USENIX Association, 2012.

[225] Bryce Thomas, Raja Jurdak, and Ian Atkinson. SPDYing up the Web. *Communications of the ACM*, 55(12):64–73, December 2012.

[226] Amin Tootoonchian, Stefan Saroiu, Yashar Ganjali, and Alec Wolman. Lockr: Better Privacy for Social Networks. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, pages 169–180. ACM, 2009.

[227] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. SoK: Secure Messaging. In *IEEE Symposium on Security and Privacy (SP '15)*, pages 232–249. IEEE, 2015.

[228] US Bureau of Justice Statistics. Victims of Identity Theft, 2014. `https://www.bjs.gov/`

[229] U.S.C. 18 U.S. Code 2705 - Delayed notice. Available at: `https://www.law.cornell.edu/uscode/text/18/2705`.

[230] U.S.C. 18 U.S. Code 2709 - Counterintelligence access to telephone toll and transactional records. Available at: `https://www.law.cornell.edu/uscode/text/18/2709`.

[231] U.S.C. 18 U.S. Code 3123 - Issuance of an order for a pen register or a trap and trace device. Available at: `https://www.law.cornell.edu/uscode/text/18/3123`.

[232] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, pages 137–152. ACM, 2015.

[233] Giovanni Vigna, William Robertson, Vishal Kher, and Richard A Kemmerer. A Stateful Intrusion Detection System for World-wide Web Servers. In *19th Annual Computer Security Applications Conference*. IEEE, 2003.

[234] Stephen T. Vinter and Richard E. Schantz. The Cronus Distributed Operating System. In *Proceedings of the 2nd Workshop on Making Distributed Systems Work*, EW 2, pages 1–3, New York, NY, USA, 1986. ACM.

[235] F. Wang, J. Mickens, N. Zeldovich, and V. Vaikuntanathan. Sieve: Cryptographically Enforced Access Control for User Data in Untrusted Clouds. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*, Santa Clara, CA, 2016.

[236] Helen J Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 1–16. ACM, 2007.

[237] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, pages 850–861. ACM, 2015.

[238] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. Demystifying Page Load Performance with WProf. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, pages 473–486. USENIX Association, 2013.

[239] Wang Wei. Flickr Vulnerable to SQL Injection and Remote Code Execution Flaws. `http://thehackernews.com/2014/04/flickr-vulnerable-to-sql-injection-and.html?m=1`

[240] Suzanne P. Weisband and Bruce A. Reinig. Managing User Perceptions of Email Privacy. *Communications of the ACM*, 38(12):40–47, December 1995.

[241] Andrew Whitaker, Marianne Shaw, and Steven Gribble. Scale and Performance in the Denali Isolation Kernel. *ACM SIGOPS Operating Systems Review*, 36(SI):195–209, 2002.

[242] James Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael Ernst, and Thomas Anderson. Verdi: A Framework for Formally Verifying Distributed System Implementations. In *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*, 2015.

[243] Peter Williams, Radu Sion, and Alin Tomescu. PrivateFS: A Parallel Oblivious File System. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*, pages 977–988. ACM, 2012.

[244] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos Operating System. *ACM Transactions on Computer Systems (TOCS '94)*, 12(1):3–32, 1994.

[245] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in Numbers: Making Strong Anonymity Scale. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 179–182, 2012.

[246] David Isaac Wolinsky, Ewa Syta, and Bryan Ford. Hang with your Buddies to Resist Intersection Attacks. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*, pages 1153–1166. ACM, 2013.

[247] 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 '06)*, pages 263–278, 2006.

[248] Kehuan Zhang, Zhou Li, Rui Wang, XiaoFeng Wang, and Shuo Chen. Sidebuster: Automated Detection and Quantification of Side-channel Leaks in Web Application Development. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*, pages 595–606. ACM, 2010.

[249] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, pages 283–298, 2017.