Regaining Control over Cloud and Mobile Data

Roxana Geambasu

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

Doctor of Philosophy

University of Washington

2011

Program Authorized to Offer Degree: Computer Science and Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Roxana Geambasu

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Co-Chairs of the Supervisory Committee:

_____

Steven D. Gribble

_____

Tadayoshi Kohno

_____

Henry M. Levy

Reading Committee:

_____

Steven D. Gribble

_____

Tadayoshi Kohno

_____

Henry M. Levy

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Proquest Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, 1-800-521-0600, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

Regaining Control over Cloud and Mobile Data

Roxana Geambasu

Co-Chairs of the Supervisory Committee:
Associate Professor Steven D. Gribble
Computer Science and Engineering Department

Associate Professor Tadayoshi Kohno
Computer Science and Engineering Department

Professor Henry M. Levy
Computer Science and Engineering Department

While emerging computing technologies – such as cloud computing and small, powerful, mobile devices – offer previously unimaginable global access to data and applications, they also threaten users' sense of control over data ownership, distribution, and properties. For example, uploading some data to a Web service – such as a document to Google Docs, a photo to Facebook, or an email to Hotmail – causes the user to lose control over that data. The user cannot ensure that the service deletes all copies of her data when she asks it to do so, that her data is not shared with advertisers, or that her data is replicated sufficiently to ensure its long-term availability. Similarly, storing data on a mobile device causes the user to lose control over that data when the device is stolen or lost; the user cannot ensure that the data can never be compromised and she cannot tell whether it has been compromised.

This dissertation examines the broad data security, privacy, and management challenges raised by modern technology and proposes a set of techniques that address these issues. We present four systems, each aiming to re-empower users with a specific aspect of their lost data control. Keypad offers remote access control and auditability for data stored on a stolen device. Vanish provides control over the lifetime of data stored in untrusted clouds. Comet lets clients customize the functionality of trusted cloud storage, while Menagerie provides a uniform view of a user's scattered

Web data. We present the design, implementation, and detailed evaluation for each of the four systems, demonstrating the feasibility of our approaches.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I extend special thanks to my advisors, Prof. Steven D. Gribble, Prof. Tadayoshi Kohno, and Prof. Henry M. Levy, who have supported and guided all aspects of my doctoral career. I also thank Prof. Magdalena Balazinska, Dr. Andrew Birrell, and Prof. Arvind Krishnamurthy for fruitful collaboration and valuable advice throughout my Ph.D. program. Parts of this dissertation were done in collaboration with Cherie Chung, John P. John, Paul Gardner, Amit Levy, Vinnie Moscaritolo, and Alexander Moshchuk, whom I thank for their collaboration. I also extend thanks to Peter Hornyack, Tomas Isdal, Ben Lerner, Michael Piatek, Charles Reis, David Richardson, and Mark Zbikowski for frequent feedback, advice, and collaboration. Finally, I want to help Dr. Lindsay Michimoto for her guidance throughout the program. This work was supported by the First Google Fellowship in Cloud Computing and NSF grants CNS-0132817, CNS-0614975, CNS-0430477, and CNS-0627367.

DEDICATION

To my husband, Bogdan, for his continued support of my education and career.

Chapter 1

INTRODUCTION

Two significant computer advances – cloud technology and powerful mobile devices – are revolutionizing how users work and the degree of control they experience over their data. Cloud computing is catalyzing a migration from in-house infrastructure and applications to remote public infrastructure and Web services; as a result, personal desktop applications are yielding to Web services – such as Google Docs and Photoshop Express – and IT-managed enterprise applications and datacenters to cloud-based solutions – such as Amazon AWS. Further, robust and usable small-form portable devices are encouraging users to adopt increasingly mobile computing habits. For example, travelers can carry gigabytes of data on their laptops and USB sticks, access calendars and emails on phones, and read documents with e-readers.

While cloud and mobile computing enable pervasive data access and large-scale sharing, they also threaten users' sense of control over their data's security, privacy, and management. For example, when sending an email over Hotmail, posting a photo to Facebook, or uploading a document on Google Docs, users cede all data control to the Web service. The service can decide to retain the data for months after the user requests its deletion, to share it with others for monetary or legal reasons, or to replicate it on servers outside the user's jurisdiction. However detrimental, such practices are commonplace today [38, 140, 184, 219]. Similarly, losing a mobile device means users cannot securely erase their data, prevent access to it by thieves, or identify potentially compromised data.

This dissertation examines the data security, privacy, and management challenges created by clouds and new mobile technologies, and proposes novel techniques to *re-empower users with control over their data stored on them*. By analyzing new cloud and mobile technologies, we identify three root causes for the lost data control: (1) the untrustworthy and possibly hostile computing environments in public clouds and on stolen mobile devices, (2) the lack of customizability that characterizes these environments, and (3) data dissemination across many clouds and devices. These properties challenge users' ability to perform many tasks, such as obtaining an audit log of all

accesses to data, securely erasing sensitive data, customizing various data-management properties (such as where a service should replicate data), and organizing personal data meaningfully when it is scattered across many services and devices.

To address these challenges, the thesis presents the design, implementation, and evaluation of four systems, each focused on restoring one aspect of control lost to the cloud or mobile devices. Keypad [79], an encrypted file system for theft-prone devices, allows post-theft remote data access control and fine-grained access auditing. Vanish [81], a distributed-trust self-destructing data system, lets users control the lifespan of their Web data, such as Hotmail emails, Facebook messages, or Google documents. Comet [82], an extensible key/value store, helps clients customize how their data is managed in a cloud storage service, such as Amazon S3 or peer-to-peer, Internet-wide distributed hashtables. Finally, Menagerie [77] integrates a user's scattered Web data into a uniform naming, protection, and access system.

The systems when considered as a group demonstrate that various aspects of control can be recovered without sacrificing the advantages of new technologies. Doing so involves a combination of techniques from multiple computing fields – including operating systems, distributed systems, and cryptography – to address the complex challenges raised by modern technology. For example, our use of cryptography is unconventional: while encryption has traditionally been used to protect data confidentiality, we use it to force remote access auditing in Keypad and assured data deletion in Vanish.

The thesis contains six additional chapters. Chapters 2–5 review each system's motivation, related work, design, and evaluation. Chapter 6 presents future directions, while Chapter 7 concludes. The remainder of this chapter describes the broad problem space created by new technologies, as an overarching motivation for our work.

1.1 Modern Technologies

In the past, users relied on in-house systems, such as desktop PCs and private datacenters, for most of their application and storage needs. Today, the traditional in-house computing world is radically evolving in two directions: much of the computing and storage is moving onto giant-scale public

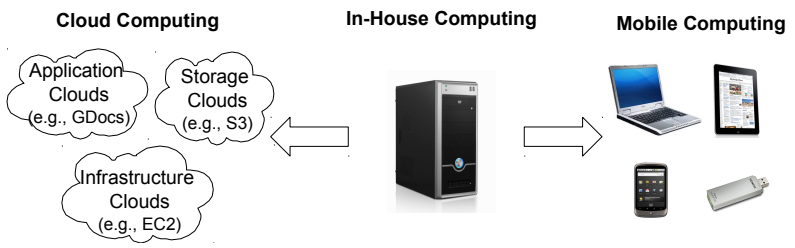


Figure 1.1: **Technological Trends.** Computing technology is transitioning from in-house environments, such as desktop PCs, to public clouds and portable mobile devices.

clouds and the rest is directed to tiny, yet powerful, mobile devices, as shown in Figure 1.1. We describe each transition next.

Cloud Computing. The transition to cloud computing is being fostered by *application clouds*, *storage clouds*, and *infrastructure clouds*. Application clouds provide software-as-a-service solutions, which are quickly replacing in-house applications. Examples include Google Docs, Photoshop Express, and Quicken Online for individual users, and Salesforce.com, Google Apps, and Gmail for enterprises. Storage clouds – such as Amazon S3, Mozy, and Dropbox – are replacing in-house storage and backup solutions for both individuals and enterprises [63, 192]. Finally, infrastructure clouds – such as Amazon's EC2, Google's AppEngine, and Microsoft's Azure – are attracting many enterprises [105, 190] and promise to become the Web's preferred deployment model [78].

Public clouds offer many advantages over in-house computing environments. Users can access their data and applications through multiple devices from anywhere in the world and can avoid the pains of installing and managing software. Similarly, enterprises can significantly diminish their hardware and software maintenance costs, avoid over-provisioning, and scale their services based on load. Finally, public clouds greatly facilitate sharing and collaboration across users all around the world.

Powerful Mobile Devices. Desktop computers in homes and offices are giving way to small-form mobile devices that can be carried everywhere: at home, to work, on the street, and on trips around the world. This transition is enabled by recent advances in mobile technology that have endowed pocket-sized devices with computing and communication capabilities similar to the desktops of only

a few years ago. For example, laptops have tremendous computing power and run unmodified desktop applications, while tablets and phones are sufficiently powerful to run trimmed-down versions of desktop operating systems and popular applications, such as Web browsers, calendars, email, and even document editors for the iPad.

Mobile devices offer important advantages. Users can increase their on-the-go productivity using laptops, netbooks, tablets, and phones, and they can easily transport and share large amounts of information using USB sticks. Unfortunately, mobile devices also pose significant challenges, e.g., their small form and mobile nature make them highly susceptible to theft or loss: one in ten laptops is lost or stolen within a year of purchase [142] and dry cleaners in the U.K. found over 4,000 USB sticks in pockets in 2009 [193]. The loss of devices compromises the privacy of the data stored on them, a topic that we address below.

1.2 The Problem: Losing Data Control

Despite their advantages then, clouds and mobile devices challenge users' ability to control data security, privacy, and management. We now provide a broad view of this problem space to establish the context for the rest of the thesis. To do so, we use a simple scenario to expose the various forms of lost data control and the root causes for such losses. Section 1.3 then restricts the scope to the specific challenges addressed in this thesis and summarizes our contributions.

1.2.1 An Example Scenario

Ann, a relatively tech-savvy person, owns a growing business commercializing office supplies. In the past, Ann handled all of her company-related computing using her desktop PC.[1] She edited customer contracts and other documents using Microsoft Office, kept her books with Quickbooks, backed her data onto external Seagate disk drives, and ran Apache to publish her company's Web site. Over time, these self-managed solutions became cumbersome and unsupportive of Ann's expanding travel schedule and data-sharing needs. Therefore, she switched to the new cloud-and-mobile world, where she uses Google Docs to collaborate with her employees, Netsuite to keep her books, Mozy to back up new data, and Amazon EC2 to host her Web site. Moreover, Ann uses

[1] We use a single-computer scenario for simplicity, but similar conclusions can be reached for private datacenters.

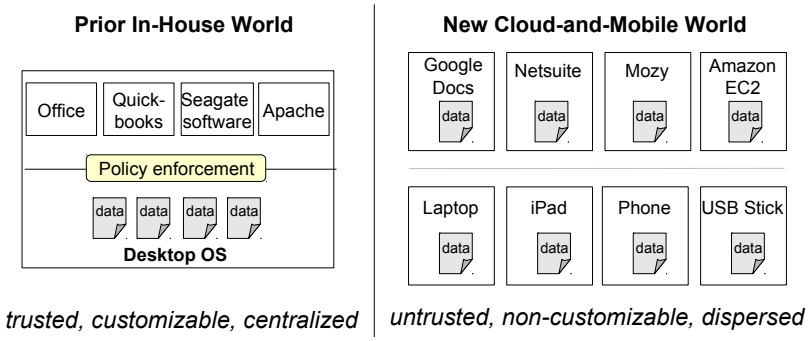


Figure 1.2: **Example Scenario.** Compares the in-house computing environment to the cloud and mobile environments. Cloud and mobile technologies fragment and replicate data on scores of untrusted or non-customizable locations.

mobile devices to interact with her data when traveling and replicates some customer data on those devices for disconnected use (e.g., on planes). See Figure 1.2.

Ann is vigilant about protecting her data's security and privacy. For example, she wants to ensure that her customer and financial data are kept private at all times, yet she wants to collaborate on a subset of documents with her employee, Bob. She also wants her data to remain persistent and unaltered for years or decades, but she seeks its immediate deletion upon request to avoid future liabilities or exposure.

In the past, Ann could achieve all her goals, albeit with some effort. She protected her data against unauthorized accesses by locking her server in a monitored room and installing a variety of security tools, such as firewalls, antivirus programs, and network intrusion detection systems. She shared her documents only with Bob, using the file system's access control mechanisms. She enabled fine-grained auditing to monitor all data accesses, installed a file system extension to securely erase data upon request, and customized her backup application to use high levels of replication to ensure long-term data persistence.

Ann would like similar controls over data security, privacy, and management in the new cloud-and-mobile world. However, getting them is challenging for reasons we now describe.

1.2.2 Shortcomings of the Cloud and Mobile World

The preceding scenario highlights important properties of the in-house computing environment that allowed Ann to achieve her data security goals. First, it is *trusted*. For example, when all of her data, applications, and infrastructure were on-premise, Ann could gain trust in her system by deploying physical and software security mechanisms (e.g., firewalls). Second, Ann could *flexibly customize* her environment for her specific needs by adding new components (e.g., the secure-deletion and auditing file system modules) or tailoring existing ones (e.g., her backup system). Third, all of Ann's data was stored in a single location – the desktop PC – which allowed the *uniform installation of specific policies* (e.g., secure deletion, auditing, replication) *across all her data*.

Unfortunately, the cloud-and-mobile world exhibits opposite properties, which challenge Ann's security goals:

- *Untrusted computing environments:* Cloud computing and mobile devices physically migrate storage, applications, and infrastructure from private premises to public, insecure grounds. Users either outsource their data to untrusted clouds or take their mobile devices into places where they can be easily stolen or lost. This off-premise data migration raises significant trust concerns. For example, Ann's cloud services could release her data for monetary or legal reasons or preserve it beyond its intended lifespan. Similarly, a stolen mobile device could leak confidential information after being physically compromised by a thief.

- *Inflexible to customization:* Trusted or not, today's cloud services and mobile devices provide narrow interfaces and one-size-fits-all semantics. For example, Ann may require access auditing, assured deletion, and high persistence guarantees for her company documents, which services such as Google Docs does not support. Similarly, mobile phones require jailbreaking to impose low-level policies (e.g., secure deletion).

- *Fragmented data dispersion:* Cloud computing and mobile devices fragment and replicate data across many Web services, intermediaries, and mobile devices. For example, Ann's financial documents are stored on Netsuite, Mozy, and a laptop; her customer contracts are on Google Docs, Mozy, and an iPad; and her product photos are being replicated by Facebook and Akamai. Each of these data-storage locations has its own security, privacy, and management policies. How can Ann enforce uniform access control policies on a photo with copies

Form of Data Control Loss	Approaches for Recovering Control
1. Cloud data confidentiality	[95, 83, 191, 18, 181, 43, 175, 177, 111]
2. Mobile data confidentiality	[130, 153, 159, 3, 47, 199]
3. Controlled data sharing	[156, 206]
4. **Data access auditing**	**Keypad**, [178]
5. Access privacy	[40, 41]
6. Data availability	[87]
7. Data long-term persistence	[104, 30, 31]
8. **Assured data deletion**	**Vanish**, [138, 148, 149, 203, 27]
9. **Data property customization**	**Comet**
10. **Data organization and search**	**Menagerie**, [147]

Table 1.1: **Data Control Challenges.** Specific kinds of data control that users have ceded to the cloud and mobile technologies and references to known approaches for recovery. Neither the problem set nor solution lists are meant to be complete. Bolded challenges are addressed in this thesis.

on Facebook, Picasa, and Mozy? How can she delete a document when copies of it are stored in many, possibly unknown, email services?

1.2.3 Data Control Challenges, Potential Solutions, and Their Limitations

Table 1.1 shows specific data control challenges created by the lack of trust, customizability, and uniformity in today's cloud and mobile environments. The space of all possible challenges is enormous, and this dissertation contributes techniques that address only a subset of them, as indicated in the figure. For reference, the figure also includes citations for example techniques known to address each challenge. A comprehensive solution list for challenges not addressed in this thesis is beyond our scope; for the specific challenges addressed here, detailed related work discussions are deferred to the respective chapters.

While the majority of identified problems have been addressed, we find that known techniques are not complete solutions to any of the problems, as described below:

1. *Cloud data confidentiality.* As noted, protecting cloud data from unauthorized parties is challenging. A malicious or buggy service may reveal information to unintended parties,

such as advertisers, legal authorities, and other co-residents. For example, a recent glitch in DropBox's authentication logic allowed any user to log in and access the data of any other user [127], and Amazon EC2 instances have been found vulnerable to timing attacks by co-residents [167]. Techniques to improve confidentiality include: different forms of encryption with varied applicability (e.g., traditional encryption [95], searchable encryption [191], and fully-homomorphic encryption [83]), distributing data and computation across multiple services using threshold cryptography (e.g., secure multi-party computation [18] and threshold secret sharing [181] for securely distributing computation and data, respectively), and using trusted hardware to bootstrap greater assurance in the cloud [43, 111, 175, 177]. Unfortunately, such techniques are not complete solutions. For example, even the most generally applicable encryption scheme – fully-homomorphic encryption, a generic primitive that allows the construction of services that operate on encrypted data – has been proven to have limited applicability [211]. Similarly, trusted-hardware-based solutions require deployment of special hardware and an identifiable trusted computing base in all clouds operating on the user's data. In today's data-dispersed world, where data is forwarded and replicated across many services and intermediaries (such as content delivery networks and Facebook applications), meeting this requirement is difficult.

2. *Mobile data confidentiality.* Maintaining confidentiality on mobile devices proves challenging, because the devices can be easily stolen or lost. Techniques to increase mobile data confidentiality include encryption [130, 153, 159], using advanced authentication mechanisms [3, 37, 47, 91, 102, 132, 199], and disabling data accesses after device loss is detected [11, 97]. As will be shown, all of these data-protection techniques can and do fail in the real world for a variety of reasons, including poor user policy configuration [224] and physical attacks [90, 169]. We address these limitations by complementing data confidentiality protection with data access auditing in Keypad.
3. *Controlled data sharing.* Related to confidentiality, fine-grained, controlled sharing also challenges today's inflexible Web services. For example, Ann may wish to restrict access to her Facebook photos to only a subset of her friends, but Facebook's access control scheme is too

coarse for her needs.[2] For example, what if Ann wants to share her data only with users located in the U.S.? Cryptographic techniques, such as attribute-based encryption [156], have been developed to provide users with flexible access control despite inflexible cloud services.

4. *Data access auditing.* Ann may want to obtain a log of all accesses to her data. For example, she could be legally required to maintain an audit log of all accesses to customer information stored on various services and devices. Unfortunately, implementing this policy is challenging when her customer data is scattered across many untrusted online services and employee laptops. One partial solution is secure logging [178], which lets users recognize a tampered log provided they have access to that log. In many cases, however, simply recognizing tampering may be insufficient (e.g., Ann may need to know exactly what was accessed even if the thief has tampered with the log), and obtaining the log may be impossible (e.g., Ann may never recover her stolen device). We therefore developed Keypad, a file system for mobile devices that guarantees remotely accessible, accurate logs of data accesses on a stolen device *without* assuming device recovery or connectivity post-theft.
5. *Access privacy.* Keeping various user actions private from "Big Brother" services can be daunting. For example, Google can track when Ann works on her online documents, her location, and with whom she collaborates. Private information retrieval (PIR) [40, 41] alleviates this problem by letting users request data from a database without revealing what data they requested. However, despite years of research, PIR remains extremely heavyweight [187].
6. *Data availability.* While availability is often cited as a cloud advantage, a cloud's failure can be catastrophic because it affects many users of potentially critical applications and individual users cannot influence its recovery. Examples of cloud blackouts and their vast impacts on businesses litter the news [13, 117, 121, 223]. Some have argued that cross-cloud redundancy is needed to cope with massive cloud failures [87], though no such technical solutions exist today.
7. *Data persistence and integrity.* Long-term persistence and integrity pose complex challenges in the cloud. The cloud provider can change its policies over time, decide to drop long-unaccessed data to save on storage costs, or go out of business altogether. Techniques to

[2] In response, Google+ recently introduced circle-based access control [206], which allows more fine-grained, configurable access control, though it is still unclear that this new scheme supports all possible user needs.

address such problems include proofs of recoverability [30, 104], which let an auditor efficiently monitor the availability and integrity of cloud data without having to download it, and black-box monitoring mechanisms for checking that a data object is sufficiently replicated in the cloud [31].

8. *Data deletion.* Data destruction is as crucial as persistence [122, 144]. Unfortunately, deletion is challenging in a world where data is replicated across many devices and services that fail to guarantee deletion upon request. For example, Facebook does not erase data promptly after a user request [38], nor in some cases does it do so after users close their accounts [189]. Techniques for ensuring data deletion rely on encrypting data and destroying the key at deletion time [27, 138, 148, 149, 203]. The common limitation among known techniques is their reliance on trusted centralized key services to delete data, which might raise significant concerns for users: if users do not trust Facebook or Google to delete their data, why would they trust a key service to do so? To alleviate trust assumptions, we built Vanish, a distributed-trust, self-destructing system that combines threshold secret sharing with large-scale distributed systems.
9. *Data property customization control.* As argued in our simple scenario, some users may need more specific control over their data's management. For example, Ann wants her financial data to be stored only on US servers, her irreplaceable documents to always be replicated on at least seven servers, and accesses to her health records to be tracked in a forensic log. Support for such customization is rare given today's inflexible Web services. We therefore developed Comet, a system that provides extensibility to a specific type of cloud service – a key/value storage cloud.
10. *Data organization and search.* Organizing data meaningfully is difficult when it is scattered across many services and devices. For example, how can Ann find all client data when it is in emails on Gmail, contracts on Google Docs, and financial records on Netsuite? We present Menagerie, a system that provides uniform naming, protection and access interfaces to all user data, thereby letting users to organize their scattered data into heterogeneous collections that can be shared with friends and family.

1.3 Contributions

Through our work building systems to address some key difficulties encountered when using cloud and mobile device environments, we have contributed many new techniques and insights. We now briefly overview the context and motivation for our work on each system and present our specific contributions to the goal of empowering users with control over their cloud and mobile data.

Keypad: Remote Access Control and Auditing for Stolen Devices. Traditional data protection tools, such as encrypted file systems [70, 130], are insufficient to guarantee data confidentiality on a stolen mobile device. For example, many users configure poor passwords; they write them down on sticky notes; and they co-locate their smartcards with their mobile devices. Keypad enhances current encrypted file systems through auditing and remote access control. First, Keypad's fine-grained file access auditing lets users obtain *explicit evidence* of whether any files have been accessed *after* a device is lost. Second, users can disable future file accesses after realizing that a device is missing, even in the *absence* of device network connectivity. Keypad's main contribution is to combine encryption and remote key management to weave these two properties into the file object itself. By encrypting files locally but storing decryption keys remotely, Keypad *requires* the involvement of an audit server with every file access. By alerting the audit server to refuse to return a particular file's key, the user can prevent post-theft accesses. This problem space raises difficult performance, privacy, and disconnection challenges. Keypad overcomes these challenges through novel combinations of techniques from distributed systems (key caching, prefetching, and paired architectures) and cryptography (identity-based encryption). Overall, Keypad demonstrates that, perhaps unintuitively, it is possible for a user to retain control over her data on a mobile device even *after* she loses control over the physical device. Chapter 2 details Keypad's design.

Vanish: Lifetime Control with Self-destructing Data. To address users' inability to delete their Web data, we built Vanish, a system that lets users control the lifetime of their data stored on untrusted Web services by providing a new abstraction, called *self-destructing data* [81, 80]. With self-destructing data, users can time-limit Web data by specifying a timeout for it before uploading it to the cloud. After the timeout, all copies of the self-destructing data object will become forever unreadable without any explicit user action and regardless of whether the Web services cooperate. To create the self-destruction mechanism without having to trust any centralized services, Vanish

encrypts the data with a symmetric key, breaks the key into pieces, and scatters the pieces across nodes in a large-scale, distributed-trust, peer-to-peer system. The nodes are designed to independently erase key pieces after a specified time. Until the timeout, data can be decrypted by retrieving the key pieces and reconstructing the key. After the timeout, however, key pieces will have disappeared, rendering the data forever undecryptable. Chapter 3 describes Vanish, our experience with building it on top of an existing commercial, million-node, peer-to-peer system – the Vuze distributed hash table (DHT) – and subsequent redesigns of the DHT to better accommodate the Vanish goals.

Comet: Cloud Storage Customization with Active Storage Objects. Whether trusted or not, today's provide users with very little control over their data. We built *Comet*, an extensible storage service, to support user-specific customization of trusted cloud storage services by providing a new abstraction, called an *active storage object* (ASO) [82]. Using Comet, clients store ASOs into the service instead of passive data. An ASO contains a piece of code that controls the object's behavior inside the storage system, in addition to the data. For example, the active code in a Comet ASO can control the data's lifetime, its replication scheme, or its access control policies. An ASO can also perform its own application-specified filtering or forensic logging. In Comet, we built upon extensible systems concepts and language sandboxing mechanisms to create a secure, customizable storage service. Chapter 4 describes Comet's design, which can be viewed as a case study for how to build extensibility into today's inflexible Web services.

Menagerie: Unified Web-data Control. To address the data management challenges created by the data's dispersion across the Web, we describe Menagerie, a system that provides users with uniform control over their Web data [77]. Today, it is extremely difficult to gather all of the data related to a recent trip (e.g., photos, videos, documents, and so on), archive it into a uniform collection, and share that collection with friends and family: each Web service presents its own data access interface and protection scheme, which thwarts integration. With Menagerie, users can build heterogeneous collections of their Web objects, share these collections with other users, and process them uniformly with various applications. Menagerie's key contribution is to impose a uniform naming, protection, and access interface, which helps create generic applications for personal data organization on the Web. Menagerie's design is detailed in Chapter 5.

Overall Contribution. At the highest level, our work demonstrates that carefully designed abstractions and mechanisms can help recover some of the lost data control without sacrificing the advantages of new technologies. Each contributed system should be considered as a case study for how to recover one specific aspect of data control.

A useful mechanism for recovering control is the introduction of *self-managing data abstractions* – such as Vanish's self-destructing data abstraction, Comet's active storage objects, and Keypad's audited files – in which *the data object itself is extended to provide desired data control properties*. Traditionally, data management policies, such as data retention, replication, and forensics policies, are implemented as properties of the system storing that data, such as the file system on a mobile device or a Web service's persistent data store. With self-managing data, such policies are organically built into the data object itself and do not depend on the storage system's explicit support. For example, self-destructing data objects disappear on their own after a pre-specified time regardless of service-side cooperation; active storage objects encapsulate arbitrary policies of how the data is handled in a storage service; and with audited files, remote access control and auditing are properties guaranteed by the data object itself and not by the software running on the device. Our experience shows that self-managing data abstractions are an effective approach for regaining control over various aspects of the users' data when it is stored on untrusted, inflexible, or non-uniform locations, such as cloud services and mobile devices.

Chapter 2

KEYPAD: REMOTE ACCESS CONTROL AND AUDITING FOR THEFT-PRONE DEVICES

This chapter presents Keypad, an auditing file system for theft-prone devices, such as laptops, tablets, and USB sticks. Keypad was originally described in a 2011 paper [79]. We begin by providing a high-level overview of Keypad's motivation and architecture.

2.1 Motivation and Overview

As described in Chapter 1, mobile devices, such as laptops, tablets, and USB memory sticks, create not only great advantages but also significant risks due to their susceptibility to theft and loss. The loss of such devices is most concerning for organizations and individuals storing confidential information, such as medical records, social security numbers (SSNs), and banking information.

Conventional wisdom suggests that standard encryption systems, such as BitLocker [130], PGP Whole Disk Encryption [153], and TrueCrypt [70], can protect confidential information. However, encryption alone is sometimes insufficient to meet users' needs. Two reasons are relevant for this discussion. First, traditional encryption systems can and do fail in the world of real users. As described in the seminal paper "Why Johnny Can't Encrypt" [224], security and usability are often at odds. Users find it difficult to create, remember, and manage passphrases or keys. As an example, a password-protected USB stick containing private medical information about prison inmates was lost along with a sticky note revealing its password [176]. Encrypted file systems often rely on a locally stored key that is protected by a user's passphrase. User passphrases are known to be insecure; a recent study of consumer Web passwords found the most common one to be "123456" [96]. Finally, in the hands of a motivated data thief, devices are open to physical attacks on memory or cold-boot attacks [90] to retrieve passphrases or keys. Even physical attacks on TPMs and "tamper-resistant" hardware are possible [7, 169].

Second, when encryption fails, it fails *silently*; an attacker might circumvent the encryption

without the data owner ever learning of the access. The use of conventional encryption can therefore lead mobile device owners into a false sense of protection. For example, a hospital losing a laptop with encrypted patient information might not notify patients of its loss, even if the party finding the device has circumvented the encryption and accessed that information.

This chapter presents the design, implementation, and evaluation of *Keypad*, a file system for loss- and theft-prone mobile devices that addresses these concerns. The principal goal of Keypad is to provide *explicit evidence* that protected data in a lost device either has or has not been exposed after loss. Specifically, someone who obtains a lost or stolen Keypad device *cannot* read or write files on the device without triggering the creation of access log entries on a remote server. This property holds even if the person finding the device also finds a note with the device's encryption password.

Keypad's forensic logs are detailed and fine grained. For example, a curious individual who finds a laptop at the coffee shop and seeks to learn its owner might register audit records for files in the home directory, but not for unaccessed confidential medical records also stored on the device. However, the professional data thief will register accesses to all of the specific confidential medical files that they view. Furthermore, Keypad lets device owners disable access to files on the mobile devices once they realize their devices have been lost or stolen, even if the devices have no network connectivity, such as USB memory sticks (in contrast to systems like Apple's MobileMe).

Keypad's basic technique is simple yet powerful: it tightly entangles the process of file access with logging on a *remote auditing server*. To do this, Keypad encrypts protected files with *file-specific* keys whose corresponding decryption keys are located on the server. Users never learn Keypad's decryption keys and thus they cannot choose weak passwords or accidentally reveal them; it is therefore computationally infeasible for an attacker to decrypt a file without leaving evidence in the log. When a file operation is invoked, Keypad logs the file operation remotely, temporarily downloads the key to access the file, and securely erases it shortly thereafter. Keypad is implemented on top of a traditional encrypted file system; obviously users should choose strong passwords (or use secure tokens, etc.) for that underlying file system, but Keypad provides a robust forensic trail of files accessed even if users choose weak passwords or the traditional system's keys are otherwise compromised.

While conceptually simple, making this vision practical presents significant technical challenges

and difficult tradeoffs. For example, neither the user nor Keypad can predict when a device will be lost or stolen. As a result, the system must provide both an accurate fine-grained forensic record, which is critical after loss, and acceptable performance, which is critical prior to loss.

The tension between performance and forensics is pervasive. As an example, consider the creation of a file. For forensic purposes, a naïve Keypad architecture might first pre-register newly created files and their corresponding keys with the remote server prior to writing any new data to those files. However, pre-registration would incur at least one full network round-trip, which could be problematic for some workloads over slow mobile networks, such as 3G or 4G. Delaying the registration is an obvious optimization, yet doing so would leave a loophole that a device thief could exploit to access files without triggering a log entry in the remote server. Overall, our experience demonstrates that we can achieve both forensic fidelity and acceptable performance by combining conventional systems techniques with techniques from cryptography, including identity-based encryption [26, 182].

The remainder of this chapter is organized as follows. We begin with a list of motivating examples and goals in the remainder of this section. Keypad's architecture is presented in Section 2.3 and its implementation in Section 2.4. Section 2.5 provides a detailed evaluation of our prototype and Section 2.6 discusses its security. Section 2.7 reviews related work and we conclude in Section 2.8.

2.1.1 Example Scenarios

Keypad is designed to increase assurances offered to owners of lost or stolen mobile devices. The mobile devices might have computational capabilities (e.g., laptops and phones) or might be simple storage devices (e.g., USB sticks). We view Keypad as particularly valuable to users storing personal or corporate documents, banking information, SSNs, medical records, and other highly sensitive data.

To illustrate the need for Keypad, we extend the simple example scenario defined in Section 1.2.1. Bob, an employee in Ann's company, carries a company laptop that stores documents containing trade secrets and sensitive customer information (such as credit card numbers). The company's IT department installs Keypad on the laptop, configuring it to track all accesses to files in her "company documents" folder. After returning to his hotel from a two-hour dinner, Bob notices

that her laptop is missing. He immediately reports the loss to the IT department, which disables any future access to files in the "company documents" folder. The IT department also produces an audit log of all files accessed within the two-hour window since he last controlled his laptop, confirming that no sensitive files were accessed. If needed, Ann, the business owner, can instruct the legal department to use that confirmation in court to prove that the company laptop's loss has led to no customer data disclosures.

As a second example, at tax preparation time, Charlie scans all of her personal tax documents, places them on a USB stick, encrypts it with a password, and physically hands the stick and password to his accountant. A few weeks later, Charlie can no longer find his thumb drive and can't remember whether his accountant kept it or whether he lost it in the intervening weeks. Fortunately, Charlie's stick was monitored with Keypad and Charlie uses a Web service provided by his drive manufacturer to view an audit log of all accesses to the drive. He sees that there were many accesses to his tax files over the previous week and he learns the IP addresses from which those accesses were made. Charlie therefore places fraud alerts on his financial accounts and notifies the appropriate authorities.

In these scenarios, users benefit from additional advantages that Keypad has over traditional encrypted file systems. First, Keypad provides highly accurate, remotely readable forensic records of which files were accessed post-loss. If a file does not appear in those records, that suggests that no one accessed the file after device loss; if a file does appear in those records, this suggests that data was accessed and the owner should take appropriate mitigating actions. Second, by preventing key access, Keypad can prevent adversaries from accessing protected files post-loss, even in the absence of network connectivity, e.g., for a disconnected USB stick or an extracted laptop hard drive.

2.2 Goals and Assumptions

Figure 2.1 shows a high-level timeline of three periods in the life of a lost or stolen device, along with the properties the user requires during each. First is the normal use period during which the user has control of her device. The user loses control of the device at T_{loss}; however, the user may not know exactly when this occurs, so she must consider T_{loss} to be the last point at which she remembers having control. T_{notice} is the time at which the user realizes that she has lost her device,

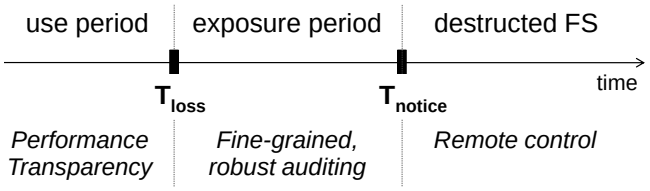


Figure 2.1: **Timeline of Theft/Loss.** This timeline shows the two critical events during the lifetime of a device: the device loss and the user noticing that the device has been lost. For each period, we enumerate the Keypad properties that matter in that period.

at which point she should take action. In our Alice scenario, the exposure period (T_{loss} to T_{notice}) is the full two-hour dinner window.

Our primary Keypad goal is to provide strong *audit security*. If an adversary gains control of a device and accesses a Keypad-protected file, at least one audit log entry should be produced on a remote audit server. Further, the adversary cannot tamper with the contents of the audit log or otherwise make it unavailable to the victim. Specifically, our goals are:

- *Robust auditing semantics:* Keypad must provide robust semantics by preventing unrecorded file accesses. To achieve this, the remote auditing server must observe data and metadata operations performed on the client.
- *Performance:* File access latency and throughput should be acceptable for Keypad-protected data. We mainly target office productivity and mobile workloads, rather than server- or engineering-oriented workloads. We also assume multiple network environments: at the office (LANs), at home (broadband), and on the road (3G or 4G). We seek minimal overhead at work or home, but will tolerate some increased latency in challenging mobile environments in exchange for Keypad's properties.
- *Fine granularity:* Keypad should produce detailed access logs of read and write accesses to individual Keypad-protected files. Administrators can control the granularity and coverage of these logs; e.g., configuring Keypad to produce audit logs for an entire file system or only for specific files identified as sensitive.
- *User transparency:* We assume that users are not technically sophisticated; therefore, Keypad's

operation should be largely transparent to them and its auditing security should be independent of users' technical competence.

- *Remote access control:* The victim should be able to disable access to protected files after device loss, even if the device has no network or computational capabilities. If an adversary has not yet accessed a protected file, then disabling access prevents any access to the file in the future. If an adversary has already accessed the file, we provide no guarantees about repeat accesses.

These goals mean that device owners will have accurate information about which files have been accessed post-loss. While we will consider optimizations that may introduce extra entries in the audit log, maintaining a *zero false-negative rate* is critical. If a file does not appear in the audit log, then one can confidently say that the file was not accessed. In addition, these audit goals must hold after T_{loss} even if an attacker uses his own software and hardware (and not Keypad) to access the files stored on the device.

We also have several non-goals for Keypad. First, we do not attempt to ensure the device's physical or software integrity after theft/loss. If a user recovers a lost device, he should assume that it has been tampered with, and inspect and reinstall the device from scratch to ensure that no keyloggers or malware have been installed. Second, Keypad deals with device theft/loss that is detectable by a user, and not with surreptitious attacks where an adversary might undetectedly access data on a user's device while he is away. This excludes evil-maid attacks from our threat model [205].

Third, Keypad ensures auditability and remote control solely at the file system interface level. Auditability and control of clear-text data cached in applications' memories is out of Keypad's scope. Fourth, we do not seek to improve the confidentiality of protected files over traditional encryption. Instead, Keypad provides a secure audit log of file accesses if that traditional encryption fails. Finally, we do not guarantee that users can always access Keypad-protected files in the absence of network connectivity (which we consider increasingly rare, given ubiquitous cellular and WiFi networks). However, we do introduce a "paired-device" mechanism to mitigate the impact of disconnection while still maintaining auditability.

2.3 Keypad Architecture

Keypad augments encrypted file systems with two properties: auditability and remote data control. The basic idea is simple yet powerful. Keypad: (1) encrypts each file with its own symmetric key, (2) stores all keys on a remote audit service, (3) downloads the key for a file each time it is accessed, and (4) destroys the key immediately after use. This approach supports our auditability and remote data control goals. By configuring the audit service to log all storage accesses, we obtain fine-grained auditability; by disabling all keys associated with a stolen device on the service, we prevent further data access.

Despite its simplicity, designing a practical file system to achieve our goals poses three challenges. First is performance: each file access requires a blocking network request, which could harm application performance and responsiveness over high latency cellular networks. Second is disconnection: involving the network on all file accesses prohibits file use during network unavailability. While we treat this as an exception, we still wish to support disconnected operation. Third is metadata: an auditor requires user-friendly, up-to-date metadata for each key to interpret access logs appropriately. As will be shown, efficiently maintaining metadata is complex, but possible. This section shows how Keypad's design addresses these three challenges.

2.3.1 Architectural Overview

Figure 2.2 shows Keypad's architecture. On the client device, each file F has a unique identifier (called the *audit ID* – ID_F) stored in its header, and the file's data is encrypted with a unique symmetric key, K_F. A remote *key service* maintains the mappings between audit IDs and keys. When an application wants to read or write a file, Keypad looks up the file's audit ID in its header and requests the associated key from the service. Before responding to the request, the service durably logs the requested ID and a timestamp. This process ensures that after T_{notice}, the user will be able to identify all compromised audit IDs for which there is a log entry after T_{loss}.

In addition to the key service, Keypad contains a *metadata service* that maintains information needed by users to interpret the logs. The information (called *file metadata*) includes a file's path, the process that created it, and the file's extended attributes. The metadata and key services fulfill conceptually independent functions; they could be run by a single provider or by distinct providers.

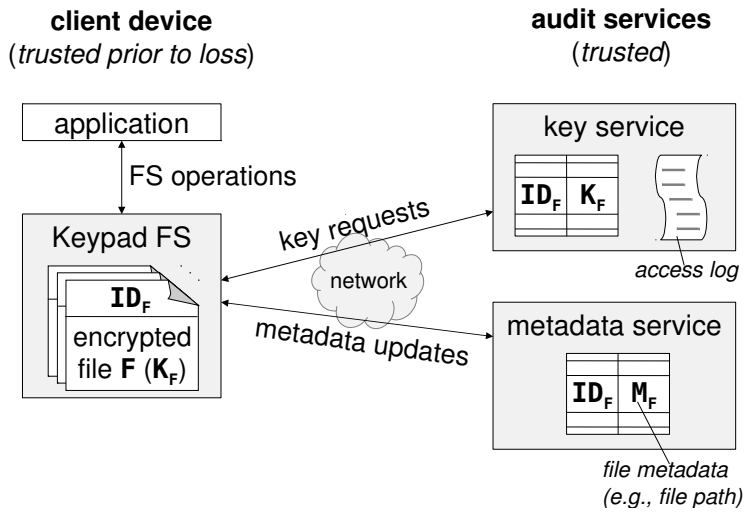


Figure 2.2: **The Keypad System Architecture.** Each file is encrypted with its own random symmetric key. Keys are stored remotely on a key service. To enable forensics, a (separate) metadata service stores file metadata.

Using distinct providers helps to mitigate privacy concerns that could arise if a single party tracked all file access information. The key service sees only accesses to opaque IDs and keys, while the metadata service learns the file system's structure, but not the access patterns. Thus, privacy-concerned users can avoid exposing full audit information to any audit service by using different key and metadata providers.

To meet our goal of robust auditing semantics, Keypad must carefully manage file metadata. For example, when an application creates a new file with name G, Keypad: (1) locally allocates an ID_G for the file, (2) sends a request to the key service to create a new key K_G and bind it to ID_G, and (3) sends a request to the metadata service to register the name G with ID_G. While steps 2 and 3 can occur concurrently, Keypad must confirm that both requests complete before it allows access to the new file. This ensures that file metadata is associated with keys prior to T_{loss}, so that any compromised keys can be correlated with their metadata after T_{notice}.

Similarly, during a file's lifetime, Keypad must keep the service's metadata current to ensure that a user will have fresh information in case of compromise. For example, whenever an application renames a file, Keypad sends a metadata-update request to the metadata service. Keypad must

ensure that a thief cannot overwrite the user's metadata with bogus information after theft. For this reason, we implement the metadata store as an append-only log.

2.3.2 Semantics and Challenges

Keypad provides users with strong auditing semantics at audit time (i.e., post T_{loss}). We formulate an *ideal* invariant describing these semantics as follows:

> For any file **F** with identifier $\mathbf{ID_F}$ that was accessed after $\mathbf{T_{loss}}$ the following properties hold:
>
> (1) the key service shows an $\mathbf{ID_F}$ log entry after $\mathbf{T_{loss}}$, and
>
> (2) the metadata service shows all metadata updates that occurred on $\mathbf{ID_F}$ before $\mathbf{T_{loss}}$.

For (2), the metadata server must contain the latest file metadata (such as file pathname or other attributes) that the user assigned to the file. For example, suppose a user has downloaded a blank IRS tax form into `/tmp/irs_form.pdf`, renamed it as `/home/prepared_taxes_2011.pdf`, and filled it with sensitive information. Then, at forensics time, the user will need to have this latest path available on the service side to interpret a compromise of the taxes file accurately. Hence, maintaining up-to-date service-side metadata is vital to enable meaningful forensics.

In theory, we could achieve semantics arbitrarily close to this ideal invariant. If Keypad downloaded a file's key *every time* a block in the file is accessed and erased the key from memory immediately after using it, then we would obtain the first part of the invariant. Similarly, if Keypad waited for *every* metadata update to be acknowledged by the metadata service before completing that operation on the local disk, then we would obtain the second part.

In practice, however, achieving the ideal invariant is challenging at best. If Keypad must wait a full network round-trip for every block access and for every metadata operation (e.g., `rename`), then the system would be unacceptably slow over high-latency networks. Similarly, disconnected access would be impossible. The remainder of this section describes a combination of new techniques and re-purposed traditional mechanisms that help overcome these challenges. While each technique slightly weakens the invariant, we believe that the semantics remain clear and easy to grasp, and that we achieve our goals in nearly all realistic cases.

2.3.3 Encryption Key Caching and Prefetching

Many of Keypad's critical-path operations are remote key-fetching requests, e.g., issued whenever an application performs a file `read` or `write`. The number of such key requests can be minimized using standard OS mechanisms, such as caching and prefetching. For instance, instead of erasing a key immediately after use, Keypad can cache it locally. Similarly, on access to a file F, Keypad can prefetch keys for other related files, such as those in the same directory. Key caching and prefetching remove key retrieval from the critical path of many file accesses, dramatically improving performance (Section 2.5).

While caching and prefetching are well understood, they have non-standard implications in our system. First, these techniques cause keys to accumulate in the device's memory, affecting what users can deduce from the audit log of a lost device. Keys that are cached at time T_{loss} are susceptible to compromise: if an adversary can extract them from memory he can permanently remember those keys and bypass audit records for those files. The victim must thus make the worst-case assumption that all keys cached at T_{loss} are compromised. Second, key prefetching creates false positives in the audit log: some prefetched keys may not be used, although records for those keys will appear in the logs.

Keypad must therefore use caching and prefetching carefully to ensure good auditing semantics. For caching, we impose short lifetimes (T_{exp}) on keys and securely erase them at expiration. This bounds key accumulation in memory; the shorter the T_{exp}, the fewer keys will be exposed after T_{loss}. Experimentally, we find that key expirations as short as 100 seconds reap most of the performance benefit of caching, while exposing relatively few keys in memory at a given time. For prefetching, we designed a simple scheme to prefetch keys only when a file-scanning workload is detected (e.g., recursive file search or file hierarchy copying). This benefits file-system-heavy workloads where prefetching is the most useful, while maintaining high auditing precision for light workloads (e.g., interacting with a document). We discuss further prefetching alternatives in Section 2.4.

Key caching and prefetching alter Keypad's auditing semantics in a clear way: a user must now consider as compromised all files with audit records after $T_{loss} - T_{exp}$. Doing so ensures that the user will never experience false negatives. Hence, these techniques alter the invariant introduced in Section 2.3.1 in the following way: key and metadata service information must be present for any

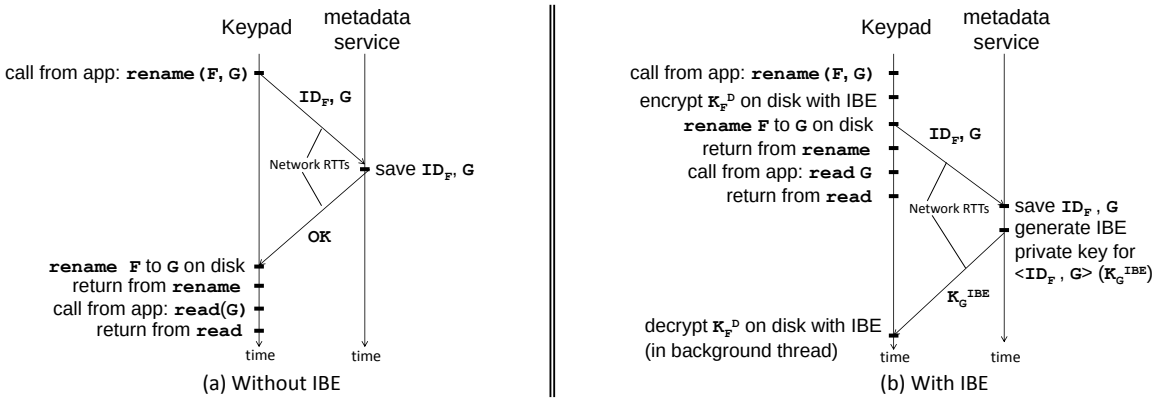


Figure 2.3: **Timelines for Handling Metadata Operations without IBE (a) and with IBE (b).** The application is assumed to issue a `rename(F, G)` followed by a `read(G)`. Assuming that a copy of F's decryption key is cached in memory, IBE allows overlap of accesses to F with the metadata service request until the cached key times out (1 second in our system).

file F that was accessed after $T_{loss} - T_{exp}$. In Section 2.5.2 we quantify the effects of caching and prefetching on auditing.

2.3.4 Identity-Based Encryption for Metadata Updates

Metadata-update file system operations (such as file `create` and `rename`) account for a significant portion of file system operations in many workloads. For example, an OpenOffice file save invokes 11 file system operations, of which 7 are metadata operations that create and then rename temporary files. This large number of metadata operations would result in poor performance over slow networks if Keypad were to wait for an acknowledgement from the metadata service upon *every* metadata update before committing the update to disk, as required by our ideal auditing semantics. Figure 2.3a shows this scenario.

Overlapping local metadata updates with remote metadata service updates seems like a tempting optimization, however, it opens Keypad to possible attacks and frustrates our semantics. For example, consider a user who creates a new file called `/home/taxes_2011`, writes sensitive tax information inside, and closes the file and editing application. Suppose that due to network failures the create request does not reach the metadata service and therefore the service does not learn the

new file's name. If a thief steals the device and reads the tax file ten minutes later, the access will produce an audit trail on the key service; however, no file metadata will be available for the user to interpret the log. Worse, the thief could block Keypad's metadata retries and send a bogus request to the service, e.g., declaring the new file's path as `/tmp/download` to mislead the user.

To respond to this challenge, Keypad leverages identity-based encryption (IBE) [26, 182] in a way that both eliminates the network from the critical path of metadata updates and retains its strong auditing semantics. IBE allows a client to perform public-key encryption using any key string it chooses as the public key. A server called a private key generator (PKG) is required to generate the decryption key for the arbitrary public key. Most importantly for our use, the PKG need not know the public key string in advance, but the public key string must be provided to the PKG to learn the decryption key.

We modified Keypad to use IBE as follows. First, we add a level of indirection for file encryption keys. A file F's content is encrypted using a locally-generated random *data key* (denoted K_F^D) stored in the file's header. The data key is *itself* encrypted under the remote key, which in turn is stored on the key server. Section 2.4 provides more detail.

Second, Keypad's metadata service acts as a PKG, as shown in Figure 2.3b. When an application invokes a metadata operation (such as `rename`) for a file F, Keypad "locks" its encrypted data key K_F^D in the on-disk file header by encrypting it with IBE, using the new file's pathname as the public key string. While the metadata request is in flight, reads and writes can proceed *as long as* a copy of the file's cleartext data key K_F^D is cached in memory. Because files with metadata updates in flight are vulnerable to attacks, we reduce the key expiration time for such files to the bare minimum necessary to hide network latencies on cellular networks. For example, our prototype expires cached keys with in-flight metadata updates in *one second*, minimizing attack opportunity. After the cached key times out, the file is essentially "locked" on disk by the IBE encryption, preventing subsequent file accesses until the metadata service confirms its success. On confirmation, the metadata service returns the IBE private key, allowing Keypad to "unlock" the file.

Suppose an attack or network failure prevents the service from registering the new metadata and subsequently the device is stolen. In the (extremely likely) case that the theft occurred more than one second after the user's rename request, the file's cached data key will have expired and the thief will need to obtain the IBE private key in order to unlock the file for access. As a result, the thief is

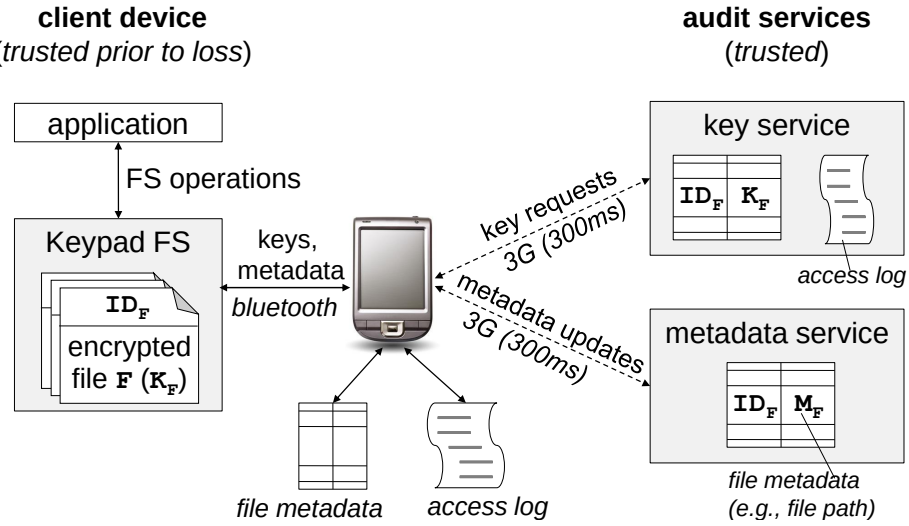


Figure 2.4: **Paired-device Architecture.** By pairing a laptop with a mobile phone, Keypad supports disconnected operation and may even improve performance.

forced to supply the *correct* file pathname to the metadata service if he desires to read the file; lying or avoiding the metadata update will prevent him from gaining access. Therefore, the thief cannot access the file without causing an audit record associated with correct and up-to-date metadata to be logged on the corresponding audit services.

2.3.5 Using Paired Devices for Disconnected Access

Although disconnected operations are assumed to be the exception rather than the rule, Keypad must still support them. One option is to cache keys for an extended period of time and accumulate metadata registrations locally. However, this forces the user to give up auditability for the disconnected duration, which can be dangerous. Further, caching is not applicable to storage-only devices like USB sticks. To address this issue, we developed a *paired-device* extension to Keypad that supports disconnected operations without sacrificing auditability semantics.

Many of today's users carry multiple devices when they travel, such as a laptop as well as a smart phone or a tablet. These devices support short-range, low-latency networks, such as Bluetooth. The paired-device architecture, shown in Figure 2.4, uses a cell phone as a transparent extension of

the Keypad key and metadata services. Keypad on the laptop is configured as usual, using strict caching, prefetching, and metadata registration policies to ensure fine-grained auditing. The phone is configured to hoard [108] any recently used keys, cache them until connectivity is restored, log any accesses and metadata updates to the local disk, and upload the logs when connectivity returns. If only the laptop is lost, the phone is used along with the audit service logs to provide a full audit trail. If the phone is stolen along with the laptop, then the audit service will list more files as exposed than if the laptop were stolen alone.

In addition to supporting (increasingly rare) disconnected cases, the paired-device architecture has another advantage: it can improve performance over slow mobile networks without sacrificing auditing. Because the laptop–phone link is relatively efficient, the paired phone can improve laptop performance by acting as a cache for it. Here the phone is configured to perform aggressive directory-level key prefetching and caching. On a key miss, the laptop contacts the phone via bluetooth and the phone returns the key, if available; otherwise the phone fetches the missed key and other related keys from the key service and returns the key to Keypad. Section 2.5 evaluates the performance improvement for this solution. As before, auditing properties are preserved if only the laptop is stolen. If both devices are stolen, then auditing is at a directory-level granularity.

2.3.6 Partial Coverage

Not all files necessarily require audit log entries. For example, as a trivial optimization we could exclude non-sensitive files such as binaries, libraries, and configuration files from Keypad's audited protection domain. In this scenario protected files are encrypted locally and their keys and metadata are stored remotely; unprotected files are (optionally) encrypted locally, but their encryption keys are derived from the user's login credentials.

The benefits of this optimization are obvious: Keypad's performance and availability costs are only incurred for protected files. There is also a risk: if a sensitive file is accidentally placed in an untracked file or directory, the audit logs will not reveal accesses to that sensitive data. One reasonable protection policy is to track accesses to any file in crucial directories, such as the user's home and temporary directory (e.g., `/home` and `/tmp` on Linux).

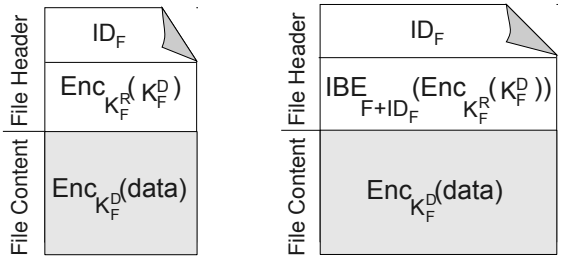
2.3.7 Summary

Keypad provides strong guarantees to its users. If a protected file is accessed, then at least one record related to that access will appear in the remote audit logs, and up-to-date metadata about the file will be available online. As we have shown, one challenge Keypad faces is preserving this strong property while overcoming the performance impact of communicating with remote services in the critical path of file accesses. We introduced a series of novel techniques to meet this challenge. Though some of these techniques have an impact on the quality of the information in the audit logs, we show in Section 2.5.2 that this impact is small.

2.4 Prototype Implementation

We implemented a Keypad prototype including the client-side Keypad filesystem, the key service, and the metadata service as shown in Figure 2.2. All components are coded in C++ and communicate using encrypted XML-RPC with persistent connections. Our client-side Keypad file system is an extension of EncFS [64], an open-source block-level encrypted file system based on FUSE [75]. EncFS encrypts all files, directories, and names under a single volume key, which is stored on disk encrypted under the user's password. Keypad extends EncFS in two ways. First, we modified EncFS to encrypt each file with its own per-file key. The single volume key is still used, however, to protect file headers and the file system's namespace, e.g., file and directory names. Second, Keypad stores all file keys on a remote key server and maintains up-to-date metadata on a metadata server. To support forensic analysis we built a simple Python tool; given a T_{loss} timestamp and an expiration time, T_{exp}, the tool reconstructs a full-fidelity audit report of all accesses after $T_{loss} - T_{exp}$, including full path names and access timestamps.

Keypad File Structure. Figure 2.5(a) shows the internal structure of a Keypad file F, which consists of two regions: the file's header and its content. The file's header is fixed size and is encrypted using EncFS' volume key. For the file's content, our implementation adds a level of indirection for encryption keys to support techniques such as IBE efficiently. Specifically, file F's content is encrypted using a 256-bit random *data key*, denoted K_F^D. The data key is stored in the file's header encrypted under the *remote key*, denoted K_F^R. The remote key is stored on the key server and is identified by the file's audit ID (ID_F), which is a randomly generated 192-bit integer that is stored



(a) Keypad File. (b) IBE-Locked File.

Figure 2.5: **Keypad File Formats.** Keypad on-disk file structure for the normal case (a) and the IBE-locked case (b).

in the file's header along with the encrypted data key. This internal file structure is transparent to applications, which see only the decrypted contents of a file.

FS Operations. Keypad intercepts and alters two types of EncFS operations: file-content operations (`read`, `write`) and metadata-update operations (`create`, `rename` for files or directories). When an application accesses file content, Keypad: (1) looks up the file's audit ID from its header, (2) retrieves the remote key K_F^R, either from the local cache or the key service, (3) decrypts the data key K_F^D using K_F^R, (4) caches K_F^D temporarily, and (5) decrypts/encrypts the data using K_F^D.

When an application creates or updates file metadata, Keypad: (1) locks the data key using IBE, if enabled, and (2) sends the new metadata to the metadata service. The metadata is the file's path reported as a tuple of the form `directoryID/filename`. The names of Keypad directories are also kept current on the metadata service. While our current prototype applies IBE for file metadata update operations (e.g., file `create`, `rename`), it does not apply it to directory metadata operations (e.g., `mkdir` or directory `rename`), although this should be possible to add.

Key Expiration. Keypad caches keys for a limited time for performance. A background thread purges expired keys from the cache. If a key has been reused during its expiration period, the thread requests the key from the key service again, causing an audit record to be appended to the access log for that audit ID. If a response arrives before the key expires, the key's expiration time is updated in the cache, otherwise the key is removed. As a result, absent network failures, keys in Keypad never expire while in use. This ensures that long-term file accesses, such as playing a movie, will not exhibit hiccups due to remote-key fetching.

Key Prefetching. Key prefetching attempts to anticipate future file accesses by requesting file keys before the files are accessed. For our prototype, we sought a simple policy that would have both reasonable performance and little impact on auditability. We have experimented with two policies: (1) a random-prefetch scheme that prefetches random keys from the local directory upon every key-cache miss and (2) a full-directory-prefetch scheme that prefetches all keys in a directory when it detects that the directory is being scanned by an application. Our experiments indicated that the latter policy provided equally good performance, while incurring fewer false positives in the audit logs. Hence, our Keypad prototype uses it by default. The intuition behind our full-directory prefetch design is to avoid producing false positives for targeted workloads (such as interacting with a document, viewing a video, etc.) and to improve performance for scanning workloads (such as grepping through the files in a directory or copying a directory). Our full-directory-prefetch scheme avoids recursive prefetches to ensure that any false positives are triggered by real accesses to (related) files in the same directory. While other more effective prefetching policies may exist, our results show that our full-directory-prefetch policy, combined with our caching policies, reduce the number of blocking key requests to a point where the performance bottleneck shifts from blocking key requests to metadata requests (see Section 2.5).

IBE. To avoid blocking for metadata-update requests, our prototype implements IBE-based metadata registration, using an open-source IBE package [145]. On a metadata-update operation, Keypad locks the file until the metadata service confirms the receipt of the new file path; however, file operations can proceed for a one-second window, as previously described, to absorb network latency. Figure 2.5(b) shows the structure of an IBE-locked file. Its encrypted data key is further encrypted using IBE under a public key consisting of the file's path (`directoryID/filename`) and the audit ID (ID_F). Embedding ID_F into the public key strongly binds ID_F and the path together at the metadata server. Handling updates for other types of file metadata functions (such as `setfattr`) works similarly, although our current prototype only supports pathnames as metadata. An attacker cannot pre-obtain private IBE keys for popular file paths from the metadata server prior to stealing the device, because directory and file IDs are drawn at random from a gigantic space (2^{192}).

Android-Based Paired-Device Prototype. We implemented a prototype of the paired-device architecture (Figure 2.4) using the Google Nexus One phone. A simple daemon (431 lines of Python)

on the phone accepts key requests from the laptop over Bluetooth, saves accesses to a local database, responds to the laptop, and uploads access and metadata information to Keypad servers in bulk over wireless. It leverages the key derivation mechanism to easily fetch directory keys upon a key-cache miss. For example, when the laptop requests a file key, the Nexus will fetch the parent directory's key from the key server, cache it, compute the file's key by applying HMAC to the directory key, and return the file key to the laptop. Thus, when pairing with the Nexus, the key server provides directory-level auditing, while the phone offers fine-grained auditing.

2.5 Evaluation

This section quantifies Keypad's performance and auditing quality. Keypad must be fast enough to preserve the usability of desktop and mobile applications, even in the face of adverse network conditions (e.g., 3G), while providing high quality auditing.

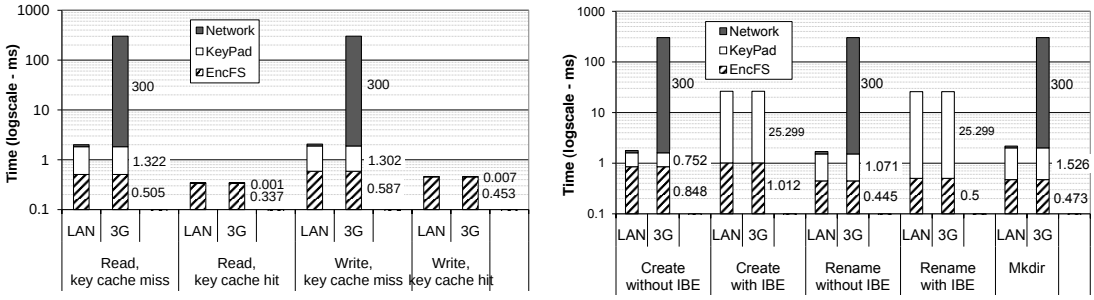
For our experiments, we used an eight-core 2GHz x86 machine running Linux 2.6.31 as our client. Our key service and name service daemons ran on 8 core 2.6GHz servers with 24GB of RAM, connected via gigabit Ethernet. We used Linux's traffic control utility to emulate different network latencies. We did not emulate different bandwidth constraints, however, Keypad's bandwidth requirements are very low. During a 12-day period in which one of our authors used Keypad continuously, average Keypad bandwidth was under 5 kb/s, with occasional spikes up to 45 kb/s.

Throughout the evaluation, we emulate the following RTTs for various networks: 0.1ms RTT for a LAN, 2ms RTT for a wireless LAN (WLAN), 25ms RTT for broadband, 125ms RTT for a DSL network, and 300ms RTT for a 3G cellular network. To illustrate network latency effects on Keypad performance, we often use examples from extreme network conditions, such as fast LANs and slow 3G networks, even though popular mobile connections today rely on WLAN and 4G.

2.5.1 Performance

To understand where the time goes for Keypad operations, we microbenchmarked file content (`read` and `write`) and metadata (`create`, `rename`, and `mkdir`) operations. Our measurements included client, server, and network latencies, as well as latency contributions for EncFS and Keypad.

Figure 2.6(a) shows the latency of file read and write operations for two cases: key-cache misses,



(a) FS Content Operations: `read`, `write`.

(b) FS Metadata Operations: `create`, `rename`, `mkdir`.

Figure 2.6: **File Operation Latency.** The latency of Keypad (a) content and (b) metadata-update operations. For each, we show the time spent in EncFS code, Keypad client and server code, and on the network. Labels on the graph show the latency for each component in the 3G 300ms RTT case. Results are averaged over 10 trials with a warm disk buffer cache.

which must fetch the key from the server, and key-cache hits, which use a locally cached key. For each case we show data for two extreme networks: a fast 0.1ms-RTT LAN and a slow 300ms-RTT 3G network. The results show that misses are expensive on both networks, but for different reasons. On a LAN, the network is insignificant, but Keypad adds to the base EncFS time due to the XML-RPC marshalling overhead. On 3G, network latency dominates. When the key-cache hits, both the network and marshalling costs are eliminated; a file read with a cached key is only 0.01ms slower than the base EncFS read time of 0.337ms. This shows the importance of key caching to avoid misses, which we accomplish by carefully choosing our expiration and prefetching policies.

Figure 2.6(b) shows the latency of file metadata update operations. For `create` and `rename`, we show latency with and without IBE; `mkdir` is shown only without IBE, since it does not benefit from this optimization in our prototype. Without IBE, metadata update latency is driven primarily by network RTT: file creation takes 1.618ms on a LAN, and 302ms over 3G. With IBE, metadata update latency is independent of network delay and is dominated by the computational cost of IBE itself. The figure shows that IBE meets its goal of improving performance of metadata updates over 3G. While IBE would add overhead for a LAN, it is unnecessary and would be disabled in the LAN environment.

Optimizations

We now demonstrate the effectiveness of our optimizations on a challenging workload: Apache compilation. While this workload is not characteristic of mobile devices, its complex nature make it ideal for evaluating the impact of our optimizations. In Section 2.5.1, we extend our evaluation to more typical workloads for mobile devices. As baselines, the Apache compilation takes 112s using the unmodified EncFS encrypted file system (i.e., with encryption but without auditing) and 63s on `ext3` (i.e., without encryption or auditing). Because Keypad enhances EncFS, the fair baseline comparison for Keypad is EncFS, and not `ext3`.

In what follows, we inspect the effect of optimizations as we enable one optimization after the other. We begin by showing the effect of purely key caching with no other optimizations, then we add prefetching, then IBE, and finally we add the paired-device optimization.

Key Caching and Expiration. Key caching is crucial to performance. Even a cache with one-second expiration time has significant impact: 18% improvement on a LAN and 4.9x on 3G, relative to no caching at all. Figure 2.7 shows additional improvements for Apache compilation time as expirations are lengthened beyond one second. No optimizations other than caching are enabled here. Our results suggest that short expiration times are sufficient to extract nearly all the benefits. For LAN, Broadband, or DSL latencies, an expiration of 10s or so is optimal. Over 3G, a 100s key expiration time achieves all the benefit and provides 8.6x improvement over 1s (from 79.4 minutes down to 9.2 minutes). In comparison to EncFS, Keypad's performance degradation for 100s expiration times is already small over a LAN (5.3% overhead over EncFS), while for the other network types, further optimizations are required for performance.

Note that a 100s timeout is extremely small. To benefit from cached keys, a thief needs to steal the device within 100 seconds of the user's last access. Even in such cases, the user will know which files were exposed. We therefore believe that we can achieve both good performance and accurate auditing with these parameters.

Directory-Key Prefetching. Key caching alone avoids many key service requests: of the 75,744 reads and writes in the Apache compilation, only 486 involve the server when using a 100s expiration time. Directory-key prefetching avoids additional server requests. Prefetching a directory key on the first, third, or tenth miss in a directory results in 101, 249, and 424 key-cache misses, which

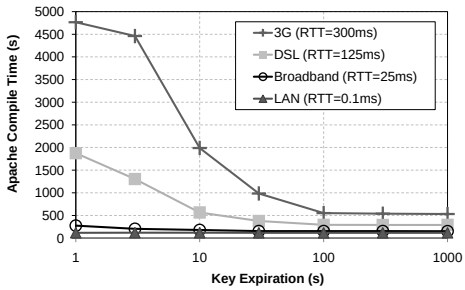
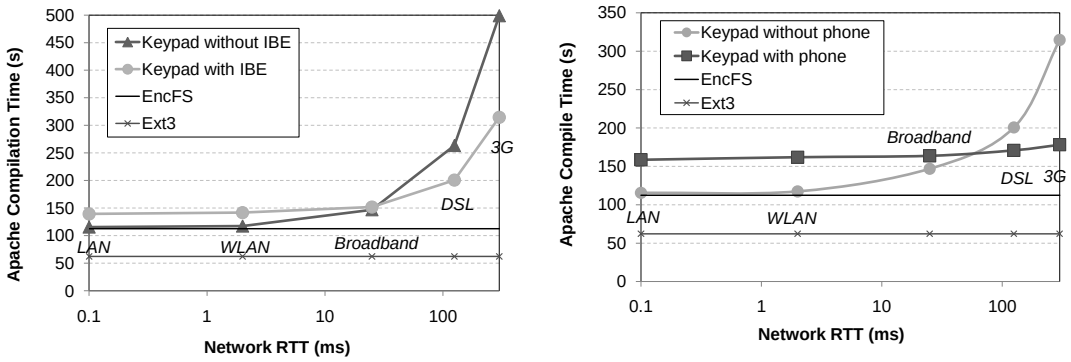


Figure 2.7: **Effect of Key Expiration Time.** This graph shows the effect of key expiration without any other optimizations enabled. A 100-s key expiration time is nearly optimal, and achieves compilation times of 115s, 153s, 292s, and 551s over a LAN, Broadband, DSL, and 3G, respectively. For comparison, the Apache compilation takes 112s on the unmodified EncFS and 63s on `ext3`.

translates into 63.3%, 24.1%, and 2.4% improvements, respectively, over not using directory-key prefetching over 3G. We adopted a prefetch-on-third-miss policy to strike a good balance between performance and auditing quality (which is evaluated in Section 2.5.2). Over fast networks, such as a LAN and WLAN, the prefetch-on-third-miss policy coupled with 100-s key caching results in negligible performance overheads compared to EncFS: 2.8% for LAN and 4.3% for WLAN. Over slower networks, especially 3G, other smarter prefetching policies may improve performance by further eliminating blocking key requests. However, we find that with our simple prefetching policy, the dominating runtime component now becomes the blocking metadata requests (932 blocking metadata requests compared to the 249 blocking key requests). We next focus on optimizing metadata requests.

IBE. IBE hides the latency of metadata updates over slow mobile networks. Figure 2.8(a) shows the impact of IBE on Apache compilation as a function of network RTT. As we see in the figure, IBE provides dramatic improvements on high-latency networks, including 3G- and 4G-class networks. For example, IBE improves the benchmark's performance on 3G by 36.9%. The crossover for IBE is around 25ms, i.e., it should be used only for networks with RTTs over 25ms and disabled otherwise. As mentioned above, for faster networks, such as LANs or WLANs, IBE is not even necessary, as Keypad's overhead is already negligible after applying key caching and prefetching.



(a) Effect of IBE.

(b) Effect of Device Pairing.

Figure 2.8: **Effect of IBE and Device-pairing Optimizations.** (a) Effect of applying the IBE optimization atop a 100-s key caching policy and a third-miss prefetching policy; no device pairing is used here. (b) Effect of applying the device-pairing optimization atop the optimization setup in (a).

The Paired Device. Our paired device design is aimed at facilitating disconnected operation, but it can also provide performance benefits for high-latency network environments. Figure 2.8(b) shows the effect on the Apache workload of using a paired device as a caching proxy for key and metadata services. Two conclusions can be reached from the figure. First, performance for disconnected operation over Bluetooth should be similar to or better than that of a broadband connection (the latencies are similar). Second, pairing with another device is always beneficial for performance over cellular networks, because most operations only traverse the lower latency Bluetooth link. Obviously the paired device should not be used if fast networks are available, where Keypad is already efficient enough compared to EncFS.

Office-Oriented Workloads

Figure 2.9 shows the impact of our optimizations on more typical office-oriented workloads. We add optimizations incrementally, reporting additional improvement as more optimizations are added. The labels on top of each bar group show the total improvement with all three optimizations enabled. Different workloads benefit the most from different optimizations, depending primarily on

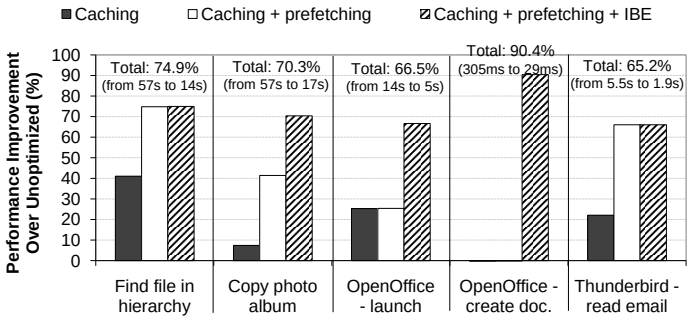


Figure 2.9: **Impact of Optimizations on Various Applications.** Impact of three of the optimizations on an emulated 3G network; labels indicate the total performance improvement when using all three optimizations over the unoptimized case, as well as the absolute numbers for the unoptimized and optimized.

the relative frequency of those operations. For example, caching and prefetching are important for a read-intensive workload such as a recursive grep ("Find file in hierarchy"). IBE provides large improvements for workloads that create files ("OpenOffice – create doc"). For mixed content/metadata workloads, such as copying a photo album across directories, all optimizations are important.

To better understand performance across many applications, we benchmarked the time to perform a number of popular tasks using EncFS and Keypad over several emulated networks (Table 2.1). For Keypad, we show both warm and cold key-cache times. A user will likely experience both, but with well-chosen key expiration times many operations will be absorbed by a warm cache.

From a user's perspective, Keypad performs roughly identically to EncFS over fast networks, such as a LAN and a wireless LAN. Hence, while at the office, the user should never feel our file system's presence, whether its key cache is warm or cold. With only a few exceptions, the user should perceive similar application performance over broadband with Keypad and the unmodified EncFS. Over mobile networks, the user may notice some application slowdown, especially after extended periods of inactivity.

The table and our own experience confirm that application launches are particularly expensive over 3G networks, as they often encounter a cold cache and many file system interactions. Keypad could optimize launch by profiling applications and prefetching needed keys; other file systems, such as NTFS, perform similar special-case optimizations to speed up application launch.

Application	Task	Time (seconds)					
			Keypad				
		EncFS	LAN (RTT=0.1ms)	WLAN (RTT=2ms)	Broadband (RTT=25ms)	DSL (RTT=125ms)	3G (RTT=300ms)
OpenOffice Word Processor	Launch	0.5	0.5 | 0.5	0.6 | 0.6	1.3 | 1.3	2.7 | 2.7	4.6 | 4.6
	New document	0.0	0.0 | 0.0	0.0 | 0.0	0.0 | 0.0	0.0 | 0.1	0.0 | 0.3
	Save as	1.4	1.4 | 1.4	1.4 | 1.4	1.5 | 1.5	1.6 | 1.8	2.0 | 2.3
	Open	1.7	1.7 | 1.7	1.8 | 1.8	2.0 | 2.2	2.1 | 4.0	2.1 | 7.5
	Quit	0.1	0.1 | 0.1	0.1 | 0.1	0.3 | 0.4	0.4 | 0.7	0.4 | 1.2
Firefox	Launch	3.7	3.7 | 3.7	3.8 | 3.8	4.4 | 4.4	6.0 | 6.0	8.8 | 8.8
	Save a page	0.7	0.7 | 0.7	0.7 | 0.7	0.7 | 0.8	0.9 | 1.5	1.3 | 2.8
	Load bookmark	4.5	4.5 | 4.5	4.5 | 4.5	4.5 | 4.6	4.5 | 5.0	4.5 | 5.7
	Open tab	0.2	0.2 | 0.2	0.2 | 0.2	0.2 | 0.2	0.2 | 0.4	0.2 | 0.8
	Close tab	0.0	0.0 | 0.0	0.0 | 0.0	0.0 | 0.0	0.0 | 0.1	0.0 | 0.3
Thunderbird	Launch	1.3	1.3 | 1.3	1.3 | 1.3	1.4 | 1.4	2.0 | 2.0	3.1 | 3.1
	Read email	0.3	0.4 | 0.4	0.4 | 0.4	0.5 | 0.6	1.0 | 1.5	1.9 | 2.5
	Quit	0.2	0.2 | 0.2	0.2 | 2.2	0.2 | 0.4	0.2 | 1.3	0.2 | 2.9
Evince PDF Viewer	Launch	0.1	0.1 | 0.1	0.1 | 0.1	0.1 | 0.1	0.1 | 0.1	0.1 | 0.4
	Open document	0.1	0.1 | 0.1	0.1 | 0.1	0.1 | 0.1	0.2 | 0.2	0.4 | 0.4
	Quit	0.0	0.0 | 0.0	0.0 | 0.0	0.0 | 0.0	0.0 | 0.0	0.0 | 0.0

x | y: x = time with warm key cache
y = time with cold key cache

Table 2.1: **Typical Application Performance Over Keypad.** For Keypad, we show both warm and cold key-cache times, separated by a |.

Comparison to Other File Systems

A networked file system might be an alternative to Keypad; instead of just storing keys remotely, all file system content would be remote. NFS provides a reasonably fair comparison to Keypad, since its short-term caching might provide audit properties comparable to ours. In contrast, for AFS and Coda, their long-term, coarse granularity caching policies might interfere more with precise audit semantics.

Figure 2.10 shows the relative performance of Keypad to (remote) NFSv3 and (local) EncFS for Apache compilation. We configured NFS with asynchronous batched writes and its default caching policy; this improves its performance, but would have some impact on auditing. Note that for these experiments, as before, we emulated different network RTTs but we did not constrain network bandwidth; thus, our results are upper bounds of NFS performance. Over actual 3G links, NFS performance would be significantly degraded because of wireless bandwidth constraints.

With LAN latencies, Keypad's performance is almost identical to EncFS with only a 2.78% increase in runtime, but worse than NFS, with a 75% increase. For reference, the unmodified EncFS

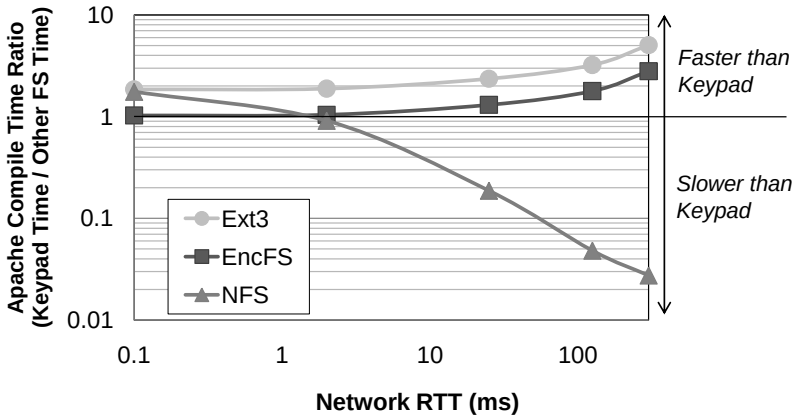


Figure 2.10: **Comparison to EncFS and NFS.**

itself is 71% slower compared to NFS with LAN-like latencies. As RTT grows, NFS degrades significantly. Even with an RTT of 2ms, NFS is 8.8% *slower* than Keypad, while for 3G network latencies of 300ms, NFS is 36.4x slower than Keypad! In contrast, Keypad is only 2.7x slower than EncFS over a 300ms network.

On large-RTT networks, NFS impacts interactivity. For example, launching OpenOffice over NFS with 3G latency takes 50.6 seconds, loading a bookmark in Firefox takes 27.6 seconds, and opening an email in Thunderbird takes 12.5 seconds, which we believe is unacceptable performance for these user-facing tasks.

Anecdotal Experience

Anecdotally, one co-author used Keypad continuously to protect his laptop's `$HOME` and `/tmp` directories over a 12-day period, with an emulated 300ms client-to-server latency. Overall, the experience was positive: in most cases, there was no noticeable performance impact. Some activities, such as file system intensive CVS checkouts or recursive copies, were slower but usable. Other more typical activities, such as browsing the Web, editing documents, and exchanging email, had no noticeable performance degradation.

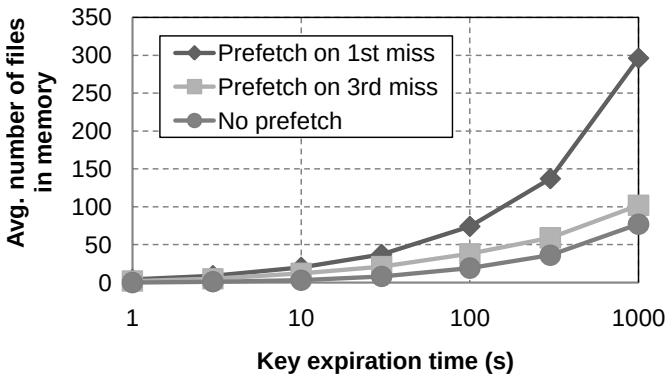


Figure 2.11: **Effect of Optimizations on Auditability.** The average number of keys that reside in memory at any point in time, under various key expiration times and prefetching policies.

2.5.2 Auditing Properties

We now evaluate our optimizations' impact on auditability.

In-memory Key Sets. As described in Section 2.3.3, keys for recently-accessed or prefetched files stay in memory for their expiration period T_{exp}. This is not an issue for a thief who steals a passive storage device, such as a USB stick. For a laptop, because a thief can theoretically access cached-key files without triggering a server-side audit log, users must consider all files whose keys were retrieved between $T_{loss} - T_{exp}$ and T_{loss} as compromised. The size of this set at any point in time depends on the user's workload and on the aggressiveness of the caching and prefetching schemes.

To quantify this issue we used a trace gathered during our twelve-day deployment experience (Section 2.5.1) to calculate the impact of various optimizations on auditability. Figure 2.11 shows the size of the in-memory key set at any point in time averaged over use periods, for different key expiration times and prefetching policies. The graph shows that for reasonable key expiration and prefetching strategies, the average number of in-memory keys is small. For example, with a 100-second key expiration time and a prefetch-directory-on-third-miss strategy, on average there are 38 keys in memory at any instant. This is a small number and furthermore we observed that most of these keys exist as a side-effect of prefetching; i.e., they are files in the same directory as a file that was accessed by a user or program.

False Positives. Prefetching affects forensics by introducing false positives in the audit log. The rate of false positives depends on the prefetching policy as well as the thief's workload, since false positives only concern time post-T_{loss}. In the absence of an accepted "thief workload," we created a few scenarios that a thief might follow. Our goal was to gauge the impact of various prefetching policies on the rate of false positives, as a thief tries to find sensitive information on a captured device. We investigated three scenarios: (1) the thief launches Thunderbird, reads a few emails, browses folders, and searches for emails with a particular keyword; (2) he launches a document editor and looks at a few files; and (3) he inspects the history, bookmarks, cookies, and passwords in a Firefox window. For these workloads, our default prefetch policy (prefetch directory keys on the 3rd miss) leads to the following ratios between false positives and total accessed keys: 3:30, 6:67, and 0:12 for our Thunderbird, document editor, and Firefox workloads, respectively. Audit precision is high for these scenarios.

We have also discovered bad scenarios; if the thief navigates to a web page in Firefox, loading several files from the cache directory causes Keypad to prefetch the entire directory. While this causes several false positives, the user correctly learns that activity happened in the Firefox cache directory. Even in such cases, the auditing implications of our non-recursive prefetching policy are minimal, since all false positives are localized to one directory.

2.5.3 Summary

We measured the performance of our Keypad prototype on several workloads. Our measurement results and our experience using the system show that Keypad meets its goals of adding little overhead in the office or home environment, while remaining highly usable over cellular networks, such as 3G. Overall, our results show that with properly parameterized optimizations, Keypad can provide good performance while also maintaining good auditing fidelity. Furthermore, with current and future improvements of cellular network connectivity (e.g., 4G), we expect Keypad to have even better performance.

2.6 Security Analysis

Keypad is designed to provide strong audit guarantees for encrypted file systems if the first layer of defense, encryption with a password or cryptographic token, is breached. Keypad can additionally destroy the ability to read files after a mobile device is reported lost or stolen. Although we evaluated security properties extensively inline above, we now return for a unified discussion.

Context and Threat Model. We designed Keypad assuming that individuals who find or steal a mobile device range in sophistication, degree of planning, and interest. Curious individuals may insert a found USB stick into their computer, enter the password on the attached sticky note, and browse through a few files trying to find the device owner. Petty thieves may grab laptops opportunistically but have no real interest in accessing confidential files. Corporate spies may plan and execute device theft carefully, with the goal of accessing confidential files before the victim reports the device missing. We refer to all such individuals as "attackers."

Because a user has no way of knowing the motivation and skill of a potential attacker, Keypad assumes the worst. We assume that an attacker has full access to the lost device's hardware (for laptops and USB sticks) and software (for laptops). The attacker can perform cold-boot attacks on laptops, install new software, and manipulate or sever the device's network traffic. The attacker can also perform lower-level activities, such as physically extracting the hard drive from a laptop or memory from a USB stick and interrogating it with custom hardware. However, we do not consider attacks in which the adversary gains control of the device, modifies it, and returns it to the victim without his knowledge (see our non-goals discussion in Section 2.1.1). Any attacker with control over a device while in the user's possession could mount a slew of malicious attacks outside the scope of a forensic file system, ranging from online data exfiltration to the installation of password key loggers. Malware is therefore also explicitly outside our threat model.

Analysis. We begin with the premise that the audit servers are trusted and secure. The key and metadata servers are trusted to maintain accurate logs, and they are assumed to incorporate strong defenses to adversarial comprise, routinely back up their state, and have their own audit mechanisms. Neither the key server nor the metadata server is, however, fully trusted with the private information about a user's file access patterns prior to T_{loss}; accessing that information requires collusion between both servers or the device owner's invocation of the Keypad post-loss audit mechanisms.

The unavailability of servers can deny access to files; for highly sensitive data, we argue that users would prefer unavailability over the potential for unaudited future file disclosure. Further, although not implemented in our prototype, the communications between the Keypad file system and the servers should be encrypted to ward off attackers who intercept network communications prior to device theft. The keys must change every T_{exp} seconds to ensure that an attacker who extracts the current network encryption key from the device cannot decrypt past intercepted data.

Consider now an attacker who obtains a lost or stolen Keypad device. If the device is cold, such as a powered-down laptop or a USB stick, then any successful attempt to access a protected file must generate at least one log record on the Keypad audit servers. This is true whether the attacker uses the Keypad file system or his own hardware or software to perform the access. All of Keypad's mechanisms – the storing of K_F^R on remote servers, the entangling of the metadata server and key server states to ensure consistency, and our method for using IBE – enforce this property. Additionally, the selection of 192-bit audit IDs at random makes it infeasible for an attacker to request information about valid audit IDs from the key and metadata servers prior to physically obtaining the protected device; such requests are additionally thwarted by authenticating the device to the servers.

Attackers who obtain warm, computational devices – such as running or hibernated laptops – may seek to violate the properties of Keypad by directly accessing the device's memory. Cached keys K_F^R should be evicted from memory upon device hibernation, and such evictions should be recorded on the audit servers. For fully running devices, we must assume that an attacker has accessed any file with an audit log entry after $T_{loss} - T_{exp}$. Although Keypad's focus is on providing file system auditing, a forensic analyst must also acknowledge that applications may have sensitive data in memory. A conservative analyst might use various heuristics to identify potentially vulnerable cleartext data. For example, he might mark as compromised any file opened since the device's last boot or hibernation, events that could be recorded on the audit servers. A potentially better future solution to this problem might be to employ encrypted memory technology [159], possibly coupled with auditing.

Most importantly, even against an attacker who obtains warm computational devices, Keypad preserves the following invariant: if an analyst does not mark a file as accessed, then one can confidently conclude that the file has indeed not been accessed by an attacker. Finally, because

entries in the key service are identified per-device, the service can deny access to all relevant keys if a device is reported missing.

For completeness, we must also consider an attacker who attempts to generate spurious entries in the remote audit logs. While such spurious entries might complicate the task of a forensic analyst, an attacker cannot use such actions to hide their actual accesses of confidential data.

2.7 Related Work

Keypad is related to previous work in three areas: (1) theft-protection systems, (2) data-protection systems, and (3) distributed file systems. We next review these solutions to provide a clear differentiation with Keypad, and complement the discussion with an overview of the third type of related system.

Theft-Protection Systems. Theft-protection systems, such as Adeona [166] and Apple's MobileMe [11], provide best-effort auditing semantics, where audit logs are batched and uploaded to a trusted server, making them vulnerable to hardware and disconnection attacks. Keypad provides strong forensic and data-destruction capabilities even against thieves who use their own hardware and software to attack a Keypad-protected file system or (temporarily) block the device's access to the network. Unlike in MobileMe and Adeona, where the audit log for a file access occurs after the fact, the audit log in Keypad is produced *before* the access can occur, making it mandatory.

Data-Protection Systems. Encrypted file systems exist in academia (e.g., [24]) and industry (e.g., BitLocker, PGP Whole Disk, TrueCrypt). None provide remote auditing capabilities, therefore a security breach may go undetected. Keypad's forensic and data-destruction capabilities are orthogonal to work increasing the resilience of encrypted file systems to breach. Keypad can compose with new advances in encrypted file systems, providing both stronger barriers to access and a forensic trail if that barrier is breached.

ZIA [47] and follow-on work [48] protect files on a device with transient authentication. In ZIA, users wear tokens that broadcast their presence, and the device decrypts or encrypts itself depending on whether the user is located close to the device or not, respectively. Protection is lost if an attacker obtains both the device and the token, with no forensic guarantees. Keypad does not require a paired

device, but if one is used, Keypad still provides a forensic trail of potential accesses even if both are lost or stolen. Keypad could be combined with ZIA for additional defense in depth.

Keypad's remote key-escrow architecture has been used frequently in the past to achieve a number of security and privacy goals. First, capture-resilient cryptography [119] uses a key server to prevent dictionary attacks against login passwords on stolen devices, as well as to enable remote wipe-out. Second, location-aware encryption [199] uses a remote key server to dynamically adapt a device's data protection level based on its location. While the device is at a trusted location (e.g., at its owner's home), the server provides the decryption key; when the device is at an unknown or untrusted location, the server will require the user to enter a special password to return obtain the decryption key. Third, assured-delete systems, such as the Ephemerizer [149], revocable backup systems [27], and our own Vanish distributed-trust self-destructing data system [81] adopt the key-escrow architecture to ensure the deletion of sensitive data stored in backup systems or on Web services. Keypad resembles all of these systems in its remote key-escrow architecture and its secondary goal: post-theft data destruction. It differs from these systems in its primary goal: fine-grained auditability of mobile device data accesses.

In general, today's data-protection systems differ from our system in that they focus on data exposure *prevention*, whereas Keypad focuses on data exposure *detection* should prevention systems fail. In that sense, they should be considered as complementary rather than competitors.

Networked File Systems. Work in distributed file systems has aimed at providing shared and available remote storage (e.g., [94, 113, 174]). Bayou [150] and Coda [134] support mobility, disconnected operation and data consistency. Coda's disconnected operation [108] relies on data caching, whereas Keypad uses device pairing, coupled with key caching, to support offline accesses. Coda supports encrypted communication but not storage. LBFS [135] uses compression to reduce latency for interactive file access over slow wide-area networks. SFS [124, 74] is a network file system that supports secure network file transfers, avoiding the need for distributed key infrastructure by embedding public keys in file pathnames. SFS is concerned with secure communication, not with protecting a user's stored data from theft; it does not encrypt data on disk and does not support auditing.

In general, these systems do not support encryption and auditing. While they could be modified

to support both on the server, there are significant performance issues, e.g., streaming an NFS-hosted video over 3G or wireless is slow and expensive. Finally, all of these systems are concerned with the transfer of *file data* between a client and server; in contrast, Keypad is concerned with *key management* and the transfer of encryption keys between a file system and a remote key server. Keypad is unique in its support for (and integration of) encryption and audit logging; it demonstrates the advantage of separating encryption and key management to enforce auditing for mobile device data.

2.8 Summary

This chapter described Keypad, an auditing file system for loss- and theft-prone devices. Unlike basic disk encryption, Keypad provides users with evidence that sensitive data either was or was not accessed following the disappearance of a device. If data was accessed, Keypad gives the user an audit log showing which directories and files were touched. It also allows users to disable file access on lost devices, even if the device has been disconnected from the network or its disk has been removed. Keypad achieves its goals through the integration of encryption, remote key management, and auditing. Our measurements and experience demonstrate that Keypad is usable and effective for common workloads on today's mobile devices and networks.

Chapter 3

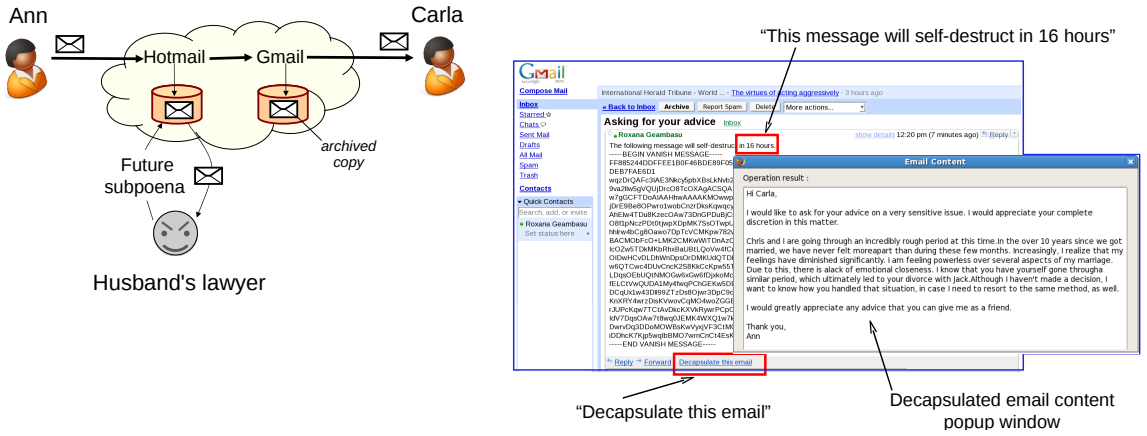
VANISH: DATA LIFETIME CONTROL WITH SELF-DESTRUCTING DATA

Today's technical and legal landscape presents formidable challenges to personal data privacy. First, as described in Chapter 1, our increasing reliance on Web services causes personal data to be cached, copied, and archived by third parties, often without our knowledge or control. Second, the disclosure of private data has become commonplace due to carelessness, theft, or legal actions. This chapter proposes Vanish, a self-destructing data system that provides users with control over the lifetime of their data stored in untrusted Web services. Vanish ensures that *all* copies of certain data – such as emails on Hotmail, photos on Facebook, or documents on Google Docs – become unreadable after a user-specified time. Vanish's initial design was introduced in a 2009 paper [81] and subsequently revised for increased security in a 2011 technical report [80].

3.1 Motivation and Overview

We target the goal of creating data that self-destructs or vanishes *automatically* after it is no longer useful. Moreover, it should do so *without* any explicit action by the users or any party storing or archiving that data, in such a way that *all* copies of the data vanish simultaneously from all storage sites, online or offline.

Numerous applications could benefit from self-destructing data. As one example, consider the case of email. Emails are frequently cached, stored, or archived by email providers (e.g., Gmail, or Hotmail), local backup systems, ISPs, etc. Such emails may cease to have value to the sender and receiver after a short period of time. Nevertheless, many of these emails are private, and the act of storing them indefinitely at intermediate locations creates a potential privacy risk. For example, imagine that Ann sends an email to her friend discussing a sensitive topic, such as her relationship with her husband, the possibility of a divorce, or how to ward off a spurious lawsuit (see Figure 3.1(a)). This email has no value as soon as her friend reads it, and Ann would like that *all* copies of this email — regardless of where stored or cached — be automatically destroyed after a certain



(a) Example Scenario. (b) Vanishing Emails plugin for Gmail.

Figure 3.1: **Example Scenario and Vanish Email Screenshot.** (a) Ann wants to discuss her marital relationship with her friend, Carla, but does not want copies stored by intermediate services to be used in a potential child dispute trial in the future. (b) The screenshot shows how Carla reads a vanishing email that Ann has already sent to her using our Vanish Email Firefox plugin for Gmail.

period of time, rather than risk exposure in the future as part of a data breach, email provider mismanagement [140], or a legal action. In fact, Ann would prefer that these emails disappear early — and *not* be read by her friend — rather than risk disclosure to unintended parties. Both individuals and corporations could benefit from self-destructing emails.

More generally, self-destructing data is broadly applicable in today's Web-centered world, where users' sensitive data can persist "in the cloud" indefinitely. With self-destructing data, users can regain control over the lifetimes of their Web objects, such as private messages on Facebook, documents on Google Docs, or private photos on Flickr. Similarly, this concept could be used to protect the privacy of sensitive SMS and MMS text messages, which are increasingly being used as evidence in divorce cases and other legal actions [219, 146]. As a news article states, "don't ever say anything on e-mail or text messaging that you don't want to come back and bite you [141]." Finally, some have argued that the right and ability to destroy data are essential to protect fundamental societal goals like privacy and liberty [122, 144].

Observation and Goals. A key observation in these examples is that users need to keep certain data for only a limited period of time. After that time, access to that data should be revoked for *everyone*

– including the legitimate users of that data, the known or unknown entities holding copies of it, and the attackers. This mechanism will not be universally applicable to all users or data types; instead, we focus in particular on sensitive data that a user would prefer to see destroyed early rather than fall into the wrong hands.

Motivated by the above examples, as well as our observation above, we ask whether it is possible to create a system that can permanently delete data after a timeout: (1) even if an attacker can retroactively obtain a pristine copy of that data *and* any relevant persistent cryptographic keys and passphrases from *before* that timeout; (2) without the use of any explicit delete action by the user or the parties storing that data; and (3) without relying on any dedicated centralized service.

A system achieving these goals would be broadly applicable in the modern digital world as we've previously noted, e.g., for files, private blog posts, on-line documents, Facebook entries, content-sharing sites, emails, messages, etc. In fact, the privacy of any digital content could potentially be enhanced with self-deleting data. However, implementing a system that achieves this goal set is challenging. Section 3.2.1 describes many natural approaches that one might attempt and how they all fall short. In this chapter we focus on a specific self-deleting data scheme that we have implemented, using email as an example application.

Our Approach. The key insight behind our approach and the corresponding system, called *Vanish*, is to leverage the services provided by existing decentralized, global-scale infrastructures, such as peer-to-peer distributed hash tables (DHTs) or the World-Wide-Web. Intuitively, Vanish encrypts a user's data locally with a random encryption key not known to the user, *destroys* the local copy of the key, and then sprinkles pieces (Shamir secret shares [157]) of the key across random Internet locations in the decentralized system. For example, a DHT-based Vanish implementation selects random indexes and stores the key pieces at the nodes responsible for storing those indexes; a WWW-based implementation selects random Web sites and stores or computes shares to/from them.

In this chapter, we present Vanish's design, implementation, and evaluation, focusing on a DHT-based approach. In Section 3.7, we show how Vanish can be extended to leverage other global-scale decentralized infrastructures, such as the WWW. Our initial choice of DHTs as storage systems for Vanish stems from three intuitive properties that make DHTs attractive for our data destruction goals. First, their huge scale (over 1 million nodes for the Vuze DHT [67]), geographical distribu-

tion of nodes across many countries, and complete decentralization make them robust to powerful and legally influential adversaries. Second, DHTs are designed to provide reliable distributed storage [123, 171, 198]; we leverage this property to ensure that the protected data remains available to the user for a desired interval of time. Finally, DHTs have an inherent property that intuitively helps with data destruction: the fact that the DHT is constantly changing means that the sprinkled information will disappear over time as the DHT nodes churn or internally cleanse themselves by explicitly erasing old values, thereby rendering protected data permanently unavailable. In fact, it may be impossible to determine retroactively which nodes were responsible for storing a given value in the past. Unfortunately, Sybil attacks [62] and aggressive replication schemes in today's deployed DHTs challenge these intuitive properties in practice. Hence, in addition to the Vanish design, we also demonstrate how DHTs can be re-designed to match Vanish's needs by describing our own large-scale deployment of changes to an existing, million-node DHT (Vuze DHT).

Implementation and Evaluation. To demonstrate the viability of our approach, we implemented three prototype Vanish implementations: one that relies on Bittorrent's Vuze DHT client [214], one that leverages the WWW as its key store, and one that allows the combination of two or more key stores – such as DHTs and the WWW – for increased security and defense-in-depth. On top of all three Vanish prototypes, we built two applications: a Firefox plugin for Gmail and other Web sites, and a self-destructing file management application. Figure 3.1(b) shows how a user can decapsulate a vanishing email from her friend using our Gmail plugin (see Section 3.4 for detailed interface description). Our performance evaluation shows that simple optimizations can support even latency-sensitive applications, such as our Gmail plugin, with acceptable user-visible execution times.

Security is critical for our system and hence we consider it in depth. Vanish targets *post-facto, retroactive attacks*; that is, it defends the user against future attacks on old, forgotten, or unreachable copies of her data. For example, consider the subpoena of Ann's email conversation with her friend in the event of a divorce. In this context, the attacker does not know what specific content to attack until *after* that content has expired. As a result the attacker's job is very difficult, since he must develop an infrastructure capable of attacking *all* users at *all* times. We leverage this observation to estimate the cost for such an attacker. We target no formal security proofs, but rather evaluate the security of our system analytically and experimentally.

In particular, DHT-specific attacks are a major concern for security, and we dedicate thorough analysis to some of them, including detailed, large-scale experimentation with deployed defenses against select attacks on the Vuze DHT. However, we explicitly do not seek to deploy or measure defenses against *all* known DHT-specific attacks, but rather direct the reader to past research on defenses against some of them (e.g., [28, 35, 45, 62, 160]). For improved security of our system against still unpatched vulnerabilities, we present extensions to Vanish's core architecture that allow its use of other key stores and combinations thereof for defense in depth.

Contributions. While the basic idea of our approach is simple conceptually, care must be taken in handling and evaluating the mechanisms employed to ensure its security, practicality, and performance. Looking ahead, and after briefly considering other tempting approaches for creating self-destructing data (Section 3.2.1), the key contributions of this work are to:

- identify the principal requirements and goals for self-destructing data (Section 3.2.2);
- propose a method for achieving these goals that combines cryptography with decentralized, global-scale systems, such as DHTs (Section 3.3);
- present a Vanish prototype implementation on top of the existing, million-node Vuze DHT, as well as applications on top of it (Section 3.4);
- evaluate Vanish's security analytically and experimentally (Section 3.5); our analysis includes the design, large-scale deployment, and experimental evaluation of a new security-oriented Vuze DHT design that better meets Vanish's security requirements (Section 3.6); and
- describe and evaluate a set of architectural extensions that increase Vanish's security in front of retroactive attacks (Section 3.7).

Together, these contributions provide the foundation for empowering users with greater control over the lifetimes of private data scattered across the Internet.

3.2 Candidate Approaches, Goals, and Threat Models

We now review potential options for protecting Web data against retroactive attacks against privacy and their limitations, after which we formulate our own goals, assumptions, and threat models for Vanish.

3.2.1 Potential Approaches and Their Limitations

Although prior to our Vanish contribution lifetime control for Web data had been largely unaddressed, a number of existing and seemingly natural approaches may appear applicable to achieving our objectives. Upon deeper investigation, however, we find that none of these approaches are sufficient to achieve the goals enumerated in Section 3.1. We consider these strawman approaches here and use them to further motivate our design constraints in Section 3.2.2.

One obvious approach might be to use a standard public key or symmetric encryption scheme, as provided by systems like PGP and its open source counterpart, GPG. However, traditional encryption schemes are insufficient for our goals, as they are designed to protect against adversaries without access to the decryption keys. Under our model, though, we assume that the attacker will be able to obtain access to the decryption keys, e.g., through a court order or subpoena.[1]

A potential alternative to standard encryption might be to use forward-secure encryption [17, 34], yet our goal is strictly stronger than forward secrecy. Forward secrecy means that if an attacker learns the state of the user's cryptographic keys at some point in time, they should not be able to decrypt data encrypted at an earlier time. However, due to caching, backup archives, and the threat of subpoenas or other court orders, we allow the attacker to either view past cryptographic state or force the user to decrypt his data, thereby violating the model for forward-secure encryption. For similar reasons, plus our desire to avoid introducing new trusted agents or secure hardware, we do not use other cryptographic approaches like key-insulated [16, 59] and intrusion-resilient [57, 58] cryptography. Finally, while exposure-resilient cryptography [32, 60, 61] allows an attacker to view parts of a key, we must allow an attacker to view all of the key.

For online, interactive communications systems, an ephemeral key exchange process can protect derived symmetric keys from future disclosures of asymmetric private keys. A system like OTR [4, 29] is particularly attractive, but as the original OTR paper observes, this approach is not directly suited for less-interactive email applications, and similar arguments can be made for OTR's unsuitability for the other above-mentioned applications as well.

An approach with goals similar to ours is the Ephemerizer family of solutions [138, 148, 149],

[1] U.S. courts are debating whether citizens are required to disclose private keys, although the ultimate verdict is unclear. We thus target technologies robust against a verdict in either direction [84, 139]. Other countries such as the U.K. [143] require release of keys, and coercion or force may be an issue in yet other countries.

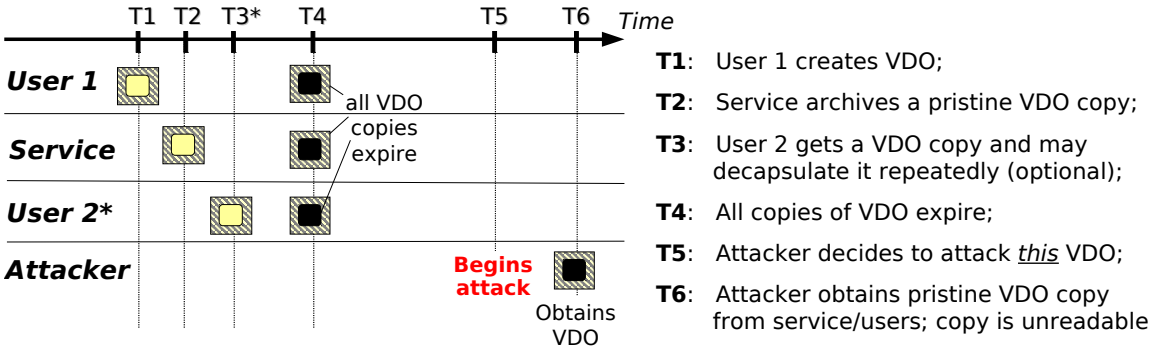


Figure 3.2: **Timeline for VDO Usage and Attack.**

along with revocable backup systems [27]. These approaches require the introduction of one or more (possibly thresholded) trusted third parties which (informally) escrow information necessary to access the protected contents. These third parties destroy this extra data after a specified timeout. The biggest risks with such centralized solutions are that they may either not be trustworthy, or that even if they are trustworthy, users may still not trust them, hence limiting their adoption. For example, if Ann does not trust Gmail or Facebook to delete her data, why would she trust another centralized service to do so? Indeed, many users may be wary to the use of dedicated, centralized trusted third-party services after it was revealed that the Hushmail email encryption service was offering the cleartext contents of encrypted messages to the federal government [184]. This challenge calls for a decentralized approach with fewer real risks *and* perceived risks.

3.2.2 Goals and Assumptions

To support a broad range of target applications (self-destructing email, Facebook messages, text messages, trash bins, etc.), we introduce the notion of a *vanishing data object* (*VDO*). A VDO encapsulates the user's data (such as a message, photo, or file) and prevents its contents from persisting indefinitely and becoming a source of retroactive information leakage. Regardless of whether the VDO is copied, transmitted, or stored in the Internet, it becomes unreadable after a predefined time period even if an attacker *retroactively* obtains both a *pristine* copy of the VDO from *before* its expiration and all the user's past persistent cryptographic keys and passwords. Figure 3.2 illustrates these VDO properties by showing the timeline for typical VDO usage and attack.

Our VDO abstraction and Vanish system make several key assumptions:

1. *Time-limited value.* The VDO encapsulates data that is of value to the user and the user's trusted communicants for only a limited period of time (e.g., a few days or weeks for emails, Web objects, and SMSs).
2. *Known timeout.* Users creating a VDO know the approximate VDO lifetime they desire.
3. *Internet connectivity.* Users are connected to the Internet when interacting with VDOs. Internet connectivity is presently required for many applications, such as sending and receiving email or interacting with the Web, and the move towards ubiquitous connectivity through WiFi and 3G makes this assumption reasonable for many other applications, as well. Such connectivity is not required for deletion; a VDO will become unreadable even if connectivity is removed from its storage site (or if that storage site is turned off).
4. *Dispensable under attack.* Vanish is designed for use with data that is private but whose persistence is not critical. Rather than risk exposure, users prefer that the VDO be destroyed prematurely.
5. *Trusted communicants.* Users sharing VDOs trust each other not to save decapsulated (cleartext) copies.

Based on these assumptions, we formulate the following goals for self-destructing data and Vanish:

1. *Destruction after timeout.* A VDO must expire automatically and *without* any explicit action on the part of its users or any party storing a copy of the VDO. Once expired, the VDO must also be inaccessible to any party who obtains a *pristine* copy of the VDO from *prior* to its expiration.
2. *Accessible until timeout.* During its lifetime, a VDO's contents should be available to legitimate users.
3. *No single or small number of trusted parties.* A VDO's post-timeout privacy should not depend on the correct functioning of one or a few trusted parties.
4. *Leverage existing infrastructures.* The system must leverage existing infrastructures. It must not rely on external, special-purpose dedicated services.
5. *No secure hardware.* The system must not require the use of dedicated secure hardware.

A corollary of goal (1) is that the VDO will become unavailable to the legitimate users after the timeout, which is compatible with our applications and assumption of time-limited value.

Our desire to leverage existing infrastructure (goal (4)) and to have no single point of trust stems from our belief that special-purpose services may hinder adoption. As noted previously, Hushmail's disclosure of the contents of users' encrypted emails to the federal government [184] suggests that, even if the centralized service or a threshold subset of a collection of centralized services is trustworthy, users may still be unwilling to trust them.

3.2.3 Threat Model

The above list enumerates the intended properties of the system *without* the presence of an adversary. We now consider the various classes of potential adversaries against the Vanish system, as well as the desired behavior of our system in the presence of such adversaries.

The central security goal of Vanish is to ensure the destruction of data after a timeout, despite potential adversaries who might attempt to access that data after its timeout. Obviously, care must be taken in defining what a plausible adversary is, and we do that below and in Section 3.5. But we also stress that we explicitly do *not* seek to preserve goal (2) — accessible prior to a timeout — in the presence of adversaries. As previously noted, we believe that users would prefer to sacrifice availability pre-timeout in favor of assured destruction for the types of data we are protecting. For example, we do not defend against denial of service attacks that could prevent reading of the data during its lifetime. Moreover, we explicitly do not seek to protect the data's privacy against pre-timeout attacks; such a property is either provided by traditional encryption mechanisms (if the attacker does not know the user's keys) or seems impossible (if the attacker has access to the encrypted data and the user's keys). Making these assumptions let us focus on the primary novel insights in this work: methods for leveraging decentralized, large-scale, distributed systems in order to ensure data destruction.

We therefore focus our threat model and subsequent analyses on attackers who wish to compromise data privacy. Two key properties of our threat model are:

1. *Trusted data owners.* Users with legitimate access to the same VDOs trust each other.
2. *Retroactive attacks on privacy.* Attackers do not know which VDOs or users they want to access until *after* the VDOs expire.

3. *Unrestricted access to past data and passphrases following the attack.* Attackers are allowed access to all data stored on persistent storage prior to the start of the retroactive attack, as well as access to all the user's past passphrases.
4. *Possible precomputation.* While attackers do not know which VDOs or users they want to access until *after* the VDOs timeout, they *can* do arbitrary VDO-agnostic precomputations at any time in preparation for a possible future attack.

The first aspect of the threat model is straightforward, and in fact is a shared assumption with traditional encryption schemes: it would be impossible for our system to protect against a user who chooses to leak or permanently preserve the cleartext contents of a VDO-encapsulated file through out-of-band means. For example, if Ann sends Carla a VDO-encapsulated email, Ann must trust Carla not to print and store a hard-copy of the email in cleartext.

The second aspect of the threat model – that the attacker does not know the identity of a specific VDO of interest until *after* its expiration – was discussed briefly in Section 3.1. For example, email or SMS subpoenas typically come long after the user sends a particular sensitive email. Therefore, our system defends the user against *future attacks against old copies of private data*.

The third aspect of the threat model implies that attackers have access to all past, persistent cryptographic keys (conceptually, any cryptographic information that must survive a system reboot). While some implementations may strive to make such access as difficult as possible – such as storing keys on tamper-responding secure tokens – we seek a solution that provides a high level of security even without access to such storage mechanisms, which may be unavailable or may have unknown vulnerabilities, e.g., following [8].

The retroactive restriction (bullet (2) in the threat model) describes an attacker who does not know which VDOs or users he will wish to attack until sometime after those VDOs timeout. This reflects a natural class of threats. However, an attacker may know that he'll wish to expose some (as yet unknown) VDOs in the future. Such an attacker might try to maximize his potential success *prior* to knowing which VDOs will be of interest. Referring to Figure 3.2, an attacker might attempt some form of precomputation against all VDOs created by all users at all times, and then try to leverage the results of that precomputation once the target VDO is known at time T5. This is captured by bullet (4) of the threat model. The precise form of precomputation will depend on the adversary in question, as well as our system's design.

3.3 Vanish Architecture

We designed and implemented Vanish, a system capable of satisfying all of the goals listed in Section 3.2.2. A key contribution of our work is to leverage existing, decentralized, large-scale Distributed Hash Tables (DHTs). After providing a brief overview of DHTs and introducing the insights that underlie our solution, we present our system's architecture and components.

Overview of DHTs. A DHT is a distributed, peer-to-peer (P2P) storage network consisting of multiple participating *nodes* [123, 171, 198]. The design of DHTs varies, but DHTs like Vuze generally exhibit a put/get interface for reading and storing data, which is implemented internally by three operations: `lookup`, `get`, and `store`. The data itself consists of an $(\textit{index}, \textit{value})$ pair. Each node in the DHT manages a part of an astronomically large index name space (e.g., 2^{160} values for Vuze). To store data, a client first performs a `lookup` to determine the nodes responsible for the index; it then issues a `store` to the responsible node, who saves that $(\textit{index}, \textit{value})$ pair in its local DHT database. To retrieve the value at a particular index, the client would `lookup` the nodes responsible for the index and then issue `get` requests to those nodes. Internally, a DHT may replicate data on multiple nodes to increase availability. Finally, DHTs typically enforce limits on the time some data item can be stored at a particular index, to protect storage nodes from storage denial-of-service (DoS) attacks. For example, a Vuze DHT node storing some object erases that data at a configurable time that must be less than three days.

Numerous DHTs exist in the Internet, including Vuze, Mainline, and KAD. These DHTs are *communal*, i.e., any client can join, although DHTs such as OpenDHT [164] only allow authorized nodes to join.

DHT-related Insights. Three intuitive properties of DHTs make them appealing for use in the context of a self-destructing data system:

1. *Availability.* Years of research in availability in DHTs have resulted in relatively robust properties of today's systems, which typically provide good availability of data prior to a specific timeout.
2. *Scale, geographic distribution, and decentralization.* Measurements of the Vuze and uTorrent DHTs estimate in excess of one million [67] and five million simultaneously active DHT

nodes, respectively. The data in [194] shows that while the U.S. is the largest single contributor of nodes in Vuze, a majority of the nodes lie outside the U.S. and are distributed over 190 countries.

3. *Inherent Cleansing Properties.* DHTs evolve naturally and dynamically over time as new nodes constantly join, old nodes leave, and nodes internally cleanse themselves. We call this evolution *churn*. The average lifetime of a node in the DHT varies across networks and has been measured from minutes on Kazaa [88] to hours on Vuze/Azureus [67]. Similarly, all DHTs present internal cleansing functions that protect them against storage DoS attacks. For example, Vuze imposes a maximum 72-hour time limit and OpenDHT imposes a one-week time limit.

The first property provides us with solid grounds for implementing a useful system. The second property makes DHTs more resilient to certain types of attacks than centralized or small-scale systems. For example, while a centrally administered system can be compelled to release data by an attacker with legal leverage [184], obtaining subpoenas for multiple nodes storing a VDO's key pieces would be significantly harder, and in some cases impossible, due to their distribution under different administrative and political domains.

Traditionally, DHT research has tried to counter the negative effects of churn on availability. For our purposes, however, the constant churn in the DHT is an advantage, because it means that data stored in DHTs will naturally and irreversibly disappear over time as the DHT evolves.

Vanish. Vanish is designed to leverage one or more DHTs. Figure 3.3 illustrates the high-level system architecture. At its core, Vanish takes a data object D (and possibly an explicit timeout T), and encapsulates it into a VDO V.

In more detail, to encapsulate the data D, Vanish picks a random data key, K, and encrypts D with K to obtain a ciphertext C. Not surprisingly, Vanish uses threshold secret sharing [181] to split the data key K into N pieces (*shares*) $K_1, \ldots, K_N$. A parameter of the secret sharing is a *threshold* (M) that can be set by the user or by an application using Vanish. The threshold determines how many of the N shares are required to reconstruct the original key. For example, if we split the key into $N = 20$ shares and the threshold is 10 keys, then we can compute the key given any 10 of the

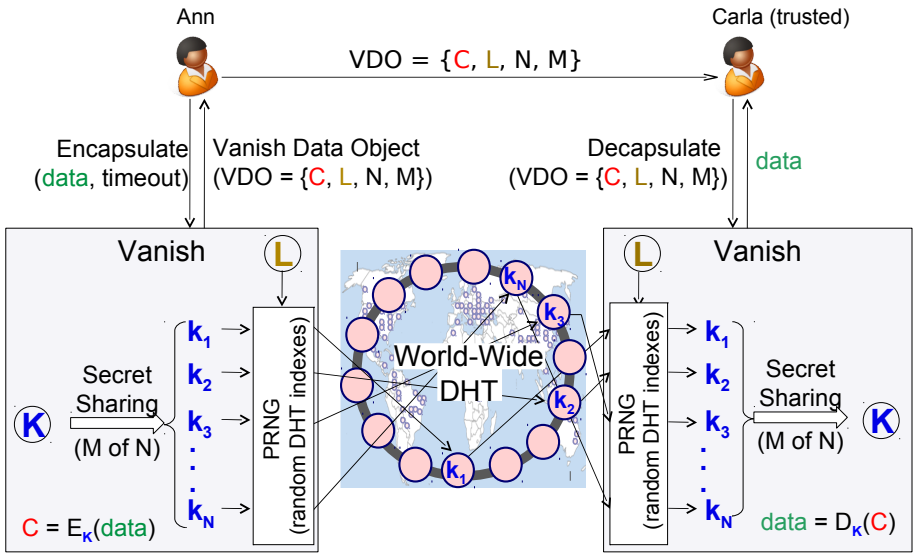


Figure 3.3: **The Vanish System Architecture.** Shows the encapsulation and decapsulation operations. PRNG denotes a secure pseudo-random number generator.

20 shares. In this section we often refer to the *threshold ratio* (or simply threshold) as the percentage of the N keys required, e.g., in the example above the threshold ratio is 50%.

Once Vanish has computed the key shares, it picks at random an *access key*, L. It then uses a cryptographically secure pseudorandom number generator [25], keyed by L, to derive N indices into the DHT, $I_1, \ldots, I_N$. Vanish then sprinkles the N shares $K_1, \ldots, K_N$ at these pseudorandom locations throughout the DHT; specifically, for each $i \in \{1, \ldots, N\}$, Vanish stores the share K_i at index I_i in the DHT. If the DHT allows a variable timeout, e.g., with OpenDHT, Vanish will also set the user-chosen timeout T for each share.

The final VDO V consists of (L, C, N, M) and is sent over to the email server or stored in the file system upon encapsulation. The decapsulation of V happens in the natural way, assuming that it has not timed out. Given VDO V, Vanish (1) extracts the access key, L, (2) derives the locations of the shares of K, (3) retrieves the required number of shares as specified by the threshold, (4) reconstructs K, and (5) decrypts C to obtain D.

VDO destruction occurs due to independent loss of the key shares by their storage peers. Two processes cause share loss in a DHT. First, each of the nodes is programmed to erase key shares after the specified time. All existing DHTs have an inherent data expiration function, which is essential

for them to protect themselves against storage DoS attacks. VDO timeout flexibility depends on the underlying DHT. For example, the Vuze DHT currently supports hourly timeouts of up to three days and OpenDHT allows timeouts of up to one week. Second, churn causes key shares to be lost from the DHT, as nodes leave the network and the DHT's structure itself evolves. As a result, the DHT will lose key shares over time. Once more than $(N - threshold)$ shares are lost, the VDO becomes permanently unavailable. Hence, in Vanish, key share disappearance from the independent and automonous DHT nodes makes the data disappear on its own without relying on actions by any centralized party.

Threshold Secret Sharing, Security, and Robustness. For security we rely on the property that the shares $K_1, \ldots, K_N$ will disappear from the DHT over time, thereby limiting a retroactive adversary's ability to obtain a sufficient number of shares, which must be $\geq$ the threshold ratio. In general, we use a ratio of $< 100\%$, otherwise the loss of a single share would cause the loss of the key. DHTs do lose data due to churn, and therefore a smaller ratio is needed to provide robust storage prior to the timeout. We consider all of these issues in more detail later; despite the conceptual simplicity of our approach, significant care and experimental analyses must be taken to assess the durability of our use of large-scale, decentralized DHTs.

Using multiple or no DHTs. As an extension to the scheme above, it is possible to store the shares of the data key K in *multiple* DHTs. For example, one might first split K into two shares K' and K'' such that both shares are required to reconstruct K. K' is then split into N' shares and sprinkled in the Vuze DHT, while K'' is split into N'' shares and sprinkled in OpenDHT. Such an approach would allow us to argue about security under different threat models, using OpenDHT's closed access (albeit small scale) and Vuze's large scale (albeit communal) access.

An alternate model would be to abandon DHTs and to store the key shares on distributed but managed nodes. This approach bears limitations similar to Ephemerizer (Section 3.2.1). A hybrid approach might be to store shares of K' in a DHT and shares of K'' on managed nodes. This way, an attacker would have to subvert both the privately managed system *and* the DHT to compromise Vanish. Inspired by these ideas, we propose a set of architectural extensions for Vanish that allow it to (1) sprinkle key shares across random Web sites in the Internet and (2) combine multiple key stores, such as DHTs and Web sites, for increased security (Section 3.7).

Forensic Trails. Although not a common feature in today's DHTs, a future DHT or managed storage system could additionally provide a forensic trail for monitoring accesses to protected content. A DHT could, for example, record the IP addresses of the clients that query for particular indices and make that information available to the originator of that content. The existence of such a forensic trail, even if probabilistic, could dissuade third parties from accessing the contents of VDOs that they obtain prior to timeout. Chapter 4 shows how customizable DHTs could in the future support particular features like forensic trails without the need for large-scale deployments.

Composition. Our system is not designed to protect against all attacks, especially those for which solutions are already known. Rather, we designed both the system and our applications to be composable with other systems to support defense-in-depth. For example, our Vanish Gmail plugin can be composed with GPG in order to avoid VDO sniffing by malicious email services. Similarly, our system can compose with Tor to ensure anonymity and throttle targeted attacks.

3.4 Prototype, Applications, and Performance Evaluation

We have implemented a Vanish prototype capable of integrating with both Vuze and OpenDHT. To integrate Vanish with the Vuze DHT, we made two types of changes to the Vuze client: local changes and deployed changes. The local changes needed patching of only the Vanish user's Vuze code and included: (1) a security measure to prevent `lookup` sniffing attacks (see Section 3.6) and (2) several performance optimizations suggested by prior work [67] to achieve reasonable performance for our applications. The deployed changes required full-scale deployment on the live, million-node Vuze DHT before they could take advantage. These included: code that parameterizes data timeouts to one-hour increments (prior to our changes, the Vuze DHT used to impose fixed 8-hour limits on data lifetime in the DHT), a new replication scheme that limits the amount of data dissemination in the DHT for security reasons, and a new lightweight anti-Sybil defense. Section 3.6 describes the most relevant modifications.

3.4.1 Vuze Background

The Vuze (a.k.a. Azureus) DHT is based on the Kademlia [123] protocol. Each DHT node is assigned a "random" 160-bit ID based on its IP and port, which determines the index ranges that it will store. To store an $(\text{index}, \text{value})$ pair in the DHT, a client looks up 20 nodes with IDs closest

(a) Vanishing Facebook messages.

(b) Google Doc with vanishing parts.

Figure 3.4: **Vanish Applications.** Screenshots of two example uses of vanishing data objects on the Web. (a) Carla is attempting to decapsulate a VDO she received from Ann in a Facebook message. (b) Ann and Carla are drafting Ann's divorce document using a Google Doc; they encapsulate sensitive, draft information inside VDOs until they finalize their position.

to the specified index and then sends `store` messages to them. Vuze nodes republish the entries in their cache database every 30 minutes to the other 19 nodes closest to the value's index in order to combat churn in the DHT. Nodes further *remove* from their caches all values that were stored more than a given timeout in the past. The timeout can be specified by a client upon `store` on a one-hour granularity and cannot exceed 72 hours.[2]

3.4.2 Vanish Applications

We built two prototype applications that use a Vanish daemon running locally or remotely to ensure self-destruction of various types of data.

FireVanish. We implemented a Firefox plugin for the popular Gmail service that provides the option of sending and reading self-destructing emails. Our implementation requires no server-side

[2]Originally, Vuze had a fixed 8-h timeout for all data; with the help of Vuze engineers, we have deployed a parameterized timeout of maximum 72h.

changes. The plugin uses the Vanish daemon both to transform an email into a VDO before sending it to Gmail and similarly for extracting the contents of a VDO on the receiver side.

Our plugin is implemented as an extension of FireGPG (an existing GPG plugin for Gmail) and adds Vanish-related browser overlay controls and functions. Using our FireVanish plugin, a user types the body of her email into the Gmail text box as usual and then clicks on a *"Create a Vanishing Email"* button that the plugin overlays atop the Gmail interface. The plugin encapsulates the user's typed email body into a VDO by issuing a VDO-create request to Vanish, replaces the contents of the Gmail text box with an encoding of the VDO, and uploads the VDO email to Gmail for delivery. The user can optionally wrap the VDO in GPG for increased protection against malicious services. In our current implementation, each email is encapsulated with its own VDO, though a multi-email wrapping would also be possible (e.g., all emails in the same thread).

When the receiving user clicks on his email, FireVanish inspects whether it is a VDO email, a PGP email, or a regular email. Regular emails require no further action. PGP emails are first decrypted and then inspected to determine whether the underlying message is a VDO email. For VDO emails, the plugin overlays a link "*Decapsulate this email"* atop Gmail's regular interface (shown previously in Figure 3.1(b)). Clicking on this link causes the plugin to invoke Vanish to attempt to retrieve the cleartext body from the VDO. If the VDO has not yet expired, the plugin pops up a new window showing the email's cleartext body; otherwise, an error message is displayed.

FireVanish Extension for the Web. Self-destructing data is broadly applicable in today's Web-oriented world, in which users often leave permanent traces on many Web sites [189]. Given the opportunity, many privacy-concerned users would likely prefer that certain messages on Facebook, documents on Google Docs, or emails on Gmail disappear within a short period of time.

To make Vanish broadly accessible for Web usage, FireVanish provides a simple, generic, yet powerful, interface that permits all of these applications. Once the FireVanish plugin has been installed, a Firefox user can select text in any Web page input box, right click on that selected text, and cause FireVanish to replace that text *inline* with an ecapsulated VDO. Similarly, when reading a Web page containing a VDO, a user can select that VDO and right click to decapsulate it; in this case, FireVanish leaves the VDO in place and displays the cleartext in a separate popup window. Some Web sites, such as Gmail, may store draft copies of the text as it is being written;

N	Time (seconds)		
	Encapsulate VDO		Decapsulate VDO
	Without prepush	With prepush	
10	10.5	0.082	0.9
20	16.9	0.082	2.0
50	32.8	0.082	4.7
100	64.5	0.082	9.2
150	94.7	0.082	14.0
200	124.3	0.082	19.0

Table 3.1: **VDO Encapsulation and Decapsulation Performance.**

for such cases, we recommend that the user either install additional existing plugins to disable that functionality or type the email in a separate textbox.

Figure 3.4 shows two uses of FireVanish to encapsulate and read VDOs within Facebook and Google Docs. The screenshots demonstrate a powerful concept: FireVanish can be used seamlessly to empower privacy-aware users with the ability to limit the lifetime of their data on Web applications that are unaware of Vanish.

Vanishing Files. Finally, we have implemented a vanishing file application, which can be used directly or by other applications, such as a self-destructing trash bin or Microsoft Word's autosave. Users can wrap sensitive files into self-destructing VDOs, which expire after a given timeout. In our prototype, the application creates a VDO wrapping one or more files, deletes the cleartext files from disk, and stores the VDO in their place. This ensures that, even if an attacker copies the raw bits from the laptop's disks after the timeout, the data within the VDO will be unavailable. Like traditional file encryption, Vanishing Files relies upon existing techniques for securely shredding data stored on disks or memory.

3.4.3 Performance Evaluation

Although performance is not a primary concern for Vanish and its applications, we now present a brief performance evaluation aimed at demonstrating that Vanish is practical from a performance perspective. In the next section, we switch our evaluation criteria to more critical aspects of our system, such as security and availability, which we evaluate in depth.

We measured the performance of our Vuze-based SDDS, focusing on the times needed to encap-

sulate and decapsulate a VDO. Our measurements used an Intel T2500 2.0GHz DUO with 2GB of RAM, Java 1.6, and a broadband network. In all experiments, Vuze operations (`put`, `get`) accounted for 99% of all execution times for medium-sized data (up to tens of MB, like most emails). Encryption, Shamir secret sharing operations, and encryption/decryption accounted for the remaining 1%. For much larger data sizes (e.g., files over hundreds of MB), encryption/decryption becomes the dominating component.

Table 3.1 shows optimized encapsulation and decapsulation times for 2KB of data as a function of the number of shares (N) (threshold is 0.83% for all values of N). Our experiments revealed the importance of configuring Vuze DHT parameters on our latency-aware applications. With no special tuning, Vuze took over 4 minutes to store 60 shares, even using parallel stores. By employing several Vuze performance improvements suggested in [67], we reduced the store time to 32 seconds, as shown in the *without prepush* column of the Table. While this time is 7x better than the non-tuned Vuze, we still judged it as too slow for our application. We therefore implemented a simple optimization: to mask storage delays from the user, our system proactively generates data keys and pre-pushes shares into the DHT. This optimization led to an unnoticeable DHT encapsulation time of $82ms$, shown in the *with-prepush* column. The graph shows that decapsulation is relatively fast — under 5 seconds for $N = 60$, which is reasonable for emails and similar content.

3.5 Security Analysis

To evaluate Vanish's security, we seek to assess two key properties: that (1) Vanish does not introduce any *new* threats to privacy, and (2) our Vuze-based prototype can be made secure against adversaries attempting to retroactively read a VDO post-expiration.

It is straightforward to see that Vanish adds no new privacy risks. In particular, the key shares stored in the DHT are *not* a function of the encapsulated data D; only the VDO is a function of D. Hence, if an adversary is unable to learn D when the user does not use Vanish, then the adversary would be unable to learn D if the user does use Vanish. There are three caveats, however. First, external parties, like the DHT, might infer information about who is communicating with whom (although the use of an anonymization system like Tor can alleviate this concern). Second, given the properties of Vanish, users might choose to communicate information that they might not

communicate otherwise, thus amplifying the consequences of any successful data breach. Third, the use of Vanish might raise new legal implications. In particular, the new "eDiscovery" rules embraced by the U.S. may require a user to preserve emails and other data once in anticipation of a litigious action. The exact legal implications to Vanish are unclear; the user might need to decapsulate and save any relevant VDOs to prevent them from automatic expiration.

The remainder of this security analysis, along with the evaluation of our deployed DHT security defenses in Section 3.6, focuses on *retroactive attacks against old data privacy*. These attacks, described in Section 3.2.3 and the timeline in Figure 3.2, are targeted at revoking the privacy of data encapsulated within expired VDOs. This section provides a broad analysis of such attacks and possible defenses, after which Section 3.6 dives deeply into attacks that integrate adversarial nodes directly into the DHT.

Retroactive Attackers. Our motivation is to protect against retroactive data disclosures, e.g., in response to a subpoena, court order, malicious compromise of archived data, or accidental data leakage. For some of these cases, such as the subpoena, the party initiating the subpoena is the obvious "attacker." The final attacker could be a user's ex-husband's lawyer, an insurance company, or a prosecutor. But executing a subpoena is a complex process involving many other actors, including potentially: the user's employer, the user's ISP, the user's email provider, unrelated nodes on the Internet, and other actors. For our purposes, we define *all* the involved actors as the "adversary."

Attack Strategies. The architecture and standard properties of the DHT cause significant challenges to an adversary who does *not* perform any computation or data interception prior to beginning the attack. First, the key shares are unlikely to remain in the DHT much after the timeout, so the adversary will be incapable of retrieving the shares directly from the DHT. Second, even if the adversary could legally subpoena the machines that hosted the shares in the past, the churn in Vuze makes it difficult to determine the identities of those machines; many of the hosting nodes would have long disappeared from the network or changed their DHT index. Finally, with Vuze nodes scattered throughout the globe [194], gaining legal access to those machines raises further challenges. In fact, these are all reasons why the use of a DHT such as Vuze for our application is compelling.

We therefore focus on what an attacker might do *prior* to the expiration of a VDO, with the goal

of amplifying his ability to reveal the contents of the VDO in the *future*. We consider three principal strategies for such precomputation.

Strategy (1): Decapsulate VDO Prior to Expiration. An attacker might try to obtain a copy of the VDO and revoke its privacy *prior* to its expiration. This strategy makes the most sense when we consider, e.g., an email provider that proactively decapsulates all VDO emails in real-time in order to assist in responding to future subpoenas. The natural defense would be to further encapsulate VDOs in traditional encryption schemes, like PGP or GPG, which we support with our FireVanish application. The use of PGP or GPG would prevent the web-mail provider from decapsulating the VDO prior to expiration. And, by the time the user is forced to furnish her PGP private keys, the VDO would have expired. For the self-destructing trash bin and the Vanishing Files application, however, the risk of this attack is minimal.

Strategy (2): Sniff User's Internet Connection. An attacker might try to intercept and preserve the data users push into or retrieve from the DHT. An ISP or employer would be most appropriately positioned to exploit this vector. Two natural defenses exist for this: the first might be to use a DHT that by default encrypts communications between nodes. Adding a sufficient level of encryption to existing DHTs would be technically straightforward assuming that the ISP or employer were passive and hence not expected to mount man-in-the-middle attacks. For the encryption, Vanish could compose with an ephemeral key exchange system in order to ensure that these encrypted communications remain private even if users' keys are later exposed. Without modifying the DHT, the most natural solution is to compose with Tor [56] to tunnel one's interactions with a DHT through remote machines. One could also use a different exit node for each share to counter potentially malicious Tor exit nodes [125, 232], or use Tor for only a subset of the shares.

Strategy (3): Integrate into DHT. An attacker might try to integrate itself into the DHT in order to: create copies of all data that it is asked to `store`; intercept internal DHT `lookup` procedures and then issue `get` requests of his own for learned indices; mount a Sybil attack [62] (perhaps as part of one of the other attacks); or mount an Eclipse attack [186]. Such DHT-integrated attacks deserve further investigation, and we provide such an analysis in Section 3.6.

As foreshadowing to Section 3.6, we show that small, practical modifications deployed in the Vuze DHT can increase `store`-based sniffing attack cost by three orders of magnitude, making it

impractical for all but the most powerful attackers. Specifically, we have designed, implemented, deployed at scale, and measured two practical DHT defenses in the live, million-node Vuze DHT: (1) a conditional replication scheme that limits data an integrated attacker's sniffing opportunities and (2) an anti-Sybil defense. Combined, these techniques require to have extreme IP diversity for a successful `store`-based attack: the adversary must continuously control 10,000 IPs scattered in 1,000 distinct /24 IP prefixes and 20 distinct /15 IP prefixes. Moreover, we show how `lookup`-based attacks can be easily thwarted using localized changes to Vanish clients that require no deployment in the Vuze DHT.

Deployment Decisions. Given attack strategies (1) and (2), a user of FireVanish, Vanishing Files, or any future Vanish-based application is faced with several options: to use the basic Vanish system or to compose Vanish with other security mechanisms like PGP/GPG or Tor. The specific decision is based on the threats to the user for the application in question.

Vanish is oriented towards personal users concerned that old emails, Facebook messages, text messages, or files might come back to "bite" them, as eloquently put in [141]. Under such a scenario, an ISP trying to assist in future subpoenas seems unlikely, thus we argue that composing Vanish with Tor is unnecessary for most users. The use of Tor seems even less necessary for some of the threats we mentioned earlier, like a thief with a stolen laptop.

Similarly, it is reasonable to assume that email providers will not proactively decapsulate and archive Vanishing Emails prior to expiration. One factor is the potential illegality of such accesses under the DMCA, but even without the DMCA this seems unlikely. Therefore, users can simply employ the FireVanish Gmail plugin without needing to exchange public keys with their correspondents. However, because our plugin extends FireGPG, any user already familiar with GPG could leverage our plugin's GPG integration.

Data Sanitization. In addition to ensuring that Vanish meets its security and privacy goals, we must verify that the surrounding operating environment does not preserve information in a non-self-destructing way. For this reason, the system could leverage a broad set of approaches for sanitizing the Vanish environment, including secure methods for overwriting data on disk [89], encrypting virtual memory [159], and leveraging OS support for secure deallocation [42]. However, even absent those approaches, forensic analysis would be difficult if attempted much later than the data's

expiration for the reasons we've previously discussed: by the time the forensic analysis is attempted relevant data is likely to have disappeared from the user's machine, the DHT's natural churn and internal cleansing mechanisms would have made relevant shares and nodes disappear irrevocably.

3.6 Designing a Security-Sensitive DHT

We now focus on DHT-integrated adversaries and show how DHTs should be designed to be robust against them. DHT-integrated attacks are of two types: (1) generic attacks, which are applicable to any DHT application (e.g., Sybil [62], Eclipse [186]) and (2) Vanish-specific attacks, which are only applicable to our system (e.g., harvesting shares by sniffing `store` and `lookup` requests). This section presents our experience deploying and measuring at full scale practical defenses against *Vanish-specific* attacks in the million-node Vuze DHT (Sections 3.6.2 and 3.6.3), followed by a review of existing techniques for *generic* attacks (Section 3.6.4). To the best of our knowledge, ours are the first DHT defenses deployed and evaluated experimentally in a large, live, commercial DHT. We begin by defining DHT-integrated adversaries, after which we present the defenses.

3.6.1 DHT-Integrated Adversaries

An adversary who interacts with the DHT *prior* to a VDO's expiration can, in the future, aid in retroactive attacks against the VDO's privacy. During such a precomputation phase, however, the attacker does not know which VDOs (or even which users) he might eventually wish to attack. While the attacker could compile a list of worthwhile targets (e.g., politicians, actors, etc.), the use of Tor would thwart such targeted attacks. Hence, the principle strategy for the attacker would be to create a copy of as many key shares as possible. Moreover, the attacker must do this continuously – 24x7 – thereby further amplifying the burden on the attacker.

Such an attacker might be *external* to the DHT — simply using the standard DHT interface in order to obtain key shares — or *internal* to the DHT. While the former may be the only available approach for DHTs like OpenDHT, the approach is also the most limiting to an attacker since the shares are stored at pseudorandomly generated and hence unpredictable indices. An attacker integrating into a DHT like Vuze has significantly more opportunities and we therefore focus on such DHT-integrating adversaries here.

Attack vectors for DHT-integrated adversaries are enumerated in Section 3.5 (Strategy (3)). They include: sniffing shares by recording `store` requests issued when they are initially placed in the DHT or subsequently as they are being replicated (we call this attack a *data-crawling attack*, following the naming in [226]); sniffing share locations from `lookup` requests as they are being accessed by Vanish during decapsulations; and a variety of generic attacks that are known in the literature. We next present defenses against the first two attack types, after which we overview well-known techniques for dealing with the last attack type.

3.6.2 Defending Against Data-Crawling Attacks

A fundamental challenge in leveraging an existing DHT to build Vanish is the tension between availability and security. Because of churn in the DHT, good availability requires some replication of key shares and low secret sharing threshold ratios to withstand nodes that crash or leave the system. On the other hand, a high replication level increases the likelihood that a key share might be exposed to an untrustworthy node. Additionally, a low secret sharing threshold increases the likelihood that multiple nodes under an adversary's control may capture enough shares to reconstruct the key for a VDO. Therefore, good security demands little replication and high secret sharing threshold ratios.

We experimented with the Vanish prototype on two versions of the Vuze system: (1) the original Vuze prior to any Vanish-specific modifications, and (2) the current Vuze, which incorporates new security and functional enhancements that we designed and deployed on Vuze. Version v4.4 was the first to include our modifications; hence the remainder of this section denotes the original DHT as *pre-v4.4 Vuze* and the security-enhanced DHT as *post-v4.4 Vuze*. Our original SDDS prototype implementation was built on top of pre-v4.4 Vuze [81]. This section describes vulnerabilities in pre-v4.4 Vuze, which made it susceptible to `store`-sniffing attacks, and proposes two techniques, which combined raise the cost of a `store`-sniffing attack by three orders of magnitude.

Vulnerabilities in pre-v4.4 Vuze

Two vulnerabilities in pre-v4.4 Vuze design made our prototype particularly susceptible to the attack demonstrated in the Clearview paper [226]. First was an extremely aggressive replication system consisting of two unnecessarily eager mechanisms. Vuze's *push-on-join replication* scheme caused

neighbors of a new node to push copies of their contents to a node when it joined the DHT. A malicious node could therefore quickly obtain data in its ID-space vicinity and then "jump" from ID to ID (by picking different ports to assume different IDs) to sweep data in multiple regions. In addition, Vuze's *20-way replication* caused every node to replicate each of its values to its 19 closest neighbors every 30 minutes. These replication mechanisms made for a very efficient attack, yet as our measurements will show, these mechanisms were unnecessary for availability. Second, Vuze's lack of Sybil defense let an attacker scatter as many nodes as it pleased at any locations in the DHT, which allowed it to benefit from the eager replication traffic.

Guided by these observations, we modified Vuze in two ways:

1. *Conditional Replication:* We designed a new security-sensitive replication algorithm, called *conditional replication*, that significantly increases security while preserving availability.
2. *Anti-Sybil Defense:* We designed a new DHT ID fabrication function that imposes harsh limits on the set of IDs that can be fabricated from one physical machine and from various network IP *prefixes* (e.g., from within the network of a corporation or an organization).

We now describe and evaluate each of these modifications separately. Because neither of these defenses is enough to protect against data crawling, we defer an overall assessment of the security of the current Vuze-based Vanish prototype to Section 3.6.2.

Defense 1: Conditional Replication

In general, it is impossible to scan the Vuze DHT by guessing indexes – the SID_i values for a Vuze-based SDDS are chosen randomly from an astronomically large address space; however, the aggressive replication scheme of pre-v4.4 Vuze design made it possible for malicious DHT nodes to learn a large number of values simply by listening to replication messages from nearby nodes. We call this a *data crawling* precomputation attack because malicious nodes attempt to "crawl" the hash table simply through large-scale infiltration and protocol listening.

To limit the opportunities for data crawling, we leverage the following crucial observation. Thanks to our use of secret sharing, Vanish can tolerate some data loss and in fact *prefers it to over-replication*. Guided on this observation, we altered Vuze's replication mechanism in three ways. First and most obvious, we completely eliminated push-on-join replication, which allowed an

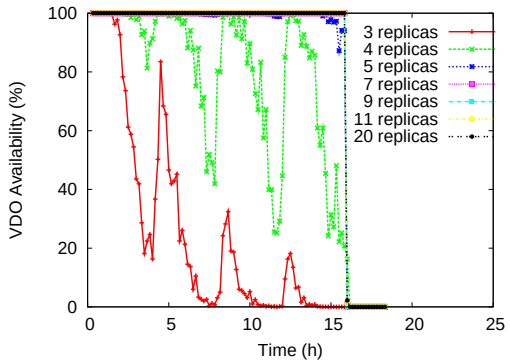


Figure 3.5: **VDO Availability Under Conditional Replication.** Availability for 60 shares, threshold ratio 85%, and various replication schemes: every-4h conditional replication with 3–11 replicas and default 20-way, every-30-minute Vuze replication.

attacker to capture a large number of the shares during an 8-hour period, even with a small number of simultaneous nodes [226].

Second, in place of Vuze's default 20-way replication scheme, we designed and deployed a new replication algorithm, called *conditional replication*, which severely limits the amount of replication going on in the system. With conditional replication, a Vuze node considers replicating a value only when a specified *minimum replication interval* has passed since the value was last replicated or stored. Imposing a minimum replication interval helps us protect against attacks where colluding nodes might attempt to force a node into replicating prematurely (a variant of the cuddling attack described in [226]). Moreover, a node first checks to see how many replicas exist for the value before replicating. If the number of existing replicas is at or above a specified *replication factor*, no replication is performed; otherwise, the node bumps the number of value replicas back to the replication factor. Finally, we parameterized Vuze's replication factor and, using experiments, were able to show that a mere replication factor of 5 (instead of 20) is needed to guarantee high VDO availability (99.99%) in face of today's churn condition in Vuze.

Thus, conditional replication reduces data dissemination caused by replication traffic. However, without caution that reduction has the potential to harm data availability. Below we discuss the availability/security tradeoff of conditional replication and examine its impact on crawling attacks.

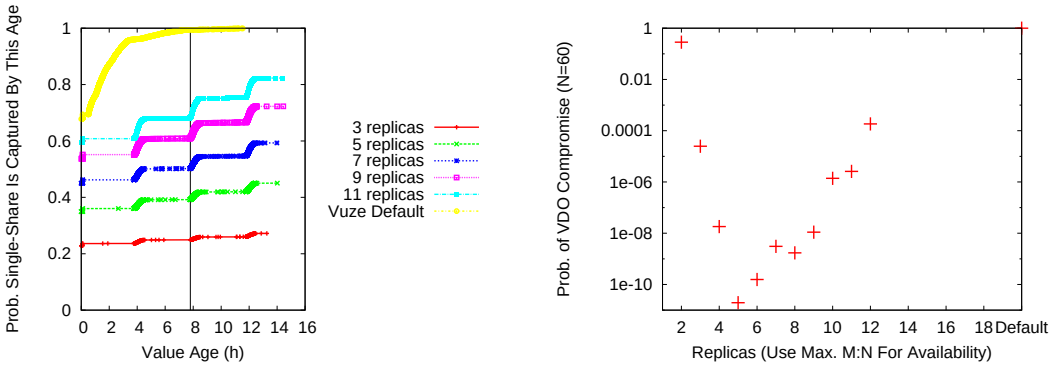
Availability Under Conditional Replication. Figure 3.5 illustrates the effect of secret sharing on VDO availability for conditional replication with different maximum replica factors. All curves shown use a four-hour replication interval, 60 key shares, and a threshold ratio of 85% (i.e., 51 shares are required to reconstruct the key). The bottom curve shows VDO availability with a replication factor of three. Here we see a rapid falloff of availability over time, with a spike up every four hours when a replication event occurs. The curve for four replicas is a little better but shows similar characteristics; in both cases the replication factor is too low to make up for natural share loss and results in poor VDO availability.

However, the graph also shows that conditional replication using a replication factor of five or more achieves over 99% VDO availability (these curves all run along the top of the graph). These results confirm our assumption that Vuze's default replication strategy – 20-way replication every 30 minutes – is significantly over-engineered and over-provisioned for real churn conditions. For the rest of our measurements, we therefore use conditional replication with a replication factor of five and a four-hour replication interval.

Impact of Conditional Replication on Data Crawling Attacks. Relative to the original Vuze replication scheme, conditional replication should greatly reduce the data-harvesting ability of an attacker who has infiltrated the DHT. To evaluate this impact, we joined the DHT using a large number of Sybil attack nodes, with each Sybil hopping to a new DHT ID every replication interval. Figure 3.6(a) shows the probability of capturing individual DHT values (not VDOs) with $25,000$ simultaneous attack nodes as a function of the ages of the values (i.e., the time since the values were stored in the DHT). Lines are shown for different replication factors, all using a 4-hour minimal-replication interval.

The figure quantifies the probability of the two types of captures: those due to direct `puts` (the points at $age = 0$) and those due to replication (the points around $age = 4$, 8, and 12 hours). Conditional replication significantly reduces the attacker's share capture: the top curve shows the original Vuze 20-way replication policy, which results in nearly 100% capture at 8 hours, compared to 40% capture at eight hours for conditional replication with a replication factor of five.

Thanks to our use of secret sharing, compromising a VDO is much more difficult than compromising a single DHT value. Figure 3.6(b) illustrates the dramatic thresholding effect of secret



(a) Share Compromise vs. Time.

(b) VDO Compromise vs. Replicas.

Figure 3.6: **The Data-Crawling Attack Under Conditional Replication.** (a) Single-share compromise probability by a 25K-node attacker, 4-h minimum replication interval, and replication factors 3–11. Points labeled "Vuze Default" correspond to 20-way, every-30-minute replication. (b) VDO compromise probability with the number of replicas, 60 shares, using the maximum $M : N$ ratio to ensure availability for 8 hours with high probability (99.9%).

sharing on VDO security. For 2 to 12 replicas, the figure shows the probability of the attacker capturing a given VDO (with $N = 60$ shares, threshold of 0.85) using $25,000$ simultaneous attack nodes hopping every 4 hours. For each of the replication factors, we use the maximum threshold ratio allowable to ensure VDO availability for the default 8h timeout (99.9% availability throughout its lifetime).

The graph's V shape illustrates an interesting tradeoff and the presence of an optimum replication factor. For very small replication factors (e.g., two replicas), churn greatly affects single-share availability, requiring us to use extremely low secret sharing threshold ratios, which results in poor security. The graph clearly shows that five replicas is the optimal replication factor for security, providing a probability of VDO compromise of approximately 10^{-10}. As the replication factor increases above five, the small increase in allowable threshold ratio does not offset the increase in per-share capture that we saw in Figure 3.6(a), resulting again in weaker security.

Thus, conditional replication severely limits the number of opportunities that an attacker has to capture a share during its lifetime. This implies that the attacker must hop less frequently and

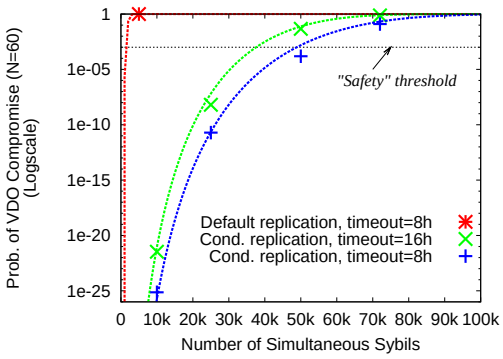


Figure 3.7: **Probability of VDO Compromise vs. Attack Size Under Conditional Replication.** Conditional replication significantly increases the resistance to attack compared with original Vuze's default replication policy.

instead maintain a much larger *continuous* DHT presence. Figure 3.7 shows the probability of VDO compromise for an increasingly powerful attacker, measured by the number of simultaneous nodes it maintains in the DHT at all times. The graph compares conditional replication under our recommended parameters (5 replicas, 4-hour replication interval) with the default Vuze post-v4.4 Vuze 20-way, every 30-minute replication policy, a sterner replication policy than the one Clearview attacked (recall that Vuze, at the time of the Clearview attack, also implemented aggressive push-on-join replication). For conditional replication, we show results for two different timeouts: 8h (the default in our SDDS prototype) and 16h. Points on the graph indicate results directly obtained from our experiments with various simultaneous attacker nodes integrated into the million-node DHT with conditional replication enabled. Lines in the figure indicate the predictions of a simple probabilistic model seeded with results from the 10,000-simultaneous-node experiment. For interpretation, the "safety threshold" line marks the 10^{-3} capture probability, which we believe is a conservative estimate for the success rate required by an adversary to mount a continuous data-crawling attack.

With conditional replication and VDO timeouts of 8 and 16 hours, attackers would require, to maintain at least $50,000$ and $37,000$ nodes, respectively, continuously integrated in the 1M-node Vuze DHT (24x7x365) in order to have even a slim chance (10^{-3}) of capturing any given VDO.

Comparing to the pre-v4.4 Vuze evaluated by the Clearview paper [226], conditional replication makes a radical improvement. That paper indicates that fewer than 500 simultaneously connected nodes were necessary to obtain 25% of the VDOs by hopping once every 150s. Hence, our modifications to the production Vuze DHT represent at least a two-order-of-magnitude increase in the number of Sybils required by the attack.

Conditional replication has a significant impact on the number of nodes an attacker would need to infiltrate the DHT to learn even a small number of VDOs. To complement this impact, we now describe a practical mechanism that raises the bar against Sybil attacks, which could otherwise enable such large-scale attacks from a limited number of physical machines.

Defense 2: Anti-Sybil Defense with Limited Node Identifiers

The pre-v4.4 Vuze was extremely susceptible to Sybil attacks, in which one physical node could inject a large number of malicious DHT nodes into the DHT. In response, we designed a simple-to-deploy Sybil defense to limit an attacker's ability to infiltrate the DHT at very large scale. Many anti-Sybil techniques have been proposed in the literature, including [28, 53, 55, 62, 114, 231] (see also Section 3.8). In choosing a Sybil defense for Vuze, we were guided by practical concerns. We contemplated many solutions, including the use of node certifications obtained from Vuze and BGP-prefix-based DHT IDs, but Vuze, Inc. deemed these to be too heavyweight or inconvenient.

Hence, to limit the number of nodes that an attacker can emulate, we devised a new lightweight, yet effective, function for computing DHT IDs. The revised node ID calculation acts as an *admission control mechanism*, capping the number of nodes that an attacker with limited IP diversification can create. Previous systems have relied on IP diversity to defend against routing attacks in unstructured P2P systems; e.g., nodes in Tarzan [73] select relays from diverse jurisdictional and operational boundaries to create anonymizing paths through a mix network. However, we believe that to date no one has incorporated IP diversity requirements in *DHT ID calculations* to limit Sybil attacks.

In Vuze and many other DHTs, a joining node's ID is generated by computing the SHA1 hash of the node's publicly visible address (IP) and port number (P), i.e., $H(IP, P) = \texttt{SHA1}(\texttt{IP} \mid\mid \texttt{P})$, where $||$ is the bitstring concatenation. We created a new DHT ID assignment function that: (1) limits the number of nodes in the DHT from a single IP address, and (2) also limits the number of nodes that can participate in the DHT *from a given IP prefix*. Let $IP_1, \ldots, IP_4$ be the first through

fourth bytes of an IP, with IP_1 being the most significant (e.g., 128 in the case of IP 128.18.15.3). The following function generates IDs for nodes joining the DHT and determines their locations in the DHT:

$$H(IP, P) = \texttt{SHA1}(\texttt{IP}_1 \,||\, (\texttt{IP}_2 \,||\, (\texttt{IP}_3 \,||\, (\texttt{IP}_4 \,||\, (\texttt{P} \,\%\, \texttt{k}_4)) \,\%\, \texttt{k}_3) \,\%\, \texttt{k}_2) \,\%\texttt{k}_1)$$

The function $H(\cdot)$ limits an IP to at most k_4 identities and caps the number of identities that can be generated by /8, /16, and /24 prefixes to k_1, k_2, and k_3, respectively. As a concrete example, the University of Washington (UW) uniquely controls a 16-bit IP prefix (128.208) and can generate IP addresses $128.208.0.0$ through $128.208.255.255$.[3] UW can therefore create up to 64K unique IP addresses that could be deployed in a Vuze Sybil attack if it were malicious. However, by setting k_2 to 2K, for example, we reduce by a factor of 32 the number of DHT positions that UW (and all other /16 owners) can occupy in the DHT – from 64K positions to 2K positions. If a successful Sybil attack requires placement at, say, 64K positions, then UW would need to co-opt at least another thirty-one /16 networks to collaborate in the attack. Moreover, assuming that we also set $k_4 = 4$ nodes, a hacker who controls one or a few IP addresses in each /16 would not be able to mount the attack. Rather, the attacker must either control the routers of all thirty-two /16 networks or control 500 different IP addresses in each /16 network. This would be a formidable task for UW. [4]

Our technique has two side effects. First, it prevents some nodes from operating as *full participants* in the DHT, e.g., only k_4 nodes could fully participate from behind a NAT. A node that cannot participate will not store values on behalf of other DHT nodes and will not be reached by other nodes' lookups. However, it can still operate as a DHT *client*, i.e., the node can still perform its own stores and lookups. Such nodes *can* use BitTorrent swarms and store or read Vanish shares. Some existing DHTs, such as OpenDHT [164], already include a distinction between client and fully participating nodes by design, so this consequence is neutral. Second, our IP-based ID limitation might reduce the DHT's size. In particular, if many nodes were prevented from participating in the DHT's maintenance traffic (e.g., storage, replication, etc.), then the DHT would be smaller than it could be, which could increase the workload on participating nodes.

To examine the benefit-cost tradeoff of our scheme, we collected data on DHT membership from

[3] IP spoofing has no value for a data harvesting attack, because the attacker would not receive return packets with data values. Route hijacking is possible, though hijacking a large number of routes over a long time (months or years) poses a great challenge.

[4] We believe that we can be as effective in filtering in classless IP systems, such as IPv6; investigation of this aspect is left for future work.

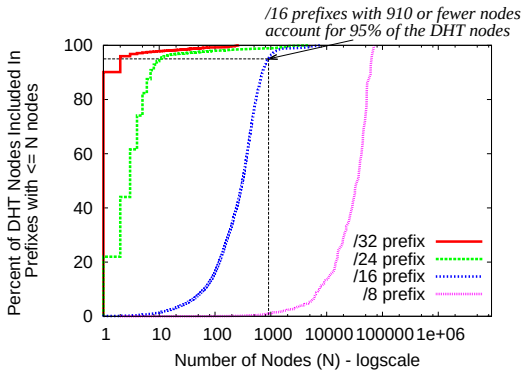


Figure 3.8: **Evaluation of Anti-Sybil Defense.** Shows the percentage of nodes that would be included by prefixes with a maximum node limit shown on the X axis. Harsh limits can be applied through ID filtering without impact on DHT size.

a Vuze, Inc. version server to which all nodes that run the default Vuze application report. In a 24h period, we saw $1,842,628$ nodes (IP-port pairs) that originated from $1,724,363$ distinct IPs. We then quantified the change in DHT size for different prefix-based ID limiting parameters.

For the standard IP prefix lengths (/8, /16, and /24), Figure 3.8 shows the percentage of DHT nodes (y axis) that would be included by prefixes with a given maximum node limit (x axis). (For the /32-prefix line, we show the percent of DHT nodes that would be included by IPs with at most a certain number of ports/IDs.) For all prefix lengths except /8, the presence in the DHT of any specific prefix is surprisingly small. For example, $/16$ prefixes with 910 or fewer nodes account for 95% of the nodes in our trace. Similarly, $/24$ prefixes with 10 or fewer nodes and IPs with 2 or fewer ports each account for 95% of the DHT nodes.

These results are significant. They suggest that harsh limits can be imposed on the number of nodes that come from each prefix. We choose our limits conservatively based on these results, as follows: $k_4 = 5$ (at most 5 IDs from any IP); $k_3 = 50$ (at most 50 IDs from any /24 prefix); $k_2 = 2,500$ (at most $2,500$ IDs from any /16 prefix). Limits on /8s are also possible and result in only a slightly greater DHT size reduction. However, given that the threat from the few companies owning /8s today is remote, we decided not to impose such limits at this time. Overall, choosing these values reduces the number of distinct IP-Port pairs in our trace from $1,842,628$ to $1,575,786$

Defense	Effect
Disable on-join-replication	Limits data dissemination
Conditional replication	Limits data dissemination
Reduce replication factor (3x impact)	Limits data dissemination
Increase min. repl. interval (80x impact)	Limits data dissemination
Prefix-based ID calculation	Limits Sybil
NAT traversal (2x on direct puts)	Increases DHT size
Port to larger DHTs (up to 6x impact)	Increases DHT size

Table 3.2: **Vuze Data-Crawling Defenses and Effects.**

distinct IDs, a reduction in DHT size of only 15% (which we believe is an overestimate due to the 24-hour-long trace). Thus, harsh per-prefix restrictions will have only a marginal impact on DHT performance but will significantly toughen the DHT's ability to withstand Sybil attacks.

Thus far, we have presented the impact of two deployed defenses – conditional replication and DHT admission control – in isolation on a data-crawling attack. We next combine these results to assess Vanish's security on top of the current, patched Vuze DHT.

Security of post-v4.4 Vuze with Data Crawling Attacks

Taken together, our new conditional replication and DHT admission control mechanism renders a Sybil-based data-crawling attack extremely challenging. Our results show that a successful Clearview-like attack would require access to 10,000 IPs scattered in 1,000 distinct /24 IP prefixes and 20 distinct /16 IP prefixes. Few attackers have access to such resources without contracting with major international companies, ISPs, or a botnet. Moreover, on today's larger DHTs such as uTorrent, with 5M nodes, an attacker must maintain about $250,000$ simultaneous DHT nodes (IDs) and $50,000$ IPs continuously, $24x7x365$, in the hope that something they learn would be of value in the future.

In addition to the two discussed security measures, we have made other design modifications to increase Vuze's security against this and other attacks. Table 3.2 summarizes the defenses we have explored. We have deployed all of the data dissemination defenses on Vuze. We deployed prefix ID calculation but have not yet enabled it, as this would have prevented us from conducting the

large-scale attack measurements needed for this section. NAT traversal is implemented in the Vuze P2P system but not yet in the DHT.

Finally, the size of the attack required for success is a linear function of the size of the DHT. The results shown here are for the 1M-node size of the Vuze DHT as of December 2009. The much larger sizes of other deployed DHTs – such as uTorrent, a DHT with over 5M simulatenous nodes – would improve Vanish's security. Specifically, an attacker would intuitively need a constant presence approximately 5X larger – or over 250,000 nodes – assuming that uTorrent implements a security-sensitive design similar to the one we have developed for Vuze. As DHT use continues to grow, then, the cost of attack should increase proportionally.

Data crawling attacks, which we investigated in this section, are one type of Vanish-specific attack. We next present defenses against another Vanish-specific attack the `lookup`-sniffing attack.

3.6.3 The Lookup Sniffing Attack

In addition to seeing `store` requests, a DHT-integrated adversary also sees `lookup` requests. Although Vuze only issues lookups prior to `storing` and `getting` data objects, the lookups pass through multiple nodes and hence provide additional exposure for VDO key shares. In a lookup sniffing attack, whenever an attacker node receives a lookup for an index, it actively fetches the value stored at that index, if any. While more difficult to handle than the passive `store` attack, the `lookup` attack could increase the adversary's effectiveness.

Fortunately, a simple, *node-local* change to the Vuze DHT thwarts this attack. Whenever a Vanish node wants to store to or retrieve a value from an index I, the node looks up an *obfuscated* index I', where I' is related to but different from I. The client then issues a `store`/`get` for the original index I to the nodes returned in response to the lookup for I'. In this way, the retrieving node greatly reduces the number of other nodes (and potential attackers) who see the real index.

One requirement governs our simple choice of an obfuscation function: the same set of replicas must be responsible for both indexes I and I'. Given that Vuze has 1M nodes and that IDs are uniformly distributed (they are obtained via hashing), all mappings stored at a certain node should share approximately the higher-order $\log_2(10^6) \approx 20$ bits with the IDs of the node. Thus, looking up only the first 20b of the 160b of a Vuze index is enough to ensure that the nodes resulted from the

lookup are indeed those in charge of the index. The rest of the index bits are useless in lookups and can be randomized, and are rehabilitated only upon sending the final `get`/`store` to the relevant node(s). We conservatively choose to randomize the last $80b$ from every index looked up while retrieving or storing mappings.

Lacking full index information, the attacker would have to try retrieving all of the possible indexes starting with the obfuscated index (2^{80} indexes), which is impossible in a timely manner. This Vuze change was trivial (10 lines of modified code) and completely local to Vanish nodes.

3.6.4 Generic DHT Attacks and Defenses

In the previous sections we offered an in-depth analysis of two data confidentiality attacks in DHTs (data crawling and lookup sniffing), which are specific in the context of our system. However, the robustness of communal DHTs to more general attacks has been studied profusely in the past and such analyses, proposed defenses, and limitations are relevant to Vanish, as well. A comprehensive survey [207] describes most known attacks including: Sybil attacks, where malicious nodes amplify their presence; Eclipse attacks, where the attacker nodes blackhole honest nodes by corrupting their routing tables; and routing and storage attacks, where malicious nodes do not follow the routing and storage protocols correctly.

Numerous defenses have been proposed to defend against all of these attacks. Sybil defenses include centralized certification, trusted measurement infrastructure, social networks, and computational puzzles [15, 28, 35, 53, 55, 170, 217, 231]. Eclipse attacks can be defended by disabling proximity optimizations, using both optimized and unoptimized routing tables, or doing anonymous auditing [14, 35, 45, 93, 185, 186, 188, 198]. Finally, common defenses to routing and storage attacks include routing through a diverse set of nodes, identifying multiple independent paths, using iterative routing as opposed to recursive routing, and storing redundant replicas at multiple locations [35, 53, 92, 93, 218].

These long-known attacks, along with some of the corresponding defenses, apply to DHT-based self-destructing data systems and should be incorporated in Vuze. For example, Vuze currently uses 20-way path redundancy, which can protect against some forms of routing attacks [92, 93], and Vuze lookups can be further strengthened by using routes that maximize AS diversity [73].

Similarly, stronger anti-Sybil defenses might be required to protect against attackers with immense IP diversity, such as botnets or ISPs.

3.6.5 Summary of Deployed Defenses

This section described the design, implementation, and evaluation of deployed DHT defenses against Vanish-specific attacks. Our techniques – conditional replication, anti-Sybil DHT admission control, and obfuscated lookups – raise the bar against attack by three orders of magnitude compared to the original Vuze, making data-crawling attacks impractical for all but the most powerful attackers.

We recognize that defenses against all of the other known attacks must also be deployed on today's DHTs before our DHT-based Vanish can be considered secure. However, from a research perspective, we have chosen to focus Vanish-specific attack types – the recently proposed data-crawling attack [226] and the `lookup` sniffing attack – and advance the state of the art in understanding and defending against this attack from a practical, measurement-driven perspective. We hope that our deployment of practical techniques to protect against these attacks on the commercial Vuze DHT will inspire researchers in the future to deploy defenses for other attacks, thereby creating a more secure large-scale, distributed, P2P platform for building distributed-trust applications, such as Vanish. Until the vision of a secure DHT is realized, Vanish can draw its security by building upon other large-scale, distributed, decentralized deployed systems. We next review such alternative solutions.

3.7 Architectural Extensions for Security

DHTs are one type of distributed, decentralized, global-scale infrastructure, but there are many others. This section shows how to use the World-Wide-Web to create data that self-destructs (Section 3.7.1) and how various distributed storage infrastructures can be combined for defense in depth (Section 3.7.2).

3.7.1 A World-Wide-Web-based Self-Destructing Data System

The Web is effectively the world's largest peer-to-peer system, composed of millions of independently administered and autonomous interacting Web sites. This section explores how to leverage existing Web infrastructure to build self-destructing data. For example, suppose that many of the

world's Website owners – understanding the need for privacy and supportive of our pro-privacy efforts – would be willing to expend minimal effort to support self-destructing data. What would be required to build a distributed-trust key store? Would we need a full-blown DHT-like structure among participating Websites or an encryption-heavy Ephemerizer-like system, or can we build something simple, easy to manage, non-intrusive, and safe? Finally, would the resulting Web-based Vanish face the same challenges as our DHT-based system, would new challenges arise, or does the Web environment include inherent properties that simplify our task?

This section presents *Tide*, a system whose goal is to organize Web sites from around the world into a distributed-trust key store for Vanish. We first describe the basic design for Tide and then evaluate our proof-of-concept Tide-based Vanish prototype.

System Design and Prototype

Tide's goal is to leverage thousands of Websites across the planet, easing deployment through the use of widely used open software platforms, such as the Apache Web server. The Tide concept is simple: Tide-modified Websites accept small pieces of data (key shares) for storage until a specified timeout, at which point they erase them. Instead of sprinkling key shares across random nodes in a DHT, Vanish now scatters them across random Tide-enabled Web sites in the Internet. Each key share in Tide is protected by association with a random 256-bit index that acts as a capability; to retrieve a share via the exposed interface, a client must know its gigantic, unguessable index.

The Tide Apache Module. We designed the Tide key store to have three important properties:

1. *Simple and easily deployable.* To maximize deployability, we implemented Tide as a small, dynamically loadable Apache module that exports a RESTful put/get interface. The module maintains an in-memory hashtable of $(index, value)$ pairs, each with an associated timeout. To minimize the risk of improper share cleanup at the timeout, we pin the hashtable into main memory. To Tide-enable a Website, the administrator simply downloads our Tide module and installs it in one of his front-end servers. Once installed, server will handle incoming `/tide` requests. The Apache module is small (826 lines of C code) and is designed to be easily understood and inspected for vulnerabilities by concerned administrators.

2. *Lightweight and safe.* Tide relies on volunteer opt-in, therefore, we must ensure that our module remains unobtrusive to the server's general functioning. Section 3.7.1 demonstrates that our system is extremely lightweight under normal conditions. To safeguard against DoS attacks, we install harsh limits on the size of each stored index and value, the maximum memory consumption by the module, and the maximum timeout. Standard IP-based rate-limiting can also be used to limit the amount of traffic serviced by our module and the number of shares being stored by a particular client. The use of secret sharing makes Tide resilient to server failure or unavailability due to excessive load; a server can opt out without impacting the overall system.

3. *Supportive of a variety of deployment strategies.* Due to its simplicity and use of a common infrastructure – the Apache Web server – Tide can be deployed in multiple ways. For example, Tide could run on a global collection of public Websites. Potential early adopters might include privacy advocates (e.g., `pgp.com`), open-information supporters (e.g., `kernel.org`, `sourceforge.net`), freedom-of-speech supporters (e.g., `rsf.org`, `eff.org`), academic institutions (e.g., `cs.washington.edu`), and worldwide testbed participants (e.g., PlanetLab members). Tide could also run on private servers known only to a small community. For example, a group of tech-savvy people could host Tide-enabled servers to support self-destructing emails for friends, or a company could run Tide for its employees.

There are several important differences between the Tide environment, based on Web servers, and the P2P/DHT environment, based on personal machines. First, a server-based environment has lower churn, which eliminates the need for cross-Website share replication. With no replication or other inter-Website communication, harvesting of key shares through such mechanisms is impossible. A Tide server can only receive shares if a client directly puts those shares to the server. Second, the Web provides the opportunity to embed the notion of node trust into the system. DHT nodes are anonymous and indistinguishable in terms of trustworthiness, hence they are all treated the same. This makes DHTs particularly vulnerable to Sybil infiltration, which caused us to develop the admissions control mechanism we described for Vuze. Infiltration attacks on a Web-based Vanish are also possible; however, in the Web environment, there have been numerous efforts to gauge the trustworthiness of Websites. These include the development and adoption of SSL/TLS with server

certificates [163], DNSSEC [12], heuristics for identifying spamming and phishing domains, and Perspectives-like [220] P2P approaches for establishing the trustworthiness of public keys in the absence of PKIs.

Finding Tide Servers Securely. The mechanism by which Vanish obtains a list of all Tide-enabled servers in the world affects security and is therefore crucial. For example, if an attacker could bias a client's selection of Tide servers, perhaps by infiltrating the list of possible servers with malicious Web sites or hiding the existence of honest sites, that would seriously degrade security. We have identified three potential alternatives for directory services:

1. *Web Search.* The simplest way to find Tide servers is to use a general-purpose search engine (such as Google, Yahoo, and Bing) to search for Tide-specific keywords. Naturally, these solutions are vulnerable to search engine compromises; however, using multiple services (e.g., Google and Yahoo and Bing) with secret sharing and some server filtering would mitigate these concerns. Our current prototype implements this option.
2. *Web-of-Trust-Based Directory Service.* A PGP-like web-of-trust scheme could serve as a distributed directory service [234]. Applications would supply the server lists from trusted communicants to Vanish. For example, if Dan trusts Doris, he might incorporate her list of trusted servers into his own list of servers, possibly with scaled-down trust levels. With such a solution, users would not have to trust any centralized directory service and would avoid the directory-hijacking attack. We leave the full investigation of this solution for future work.
3. *Communicant-Based Server Discovery.* The communicants themselves – or their employers – might host Tide servers for the timely destruction of their communications. For example, when Dan sends a self-destructing email from his account (`dan@biz1.com`) to Doris (`doris@biz2.com`), the Vanish email application on Dan's machine could automatically search for Tide servers at `http://www.biz1.com/tide` and `http://www.biz2.com/tide` and, if present, use them, requiring both in order to reconstruct the key. Such a protocol may be applicable between two companies that trust each other but do not want to rely completely on each others' data destruction mechanisms.

Server filtering can be used to further withstand infiltration by malicious entities. The space of potential filtering policies is large on the Web. To prevent infiltration of multiple URLs from a single

DNS domain – the equivalent of Sybil attacks from DHTs – our prototype eliminates duplicate URLs from any two-level domain (e.g., URLs `https://cs.washington.edu/tide` and `https://washington.edu/tide` are duplicates and one of them is dropped); this approach could be internationalized. Another potential admission control scheme might include only (or mostly) Tide servers from trusted domains, such as `.edu` domains or Fortune 500 URLs. Implementations could also ensure that domains in the local list span sufficient geographical regions or avoid certain geographical regions altogether, or that they have been registered for a long enough period of time (i.e., presumably not spam domains).

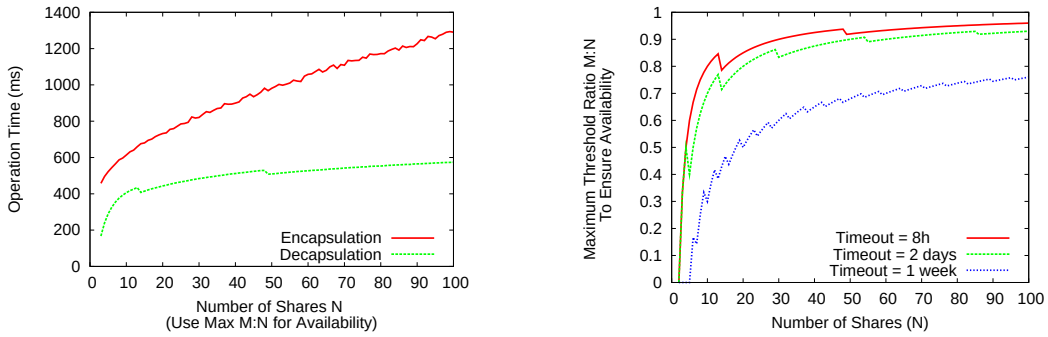
Tide-Based Vanish Prototype. We built a proof-of-concept prototype Vanish system using Tide as its backend. We have implemented a subset of the features supported by the architecture described above. In particular, our prototype integrates only Tide servers returned by search on Google and performs only rudimentary duplicate-elimination admission control. Prototype Tide Websites "register" with Google by including their Tide URLs on an indexable page or in their sitemaps. For example, a Tide server that our Website hosts is referenced from our Download page and has been indexed by Google. To find Tide-enabled sites, clients query Google's AJAX search API for URLs containing `tide/keystore` by issuing the query "*allinurl: tide keystore*." Although we find that this keyword combination is currently unpopular in URLs (our Tide server is the only result returned), our module tests URLs for a valid Tide module before including them on the local list.

We now evaluate our Tide-based Vanish prototype. Recall from Section 3.3 that threshold secret sharing and its parameters play a crucial role in the tradeoff between security and availability and hence will be a central focus of our evaluation.

Evaluation of Tide-based Vanish Prototype

We now evaluate the performance, availability, security, and lightweightness of our Tide-based Vanish prototype. To create in a realistic global setting, we deployed Tide-enabled Apache Web servers on 462 PlanetLab nodes. These nodes are servers scattered all around the world and should approximate a realistic global Tide deployment.

Performance. We evaluate Tide performance by measuring response times for encapsulation and decapsulation on our PlanetLab deployment. Figure 3.9(a) shows results for various numbers of



(a) Performance.

(b) Availability.

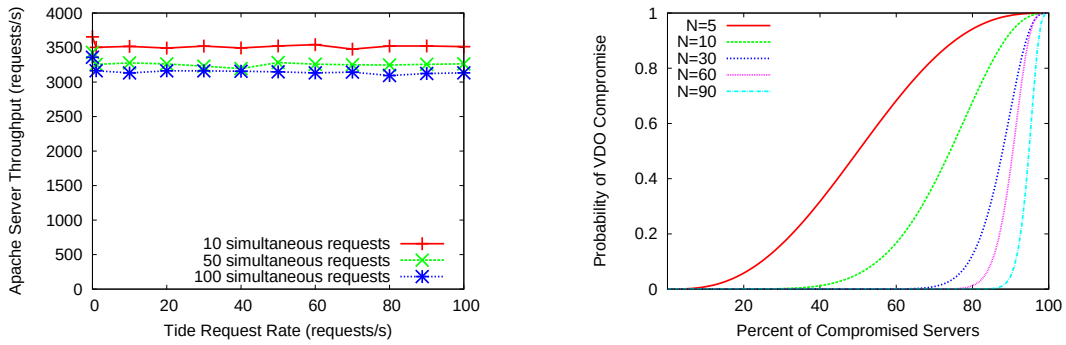
Figure 3.9: **Tide's Performance and Availability.** (a) Encapsulation and decapsulation times. (b) Maximum threshold ratio to ensure VDO availability with high probability ($> 99.999\%$).

shares (N), obtained from measurements of $10,000$ operations of each type. For each value of N, we again use the maximum threshold that ensures VDO availability for 8 hours. As the number of shares increases, VDO encapsulation times increase close to linearly, while VDO decapsulation times quickly level off. This is because encapsulation must await responses from all N servers, while decapsulation waits only for the fastest M shares to arrive. Overall, encapsulation and decapsulation response times remain under $1.3s$ and $600ms$, respectively. Building on our availability results (see below), we recommend using $N = 30$ shares and a threshold ratio of 90%, which leads to $484ms$ decapsulation times and $820ms$ encapsulation times.

VDO Availability. To estimate VDO availability, we leverage the uptime information obtained from our 462 PlanetLab servers and simulate VDOs under various numbers of shares and threshold ratios. We compute the probability that any given VDO would remain available until its timeout given crashes and reboots.

Server churn is typically small: e.g., the median node uptime in our PlanetLab trace is 59 days. Therefore, it is not difficult to achieve good availability guarantees for VDOs with timeouts of up to a week. In particular, for most values of N ($N \geq 5$), we can identify a range of secret sharing threshold ratios that ensure VDO availability with high probability. As with Vuze, our goal is to find the ratio that provides the optimal security-availability tradeoff.

Figure 3.9(b) graphs the maximum threshold ratio capable of providing VDO availability with



(a) Tide Throughput Overhead.

(b) Tide's Security.

Figure 3.10: **Tide's Security and Overhead.** (a) Probability of VDO capture as a function of the percentage of compromised servers for various values of N. E.g., for $N = 30$ an attacker must control at least 70% of the Tide servers in order to have a 10^{-3} chance of capturing the user's VDO. (b) Performance impact of Tide workload on web server throughput. Results are averaged over 10 trials.

a probability 99.9% for various numbers of shares and timeouts of 8 hours, 2 days, and 1 week. For an 8-hour timeout (our prototype's default), scattering shares across 30 servers permits a threshold ratio of 90%; i.e., with very high confidence, at least 27 (90%) of those 30 servers will remain up during the VDO's 8-hour lifetime. For larger timeouts, such as one week, required threshold ratios are much lower. In this case, again with 30 shares, the maximum ratio Tide can support is around 60% (i.e., we require over 18 of the 30 shares to reconstruct the VDO or risk losing the VDO prematurely). The reason is intuitive: during one week there is a higher chance that servers will reboot or crash compared to an 8-hour period, so we must adjust the secret sharing threshold ratio accordingly.

VDO Security. We evaluate security in the context of an adversary who controls a fraction f of the Tide servers. The adversary can achieve control either by infiltrating into or compromising some of the Web servers in a non-targeted precomputation attack, or by compromising the specific Web servers that used to store key shares for a specific VDO in a post-timeout targeted attack.

Given that VDO shares are placed at random on the servers, capturing VDOs is probabilistic. To assess the probability that an attacker captures a VDO, we use a simple combinatorial model that takes f and the VDO parameters (N, threshold ratio $M : N$) as inputs. Figure 3.10(b) shows the probability of VDO compromise as the fraction of compromised servers increases for various numbers of shares. For each value of N, we use the maximum allowable threshold ratio $M : N$ to ensure availability for the 8-hour default Cascade timeout. Using $N = 30$ again as an example, we see that an attacker who has compromised 80% of those servers (24 servers) will capture only 15% of the VDOs given the 90% threshold ratio (Figure 3.10(b), third curve from the left). Hence, in the context of a real-world deployment like our Planetlab Tide experimental setup, we conclude that using $N = 30$ shares and a threshold ratio of 90% provides both good availability for 8-hour timeouts and good security.

Tide Module Lightweightness. Because Tide relies on volunteer opt-in, we needed to verify that it is non-intrusive to the hosting server's performance. To measure the impact of handling a Tide workload on the throughput of Apache, we drove two workloads simultaneously against Apache 2.2.14 running on a modest server with a 1GH dual-core CPU and 1.5GB of memory. First was a *Tide workload* that we drove at various rates using `httperf` [133]. Second was a *Web workload* that we drove using the Apache Benchmark tool (`ab`) to measure the server's throughput when downloading a small 13-B HTML index page.

Figure 3.10(a) shows the impact of increasing the Tide workload on Apache's Web-workload throughput. We include results for increased numbers of concurrent Web-workload requests, which correspond to higher Web workloads. The points on the y axis ($x = 0$) correspond to the server's base throughput *without* Tide. The points with $x = 100$ show the server's Web-workload throughput when it is being pounded by a 100-request/second Tide workload. The almost horizontal lines demonstrate that even under heavy Tide workloads, Apache's Web-workload throughput remains within 93–96% of its base throughput (the largest penalty occurs for high Web workloads of 100 concurrent requests).

Similar results can be shown for per-Web-request latencies, where where the penalty of hosting a heavily-loaded Tide server (100 Tide requests/second) results in a 4–7% average latency increase and a 6–15% 90^{th}-percentile-latency increase. In addition, lightweight conclusions can be reached

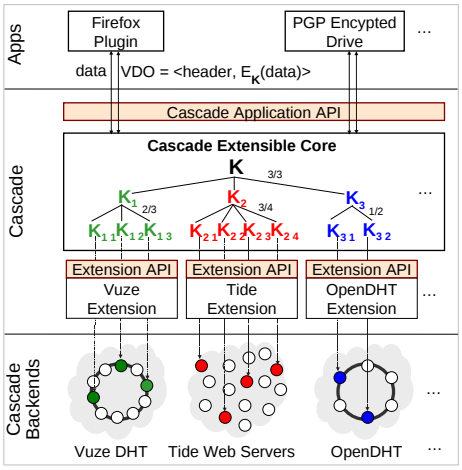


Figure 3.11: **Multi-Key-Store Self-Destructing-Data Architecture.** Distributes encryption material over i different distributed-trust key stores using hierarchical secret sharing ($i = 3$ here).

for memory overhead. Maintaining one Tide $(index, share)$ pair requires only 72 bytes; hence, with a modest 256MB of memory on each Tide server in our 462-server PlanetLab deployment, Tide could support the creation of 3-day-timeout VDOs at a rate of 200 new VDOs/second – almost a third of Facebook's status-update rate.

3.7.2 Combining Distributed Key Stores

This section briefly describes the potential for creating a single self-destructing data system that composes Tide with Vuze and/or other distributed-trust key stores. The composition of multiple key stores is relatively straightforward and has an unusual advantage: while in general a system is only as secure as its weakest link, a composed self-destructing data system is as strong as the *union* of its components, thereby moving distributed trust to a new level. As a result, a successful attack must subvert *all* of the key stores involved, rather than just one. We have extended our Vanish prototype support such combinations.

Figure 3.11 shows the architecture of the multi-key-store Vanish. The composed system uses a hierarchical secret sharing approach. The figure shows a self-destructing data system using three underlying key stores: a Vuze DHT key store, a Tide Web-based key store, and an OpenDHT key store (OpenDHT is a closed-membership DHT with different properties than Vuze). To encapsulate

data in this configuration, the random encryption key K would first be broken into three shares – one for each key store – thus requiring all three keys to reconstruct K. Each of the three per-key-store shares would then be further broken into shares. For example, we might: (1) break the Vuze share into 60 sub-shares to be stored on Vuze, requiring 85% of them for decryption; (2) break the Tide share into 30 sub-shares, requiring 90% of them; and (3) break the OpenDHT share into however many shares are appropriate for that DHT. The figure shows key-store-specific extension components, which provide a standard key-store interface and handle the protocol details and parameters for each key store. New key stores can be easily plugged into the self-destructing data system by adding new extensions.

Two reasons exist for such multi-store trust distribution. The first is an argument for N-versioning and defense-in-depth. Different key stores exhibit different strengths and weaknesses. For example, Tide might be attractive to those concerned about botnets, whereas a DHT-based system might be more vulnerable to a botnet-based attack. Conversely, an entirely P2P system might be attractive to conspiracy theorists strongly resistant to certain Tide admission control policies (such as open admission to Tide servers from `.edu` or `.gov` domains). Even different admission control policies for Tide provide different security properties. By combining different key stores – especially very distinct ones, such as Tide and Vuze – we significantly raise the bar against both perceived untrustworthiness and real attacks. Attackers well positioned to attack one key store may not be well positioned another. For example, an attacker willing to use illegal botnets might not be well positioned to collaborate with large corporations, educational institutions, or governments in hosting nefarious Tide servers, and vice versa. Second, key store composition allows the incremental deployment of new key stores. As an example, our currently deployed, significantly strengthened Vuze key store can foster the gradual deployment of our new Tide key store. Without composability, incremental deployability could be a bottleneck for Tide.

3.7.3 Summary

The Web is composed of an enormous set of interconnected services, many of them based on Apache or other open, standardized components. This section presented a unique scheme for creating a distributed-trust key-value store for self-destructing data based on this infrastructure. To the best

of our knowledge, our Tide implementation is the first easily deployable and lightweight key-value store that leverages the massive world-wide Web. Our measurements, conducted on over 400 world-wide PlanetLab nodes, demonstrate the viability of this approach in real-world settings. If Tide were deployed widely, we believe it might be useful in other contexts independent of self-destructing data.

3.8 Related Work

We discussed numerous related works inline above. We return here to a summary of two key classes of related works: cryptographic foundations for and other explorations related to self-destructing data, and past work on DHT attacks and defenses.

Key establishment protocols often target the derivation of ephemeral session keys for interactive communications [129]. Past work has also considered leveraging third parties to assist in the destruction of stored content, including the Ephemerizer [138, 148], revocable backup systems [27], and multiple commercial self-destructing email and SMS companies like Disappearing Inc. and Tiger Text. Multiple cryptographic techniques have also been developed with goals compatible with self-destructing data, including forward-secure [17, 34], key-insulated [16, 59], intrusion-resilient [57, 58], and exposure-resilient [32, 60, 61] cryptography. These techniques differ from ours in that we incorporate a high distribution of trust directly into the core of our system design and evaluation, we target attackers with significant retroactive access capabilities, we do not wish to require the use of auxiliary tokens or secure hardware, and we target the Web rather than content like local files and backup tapes.

Steganography [152], deniable encryption [33], or a deniable file system [49] is also related to, though different from, self-destructing data. Intuitively, if one could hide, deny the contents of, or deny the existence of private historical data, one would not need to destroy. These approaches are attractive but hard to scale and automate for many applications, e.g., generating plausible cover texts for emails and photos. In addition to the problems observed with deniable file systems in [49] and [126], deniable file systems would also create additional user hassles for a trash bin application, whereas our approach could be made invisible to the user.

Further afield, but focused on the Web, others have proposed limiting the lifetime of personal Web data by making that data less sensitive over time, albeit with a significant level of trust placed

on the storing Web services [212]. Others have also considered overlaying access policies on top of existing cloud storage services like S3, e.g., [203]. Self-destructing data systems are explicitly different than DRM systems: an SDDS assumes that end-users who have access to a VDO during its lifetime are *trusted* by the VDO creator, whereas DRM systems target precisely untrusted users. DRM systems are, however, related in that they target control over the lifetime and use of data, and thus [122] proposes extending DRM techniques to user-generated content. Tide is also related to hyper-encryption, an information-theoretically secure encryption scheme which leverages a decentralized collection of dedicated machines [161]. These machines continuously serve random pages of data, where each page can be read at most twice. Tide differs from hyper-encryption in its goals (self-destructing data as opposed to information-theoretic secure communications between two parties sharing a secret key) and complexity (e.g., a recent implementation of hyper-encryption [103] requires two communicants to interactively reconcile their accessed pages, whereas Tide's use of secret sharing creates no such need).

We touch on prior work on various DHT attacks and defenses in Section 3.6. We now specifically contrast our Sybil defense with earlier proposals. Our work contributes a simple yet surprisingly effective technique for limiting identity fabrication. Prior defenses include strong identities minted by a logically centralized authority [62], computational puzzles and bandwidth contributions to make peers prove that they are not Sybils [28], and leveraging social networks [114, 231]. None of these defenses have been adopted by today's DHTs like Vuze, because there was no perceived need for existing applications and many of these defenses were deemed too complex or heavyweight. We instead propose simpler measures that cap the number of DHT IDs that an attacker with limited IP diversification can create. Our mechanism relies on IP addresses as weak identities and separates service nodes from client nodes, i.e., anyone can obtain service from a DHT (`get` or `put` values), but only a limited number of clients from a given IP or prefix can serve as DHT data-storage nodes. As we noted, while previous systems have used IP diversity to route robustly in the face of malicious routing attacks [46, 73, 115], this is the first use of DHT ID assignment as an admission control mechanism, forcing extensive IP diversity to counter DHT Sybil attacks.

Section 3.6 examined the newest class of DHT attacks: the Sybil data-crawling attacks [201, 226]. No defenses against these attacks are known, so we designed, implemented, deployed, and evaluated defenses against them in the Vuze DHT. To the best of our knowledge, until this work, no

defenses against such attacks have been deployed and measured on large-scale, live DHTs.

3.9 Summary

Data deletion has become increasingly important in our litigious and online society. This chapter introduced *Vanish*, a new concept for protecting data privacy from attackers who retroactively obtain, through legal or other means, a user's stored data and private decryption keys. A new aspect in Vanish is the combination of threshold secret sharing with distributed, global-scale decentralized infrastructures, such as P2P DHTs. We prototyped Vanish on top of the Vuze global-scale DHT. Our prototype causes sensitive information, such as emails, files, or text messages, to self-destruct without any action on the user's part and without relying on any centralized or trusted service.

Our experience revealed that leveraging existing DHTs to create self-destructing data is challenging. Specifically, Vuze's design was at times misaligned with Vanish's requirements. For example, the limited 8-h timeouts in the original Vuze DHT limited Vanish's usefulness; its excessive replication exposed Vanish to data crawling attacks; and its lack of node admission controls opened our system to Sybil. To make Vuze more suitable for our needs, we deployed a series of security and functional modifications to its codebase. Using large-scale experiments, we have shown that our modifications increase Vanish's robustness to certain attacks by three orders of magnitude.

Our large-scale deployment of security measures in a commercial, world-wide DHT – the first of its kind – was difficult and time-consuming. While the changes required to make Vuze better match Vanish's requirements were conceptually simple, deploying them took months of work with Vuze's DHT designer. Moreover, testing and experimenting is difficult and can only be done after large-scale adoption of code revisions, which slows the deployment process even further.

This deployment experience motivated the design of Comet, an extensible DHT that allows each application to customize the way its data is managed in the DHT. With Comet, Vanish can control how its key shares are replicated, when they expire, and how accesses to them are entered in a forensic log. If Comet existed at the time Vanish was designed, then making the DHT fit Vanish's requirements better would not have required painful large-scale deployments of new replication schemes. The next chapter describes Comet's design and broad applicability to Vanish and other applications.

Chapter 4

COMET: DATA MANAGEMENT CONTROL WITH EXTENSIBLE STORAGE

The previous two chapters addressed security and privacy challenges raised by *untrusted* cloud and mobile environments. This chapter turns to a second problem in cloud computing: the inflexibility and lack of control over how the data is managed in *trusted* clouds. For example, how can a client specify where her sensitive Google documents should or should not be stored, how can she obtain a forensic access log of her data on Facebook, or how can she lower the replication factor on private data stored in a DHT? Even if trusted, these clouds simply do not support such levels of customization over data management and properties.

To increase control, we argue that extensibility and customizability should be built into public clouds. Specifically, clouds such as Amazon S3, Google Docs, Facebook, and the Vuze DHT should allow their clients to customize data management properties, such as data placement, availability of forensic logs, and replication schemes. This chapter takes a first step toward the realization of the extensible-cloud vision, proposing the design, implementation, and evaluation of Comet, an extensible distributed storage system initially introduced in a 2010 paper [82]. While motivated broadly by the inflexibility in today's Web clouds, Comet focuses on a particular kind of cloud service – a distributed key/value store based on peer-to-peer DHTs. Comet's design is informed by our experience building Vanish on top of the inflexible Vuze DHT. We begin by describing this context and motivate the need for extensible DHTs.

4.1 Motivation and Overview

Weakly-connected key/value stores, or DHTs, have become storage backends for many of today's applications, ranging from Web services to peer-to-peer applications. For example, Amazon's S3 [6] provides a key/value store for external Web services. Amazon's Dynamo [54], Apache Cassandra [10], and Project Voldemort [158] provide reliable and scalable DHTs for company-internal applications (for Amazon, Facebook, and LinkedIn, respectively). On the global Internet, DHTs pro-

vided by BitTorrent-based systems, such as Vuze [214] and uTorrent [208], store metadata for millions of clients using peer-to-peer file-sharing applications. And finally, researchers have developed complete file systems on top of untrusted clients in widely distributed P2P environments [2, 51, 172].

A significant limitation of today's DHTs for generic application support, however, is that different applications have different needs. As an example, each Dynamo application inside of Amazon runs its own Dynamo instance [54], even though a single instance might be logically better and more resource efficient. As shown in the previous chapter, as part of our own work on Vanish, we needed to make application-specific parameter and policy changes to Vuze (a million-node commercial DHT) in order to harden it against attack. While these changes were conceptually simple, e.g., modifying the storage replication algorithm, deploying these changes was difficult and time-consuming. Other Vuze applications may wish to make their own application-specific changes or enhancements, but doing so is neither feasible nor supportable, and it doesn't scale. We believe that with the huge consolidation benefits of shared cloud storage services, either inside or outside of the enterprise, supporting specialization of storage services can have high payoffs in the future.

We present Comet, a next-generation, flexible, distributed storage system, which opens the world of distributed storage to a new set of more complex storage applications. In particular, Comet permits multiple applications to share a single Comet instance, while enabling each application to change the behavior of its storage elements to suit its own requirements. For example, a storage element can make decisions based on its access history, its current number of replicas, the time of day, etc. Therefore Comet can easily support different storage lifetimes, access methods, access control schemes, or replication schemes for different storage-element types, in a way that makes them easy to deploy and test. Using Comet, we can also carry out interesting measurement-based experiments from *within* the DHT.

Comet implements *active storage objects* (ASOs). An active storage object consists of a key, an associated value (an untyped blob), and optionally, a set of simple *handlers*. An ASO's handlers execute as a result of common storage events on the object (such as `get` and `put`) or from timer events that its handlers request. As a result, an ASO can modify its environment, monitor its execution, and make dynamic decisions about its state.

The concept of extensible systems has been widely explored in the past in many domains, including operating systems [20, 180, 106], networks [221, 204], messages [213], routers [110],

databases [137], and browsers. We discuss these and other related works in Section 4.7. At a high level, Comet brings extensibility to a new environment – key/value stores – which creates an interesting set of design questions. For example, what features should the system provide for applications and which can (and should) be left out? What is the proper tradeoff between power and safety? How can client nodes be confident that active storage objects will not cause damage or interference? How can we prevent the use of active storage objects to mount a DDoS attack? And overall, how can we extend the storage system without losing its principal characteristics? Our Comet design considers these and other issues.

The remainder of this chapter describes our goals, architecture, experience, and evaluation of Comet. To provide concrete insight into Comet's design and potential, we implemented a Comet prototype and used it to create and deploy a set of over a dozen Comet applications. Our prototype leverages Vuze: each Comet instance is an extended Vuze client that can execute Comet active storage objects while also serving as a full participant in the million-node Vuze DHT. Comet applications are written in Lua – a common application-extension language. We modified the Lua runtime to meet our isolation and safety requirements, providing a safe sandbox for handler execution. To test our applications we ran our Comet clients from several hundred PlanetLab nodes and measured their behavior. Overall, our experience demonstrates that a highly restrictive but active distributed storage system can provide significant power to simultaneously support applications with diverse storage needs.

4.2 Goals and Assumptions

Comet is a distributed key-value storage system. Like other such systems, a Comet storage object is a <key,value> pair. Unlike previous systems, however, Comet's design facilitates extensible, active storage objects. A Comet application performing a `put` can therefore include, along with a key and value, a small set of *handlers* for that object. The node receiving the `put` stores the handlers along with the key and value, registers the handlers for events that they specify, and executes the handlers when their respective events occur.

Comet's system goals are:

1. *Flexibility.* Comet should be easily customizable to achieve our target functions.
2. *Isolation and safety.* A client node running Comet should be protected from the execution

of handlers (e.g., an executing handler cannot corrupt the node or use unlimited resources). Handlers should not be able to mount messaging attacks on other nodes.

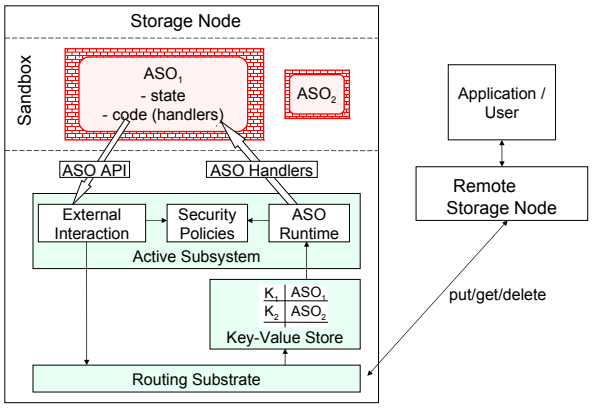
3. *Performance.* The performance of `gets/puts` on a Comet ASO with null handlers should be the same as on a non-active system, and execution of handlers should have only negligible performance impact.

Isolation and safety are particularly important to our architecture. While Comet can be used in different environments, we designed it to enable wide-scale, outside-the-firewall deployment on autonomous nodes, similar to P2P systems and DHTs. Users downloading Comet must trust it and have guarantees about its behavior. For this reason, Comet enforces four important restrictions:

1. *Limited knowledge:* an ASO is not aware of other objects or resources stored on the same node and has no direct way to learn about them.
2. *Limited access:* an object handler can manipulate only its own value and cannot modify the values of other objects on its storage node.
3. *Limited communication:* an active storage object cannot send arbitrary messages over the network.
4. *Limited resource consumption:* an ASO's resource usage is strictly bounded, e.g., the system limits the amount of computation and memory it can consume.

We are specifically *not* attempting to build a general-purpose distributed programming system, such as PlanetLab [9, 151]; such a system would be unacceptable in our target environment and inappropriate (and unnecessary) for our needs. Rather, our goal is to support relatively simple specializations or actions on simple storage objects. Even very simple specializations can provide a significantly more powerful storage system that enables new types of applications. We therefore take a lightweight and limited approach. As examples, an ASO should be able to perform the following functions:

- *Statistics gathering.* Collect statistics about its use, e.g., count the number of `gets` and `puts`.
- *Information tracking.* Log information, such as a list of IPs that performed `get` operations on its value or a recent history of the values it stored.
- *Time awareness.* Take time-based actions, e.g., to make periodic changes to its state or self-destruct after a timer has elapsed.



(a) Architecture.

getSystemTime() → UTC

getIP() → node's external IP

getID() → node's DHT ID

getKey() → ASO's key

deleteSelf(): deallocate ASO

get(key, [args]) → value, storage nodes

put(key, value[, nodes])

lookup(key) → nodes closest to a key

(b) ASO API.

Figure 4.1: **Comet Architecture and APIs.** (a) depicts the decomposition of a Comet node into two vertical components - the core Comet code, which is trusted from the node's perspective, and the ASO code which is arbitrary and, therefore, untrusted. (b) details the API exposed to ASOs.

- *Location awareness.* Make location-based decisions, e.g., choosing where to store based on nodes' network locations.
- *Access control.* Implement simple access control policies on its own.
- *Replication.* Implement different replication policies.
- *Storage system measurement.* Provide insight into the behavior of the distributed storage system as seen by clients executing within the system itself.

As we shall see, the only long-term state available to a handler is its object's value; therefore, any logs, counts, etc., must be stored as part of that value. However, an active object can choose to report only a subset of its stored value record on a `get`, or it can selectively report different values to different callers based on call parameters.

The following sections describe Comet's architecture. In particular, we discuss the tradeoffs required to provide flexibility while also achieving isolation and safety.

A fundamental assumption in Comet is that DHT nodes are trusted. While Comet applications are not trusted, a fundamental assumption in our system is that applications trust DHT nodes. That is, Comet does not strive to improve security in front of potential DHT attacks, such as byzantine nodes and Sybil attacks [62].

4.3 Comet Architecture and Implementation

This section describes Comet's active storage architecture and prototype implementation. One could imagine running Comet in various environments, e.g., an inside-the-firewall corporate deployment or a distributed environment with autonomous untrusted nodes. We focus our current architecture and prototype on the latter.

4.3.1 Architecture

Figure 4.1(a) shows the high-level architecture of our Comet distributed storage system. The Comet storage system consists of three basic components. First is the *routing substrate* (Figure 4.1(a) bottom), which implements the value/node mapping, allowing a client to find nodes that store specific data items. In the case of a DHT, for example, the routing substrate typically applies a hash function to the key to compute the IDs of nodes that store the associated value. However, other routing substrates may locate values in other ways.

The second component is the *key-value store*, which maintains a set of key-value pairs on each node. A key-value storage system typically exports a simple `get/put` interface. While existing storage systems store arbitrary, untyped byte strings, the Comet storage system stores *active storage objects* (ASOs). An ASO consists of a key and its associated *state* (i.e., a value, stored as an untyped byte string), along with optional *code* that operates on that state. The code is structured as a set of *handlers* that specify how the object behaves, i.e., how it modifies its state when certain events occur. For example, an ASO's `onGet` handler is invoked whenever a remote client performs a `get` operation to access an object. This handler might perform some simple operation, such as incrementing a counter for the number of gets or appending the client's IP address to a log structure. The counter or the log structure would be stored as part of the ASO's state that can be accessed by the handler.

The third architectural component is the *active runtime system*. The runtime system handles ASO invocations and provides the security policy and execution environment. An application running on a remote client specifies the initial state and handlers for an ASO when initially storing the object via a `put` operation. When a client performs a `get` or a `put`, it can optionally request a cryptographic checksum of the code associated with the target ASO. This can serve as an integrity

onGet(caller[, callbackID, payload]) Invoked when a *get* is performed on the ASO. Returns a value which will be passed back to the caller. Instead of returning a value immediately, the handler could also perform a *put* at the optional *callbackID* sometime in the future. The handler also takes an optional *payload* argument of arbitrary type.
onPut(caller) Invoked upon initial *put* when the object is created. Returns the value that should be stored by the node (e.g., itself or nil).
onUpdate(new_value, caller) Invoked on an ASO when a *put* overwrites an existing value. Returns the value that should be stored, e.g., *new_value* if it should be replaced, or itself if not.
onTimer() Invoked periodically. This handler has no return value. It is used to perform periodic maintenance such as replication.

Table 4.1: **ASO Handler Calls.**

check that the client's initial `put` is to a key with no associated ASO and that subsequent operations are performed on ASOs created by the application. In most implementations, a Comet node distrusts remote nodes and client applications; therefore, the runtime component of the active subsystem implements and enforces an ASO execution sandbox (Figure 4.1(a), top). Our Comet prototype uses a language sandbox based on Lua [168] to prevent a handler from accessing outside state and to constrain the ASO from consuming too many computational and memory resources on the host. The ASO runtime consults a security policy module, which specifies all execution limits.

While some applications may be satisfied by an entirely sandboxed execution, many would benefit from an ASO's limited ability to interact with or "sense" its environment. For example, to implement the conditional replication scheme we added to Vuze for Vanish, an ASO requires knowledge of the number of replicas in the DHT and the time of day (to enforce the desired minimum replication interval). For this reason, the active subsystem exposes a small API (called the ASO API) to the handlers.

4.3.2 Active Storage Object API

Table 4.1 and Figure 4.1(b) show the handler and ASO runtime APIs, respectively. The handler API supports invocations based on the primary storage functions – `put`, `get` – as well as an `onTimer` handler to be executed periodically (e.g., once every 10 minutes) during the object's lifetime. For example, an ASO could directly implement a custom replication policy in its `onTimer` handler.

The ASO runtime API is the only way for an ASO to interact with its environment outside of the sandbox. Our design supports two types of useful interactions: (1) obtaining information about the local node, and (2) executing various storage system operations. The former category includes functions to obtain the time of day, the hosting machine's external IP address, etc. The latter includes functions to interact with other storage system objects. The ASO API was not designed to be entirely general; rather, our goal was to provide a minimal interface, informed in part by our requirements of security, privacy, and isolation. We tested this interface by implementing and running over a dozen applications on our Comet prototype. Interestingly, we were able to build a relatively diverse set of applications with a surprisingly small interface, which has remained relatively stable through the project. This suggests that a small interface, like the one shown in Figure 4.1(b), can support a wide variety of applications. Naturally, there are limitations. For example, we explicitly prohibit any direct network-level interactions with remote nodes on the Internet. While this feature might be desirable to certain measurement applications, its DDoS implications would be unacceptable.

4.3.3 Language Based Sandbox

Our Comet prototype focuses on a DHT environment composed of a large number of untrusted autonomous nodes that cooperate to support the distributed active storage system. In this environment, the key challenges include providing a strong sandbox and limiting ASO resource consumption. We briefly describe how our system addresses these challenges using a language based sandbox.

The Comet prototype required an ASO programming environment that reflected our needs for simple extensibility, flexibility, performance, isolation, and safety. To meet these needs, we chose Lua [168], a lightweight and easily constrained scripting language. A dynamically typed, imperative and functional programming language, Lua is most commonly used for coding application extensions. In this context, it lets users add or modify features in video game engines, Web servers, ver-

sion control systems and other applications (specific examples include World of Warcraft, SimCity 4, Adobe Photoshop Lightroom, and Squeezebox Jive Platform). Several properties make Lua well suited for implementing ASOs. First, it employs a small set of programming constructs (including first-order functions) and a small number of data types (including tables, which are heterogeneous associative arrays). Second, Lua compiles to simple bytecode, which makes it relatively easy to sandbox. Finally, ASOs written in Lua are concise and small when serialized; the Lua ASOs we implemented are all under 1.5KB, about five to ten times smaller than Java equivalents.

Comet represents ASOs as Lua tables that encapsulate both persistent state and the handlers to be invoked on that state. Lua tables can implement basic arrays, associative arrays, or both. While an associative array can contain any name-value mappings, we treat certain associations as handlers. In particular, if the ASO table contains an associative array with the names "onGet," "onPut," "onUpdate," or "onTimer" – and those names are associated with values that are Lua functions – then the runtime invokes those functions when the corresponding events occur. Our runtime system serializes Lua tables into a byte stream for transmission to a storage node on a `put` request.

We made several modifications to the standard Lua interpreter for the Comet runtime system. We sandbox ASOs by removing all but the core libraries from the runtime, leaving only a math package, string manipulation, and table manipulation. As a result, handlers are extremely restricted: they have no direct network access, no system execution capabilities, no thread creation capabilities, and no file system access. We also strictly bound the amount of resources that a handler can consume. For example, the runtime limits both the number of bytecode instructions that a handler can execute and the amount of memory it can consume. If a handler exceeds either of these limits, the runtime terminates its execution.

The Comet runtime exposes a DHT wrapper object to handlers, which allows an ASO to communicate with its environment. The ASO can learn information about the hosting node, including the external IP address and the current system time. It can also perform a restricted set of DHT operations. For example, it can perform `get` and `put` operations on *replicated* copies of its value stored at other nodes. In the API presented in Section 4.3.2, these operations return values or neighboring node IDs. However, since these operation are slow in the DHT setting and may block for seconds or even minutes, we chose to implement them using function callbacks. Each such operation takes

an optional parameter, a function which accepts the result as its parameter. For example, instead of returning a value, a `get` operation takes a function which is eventually passed the result of the operation. The operation returns immediately with no value, and the `get` is actually performed after the ASO execution has completed. While this presents a slightly different paradigm to the user, we think this provides a greater ability to optimize the performance of Comet-based applications.

4.3.4 Comet Prototype Implementation

We built the Comet prototype on the Vuze DHT, which supports the widely used Vuze BitTorrent client. The DHT is used mainly for distributed tracking of torrents; however it has been used in research as well [98, 81].

Vuze implements the Kademlia routing protocol, in which each node is assigned a 160-bit ID based on the SHA1 hash of its IP address and port. Basic DHT operations (`get`, `put`, and `remove`) take a 160-bit key, perform a lookup to find nodes whose ID is *close* to that key, and then send a read or store RPC to those nodes.

We minimally extended the Vuze interface to conform to Comet's abstract operations. For example, we augmented `get` to allow a caller to pass an arbitrary byte-string argument. This supports a parameterized `get` operation, where the ASO can return different values depending on the parameter (analogous to the semantics for GET in HTTP).

Allowing extensibility in a DHT environment creates challenges, e.g., it has the potential to provide a platform for DDoS attacks. Therefore, in addition to the Lua resource restrictions described previously, we limit DHT communications that ASOs can perform in two ways.

First, we do not allow an ASO to perform operations on arbitrary DHT keys or nodes, but rather only on specific key-node pairs. An ASO may communicate with any of its neighboring nodes that are responsible for replicas of the ASO. We also allow the ASO to communicate with key-node pairs that have interacted with it in the past, once for each such interaction. To enable this functionality, we extended Comet requests to include the ID of the requesting node and the ID of a local key contained within the node. If an ASO receives a `get` request with a key ID specified, it gains the capability for a one-time operation on that key to the node that issued the request. The ASO can then either return a value immediately and exhaust its one-time capability, or save that capability

for future use. This mechanism allows applications to respond to DHT requests at a future point in time, especially if the requested data is not currently available. We do not allow ASOs to pass these capabilities between each other as doing so would enable a malicious node to mount DDoS attacks. In Section 4.4 we discuss signed ASOs, which do not have these restrictions.

Second, Comet imposes rate limits on the number of messages generated by an ASO, either to neighboring nodes storing replicas or to arbitrary key-node pairs that have interacted with it in the past. This prevents misbehaving ASOs from exhausting the bandwidth resources of the Comet nodes hosting them. We discuss these security issues further in Section 4.6.

4.4 Applications

This section seeks to demonstrate both the range of storage behaviors that Comet can support and the ease with which those behaviors can be implemented. To do this, we describe several of the active storage applications we have implemented, deployed, and measured on our Comet PlanetLab prototype. We provide code snippets to show how simply these actions can be programmed in our Lua-based ASO environment. In Section 4.5, we present measurements from some of these examples.

4.4.1 Customizable Replication

Most DHTs specify a fixed replication policy for stored values, requiring applications to conform to that policy. In contrast, Comet ASOs can provide their own application-specific replication mechanisms, e.g., controlling the replication factor, the replication interval, and the choice of nodes on which the object will be replicated. This flexibility is useful for applications that place varying degrees of emphasis on performance, availability, locality, and security. For instance, a security sensitive application (such as Vanish) might use a small number of replicas and long replication intervals, limiting the dispersion of its objects stored in the DHT. On the other hand, an application that values availability might replicate frequently to a large number of nodes.

Listing 4.1 shows how an ASO can define a customized replication policy. In this example, the `onTimer` handler wakes up periodically, invokes `lookup` to determine a list of nodes closest to the

ASO's key, executes `selectGoodNodes`[1] to identify a subset of nodes that will serve as replicas, and then stores a copy of itself on the selected nodes using `put`. We have also implemented a timer handler that replicates only when the number of existing replicas falls below a certain threshold; this lowers communication overhead and mitigates data harvesting attacks for security sensitive applications, reflecting the changes we made to Vuze after we published Vanish [81].

```
function aso:handleLookup(nodes)
    nodes = self.selectGoodNodes(nodes)
    dht.put(dht.getKey(), self, nodes)
end
function aso:onTimer()
    dht.lookup(dht.getKey(), self.handleLookup)
end
```

Listing 4.1: Smart Replication

4.4.2 Controlling Data Access

Comet objects can implement various policies that control how data stored in the objects is accessed. We illustrate a few such examples.

Timeouts and Limited-read values: ASOs can be used to implement objects that will be accessible for only a limited, application-specified time. Such objects are meaningful for security applications such as Vanish [81], which provide support for self-destructing digital data by storing cryptographic keys in a DHT.

Listing 4.2 shows the handler code required to implement application-specific timeouts. Each replica stores a timestamp when the object is created (stored) and then deletes the object after 60 minutes using a timer handler. In addition, the `onGet` handler prevents the object's contents from being accessed after the timeout but before it is deleted by a timer handler.

An ASO can also choose to delete itself after it has been read – providing a "limited-read value" – where each replica can be read at most once. In addition to its use for self-destructing data,

[1]The Lua code for `selectGoodNodes` is omitted for brevity. It implements an application-specific policy for choosing replicas.

```
function aso:onPut(value)
    self.timeout = dht.getSystemTime() + 60∗MINUTES
    return self
end
function aso:onTimer()
    if (dht.getSystemTime() > self.timeout) then
        −− delete local ASO
        dht.deleteSelf()
    end
end
function aso:onGet()
    if (dht.getSystemTime() > self.timeout) then
        −− delete local ASO
        dht.deleteSelf()
        return nil
    end
    return self
end
```

Listing 4.2: Timeouts

limited-read values could be used in settings where objects represent tasks and are deleted once they have been claimed by worker nodes. The object then serves as a synchronizing construct between the task's producer and consumer.

Listing 4.3 implements limited-read values. When a `get` is performed, the node records the fact that the value has been read. It then propagates the request to every other replica by overwriting them with `nil`. Note that the object does not delete itself immediately, but rather stays around for a while and periodically attempts to delete other replicas to ensure that copies on nodes with transient connectivity issues [72] are eventually deleted. Note also that concurrent `gets` issued to different replica nodes might successfully read the value. In general, as with other distributed storage systems, consistent update of replicated values would require the use of heavy-weight consensus operations. Comet does not currently provide such primitives. ASO handlers do however provide the ability for

replicas to detect and correct inconsistencies, e.g., ASOs can compare and reconcile replica contents through periodic invocations of the `onTimer` handler.

```
function aso:onGet()
    if (self.read) then return nil end
    self.read = dht.getSystemTime() + 30∗MINUTES
    dht.put(dht.getKey(), nil) −−deletes replicas
    return self
end
function aso:onTimer()
    if (self.read) then
        dht.put(dht.getKey(), nil) −−deletes replicas
        if (dht.getSystemTime() > self.read) then
            dht.deleteSelf()
        end
    end
end
```

Listing 4.3: Limited-Read Values

Data Subscription: An ASO can allow clients to "subscribe" so that they will be notified when the ASO receives a new value. In Listing 4.4, when the subscriber performs a `get`, the ASO saves the subscriber's network location (`callerNode`) and a key that will serve as the subscriber's recipient of the value (`callbackKey`). When a value `update` occurs, the ASO distributes the value to all registered subscribers – the runtime ensures that the ASO distributes these values *only* to clients who have actually performed a `get` on the ASO. In the example shown, the ASO clears its subscriber list after its `put` operations; subscribers must then re-subscribe if they're still interested. Later we will describe an implementation of a scalable publish-subscribe scheme based on this design.

Sensitive values: ASOs can implement various forms of access control policies. For instance, Listing 4.5 provides read access to the object's value only if the client can present a predetermined

```
aso.pending = {}
function aso:onGet(callerNode, callbackKey)
    if(self.value) then
        return self.value
    end
    self.pending[callerNode] = callbackKey
    return nil
end
function aso:onUpdate(callerNode, value)
    self.value = value
    for callerNode,key in pairs(self.pending) do
        dht.put(key, value, {callerNode})
    end
    self.pending = {}
end
```

Listing 4.4: Pub-sub

password akin to a feature already provided by some DHTs, like OpenDHT [164]. A client provides the password as an argument to the `get` request.

There are a few issues with the code provided above, especially if it were to be extended to support password-protected updates. A malicious node could claim to store the object but simply serve as a proxy for clients' requests and thereby implement man-in-the-middle attacks. This could be solved by exposing basic encryption primitives to the ASO, like a secure hash function and/or public key cryptographic primitives. For example, instead of passing the plaintext password to the ASO, the client hashes the concatenation of the password with its IP/port, thus the ASO can verify that the request is not being forwarded by a malicious node. The ASO's security can be further strengthened by public/private key pairs, with the ASO storing the public key and clients authenticating themselves by presenting a message signed with the corresponding private key. With these enhancements, a malicious node storing a copy of the object cannot overwrite the contents of other replicas since it doesn't possess the private key.

An application could use multiple mechanisms for controlling data access, e.g., it could use

```
function aso:onGet(caller, callerId, password)
    if (password == ‘‘mypass1234’’) then
        return ‘‘Well kept secret’’
    end
    return nil
end
```

Listing 4.5: Password

timeouts in conjunction with password-protected access. While Comet does not allow ASOs to register multiple handlers for a given storage operation, the developer can combine all of the desired mechanisms into a single handler. Though this might increase programming complexity, it allows the application developer to control how different mechanisms interact with each other and provides the basis for a predictable and deterministic execution model.

4.4.3 Measurements and Monitoring

DHT Measurements: ASOs provide a platform for instrumenting and measuring the DHT using the DHT nodes themselves. This enables a more detailed and comprehensive view of the DHT and helps provide accurate estimates of DHT properties such as churn, node lifetime distribution, transient inconsistencies, etc.

For instance, Listing 4.6 tracks the k closest nodes to the ASO and stores the information it learns as part of the object state. A measurement application can create objects of this type, store them at multiple locations within the DHT, and obtain snapshots of DHT membership by retrieving the objects' contents using `get` operations.

While this measurement could be performed by nodes that are not part of the DHT (as in earlier work [67, 195]), measurements from within the DHT can provide more accurate data. For example, the lifetime measurement could be carried out by a client that interactively crawls the routing tables of the DHT nodes and then uses heartbeat messages to monitor the uptimes of the nodes it learns about. This approach could provide faulty data, however, if the DHT contains firewalled nodes that do not receive or respond to such heartbeat messages.[2] On the other hand, firewalled nodes still

[2]In fact, about half the nodes in P2P DHTs are firewalled [76].

```
aso.neighbors = {}
function aso:handleLookup(nodes)
    self.neighbors[dht.getSystemTime()] = nodes
end
function aso:onTimer()
    dht.lookup(dht.getKey(), self.handleLookup)
end
```

Listing 4.6: Lifetime

communicate with neighbors, for example to replicate values. Therefore, measurements performed from ASOs *within* the DHT can be more accurate, as we will demonstrate later.

Monitoring uses: An ASO can also maintain audit trails, e.g., indicating where it has been stored thus far, who has read or updated the object, etc. Such tasks are particularly useful for debugging and aid in rapid prototyping. For example, this may help a developer to learn whether a new ASO replication mechanism is operating properly. Alternately, logs can also be used for forensics. Listing 4.7 illustrates a monitoring application that tracks the nodes storing and accessing a value.

This specific implementation comes with a few caveats. Each replica may have a different view of the list of nodes that have stored or read the value. To address this, the experimenter needs to get the union of the lists stored in all the replicas, consolidating them as a post-processing step.

4.4.4 Smart Rendezvous

DHTs are used for rendezvous in many distributed systems. In P2P file-sharing systems such as BitTorrent, the DHT is used as a distributed tracker either with or as a replacement for a centralized tracker. That is, peers that want to download a particular file use the DHT to identify other peers who are downloading or sharing the file. The downside with current DHT-based distributed trackers, however, is that they result in random overlay connections, as there is no mechanism to enforce more intelligent peer-matching techniques.

With Comet we can address this limitation by using ASOs to track participating nodes, as well as construct peer lists that are optimized for a requesting node. Peers could be matched in order to lower

```
replicaIps, hostIps, accessorIps = {}
function aso:onGet(callerIp)
    table.insert(self.accessorIps, callerIp)
    return self
end
function aso:onPut(caller)
    table.insert(self.accessorIps, caller.getIP())
    table.insert(self.hostIps, dht.localNode.getIP())
    return self
end
function aso:handlePut(nodes)
    for i,v in ipairs(nodes) do
        table.insert(self.replicaIps, v.getIp())
    end
end
function aso:onTimer()
    dht.put(dht.getKey(), self, 20, self.handlePut)
end
```

Listing 4.7: Monitoring

inter-node latencies [120], maximize reciprocation probability based on peer bandwidths [154], or lower ISP costs [228, 39]. We have implemented one such matching scheme that uses the nodes' network coordinates to predict inter-node latencies and provides a list of nearby peers to each joining node. We describe this in depth in Section 4.5.3.

4.4.5 Signed ASOs

The examples discussed so far adhere to the strict security policy we set out: ASOs cannot perform operations on arbitrary DHT keys or nodes. We now consider uses where we relax this assumption, but require that the ASO code be *signed* by the DHT administrator after manual verification of its security properties. As we will see below, this allows the DHT to deploy new functionality and services by using signed ASOs that access arbitrary DHT locations, but are safe (i.e., they do not

enable DoS attacks of targeted DHT nodes).[3] We have considered signed ASOs in particular as a mechanism that a DHT's developer or administrator could use for testing and evaluation of new features, before they are added to the main-line DHT code.

Recursive Get: Vuze and many other DHTs support iterative routing for key lookups. In this approach, the node performing the lookup is involved in every step of the routing operation, i.e., it identifies the target node by repeatedly querying DHT nodes to find other nodes that are closer to the target key. An alternative is to perform recursive routing, where intermediate nodes on the route pass the lookup directly to nodes that are closer to the key. Iterative lookup provides greater control to the node performing the lookup (e.g., it can control lookup parallelism), but it comes at the cost of increased latency. If both forms of lookup are available, an application would use recursive lookups by default, but fall back on iterative lookups after persistent failures [52].

With signed ASOs it is possible to implement recursive lookups even though the underlying DHT supports iterative lookup by default (as is the case with Chord, Kademlia, and Vuze). The node initiating the lookup creates a `query ASO`, which contains a reference to itself, and a local callback ID where it would like to receive the answer. When the signed ASO is created its `onPut` handler is invoked; the handler queries the local routing table to find a live node that is closest to the target key, stores a copy of the signed ASO on this node, and deletes itself from the current node. This process is repeated until one of the nodes storing the target is reached, and the `onUpdate` handler of the target ASO sends the object's value back to the original node, which initiated the request.

Caching and Hierarchical Publish-Subscribe: This idea can be extended to accomplish both caching and hierarchical publish-subscribe data delivery. For caching, the `onUpdate` handler can be modified to communicate the object not only to the requesting node but also to the intermediate node that conveyed the request. The number of intermediate nodes to which the object is replicated can be determined by gathering and analyzing statistics on object popularity (also accomplished using simple handler code), so that only popular objects are replicated at multiple nodes

[3] In some cases, the safety of the ASO code could presumably be verified automatically, e.g., by using sophisticated compile-time analysis; studying this is part of future work.

(as in Beehive [162]). To implement hierarchical publish-subscribe, intermediate nodes propagate a subscription event to the next node in the lookup process only if they haven't done so before and maintain state for subsequent queries routed to them. When a value is published, it is propagated through a dissemination tree so that the communication load is distributed across all intermediate nodes (as in Scribe and Bayeux [173, 233]).

4.4.6 Summary

This section described a set of example storage objects that we have implemented using Comet. Through these examples, it should be clear that with very small extensions (on the order of a few lines or a few tens of lines of code), a Comet application can create a wide range of powerful storage object behaviors that would be impossible in existing distributed storage systems or DHTs.

4.5 Evaluation

We deployed Comet on approximately 200 PlanetLab hosts and evaluated our design in three steps. First, we characterize the resource utilization of the various applications that we developed. Second, we measured micro-benchmarks to understand the overheads associated with active storage objects. Lastly we report on our experiences with prototyping applications using Comet.

4.5.1 Application Characteristics

Table 4.2 shows resource consumption requirements for our Comet applications. The *Max Instructions* column gives the number of dynamic Lua instructions required to execute the most expensive handler, while *Execution Time* gives the execution time for that handler. Where this value is data sensitive, we provide an estimate based on the expected maximum value. *Code Size* shows the size of each ASO with the minimum amount of data and *Max Size* is the maximum size to which the ASOs might grow for that application. From the table we see that most ASOs execute fewer than 100 Lua instructions and are smaller than 1KB in size.

Application	Max Instructions	Execution Time	Code Size	Max Size
Replication	< 10	4μs	$0.223K$	$< 1K$
Smart Replication	< 100	6μs	$0.444K$	$< 1K$
Timeouts	≈ 10	4μs	$0.434K$	$< 1K$
Limited-Read Value	≈ 10	4μs	$0.553K$	$< 1K$
Sensitive Value	< 10	4μs	$0.230K$	$< 1K$
Pub Sub	$10,000$s	54μs	$0.498K$	$100K$
Hierarchical Pub Sub	100s	6μs	$0.673K$	$1K$
Lifetime (External)	100s	6μs	$1K$	$6K$/hr
Lifetime (Internal)	< 100	6μs	$1.776K$	$\approx 3K$
Monitoring	≈ 10	4μs	$0.971K$	$3K$/hr
Smart Rendezvous	$1,000$s	14μs	$1.107K$	$10K$
Recursive Get	≈ 50	6μs	$0.714K$	$\approx 1K$

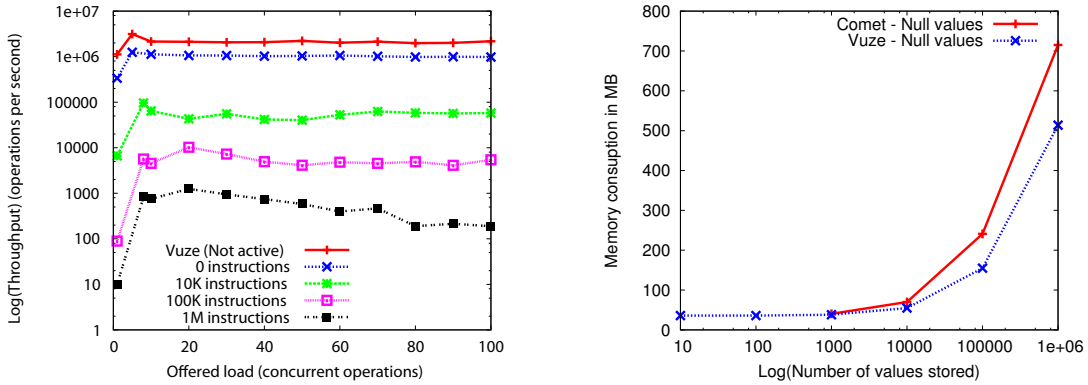
Table 4.2: Expected Application Resource Consumption.

4.5.2 Performance and Overheads

We report on simple microbenchmark measurements to compare the CPU and memory costs of Vuze and Comet. These experiments were run on an quad-core machine with Xeon processors clocked at 2.67GHz.

Single-Node Throughput. In this experiment, concurrent `get` operations are performed on many values stored in the target node. We measure the throughput of `get` requests that return successfully using a closed feedback loop. All operations are issued locally on the node, so that network latency does not affect throughput.

Figure 4.2(a) compares the throughput of objects with different ASO execution costs, expressed as the number of Lua bytecode instructions executed per handler. Both Comet and Vuze experience peak throughput when the number of concurrent operations is equal to the number of cores (eight). ASOs with zero instructions per handler are functionally equivalent to Vuze values as they simply return themselves. The peak throughput of Comet ASOs is about 60% smaller than the peak throughput of Vuze (1.4M operations per second as opposed to 3.5M operations per second). This



(a) Single-Node Throughput.

(b) Memory Footprint.

Figure 4.2: **Microbenchmark Results.**

shows the cost of the Comet/Lua execution environment. Previous measurements [194] show that the typical DHT load on Vuze clients in the wild is at most a few hundred operations per second, which makes the additional Comet overhead relatively insignificant in this context. As we increase the computational complexity of the average ASO (1K to 1M instructions per handler), the throughput decreases, but still remains well above the maximum current Vuze workload.

Operation Latency. At the 90th percentile, with maximum throughput (8 concurrent operations in our experiments), a request involving 100 Lua instructions has a latency of about 300 microseconds. For handlers with 1M instructions (two orders of magnitude more than our most compute-intensive handlers), it is 13 milliseconds. The latency for a Vuze DHT lookup is on the order of seconds, therefore the latency imposed by even extremely computationally intensive ASOs is not significant.

Memory Footprint. In this experiment, we store increasing numbers of values in the nodes. For the Vuze nodes, the string "hello world" is stored at different keys, while for Comet nodes we store an equivalent Lua ASO which returns the same string upon a `get` request. Figure 4.2(b) compares the memory footprint of the Vuze and Comet nodes as we increase the number of stored objects. Again using the median number of values stored per Vuze node (around 400), the difference in memory consumption at this level is negligible (about 36MB for both Comet and Vuze). Long lived DHT nodes can store 10,000s of values, and the highest observed is around 30,000 values [194]. In these

rare cases, our overhead relative to Vuze is about 27%, but even then the total memory footprint is still reasonable.

We next consider a workload where Comet object sizes are exponentially distributed with an average size of 10KB. In this case, a node with 500MB can store on average 50,000 values. If we assume an order of magnitude more values per node than in Vuze (4,000 instead of 400), and an order of magnitude larger values (10KB instead of 1KB limit imposed by Vuze), the median node would consume about 80MB (40MB of startup memory costs and another 40MB for the ASOs) in memory.[4]

4.5.3 Application Experience

We now report on our experiences in prototyping and deploying some of the applications described in Section 4.4.

Measuring Node Lifetimes

We revisit the experiment performed by Falkner et al. [67] to measure the lifetimes of nodes in the Vuze DHT. This experiment was done by performing random get operations from several Vuze clients in order to gather approximately 300K IPs participating in the DHT. The collection of nodes was then pinged every 2.5 minutes to check for liveness. The authors observed that nearly half the nodes were immediately unavailable after first being detected. One weakness of the methodology employed is that the clients could not differentiate nodes that are unreachable because of NATs from those that have left the DHT. Using measurement nodes that have active communication channels with NATed DHT nodes would help minimize measurement bias, but would require the measurement to be performed by nodes that are *within* the DHT.

Comet enables researchers to deploy experiments using measurement ASOs executed on nodes that are part of the DHT. To demonstrate the feasibility of this approach, we deployed Comet on 190 geographically dispersed PlanetLab nodes and integrated them into the production Vuze DHT. The measurement ASOs are stored on the Comet nodes, and they gather information about unmodified Vuze nodes that are adjacent (in the DHT) to the Comet nodes. We stored a lifetime measurement

[4] Vuze and Comet consume about 40MB without storing any values.

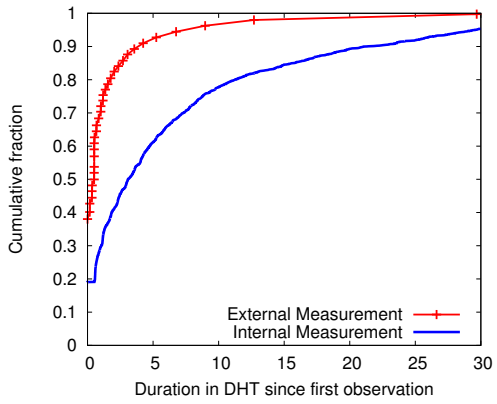


Figure 4.3: **Node Lifetimes in Vuze.**

ASO (a variant of the code shown in Listing 4.6) at each of the Comet nodes, allowed the nodes to perform measurements for several days, and then collected and analyzed the data from these nodes.[5] Figure 4.3 plots the measurement data obtained from our experiments and compared to the lifetime data obtained by measurement nodes that are not integrated into the DHT (as in [67]). We observe that the measurements performed from within the DHT provide higher estimates for node lifetimes. The reason is that DHT-internal measurement nodes are able to traverse NATs in communicating with their neighbors. The difference is significant; we measured the median node lifetime as 3.1 hours, as opposed to an estimate of 0.5 hours obtained through conventional external measurements. Measurement ASOs are thus valuable tools in characterizing DHTs and provide more accurate data for tuning parameters such as replication factor, routing parallelism, etc.

Smart Rendezvous

In Section 4.4, we proposed a way to employ intelligent peer tracking for distributed P2P trackers using ASOs. We evaluate the usefulness of this application by deploying a distributed tracker built with Comet ASOs. As with traditional distributed trackers, clients participating in a P2P swarm (such as a BitTorrent download) register their participation by storing their IP addresses under the

[5] As Comet is not currently deployed by Vuze, the measurement ASOs are stored only on the nodes that we control. A more extensive deployment would allow us to obtain more samples quickly.

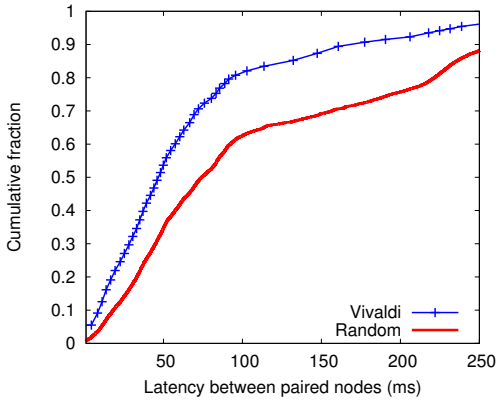


Figure 4.4: **Proximity of BitTorrent Peers.**

appropriate DHT key. In addition, clients also store their network coordinates (computed using Vivaldi [50]) along with their IP information. When clients contact the distributed tracker to obtain peer lists, the tracker ASO estimates the network latency between pairs of nodes using the supplied network coordinates and returns peers that are likely to be close to the requesting node. To evaluate this approach in practice, we deployed a tracker ASO on a Comet node in PlanetLab, while 190 PlanetLab nodes acted as peers in the swarm reporting their Vivaldi coordinates to the tracker and requesting good peers with which to communicate. Figure 4.4 depicts the effectiveness of this strategy compared to the default strategy of returning a random subset of peers to the requesting node. The graph shows a CDF of the measured latencies between peers under the two different matching schemes. The median value for the ASO-implemented Vivaldi intelligent peer matching is 47ms compared to a median of 72ms for the default scheme, a 35% latency improvement.

Vanish

Comet grew in part out of our experience specializing the Vuze DHT for Vanish [81], a self-destructing data system. Vanish used Vuze for key storage, however, Wolchock et al. [226] showed that the Vuze system was extremely open to a Sybil data harvesting attack that is able to scan the DHT for values. The attack worked in part because of Vuze's overly zealous replication policy – a high replication factor, coupled with a policy to replicate to new nodes immediately. In response,

we set out to deploy new replication mechanisms and other anti-Sybil defenses in Vuze, as described in Section 3.6. While these mechanisms were straightforward, deploying them required the co-operation of Vuze's designer and was an arduous and imperfect process. While many iterations would have been necessary to fully test and optimize policies, we often had only one shot to catch the two-month release cycle.[6] For the same reason we were unable to test individual changes in isolation as they had to be shipped in bundles in order to make progress in reasonable time.

We have used Comet to re-implement several of the changes that we deployed in Vuze. Those changes include the customizable replication scheme described in Section 4.4 (particularly a scheme that replicates only when the number of replicas falls below a threshold) and variable object lifetimes. As we showed in Section 4.4, both of these changes are trivial to program as Comet ASOs. Perhaps even more important, testing and re-deployment in Comet is significantly easier, as it does not require a redistribution of the entire DHT code base. Instead, new mechanisms can be deployed by overwriting the handler code for existing objects and using the updated bytecode for subsequently created objects, without requiring the involvement of the DHT administrators.[7] Had Comet existed at the time we deployed Vanish, it would have been possible to customize the DHT for the security requirements of the application from the start, and to optimize those policies to Vanish's requirements.

4.6 Security Analysis

The classic security goals for DHTs include resilience to attacks that: violate the system's ability to robustly store data [188], disrupt routing [188, 35], identify the participating nodes in the DHT [200, 196], and harvest copies of data stored within the DHT [226]. There are numerous well-known techniques aimed at violating these goals, including Sybil attacks [62], Eclipse attacks [186], and many others [207]. And there are also many known mechanisms for protecting against such attacks, including the use of strong identities minted by a logically centralized authority, computational puzzles and bandwidth contributions proofs [28, 53, 55, 231], and architectures built upon

[6] It takes a week or more from release until 80% of the nodes in Vuze adopt changes. This is in addition to a typical release cycle Vuze employs, which spans about a month.

[7] In general, updating the handler code for existing objects would require the application to keep track of its ASOs. In the case of applications such as Vanish, where objects are transient and have timeouts in the order of a few hours, we can also let existing objects just expire without explicitly updating them.

social network structures [114, 231]. A production DHT with ASO support must consider such classic security goals, and can leverage known countermeasures for the corresponding threats. (Although, as exemplified by Vuze and other popular DHTs, a DHT for ASOs may decide that the risks associated with these threats are minimal, and hence not deploy the known defenses.)

The security concerns of DHTs with *signed* ASOs are roughly those of conventional DHTs without ASOs (since the signed ASOs can be viewed as "vetted" parts of the DHT system itself); we therefore do not consider signed ASOs further. Empowering DHTs with *unsigned* ASOs does, however, create a *new* potential attack vector not present in conventional DHTs – namely, attacks via malicious ASOs. We seek to ensure that a malicious ASO cannot: infer private information about or damage its Comet hosting node; infer information about or affect the properties of other ASOs stored within Comet; or infer private information about or affect the properties of other Comet nodes and arbitrary computers on the Internet. To place these goals in context, we stress that while an attacker could always use her own custom software to communicate with Comet in arbitrary ways, including putting to or getting from arbitrary ASO keys and communicating with the broader Internet in arbitrary ways, our goals – if attained – imply that ASOs cannot be used to amplify the attacker's resources or capabilities. For example, an attacker should not be able to create an ASO "worm" that spreads virally, mounting a DDoS attack against a victim ASO or device on the Internet.

We find that it is possible to meet these goals using three architectural features: (1) restricting system access, (2) restricting resource consumption, and (3) restricting within-Comet communication. We consider each in turn.

Restricting system access. We designed the ASO API to be highly restrictive. The API explicitly restricts an ASO's ability to infer private information about its host or to affect the host's state. The API similarly restricts an ASO's ability to interact with arbitrary devices on the Internet. For example, the API limits an ASO's IO capabilities to explicitly defined DHT operations; arbitrary disk, network, and other IO operations are prohibited. The API also prevents an ASO from introspecting its host; e.g., although we allow the ASO to learn its host's external IP, we explicitly prevent the ASO from learning its host's internal IP. Without these restrictions, an ASO could potentially read private files on the host's disk, write sensitive files, attempt to DoS an arbitrary remote node, map

the network topology of internal IP networks, and so on. The Lua sandbox provides a simple mechanism for achieving this isolation. Namely, we removed the IO system call interface and exposed one containing only the restricted DHT operations instead.

Despite these restrictions, it may be possible for an ASO to infer (minimal) information about the hosting node via side-channels. For example, the time it takes an ASO to perform a computation could leak information to the ASO about the speed of the hosting processor. At the extreme, it may be feasible to infer modest information about other applications running on the hosting node [167]. We believe that such attacks are low risk in the Comet environment and do not consider them here.

Restricting resource consumption. Comet also significantly limits an ASO's ability to consume resources on its hosting node. Our prototype limits both the memory and CPU consumption of ASOs.

Memory. The Comet active runtime keeps a running sum of the memory footprint of an ASO. Hard limits can be set on the total memory consumption of an object; ASOs which exceed this limit are evicted. Our current prototype limits ASOs to 100kB.

CPU. The Comet runtime similarly keeps a running count of bytecode operations performed. We envision multiple policies for constraining CPU use. The naive policy limits each ASO to at most a limited number of instructions per handler invocation. Since not all Lua operations are equally costly, a more sophisticated policy would assign different weights to different Lua operations (e.g., more cost for a table lookup than an addition). The limit could also be enforced over a fixed duration of time (such as 30 minutes) rather than upon each handler invocation (which might occur much more frequently). Our current prototype implements the naive restriction and allows 100K instructions per handler invocation.

Comet provides support for exception handling in order to help debug faulty ASOs that exceed the system-imposed resource limits. Handlers can catch resource exhaustion exceptions and store the relevant handler state as part of the ASO. The developer can then retrieve this stored state and inspect it to determine why the handler exceeded the resource limits. Further, operations that return values, e.g., `gets`, provide the stack trace as a return value in the case of an exception. We found these features to be useful in debugging many of the applications that we prototyped using Comet.

Restricting within-Comet communications. We must consider two classes of communications: communications between one ASO and another, and callback communications to a caller.

Communications between ASOs. Allowing arbitrary between-ASO communications in Comet could lead to abuse. For example, suppose a malicious ASO stored under one key copies itself to a large number of other keys slowly over time, and then simultaneously all ASOs initiate connections to a victim ASO stored under some target key. Such an attack allows an attacker to amplify her resources: the attacker invests minimal effort to seed the original malicious ASO, yet the ultimate attack DDoSes nodes hosting the target key. Comet takes a Draconian approach toward protecting against such attacks: the ASO API only allows ASOs to communicate if they are stored under the *same* key, whether co-located on the same Comet node or on another node within the DHT. Our system further rate-limits communications performed by a particular ASO. Each Comet node allots a limited number of network communications per time period for every ASO it hosts. Though we have not experimentally ascertained appropriate rate-limiting parameters, the applications we present could all work with approximately the same number of network operations as is required for a value in the current Vuze DHT - about 20 every timer interval.

4.7 Related Work

The concept of extensible systems has been widely explored in the past in several domains. Extensible operating systems have been proposed that support application-specific needs [20, 180, 106]. Active networks allow code to be downloaded along with network data and executed within the network infrastructure (e.g., on routers) to extend network services [221, 204]. Active messages execute a small amount of user code with each message reception [213]. Click explored the design of an extensible router [110]. Database triggers allow applications to define procedural code that is executed in response to database operations [137].

In the context of storage systems, Watchdogs [21] extends the Unix file system, allowing a user-mode process to interpose on file operations for specific files to change access semantics. Several projects have proposed the integration of CPUs and disks to create intelligent disk storage systems that can provide on-board application-specific functions, e.g., for decision support systems, data mining, and image processing [107, 165, 1].

DHTs are increasingly used to support a variety of distributed applications, such as file-sharing, distributed resource tracking, end-system multicast, publish-subscribe systems, distributed search engines, and even data-center applications. Some of these systems (e.g., as CFS [51], i3 [197], and PAST [172]) can be implemented using the traditional put/get interface, but many others (e.g., Mercury [23], CoralCDN [71], Scribe [173], and Bayeux [233]) require customized interfaces and are implemented by altering the underlying DHT mechanisms in significant ways. Our work provides the ability to extend a DHT without requiring a substantial investment of effort to modify its implementation.

Deployed DHTs don't currently offer good semantics and security. However, people do know how to make them consistent [118, 136] and harden then against attacks [35, 53, 188, 93, 218]. The reason DHTs do not currently implement these techniques is that there has not yet been a deployed application that truly needed strong semantics and security. For example, the Vuze design perceived many threats as irrelevant [76] and deployed few defenses against them. However, after the new, more demanding Vanish application was proposed [81], the Vuze DHT responded by embracing a variety of effective security measures. In addition to enabling new applications atop DHTs, we hope to drive the design of these systems towards well-understood, yet unadopted levels of security and consistency.

4.8 Summary

Key-value stores serve as scalable distributed storage clouds for many of today's Web services and peer-to-peer applications. Despite their popularity, constructing powerful applications on top of today's key-value stores is challenging, due to their inflexibility and lack of control over how the data is managed in the store. This chapter described Comet, an extensible, active, distributed key-value store. Comet enables clients to customize a distributed storage system in application-specific ways using Comet's active storage objects. By supporting ASOs, Comet allows multiple applications with diverse requirements to share a common storage system. We implemented Comet on the Vuze DHT using a severely restricted Lua language sandbox for handler programming. Our measurements and experience demonstrate that a broad range of behaviors and customizations are possible in a safe, but active, storage environment.

While our Comet design has focused on peer-to-peer DHTs, the broad concept of extensible distributed storage systems is applicable to centralized distributed storage services, such as Amazon S3, Mozy, and DropBox, themselves inflexible clouds. We believe that the design of extensible datacenter-based storage services is an exciting avenue for future research, which is described in Chapter 6.

Chapter 5

MENAGERIE: A FRAMEWORK FOR ORGANIZING AND SHARING DISTRIBUTED WEB DATA

We now turn to the third problem investigated in this dissertation: the personal data management challenges created by data dispersal. The migration from desktop applications to Web-based services is scattering personal data across a myriad of Web sites, such as Google, Flickr, YouTube, and Amazon S3. This dispersal poses new data management challenges for users, making it more difficult for them to: (1) organize, search, and archive their data, much of which is now hosted by Web sites; (2) create heterogeneous, multi-Web-service object collections and share them in a protected way; and (3) manipulate their data with standard applications or scripts. This chapter presents Menagerie, a system that creates a unified view over the user's Web scattered data by imposing a unified naming, protection, and access system atop supportive (or proxied) Web services. Menagerie was initially introduced in a 2008 paper [77]. We begin by describing Menagerie's motivation and high-level overview.

5.1 Motivation and Overview

While Web-based services offer undeniable advantages over traditional desktop software, desktop systems have compelling advantages of their own, many of which arise from the functions provided by the desktop operating system and file system. The OS supports a set of common, beneficial services that we take for granted. Users can name, organize, and access all of their files within a single hierarchical namespace, irrespective of which applications natively operate on them (Figure 5.1a). Similarly, applications written by different software vendors can interact with each other through the protected sharing interfaces exposed by the OS, providing users with new composite functions.

As desktop-based applications, such as office productivity software, desktop email clients, and local storage are replaced by Web-based counterparts, we risk losing the advantages enjoyed by

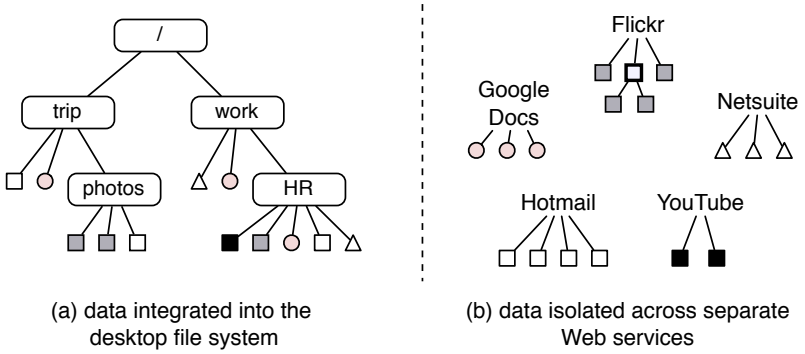


Figure 5.1: **PCs vs. Web Services.** In the desktop-centric world, users can organize and share their application data through the file system. In today's Web, data is increasingly trapped inside its Web service.

desktop systems. Users' personal data and objects are scattered across the different Web services that they use (Figure 5.1b). Consequently, users and services face a set of significant new challenges:

1. *Data organization and management.* On the desktop, a user can create a folder to hold related files and objects. On the Web, users' data is scattered across the Internet, where it is housed by a myriad of independent Web services. Given this, how can she organize, manage, archive, or search her Web objects and files *as a unit*?
2. *Protected data sharing.* While publishing is greatly simplified in the Web service environment, protected sharing, particularly at a fine grain, becomes more difficult. For example, how should one user share a specific subset of her objects with another user? Does the other user need to create accounts on all relevant Web services, and if so, do all of these services support the restricted sharing of only a select object subset?
3. *Data manipulation and processing.* Web services restrict the operations that can be performed on their objects: they typically export a limited API and expose only a small set of user commands through the browser. In contrast, the power of a system such as Unix derives, in part, from its simple data-processing commands (`cat`, `grep`, etc.) that can be composed together or extended to manipulate data in new ways. How should we balance the need for Web services to retain ownership over the data and functions they provide, with the benefits that would be gained by allowing third parties to extend services?

This paper examines these challenges. First, we discuss the principles and requirements that must underlie any solution. Next, we discuss the design and implementation of Menagerie, a proof-of-concept system that embodies our solution principles. Menagerie consists of two primary components: (1) the Menagerie Service Interface (MSI), an API that facilitates inter-Web-service communication and access control, and (2) the Menagerie File System (MFS), a software layer that allows "composite Web services" to integrate remote Web objects into a local file system namespace, reducing the engineering effort required to access and manipulate remote data.

To demonstrate the value of our approach, we have prototyped several new Web applications on top of Menagerie. Our experience shows that it is possible to combine the ease of use, publishing, and ubiquitous access advantages of Web services with the organizational, protected sharing, and data processing advantages of desktop systems.

In this section, extend the simple motivating scenario introduced in Section 1.2.1 to expose some of the shortcomings of the Web from a data management perspective. From this scenario we derive a set of required properties that a solution must have to overcome these limitations.

5.1.1 Example Scenario

Figure 5.2 illustrates our extended scenario, where Ann, the manager for a small company, has moved from a desktop environment to Web services for both her personal and business data and information processing needs. Specifically, Ann uses Flickr to manage her photo albums, Google Docs for spreadsheets and word processing files, Hotmail to communicate with colleagues and family, and Netsuite to process her and maintain her personal financial information.

Ann likes to keep her data well organized. In the past, she used her PC's desktop manager to create folders in which her related files were stored or linked. Since many of her documents are now Web-based, she would like to create virtual Web folders that are populated with links to the appropriate Web objects and collections. For example, she would like to collect all of her product marketing resources into a single folder, in spite of their spreading across many Web services.

Ann also wishes to securely share some of her virtual folders with her colleagues, granting them access to view and edit the folders' contents. However, she does *not* want her colleagues to have

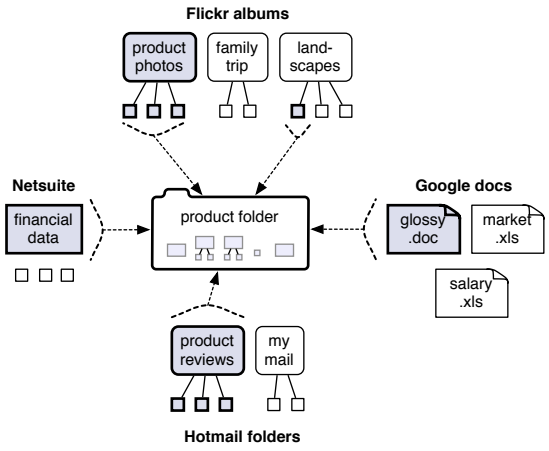


Figure 5.2: **Motivating Scenario.** Ann would like to create a new folder that links to some of her Flickr, Hotmail, Google Docs, and Netsuite objects. As well, she wants to share the folder and its contents with her colleagues, who do not have accounts on all of these services.

access to all of her business files or to her personal files. In addition, not all of her colleagues have accounts on the same Web services as Ann.

Finally, Ann is extremely careful with her valuable data and wants to prevent against accidental deletion or an operational Web service failure. She would therefore like to use a third-party archival service to maintain historical versions of all of her Web objects and virtual folders.

5.1.2 Challenges

Given the limitations of today's Web, it is extremely difficult for Ann to accomplish her goals or for third-party Web services to help her. Ann faces three classes of obstacles:

Naming. The Web services in our example provide users with the abstraction of objects that can be manipulated in various ways. Unfortunately, not all of the services expose objects with a predictable, stable URL; instead, some objects are externally presented by the Web service as a diffuse collection of HTML elements, images, frames, and JavaScript, whose URLs might be dynamically generated. Accordingly, users and third-party services have no easy way to *name* each of the objects that Ann wishes to collect into her virtual folders.

Protection. Ann needs to share some of her objects with her colleagues and with the third-party archival service, but she faces several protection-related impediments. Each Web service has imple-

mented its own particular authentication, authorization, and sharing scheme. Thus, Ann's colleagues may need to create accounts on all services to fully access her shared objects.

Even if single-sign-on accounts existed across the Web, many services fail to offer flexible and fine-grained protection. In some cases, sharing is all-or-nothing. For such services, allowing Ann's colleagues access to her professional objects may also reveal her personal data. Sharing also may be limited in some ways; for example, some Web services do not allow the sharing of different subsets of objects with different subsets of users. Finally, some services provide secure URLs that the user can hand out to grant object access, but many of these services do not support the selective granting of write access or the revocation of rights.

Ann wants to grant her associates access to a single virtual folder, implicitly giving them access to all of the objects within it. Unfortunately, those objects are scattered across many different services, each with its own authorization scheme. Short of Ann giving a third-party aggregation service her Web credentials and trusting that service with her objects, such sharing cannot be achieved.

Externalization and embedded rendering. Most Web services do not expose object data directly to users and third party services. Instead, they graphically present objects and interaction controls as embedded elements within Web pages. In contrast, on desktop systems, the filesystem permits many programs, including file managers, file sharing applications, editors, archivers, and security scanners, to process the same data objects.

To realize our scenario, Web services must provide additional functions that most of them lack today. In particular, they must export externalized representations of their objects to allow third-party services, such as archival or indexing services, to operate on that data. For simple third-party services, the structure and semantics of the externalized representation does not matter: the object can be exported as an opaque set of bytes. For richer services, a standardized or well-known representation, such as MIME for email, would be more valuable.

Finally, Ann and her colleagues rely on a third-party service to create and access virtual folders, and to browse the files within them. To support this, origin Web services should provide useful metadata and facilitate composite graphical interfaces that would allow the objects to be rendered and operated on within arbitrary Web pages. Flash movies exported by sites such as YouTube are good examples of this.

5.2 Goals and Requirements

In the PC-centric world, the operating system provides abstractions, system call interfaces, and utilities to help applications and users overcome the challenges we describe above. In the Web, there is no single trusted layer that users, browsers, and services can rely on. We therefore believe that a new service interface must be defined and adopted to provide the interoperability and integration needed to realize even our simple motivating scenario.

This service interface could be defined via conventions on top of the HTTP protocol (e.g., REST[68]), or new special-purpose protocols could be designed for this purpose. Regardless, the challenges we described motivate three clear requirements that the service interface must support:

1. *Uniform object namespace.* To address the naming challenge described above requires a single global namespace in which all personal data objects are embedded. That is, all of the objects and object collections that users manipulate should have a permanent, globally unique name within this namespace, allowing the Web service, its users, and third-party composite services to discover and depend upon these names.
2. *Fine-grained protection.* To support data sharing and composite services, a Web service must provide fine-grained protection of objects and collections. It should be possible for the user to share only a portion of her objects from a service, while keeping the other objects private. It should also be simple to aggregate and share collections of distributed objects.
3. *Unified minimal object access.* The combination of a global, hierarchical namespace and fine-grained, protected sharing of personal data allows users and services to find and share objects with each other. To be useful, however, the objects must support some standard set of access functions. As we argued above, the minimal set must include the ability for objects to be embedded and rendered within an arbitrary Web page, and for object data to be externalizable.

The next section presents the architecture and implementation of *Menagerie*, a proof-of-concept prototype we have developed to meet the challenges we have described. Menagerie allows us to experiment with new Web applications that support the organization and sharing of collections of heterogeneous Web service objects. We will describe those applications in Section 5.4.

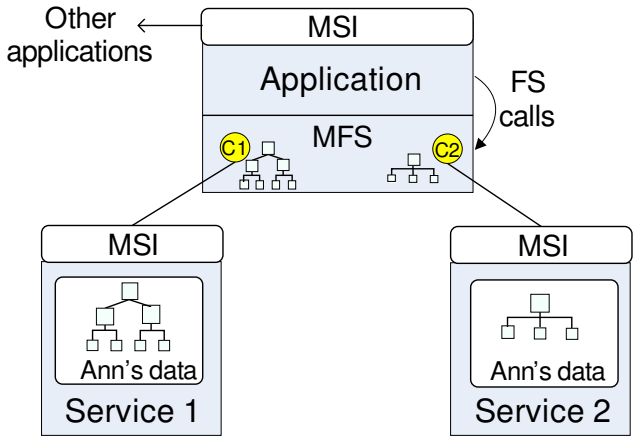


Figure 5.3: **The Menagerie Prototype.** The figure shows two Web services that export Ann's objects, a composite Web application built using the MFS layer, and the MSI capabilities (c_1 and c_2) that the application uses to access the objects.

5.3 The Menagerie Prototype

This section describes the structure and implementation of our Menagerie prototype. It consists of two principle elements: the Menagerie Service Interface and the Menagerie File System. We briefly introduce these elements here and then describe them in more depth in the remainder of this section.

The Menagerie Service Interface (MSI) is an inter-Web-service communications API that is comprised of object naming, protection, and access operations. MSI defines a uniform, hierarchical name space into which Web services *export* the names of their objects. MSI supports fine-grained sharing of Web objects through the use of *hybrid capabilities*. This protection scheme allows users without service accounts to name and access objects, while also giving services the ability to limit the actions of such users. MSI also specifies a standard set of *object-independent access functions* for Web services. These functions support object reading and writing, rendering, and metadata export. While our goal is to design an interface that Web services can easily adopt, our prototype implementation also shows that Menagerie is deployable even without Web service support.

The Menagerie File System (MFS) simplifies the development of new, composite Web applications. MFS mounts remote MSI object hierarchies into a local file system name space, allowing an

Namespace functions

list(capa, object_ID) returns *list of object names and IDs*

mkdir(capa, parent_ID, name)

getattr(capa, object_ID) returns *object attributes*

Protection functions

create_capa(capa, object_ID, rights) returns *new capa*

revoke_capa(object_capa, revoke_capa)

Content and Metadata functions

read(capa, object_ID) returns *byte[]*

write(capa, object_ID, name, content)

get_summary(capa, object_ID) returns *string*

get_URL(capa, object_ID) returns *string*

Figure 5.4: **The MSI Interface**. This table shows the parameters and return types of each function. MSI services must support the naming and protection-related functions, and may optionally support the others.

application to access remote Web objects through a standard file system interface. Figure 5.3 depicts a composite Web application that uses MFS to access the Web objects exported by two MSI-capable Web services.

The remainder of this section describes MSI's naming, protection, and content operations. Figure 5.4 shows the functions we have implemented to date. This small set was sufficient to build our example applications; as we gain more experience, we expect the interface to evolve and grow.

5.3.1 Object Naming

We designed naming in Menagerie with two goals in mind. First, users must be provided with meaningful object names that correspond to the way users name objects inside of a Web service. Second, composite applications must be provided with global, unique identifiers for the objects they access, even though those objects are scattered across heterogeneous Web services.

In Menagerie, each Web service exports an *object name hierarchy* for each of its users. This hierarchy contains the user-readable names of all objects that each user can access. The structure of this hierarchy and the granularity of each object within it are left entirely up to the service, but it typically imitates the logical structure that the service exposes to its users. For example, Flickr offers its users abstractions associated with sets of objects (photo albums) and objects within each set (photos); therefore, Flickr could choose to export a three-level name hierarchy (e.g., `Ann/Disneyland-album/Mickey-photo`).

Each object in Menagerie is identified using a *service-local ObjectID*, which is unique within the service and independent of the object's location in the hierarchy. Using the service-local ObjectIDs, Menagerie mints globally unique object identifiers by combining the service-local ObjectIDs with services' DNS names. By making ObjectIDs unique on each service (as opposed to globally unique), we give services the liberty to create and name new objects independently. By making an object's ID independent of the object's location within the service's hierarchy, we ensure that caching and other optimization opportunities are preserved even if the object can be reached via multiple paths.

Three functions in MSI support name hierarchy operations: `list`, `getattr`, and `mkdir`. Given the unique ID of a collection node in a hierarchy, `list` returns the names of all the children of that node, as well as their unique IDs. `Getattr` returns the attributes of the object with the given ID, including the type of object, a capability for the object (see Section 5.3.2), the size of the object in bytes, and various additional metadata. `Mkdir` adds a collection object to the hierarchy. Individual objects are created using the MSI `write` function, as we will see in Section 5.3.3.

5.3.2 Protection

While designing Menagerie's protection model, we considered the two fundamental access control mechanisms: capabilities and access control lists (ACLs). These mechanisms generally lie at opposite ends of a spectrum. Capabilities simplify sharing, while ACLs enable tight access control and user access tracking. While our goal is to simplify fine-grained, distributed object sharing, we must also provide services with the ability to control and track access to their data.

Menagerie therefore adopts a *hybrid capability-based protection system*, which combines the benefits of both mechanisms. A Menagerie capability is an unforgeable token that contains the globally unique ID for an object and a set of access rights. Possession of a capability gives the

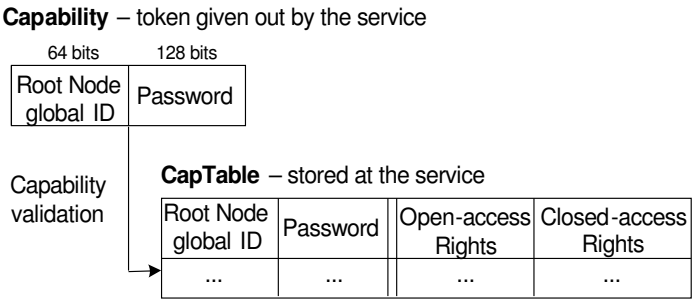


Figure 5.5: **Hybrid Capability Protection.** A capability provides access to objects within a sub-hierarchy rooted in the object identified by Root Note ID. Open-access rights allow direct object access, while closed-access operations also require user authentication.

holder the right to access the object in the specified ways. Capabilities support sharing because they are easy to pass from user to user: Menagerie's capabilities are encoded in URLs that can be emailed or embedded in Web pages.

However, a Menagerie capability is also subject to control by the Web service whose object it names. A service can divide its object rights into two types: *open-access rights* and *closed-access rights*. An open-access right gives the holder of the capability direct access to the specified operation without further authentication; e.g., if the right allows the user to read the object, then the service will return the object's contents when presented with a capability with the read bit set. Since a capability is not associated with any principal, an "open-access" operation cannot be attributed to a particular user.

A closed-access right, however, requires additional authentication. To perform an operation associated with a closed-access right, a capability with that right enabled is necessary but not sufficient: the user must also authenticate himself before the service will perform the operation. In most cases, this will require an account on that service. By "closing access" to an operation, the service can track the user that invokes the operation, or enhance the user's experience with personalized functions.

To implement capabilities, we use the *password-capability* model [36, 202]. The structure of a Menagerie capability is shown in Figure 5.5. The capability specifies a globally unique ID of a node

in a service's hierarchy and it authorizes access to the entire sub-hierarchy rooted in that node. The capability also contains a long "password" – a random field chosen from an astronomically large number space. The password is generated by the service at capability creation time and ensures that the capability cannot be guessed. A service stores information about all capabilities it creates in a table called *CapTable*, whose structure is also shown in Figure 5.5. Because the service stores the capability rights, they cannot be forged by users.

As seen in Figure 5.4, every MSI method call passes at least two parameters: a capability token for an ancestor of the accessed object within the service's hierarchy and the object's ObjectID. Upon an MSI invocation, the service checks that the ancestor relationship holds and that a corresponding (`root node ID, password`) pair can be found in its CapTable. If not, the capability is invalid and the operation fails.

MSI provides functions for creating and revoking capabilities: `create_capa` and `revoke_capa`. When a user requests a capability from a service (using `create_capa`), the service returns a URL that embeds the new capability. In this way, capability sharing is similar to URL sharing in the Web. Revocation of a capability simply zeroes the rights fields in the capability's CapTable entry. To prevent arbitrary users from revoking capabilities, revocation requires a valid capability to the same object with the `REVOCATION` right enabled.

Several current Web services already use slight variations of a hybrid-capability protection model, which confirms the applicability of our approach. As one example, Flickr and other Yahoo! services provide "browser-based authentication [230]," which is essentially a capability-based scheme; it allows users to obtain a "token" for an object, specify a set of rights enabled by that token, and pass the token to an application. As another example, Google Calendar offers users "secret URLs" to their calendars, which they can give to friends. These URLs are a type of capability that can be used to view, but not modify, the user's calendar. To share a calendar with update rights, the user must add the sharee to the service's ACL.

5.3.3 The Object Content Access Interface

MSI provides composite Web applications with two different ways to access objects. First, for mashup-style applications, Menagerie permits a composite application to embed an object from a

remote service within a Web page. The remote service is responsible for the presentation and interaction controls of that embedded object, similar to how YouTube provides embeddable, interactive objects for displaying video.

To support building expressive composite GUIs, MSI defines a set of metadata access functions, including `get_summary` and `get_URL`. The `get_summary` function returns an HTML snippet that describes the object visually. For example, `get_summary` returns an `<img>` tag for a Flickr photo's thumbnail, an `<object>` tag for a YouTube video, and a summary for a Gmail email. The Menagerie Web Object Manager application in Section 5.4.1 uses this function to present distributed collections in a visual manner.

Similarly, `get_URL` provides the link to the object's URL within the parent service. Just as today's desktop file manager applications use a file-application binding database in systems like Windows to launch the appropriate application when the user double-clicks on a file, our Menagerie Web Object Manager uses URLs to redirect the user back to the parent service when a user clicks on a particular object.

Second, for composite applications that need to directly manipulate object contents, MSI provides a small, standard set of *object-independent* access functions. These functions, which include `read`, `write`, and `delete`, allow an application to download, manipulate, and upload the objects directly from Web services. MSI does not mandate any particular object representation or format: how a service chooses to externalize an object is entirely its own choice. Some composite applications, such as the archival service we described in Section 5.1.1, do not need to understand an object's format. Others, such as an indexing, image editing, or video distillation service, will need knowledge of the object's format. Over time, we expect services will gravitate towards standard object types and formats.

5.3.4 Implementation

Figure 5.6 shows the structure of our prototype Menagerie implementation. We chose to define MSI as an XML-RPC [225] layer on top of HTTP, so that services can make use of standard Web programming toolkits and frameworks to define, access, and export MSI functions. As well, by

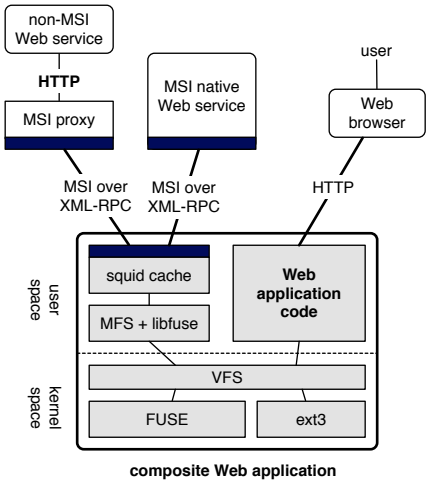


Figure 5.6: **Prototype implementation.** Our prototype system uses proxies to bridge legacy Web services to MSI. Composite Web applications can make use of MFS, which is implemented using the FUSE user-level file system framework. We have implemented MSI using XML-RPC, which is itself layered on HTTP.

using XML-RPC, we could take advantage of existing Web caching components (such as Squid) within our composite applications to improve their performance.

To experiment with composite Menagerie applications, we needed to access Web services that support MSI. As an incremental deployment strategy, we have built *MSI proxies* for existing (non-MSI) services. An MSI proxy implements the MSI functions and Menagerie protection model on behalf of a service, making it MSI compliant without needing to modify the service itself. To date, we have implemented proxies for five popular Web services: Gmail, Yahoo! Mail, Flickr, YouTube, and Google spreadsheets.

For services that provide developer APIs, we found it easy to implement proxies, as we could simply bridge between the services' REST or SOAP functions and our associated MSI functions. For services that do not provide developer APIs, building proxies was more challenging, as we had to use awkward and unstable Web scraping techniques to access the appropriate service functions and objects. Overall, however, proxies are a more secure and practical incremental deployment path than requiring each composite service to perform Web service scraping in its own way.

The Menagerie File System

The Menagerie File System (MFS) is a user-level file system based on FUSE [75] that simplifies building composite applications. MFS lets a composite application *mount the MSI name hierarchies* exported by Web services into its local file system. As a result, an application can access remote MSI objects using standard file system operations and user-level programs.

To mount a service hierarchy, the composite application must receive a capability for that hierarchy from the user and then provide the capability to MFS. Once mounted, the service can then use standard file system commands and tools, such as `cp` and `tar`, to operate on the objects. These tools issue system calls such as `getattr`, `readdir`, `read`, and `write`. The calls get passed via VFS [109] to MFS, and then translated into the corresponding MSI calls on the remote Web services. As well, metadata functions in MSI are exposed as extended file system attributes through MFS.

To boost MFS's performance, we provide composite services with two caches. MFS has an internal metadata cache for rapid retrieval of short-lived file system metadata, and it uses the Squid [69] cache to store data returned by MSI `read` and `get_summary` functions.

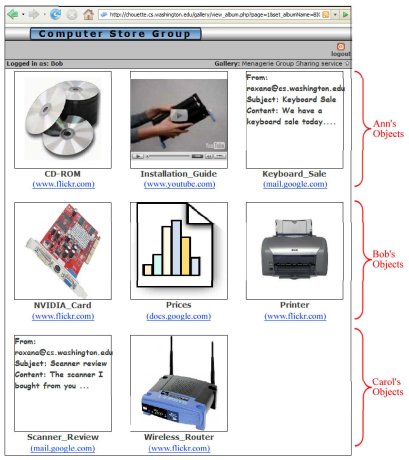
5.3.5 Summary

In this section, we described the architecture and implementation of our prototype Menagerie system. Through the use of Web service proxies and the MFS support layer, we made it possible for both existing and new Web services to communicate with each other through our Menagerie service interface. In the next section of the paper, we demonstrate the practicality and usefulness of our approach by building a set of powerful, easy to construct, composite Web applications.

5.4 Applications

As the trend towards Web-based applications continues, we believe that applications that support organizing, sharing, and manipulating distributed Web service objects will become increasingly important. This section presents several applications that we built using our Menagerie prototype. Our goal is to demonstrate the types of applications that Menagerie enables, and to show how Menagerie simplifies their implementation.

(a) Menagerie Web Object Manager (WOM)



(b) Menagerie Group Sharing Service (MGS)

Figure 5.7: **Screenshots of Two Menagerie-based Web Applications.** (a) This figure shows how Ann organizes her product-related Web objects using WOM. The right half shows the thumbnails of Ann's product photos on Flickr. The left half shows one of Ann's organizational folders, which already contains some objects. Ann is now dragging a product photo onto her new folder. (b) This figure shows how Ann and her colleagues Bob and Carol, all users of the MGS service and members of the `Computer Store` group, share objects with their group. Ann has shared some of a Flickr product photo, a YouTube video, and an email, Bob has added two Flickr photos and a Google spreadsheet of product prices, and Carol has put one Gmail email and one Flickr photo.

5.4.1 The Menagerie Web Object Manager

The Menagerie Web Object Manager (WOM) is a composite Web application that lets users organize and share their distributed Web objects. With WOM, users can create new virtual folders, populate those folders with collections of distributed Web objects, and share the folders with other users or services. WOM is a generic desktop, similar to file managers like Nautilus or Windows Explorer, but for Web objects. A WOM user can access and manipulate all of her Web service objects using the WOM Web interface; behind the scenes, WOM mounts and operates on the object hierarchies exported by the user's services.

The screenshot in Figure 5.7(a) shows how Ann organizes the Web resources for her business. When Ann first created her WOM environment, she mounted her Web service hierarchies (Flickr, Gmail, Google Docs, and YouTube) by pasting capabilities for those hierarchies into a Web form. WOM retains those capabilities and remounts the hierarchies using MFS whenever she logs in.

The WOM Web page is split in half. On the right, the user can navigate through her objects and mounted hierarchies. The expandable tree on top lists Ann's currently mounted hierarchies. In Figure 5.7(a), Ann has opened her Flickr `Product Photos` album.

The left side presents the user's virtual Web object folders. Users can create directory hierarchies and populate them by simply dragging-and-dropping objects from the right side of the interface to the left. In the figure, Ann has created a `Computer_Store` directory, containing sub-directories for `Computer_Accessories` and `Printers`. Ann is currently populating her `Computer_Accessories` folder; that directory includes two instructional YouTube videos, two customer emails from Gmail, and a folder with financial Google spreadsheets. The figure shows that Ann is dragging a Flickr product photo onto her `Computer_Accessories` directory.

WOM is only organizational; objects remain stored and managed by their respective Web services. Clicking on a object leads back to its origin Web service. For example, clicking on a Google spreadsheet in Ann's virtual folder causes Google Docs to popup a browser with that spreadsheet opened. Although the Web objects are only linked to the virtual folder, WOM can still render thumbnails of the objects. To retrieve the HTML code that displays the thumbnail for a specific object, WOM reads the object's `SUMMARY` extended attribute from MFS, which causes MFS to issue a `get_summary` call to the appropriate service.

Our WOM implementation exports MSI, which allows users to further export their new organizational structures. For example, a user can request a capability for a WOM virtual folder hierarchy and share that folder with other people and services by passing them that capability. Because WOM is a native MSI service, it requires no proxy.

WOM provides useful organization and sharing features, yet it was easy to build on top of our Menagerie prototype. One developer implemented WOM in roughly 3 days. The WOM codebase contains 275 lines of code: 131 lines of PHP code containing the application logic, and the remainder to perform HTML formatting.

5.4.2 The Menagerie Group Sharing Service

The Menagerie Group Sharing Service (MGS) is a Web application that lets users form groups and share collections of Web objects from their Web services. MGS is similar to MySpace, but it is targeted at groups rather than individuals. That is, while WOM lets a single user create and share virtual object organizations, MGS lets several users share a single virtual desktop.

We implemented MGS by modifying Gallery 1 [128], a popular Web-based photo sharing application. Hence, MGS borrows its GUI from Gallery. We enhanced Gallery to run on Menagerie, to display any type of resource (not just photos), and to support user groups. Figure 5.7(b) presents a screenshot of MGS, in which Ann, Bob, and Carol have created a group called `Computer Store Group` to share business information amongst themselves and with other colleagues. Ann has shared a photo, a video, and an email on the group page; she does not want to share her entire WOM `Computer_Store` directory with the colleagues because it contains confidential financial data. Bob has added two Flickr photos and a spreadsheet with product prices, and Carol has added an email and a photo. All resources are displayed in the group's Web page on MGS. Adding resources to the page is similar to adding resources in WOM; the user pastes a capability into a form to give MGS access to an object or hierarchy.

Modifying Gallery to build MGS took a single day for one developer. The conversion required only 73 new lines of code (32 related to HTML formatting), modification of 3 lines, and removal of 91 lines from Gallery.

5.4.3 MFS-based Applications

The WOM and MGS examples show how new Web-object management services can leverage the global naming, protection, and unified access functions that Menagerie provides. As well, the Menagerie File System lets any application treat Web objects as abstract files. As a result, services can apply *existing* file-based programs or scripting languages to remote Web objects or to the kinds of Web-object collections that Menagerie enables. Below, we give several examples of the power of this "backwards compatibility" provided by MFS.

Backup and Restore Service. Today's users have backup tools for safely archiving their desktop

data. However, for user data stored by Web services, users must trust the service to maintain their data, perhaps forever, as no generic Web object backup-and-restore application exists.

Using Menagerie, a backup-restore service that operates on distributed Web object collections can be built with a simple set of existing applications or commands, such as `tar` and `untar` in UNIX. For example, suppose that Ann wants to back up her *distributed* WOM `Computer_Store` folder. Ann provides the capability to that folder to the backup-restore service, which uses the capability to mount Ann's object hierarchy. To the service, Ann's distributed Web objects look like a local UNIX file tree. Therefore, the backup-restore service can archive Ann's objects with the following commands:

```
cd /mfs/Ann/WOM
tar -czf /backups/Ann/Computer_Store.tgz \
         Computer_Store
```

This creates a `tar` archive in the /backups folder on one of the backup-and-restore Web service's machines. Provided that all capabilities involved have the `READ` right enabled, the `tar` causes backup-restore's MFS to `read` the contents of each object recursively, first from WOM and then from the appropriate hosting service. The resulting archive will contain the entire `Computer_Accessories` folder hierarchy and the contents of all the distributed objects in it. Similarly, the service can use `untar` to restore those objects at a later time.

Changing email providers. Users may wish to migrate from one Internet mail system to another, or to consolidate multiple accounts. While some email services support interchange, this is not a general feature. Menagerie can simplify the task of email migration. For example, a new third-party Web application for migrating from one mail account (e.g., Yahoo!Mail) to another (e.g., Gmail) could be built on Menagerie through MFS using the following, perhaps surprisingly simple, command:

```
cp /mfs/Ann/Yahoo/*/*/msg  /mfs/Ann/Gmail
```

This command processes all of the folders and message directories in the user's Yahoo!Mail, copying each `msg`, which contains the contents of an individual email, to the Gmail account. The command assumes that the new changing-email-providers application has mounted Ann's Gmail and

Yahoo!Mail hierarchies. The result is to send each Yahoo message to the user's Gmail account, where it will appear in her Inbox folder.

This example needs further explanation. First, this email exchange is facilitated by the fact that our Menagerie proxies for Gmail and Yahoo!Mail implement a common XML-based schema for emails. If the services did not export the same email format, the new application would need to perform a schema mapping for each email. Second, the copy command does not recreate the same folder structure; a simple loop that first creates the folders (labels) easily solves this problem. Finally, our implementation places attachments and message content in separate files, which makes copying an email with attachments more difficult; a 10-line script (omitted here) deals with this by combining the attachment and message into a single file before copying it to Gmail.

While the full explanation of this process is more complex than the single-line `cp` command above, the example shows the power of providing UNIX file access to object hierarchies.

Synchronizing email contacts. Although some email services let users import contacts from other services, they do it in an ad-hoc manner in which each Web service knows how to fetch contacts only from the most popular other services.

With Menagerie, multi-email contact synchronization is easier because the distribution is transparent. In particular, the application need only understand contact formats and how to unify them. Since our proxies for Yahoo!Mail and Gmail export the same contact formats, as noted above, we can leverage existing file synchronization tools such as Unison. For example, a Web application for synchronizing the contacts between Gmail and Yahoo!Mail accounts can be done as follows:

```
cp /mfs/Ann/Yahoo/contacts/* /tmp/Y
cp /mfs/Ann/Gmail/contacts/* /tmp/G
unison /tmp/Y /tmp/G
cp /tmp/Y/* /mfs/Ann/Yahoo/contacts
cp /tmp/G/* /mfs/Ann/Gmail/contacts
```

In this example, the new contact synchronizer application copies the user's contacts into a local temporary file prior to running Unison because Unison creates its own temporary files in the directories it synchronizes. In Menagerie, executing Unison directly on the Web service files would result in the creation and then removal of new contacts on the Web service. To avoid this overhead, we first

download the contacts locally, run Unison on them, and then upload the unified contact set. Note that we rely on the user to resolve conflicts, since neither Gmail nor Yahoo!Mail reports the time of the last contact modification.

The MFS Desktop Bridge. MFS was designed to simplify building composite Web services, but it is also valuable as a desktop operating system component. By running MFS on a desktop, the user can mount and access her Web objects as files within the file system. As a result, the user can take advantage of desktop applications to operate on Web data: MFS acts as a bridge between the user's desktop and Web environments. For example, using MFS, we have used Adobe Photoshop to edit Flickr photos, Microsoft Excel to operate on a spreadsheet stored within Google Docs, and Nautilus to navigate through Web objects, all without changing the applications themselves.

5.4.4 Summary

In this section we presented example applications built using Menagerie. We showed how Menagerie lets services access Web objects through existing desktop applications and command languages. Our examples are not meant to be complete, but instead to stimulate the imagination of what is possible given the features that Menagerie provides. Overall, our examples demonstrate two key points. First, a common set of naming, protection, and access operations for Web services greatly simplifies the creation of new organization and sharing services for heterogeneous Web objects. Second, a file-access facility for Web objects provides a powerful path to leverage legacy command languages and applications in the new world of software as a service.

5.5 Evaluation

In this section we evaluate Menagerie, focusing on three questions. First, what additional latency does the Menagerie layer add to Web service data accesses, compared to direct access without Menagerie? Second, which internal components of Menagerie are most responsible for overhead? Third, how does the performance of the MFS Desktop Bridge compare to other methods for editing personal data?

We have not spent effort to optimize or tune Menagerie; rather, our goal was to build a straightforward and extensible framework for experimentation. Nonetheless, our results demonstrate that

Service	Oper.	Menagerie (ms)	Total (ms)	Menagerie percent
Gmail	ls	37	250	14.0%
	read	128	1,549	8.2%
Ymail	ls	35	955	3.6%
	read	121	3,943	3.0%
Flickr	ls	35	364	9.6%
	read	74	1,624	4.5%
GDocs	ls	41	348	11.7%
	read	122	3,194	3.8%

Table 5.1: **Menagerie's Latency.** Latency compared to total latency for directory listing (ls) and remote data read (rd) on several services. Menagerie is a small fraction of the total latency for existing Web services.

performance of our current prototype is competitive with other remote access Web technologies and is fast enough to be usable in practice.

For our measurements we created a Menagerie measurement service that ran Menagerie (including MFS, FUSE, and the Squid cache) and the measurement applications on an Intel P4 3.2 GHz CPU with 2GB of memory. We ran the MSI proxies for existing services on a separate machine with a similar configuration. Both machines ran Fedora Core 5, Squid 2.6, and Firefox 1.5. The two machines were connected via a 100Mbps switch.

5.5.1 Menagerie Overhead

For our first question – the additional cost of Menagerie in accessing Web service data – we measured the latency for two simple operations performed on Web services through Menagerie. Table 5.1 shows the latency for a directory listing and a remote data read invoked through Menagerie/MFS to Gmail, Yahoo Mail, Flickr, and Google Docs. The data is read by an application performing a *cat* of a 4.7MB file. The table shows that Menagerie represents only a small fraction of the total latency (less than 15%) for these operations. Not surprisingly, network latency and service time dominate.

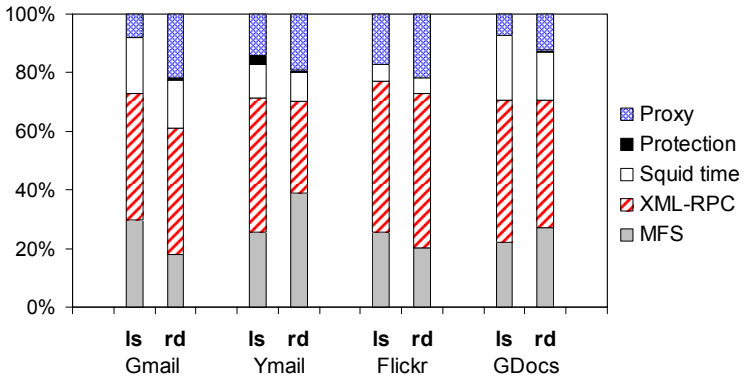


Figure 5.8: **Breakdown of Latency by Menagerie component for Directory and Read Operations on Four Web Services.** The Python-based XML-RPC library dominates the Menagerie latency; MFS is the next largest factor.

For example, the Flickr directory listing takes 364 ms to complete, of which 35 ms (9.6%) are spent in Menagerie components (MFS, MSI, and the proxy).

To answer our second question – where the time goes inside of Menagerie – we exclude the network and Web service times and account for the time spent in the various Menagerie components. For this measurement, we logged messages at key places, such as just before MFS issues an XML-RPC request to a proxy, or when the corresponding RPC function is called in the proxy, and computed the time spent in different components by subtracting the timestamps of the appropriate messages.

Factored into the Menagerie latency is the time spent in five of its components: (1) the Menagerie File System (MFS), which has both user-mode and kernel-mode components, (2) XML-RPC, (3) the Squid cache, (4) the Menagerie protection manager, including capability validation and credential translation, and (5) the Web service MSI proxy, which includes parsing and building requests to the existing Web services.

Figure 5.8 breaks down the latency for these five components. In most cases, the dominating latency is caused by the Python-based XML-RPC, which represents about half of the total latency on average. The time spent in MFS represents from 20% to 38%, due primarily to our user-level file system code; this could be reduced in an all-kernel-level implementation. The use of Web-

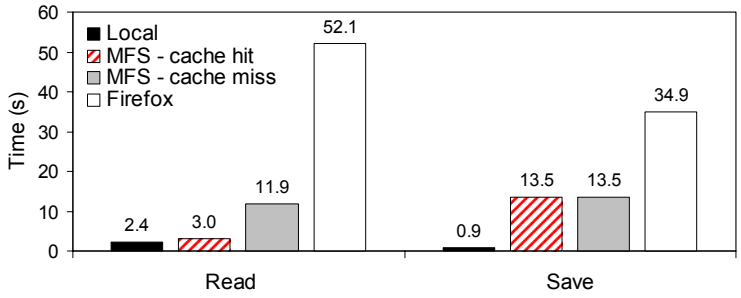


Figure 5.9: **Performance Comparison of Four Spreadsheet-handling Scenarios.** A comparison of opening and saving a spreadsheet in four cases: OpenOffice access to a local spreadsheet; OpenOffice access to a remote Google spreadsheet via the MFS bridge, shown for both a cache hit and cache miss; and Firefox browser access to a remote Google spreadsheet.

service proxies has a smaller impact on total latency, on average about 15.2%. The cost of the protection system is negligible in all cases. Overall, then, the greatest potential for improvement lies in the XML-RPC system. Given the small cost of Menagerie compared to network latency and Web service time, however, it is not clear that such optimization is warranted.

5.5.2 MFS Desktop Bridge Performance

Consider the task of opening, modifying, and saving a spreadsheet. Traditionally, users invoked desktop applications such as OpenOffice to perform this task. With the advent of rich Ajax-based interfaces for online document editing, such as Google Spreadsheets, users can now perform the same task via their browsers. The Menagerie Desktop Bridge presents a third alternative; for example, users can operate on a remote Google spreadsheet using local PC-based spreadsheet applications.

We compare these three scenarios in Figure 5.9, which shows the performance seen by the user when opening an identical spreadsheet, modifying 2 cells, and saving the file. For the MFS bridge scenario, we report the performance for two cases: OpenOffice hitting in the MFS squid cache, and missing in the cache. We used Firefox to access a remote Google spreadsheet in our third scenario.

For cache hits, the MFS solution performs nearly as well as accessing a local file. Surprisingly, saving the spreadsheet with MFS is 2.3 times faster than saving it through the browser. The slow speed of the browser solution is due mostly to the Ajax application and its required rendering and

server communication in dealing with our 200KB spreadsheet. This may be addressed in the future with more optimized Ajax engines.

Overall, Menagerie supports new functions on Web objects, as witnessed by applications in Section 5.4, and it also enables the use of existing applications to handle Web objects in a performance-competitive way.

5.6 Related Work

Menagerie builds upon many earlier efforts in Web technologies, protection system, and extensibility. The Semantic Web [19] effort, languages for describing Web service interfaces [44], and service communication protocols [68, 215, 225] enable applications to find and integrate Web service content. In this work, we identify the key components that any Web service interface must provide in order to enable a particular set of applications: generic applications for organizing, sharing, and processing Web data objects within user Web accounts.

Recently, the problems caused by the dispersal of users' data on the Web have received increasing attention from Web service providers. Web-data aggregation sites (e.g., iGoogle [85], Facebook [66], SecondBrain [179]), and Web-data processing applications [155] allow users to aggregate their Web objects from a set of supported locations, share them, or process them. Each of these applications must face the challenges of Web-account data integration (Section 5.2) on its own: it needs to devise its own naming for Web objects, often request full control from the user on his remote Web accounts, and write code to retrieve data from each service. Solving such challenges for each application is inefficient. Thus, we propose a new common service interface, which, if adopted, would facilitate the building of applications, including some of the ones enumerated above [155, 179].

Many individual Web services expose programmatic interfaces. Some social applications have agreed to support OpenSocial, a common set of JavaScript and Google Data APIs for accessing social information [86]. Menagerie and OpenSocial have very similar goals. However, Menagerie is more general, as it is not restricted to social applications, while OpenSocial's API has the benefit of being tuned towards the needs of social application programmers. This tradeoff between generality and specificy is common to many systems [65].

The need to decouple user-account data from Web services and expose it to third-party applications has been recently formulated in the W5 project [112]. While some of their concepts overlap with Menagerie's, including fine-grained protection and data access, our contribution consists of a concrete instantiation of the required common interface, a working implementation, and experience with building useful applications to validate our approach.

The idea of using operating system concepts and abstractions to address problems on the Web has been used previously. WebOS [209] provides OS abstractions for building large-scale applications over the wide-area, including global naming and authentication. Menagerie provides functions typically fulfilled by the OS on the desktop to Web applications operating on the user's Web data. Similarly, Web file systems [5, 210, 222] enable the integration of Web resources with the local file system. Unlike Menagerie, these systems do not offer any support for sharing heterogeneous collections of objects. Specific Web services provide file system interfaces that let users access their Web objects and run desktop applications on them [100, 101]. None of these supports the integration of resources from multiple Web services or the sharing of heterogeneous Web objects. Yahoo! pipes [229] allows users to integrate RSS feeds and mashup Web site data using a visual, UNIX pipe-like editor. Unlike Menagerie, Yahoo! pipes does not facilitate the fine-grained, protected sharing of personal Web objects.

Capability-based protection [116] has been used in many operating systems and distributed systems [36, 183, 202, 227]. Our hybrid capability mechanism resembles the authorized/unauthorized pointer model first used in the IBM System/38 [22], which merges capabilities with ACL-based authentication. Menagerie capabilities give Web services the choice of automatically authenticated access via capabilities or controlled access that combines capabilities and user authentication.

Single sign-on systems have been proposed to allow users to login to many services with a single account [131]. While single sign-on simplifies user account management, it does not address fine-grained sharing and support for heterogeneous collections of Web-service objects.

Some projects have looked at improving the security of mashups within the current browsers [216, 99]. Most of these provide protection mechanisms for sharing of resources within the browser, while Menagerie's protection mechanism provides controlled sharing of objects within a Web service with a third-party application.

While Menagerie is closely related to these previous systems, it is unique in its integration of:

(1) global naming and fine-grained protection for user-personal Web service objects, (2) transparent access to those objects using standard applications, and (3) extended functions supporting needed Web operations, such as embedded rendering.

5.7 Summary

The move from desktop-centric to Web-based computing and data storage poses new challenges for users and applications. This paper described the organizational, sharing, and data-processing problems faced by users and creators of modern Web services. We presented Menagerie, a software framework that supports uniform naming, protection, and access for *personal objects* stored by Web services. We designed and implemented a Menagerie prototype and integrated a set of existing Web services: Gmail, Google Docs, Flickr, YouTube and Yahoo!Mail. Using Menagerie, we built organization and sharing services for personal objects, including the Menagerie Web Object Manager and the Menagerie Group Sharing Service. Our experience with Menagerie and its applications underscores the power of this approach and its potential for enabling and simplifying the construction of new composite Web services. Our measurements show that a Menagerie-like service interface can provide performance commensurate with existing Web-object access techniques.

Chapter 6

FUTURE DIRECTIONS

We now describe several interesting future research directions in increasing users' control over data privacy, ownership, and management properties. As shown in Section 1.2.3, modern technologies cause many forms of data control loss, of which this dissertation has only dealt with four challenges. Recovering each of the other properties included in Table 1.1, along with other potential challenges, represent an interesting research direction. Other promising directions include:

The Mobile Device Operating System. The operating system on today's mobile devices requires significant re-design for security. While many mobile device providers and researchers are striving to create more secure high-level application frameworks, the underlying software stack has remained largely the same as for traditional desktop computers. For example, laptops run unmodified desktop operating systems and applications, and phones run stripped-down versions of these systems (e.g., Android OS is largely Linux and its browser is based on Chrome). Unfortunately, today's operating systems and applications were not designed to cope with the pervasive threat of theft that mobile devices face. Current mobile operating systems leave data residues everywhere, exposing sensitive data to thieves. For example, the operating system maintains gigabytes of cached data in unencrypted memory, the file system accumulates historical information on disks by not overwriting deleted blocks, and applications mismanage their sensitive data such as passwords and cookies by storing them on disk or in swappable memory. As noted earlier, encrypted file systems help but are not enough.

We believe that new operating system abstractions are required to cope with the serious threat of theft on today's mobile devices. These abstractions should allow applications to *manage sensitive data rigorously and maintain a clean environment at all times in the expectation of device theft*. While a list of required abstractions is open, some immediate ones include: encrypted and audited memory, assured-delete and self-destructing files, and secure audit logs of what code/applications/extensions have been installed. These are challenging abstractions to support and evaluate. What

does memory auditing mean and what would it cost? How do we measure the security of our new OS and what are appropriate user and theft workloads for mobile devices? Addressing these and further questions are exciting avenues for future research.

Extensible Cloud Storage. Comet (Chapter 4) introduced extensibility in a particular type of storage cloud: peer-to-peer DHTs. The concept of extensible clouds is, however, applicable to – and perhaps even more urgent in – the broad spectrum of Web services. As outline in Chapter 1, the inflexibility and lack of control over how the data is managed in today's clouds, such as Amazon S3, Facebook, Google Docs, and Hotmail, causes data security, privacy, and management challenges for the users and impedes innovation. For example, a user cannot prohibit a sensitive document on Google Docs from being stored on servers outside the US, request a log of all accesses to her profile or photos on Facebook, restrict accesses to data to people from the US or people with similar professional qualifications.

To empower users with control over their data, Web services should add such levels of customization by *exposing extensible programmatic interfaces to data management functions*. For example, a service should allow the insertion of extensions that limit the placement of data replicas in the world, provide hints on where and how many replicas to store for improved availability, impose the logging of all accesses to sensitive data, and constrain their access controls in user-defined ways. The design of an extensible Web service is likely to be informed by, though very different from, Comet. Specifically, Comet's lightweight extension model, coupled with its security-driven tight sandbox limits, imposes severe limitations on extensibility. For example, Comet allows only very lightweight handler computation and disables all network accesses. Such restrictions are reasonable for a system designed to run on limited-resource user machines, but may be unnecessarily restrictive for large server farms. Exploring the extensibility interfaces, flexible and dynamic sandboxing mechanisms, and extension interaction resolution are interesting avenues for future research.

The Cloud Operating System. Web services are transitioning from privately managed data centers to public clouds, such as Amazon's AWS, Google's AppEngine, and Microsoft's Azure. While many researchers are looking into the security, privacy, and isolation challenges raised by this transition, an interesting additional avenue for future research is its untapped opportunities. Public clouds are unique environments where many of the world's Web services share the same computing, storage,

and networking infrastructure. In fact, the entire Web might eventually be served from a small number of such clouds. This giant-scale integration offers myriad opportunities to solve some of the most complex challenges in today's Web. In the future, it may be possible to identify these opportunities and equip public clouds with the appropriate mechanisms to take full advantage of them. Following are two initial examples of untapped opportunities.

First, the public cloud could provide a set of common data controls and mechanisms that all resident Web services would inherit, thereby addressing the current Web's data scattering problem. As noted previously, Menagerie and Vanish constructed common controls on data protection, access, and deletion atop disparate Web services. These goals are important yet difficult to achieve in today's Web, which lacks a common infrastructure. Public clouds could provide these and other unified data controls "for free" as functions of a *cloud operating system*. Such a cloud OS could offer a variety of unified, Web-wide data controls, including assured data deletion, uniform data protection, full data-access logs, and the ability to control the data's geographical location. The critical design questions for the cloud OS are: (1) what controls should the cloud OS provide, (2) how can users trust it to enforce them, and (3) how can the OS support users' future, unpredictable data control needs?

Second, the cloud could simplify the Webs programming model dramatically. Building scalable Web services is hard today, because it requires each service to implement its own software stack; e.g., Flickr builds its own storage, protection, and photo indexing infrastructures. In public clouds, however, millions of Web services are connected via high-bandwidth intra-datacenter networks. Imagine a cloud OS that supports efficient, scalable, fine-grained, and secure compositions between mutually distrustful services, allowing new services to be easily built atop one another. Then Flickr might be constructed by invoking components from Photobucket, Picasa, or other co-located services, for example. Just like Map/Reduce revolutionized the large-scale batch data processing, new composition-based programming models in public clouds could revolutionize the construction of scalable Web services. Naturally, open research questions exist: (1) what should the Webs programming model be in public clouds, (2) what composition mechanisms and abstractions should the cloud OS offer, and (3) how can the OS guarantee isolation between mutually distrustful Web services that build upon one another?

Chapter 7

CONCLUSIONS

Computing technology is undergoing important transitions and the changes are creating serious data security, privacy, and management challenges. In the past, users stored and processed their data on one single machine – the desktop computer – that the user fully trusted and controlled. With the advent of mobile devices and cloud computing, this simple world is transforming into a more complex world where the data is scattered across many services and devices that the user neither trusts nor controls. This transition makes it challenging for users to organize their data meaningfully, to ensure various properties of their data, such as data lifetime or replication factors, and to preserve their data's privacy in face of malicious Web services and mobile device thieves.

This dissertation proposed a set of novel techniques to address specific data security, privacy, and management challenges raised by the adoption of new cloud and mobile technologies. The overarching goal of these techniques was to increase users' control over various aspects of their data in the cloud and on mobile devices. Keypad provides remote access auditing and control over data stored on stolen mobile devices; Vanish offers data lifetime control on the Web; Comet allows users to customize various data management properties in a storage cloud; and Menagerie allows users to regain a unified organizational view over their scattered Web data. These systems show that, with carefully crafted abstractions and mechanisms, users can regain control over various aspects of their data without losing the new technologies' advantages.

An approach common to most of this work is the introduction of new *self-managing data abstractions*, in which the control properties are built into the data object itself. In Vanish, we defined a *self-destructing data object* abstraction, which disappears on its own after a pre-specified time. In Comet, we proposed an active storage object abstraction, which controls the way the data is managed in the storage cloud. And in Keypad, we effectively defined an audited data abstraction, which weaves fine-grained data access auditing into the files using encryption and remote key management. Our experience shows that self-managing data abstractions are an effective approach for dealing with uncontrolled mobile and cloud environments.

BIBLIOGRAPHY

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evalaution. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[2] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[3] F. Ahmed and M. Y. Siyal. A novel approach for regenerating a private key using password, fingerprint and smart card. *Information Management and Computer Security*, 13(1), 2005.

[4] C. Alexander and I. Goldberg. Improved user authentication in off-the-record messaging. In *Proceedings of the ACM Workshop on Privacy in an Electronic Society (WEPS)*, 2007.

[5] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. UFO: A personal global file system based on user-level extensions to the operating system. *ACM Transactions on Computer Systems (TOCS)*, 16(3), 1998.

[6] Amazon S3. `http://aws.amazon.com/s3/`.

[7] R. Anderson and M. Kuhn. Tamper resistance: A cautionary note. In *Proceedings of the USENIX Workshop on Electronics Commerce*, 1996.

[8] R. Anderson and M. Kuhn. Low cost attacks on tamper resistant devices. In *Proceedings of the International Workshop on Security Protocols*, 1997.

[9] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet impasse through virtualization. *IEEE Computer*, 38(4), 2005.

[10] Apache Cassandra. `http://cassandra.apache.org/`.

[11] Apple MobileMe. Find your iPhone or iPad. `http://www.apple.com/mobileme/features/find-my-iphone.html`.

[12] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. *RFC 4033: DNS Security Introduction and Requirements*. The Internet Society, 2005.

[13] M. Arrington. Google App Engine goes down and stays down. `http://techcrunch.com/2008/06/17/google-app-engine-goes-down-and-stays-down/`, 2008.

[14] B. Awerbuch and C. Scheideler. Towards a Scalable and Robust DHT. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, 2006.

[15] R. A. Bazzi and G. Konjevod. On the Establishment of Distinct Identities in Overlay Networks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2005.

[16] M. Bellare and A. Palacio. Protecting against key exposure: Strongly key-insulated encryption with optimal threshold. *Applicable Algebra in Engineering, Communication and Computing*, (200), 2006.

[17] M. Bellare and B. Yee. Forward security in private key cryptography. In *Proceedings of the Cryptographers' Track at the RSA Conference (CT-RSA)*, 2003.

[18] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP – a system for secure multi-party computation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2008.

[19] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. Scientific American, 2001.

[20] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensible, safety and performance in the SPIN operating system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[21] B. N. Bershad and C. B. Pinkerton. Watchdogs – extending the UNIX file system. *Computer Systems*, 1988.

[22] V. Berstis. Security and protection in the IBM System/38. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1980.

[23] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM)*, 2004.

[24] M. Blaze. A cryptographic file system for UNIX. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 1993.

[25] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1982.

[26] D. Boneh and M. K. Franklin. Identity-based encryption from the Weil pairing. In *Proceedings of the Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 2001.

[27] D. Boneh and R. Lipton. A revocable backup system. In *Proceedings of the USENIX Security Symposium*, 1996.

[28] N. Borisov. Computational puzzles as Sybil defenses. In *Proceedings of the IEEE International Conference on Peer-to-Peer Computing (P2P)*, 2006.

[29] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use PGP. In *Proceedings of the ACM Workshop on Privacy in an Electronic Society (WEPS)*, 2004.

[30] K. Bowers, A. Juels, and A. Oprea. HAIL: A high-availability and integrity layer for cloud storage. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2009.

[31] K. Bowers, M. van Dijk, A. Juels, A. Oprea, and R. Rivest. How to tell if your cloud files are vulnerable to drive crashes. IACR Cryptology ePrint Archive, 2010.

[32] R. Canetti, Y. Dodis, S. Halevi, E. Kushilevitz, and A. Sahai. Exposure-resilient functions and all-or-nothing transforms. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, 2000.

[33] R. Canetti, C. Dwork, M. Naor, and R. Ostrovsky. Deniable encryption. In *Proceedings of the Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 1997.

[34] R. Canetti, S. Halevi, and J. Katz. A forward-secure public-key encryption scheme. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, 2003.

[35] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[36] J. Chase, H. Levy, M. Feeley, and E. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems (TOCS)*, 12(4), 1994.

[37] B. Chen and V. Chandran. Biometric based cryptographic key generation from faces. In *Proceedings of the Biennial Conference of the Australian Pattern Recognition Society on Digital Image Computing Techniques and Applications*, 2007.

[38] J. Cheng. Are deleted photos really gone from Facebook? Not always. `http://arstechnica.com/web/news/2009/07/are-those-photos-\really-deleted-from-facebook-think-twice.ars`, 2009.

[39] D. R. Choffnes and F. E. Bustamante. Taming the Torrent: A practical approach to reducing cross-ISP traffic in P2P systems. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM)*, 2008.

[40] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. Technical report, Technion, 1997.

[41] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1995.

[42] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proceedings of the USENIX Security Symposium*, 2005.

[43] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina. Controlling data in the cloud: Outsourcing computation without outsourcing control. In *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*, 2009.

[44] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web service definition language (WSDL). W3C, 2001.

[45] T. Condie, V. Kacholia, S. Sankararaman, J. M. Hellerstein, and P. Maniatis. Induced churn as shelter from routing table poisoning. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2006.

[46] F. Cornelli, E. Damiani, and S. Samarati. Implementing a reputation-aware Gnutella servent. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[47] M. D. Corner and B. D. Noble. Zero-interaction authentication. In *Proceedings of the ACM Annual International Conference on Mobile Computing and Networking*, 2002.

[48] M. D. Corner and B. D. Noble. Protecting applications with transient authentication. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2003.

[49] A. Czeskis, D. J. S. Hilaire, K. Koscher, S. D. Gribble, T. Kohno, and B. Schneier. Defeating encrypted and deniable file systems: TrueCrypt v5.1a and the case of the tattling OS and applications. In *Proceedings of the USENIX Workshop on Hot Topics in Security (HotSec)*, 2008.

[50] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM)*, 2004.

[51] F. Dabek, M. F. Kaashoek, D. Karger, R. MOrris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[52] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a dht for low latency and high throughput. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[53] G. Danezis, C. Lesniewski-laas, M. F. Kaashoek, and R. Anderson. Sybil-resistant dht routing. In *Proceedings of the European Symposium on Research in Computer Science (ESORICS)*, 2005.

[54] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[55] J. Dinger and H. Hartenstein. Defending the Sybil Attack in P2P Networks: Taxonomy, Challenges, and a Proposal for Self-Registration. In *Proceedings of the International Conference. on Availability, Reliability and Security*, 2006.

[56] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the USENIX Security Symposium*, 2004.

[57] Y. Dodis, M. K. Franklin, J. Katz, A. Miyaji, and M. Yung. Intrusion-resilient public-key encryption. In *Proceedings of the Cryptographers' Track at the RSA Conference (CT-RSA) 2003*, 2003.

[58] Y. Dodis, M. K. Franklin, J. Katz, A. Miyaji, and M. Yung. A generic construction for intrusion-resilient public-key encryption. In T. Okamoto, editor, *Proceedings of the Cryptographers' Track at the RSA Conference (CT-RSA) 2004*, 2004.

[59] Y. Dodis, J. Katz, S. Xu, and M. Yung. Key-insulated public key cryptosystems. In *Advances in Cryptology – EUROCRYPT 2002*, 2002.

[60] Y. Dodis, A. Sahai, and A. Smith. On perfect and adaptive security in exposure-resilient cryptography. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, 2001.

[61] Y. Dodis and M. Yung. Exposure-resilience for free: The case of hierarchical ID-based encryption. In *Proceedings of the IEEE International Security in Storage Workshop (SISW)*, 2002.

[62] J. R. Douceur. The Sybil attack. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[63] EMC, Inc. Emc unveils software-as-a-service strategy and its first enterprise offering, mozyenterprise. `http://www.emc.com/about/news/press/2008/012208.htm`, 2008.

[64] EncFS. http://www.arg0.net/encfs.

[65] D. R. Engler and M. F. Kaashoek. Exterminate all operating system abstractions. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 1995.

[66] Facebook. `http://www.facebook.com/`, 2007.

[67] J. Falkner, M. Piatek, J. John, A. Krishnamurthy, and T. Anderson. Profiling a million user DHT. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2007.

[68] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[69] N. L. for Applied Network Research. The Squid Internet Object Cache. `http://squid.nlanr.net`.

[70] T. Foundation. Truecrypt – free open-source on-the-fly encryption. `http://www.truecrypt.org/`, 2007.

[71] M. Freedman, E. Freudenthal, and D. Mazieres. Democratizing content publication with Coral. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[72] M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-transitive connectivity and DHTs. In *Proceedings of the Conference on Real, Large Distributed Systens (WORLDS)*, 2005.

[73] M. J. Freedman and R. M. Tarzan. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2002.

[74] K. Fu, M. F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems (TOCS)*, 20(1), 2002.

[75] FUSE: Filesystem in Userspace. `http://fuse.sourceforge.net/`.

[76] P. Gardner. personal communication, 2009.

[77] R. Geambasu, C. Cheung, A. Moshchuk, S. D. Gribble, and H. M. Levy. The organization and sharing of web-service objects with menagerie. In *Proceedings of the International World Wide Web Conference (WWW)*, 2008.

[78] R. Geambasu, S. D. Gribble, and H. M. Levy. Cloudviews: Communal data sharing in public clouds. In *Proceedings of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[79] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy. Keypad: An auditing file system for theft-prone devices. Proceedings of the ACM European Conference on Computer Systems (Eurosys), 2011.

[80] R. Geambasu, T. Kohno, A. Krishnamurthy, A. Levy, H. M. Levy, P. Gardner, and V. Moscaritolo. New directions for self-destructing data systems. Technical Report UW-CSE-11-08-01, University of Washington, 2010.

[81] R. Geambasu, T. Kohno, A. Levy, and H. M. Levy. Vanish: Increasing data privacy with self-destructing data. In *Proceedings of the USENIX Security Symposium*, 2009.

[82] R. Geambasu, A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy. Comet: An active distributed key/value store. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[83] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 2009.

[84] D. Goodin. Your personal data just got permanently cached at the US border. `http://www.theregister.co.uk/2008/05/01/electronic_searches_at_us_borders/`, 2008.

[85] Google, Inc. iGoogle. `http://google.com/ig`, 2005.

[86] Google, Inc. OpenSocial. `http://code.google.com/apis/opensocial/`, 2007.

[87] J. Gordon. Reliability and the cloud – redundancy required. `http://notes.kateva.org/2011/05/reliability-and-cloud-redundancy.html`, 2011.

[88] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[89] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the USENIX Security Symposium*, 1996.

[90] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proceedings of the USENIX Security Symposium*, 2008.

[91] F. Hao, R. Anderson, and J. Daugman. Combining crypto with biometrics effectively. *IEEE Transactions on Computers*, 2006.

[92] C. Harvesf and D. M. Blough. The Effect of Replica Placement on Routing Robustness in Distributed Hash Tables. In *Proceedings of the IEEE International Conference on Peer-to-Peer Computing (P2P)*, 2006.

[93] K. Hildrum and J. Kubiatowicz. Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks. In *Proceedings of the International Symposium on Distributed Computing*, 2004.

[94] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 1988.

[95] IETF. RFC1991 - PGP Message Exchange Formats. `http://www.ietf.org/rfc/rfc1991.txt`, 1996.

[96] Imperva. Consumer password worst practices. `http://www.imperva.com/docs/WP_Consumer_Password_Worst_Practices.pdf`, 2010.

[97] Intel Corporation. Protect laptops and data with Intel Anti-Theft technology. `http://antitheft.intel.com/Libraries/Documents/Intel_anti-theft_techbrief_final.sflb.ashx`, 2011.

[98] T. Isdal, M. Piatek, A. Krishnamurthy, and T. Anderson. Privacy-preserving P2P data sharing with OneSwarm. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM)*, 2010.

[99] C. Jackson and H. Wang. Subspace: Secure cross-domain communication for Web mashups. In *Proceedings of the International World Wide Web Conference (WWW)*, 2007.

[100] M. R. Jain. FlickrFS. `http://manishrjain.googlepages.com/flickrfs`, 2005.

[101] R. Jones. GmailFS. `http://richard.jones.name/google-hacks/gmail-filesystem/gmail-filesystem.html`, 2004.

[102] R. Joyce and G. Gupta. Identity authorisation based on keystroke latencies. *Communications of the ACM*, 33(2), 1990.

[103] J. K. Juang. Practical implementation and analysis of hyper-encryption. Master's thesis, MIT, 2009.

[104] A. Juels and B. Kaliski. PORs: Proofs of retrievability for large files. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2007.

[105] Jungle Tools LLC. JungleDisk – reliable online storage on Amazon S3. `http://www.jungledisk.com/`, 2007.

[106] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, , and K. Mackenzie. Application performance and flexibility in exokernel systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[107] K. Keetong, D. Patterson, and J. Hellerstein. A case for intelligent disks (IDISKs). *ACM SIGMOD Record*, 27(3), 1998.

[108] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1991.

[109] S. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the Summer USENIX Conference*, 1986.

[110] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1999.

[111] F. Krautheim. Private virtual infrastructure for cloud computing. In *Proceedings of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[112] M. Krohn, A. Yip, M. Brodsky, R. Morris, and M. Walfish. A world wide web without walls. In *Proceedings of the Workshop on Hot Topics in Networks (HotNets)*, 2007.

[113] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[114] C. Lesniewski-Lass and M. F. Kaashoek. Whanaungatanga: Sybil-proof distributed hash table. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[115] B. N. Levine, C. Shields, and N. B. Margolin. A survey of solutions to the Sybil attack. Technical Report 2006-052, University of Massachusetts Amherst, 2006.

[116] H. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[117] J. Lowensohn. Google Docs goes down, user data does not. http://news.cnet.com/8301-17939_109-9985608-2.html, 2008.

[118] N. A. Lynch, D. Malkhi, and D. Ratajczak. Atomic data access in distributed hash tables. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2001.

[119] P. MacKenzie and M. K. Reiter. Delegation of cryptographic servers for capture-resilient devices. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2003.

[120] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[121] O. Malik. S3 outage highlights fragility of Web services. http://gigaom.com/2008/07/20/amazon-s3-outage-july-2008/, 2008.

[122] V. Mayer-Schönberger. *Delete: The Virtue of Forgetting in the Digital Age*. Princeton University Press, 2009.

[123] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[124] D. Mazieres, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1999.

[125] D. McCoy, K. Bauer, D. Grunwald, T. Kohno, and D. Sicker. Shining light in dark places: Understanding the Tor network. In *Proceedings of the Privacy Enhancing Technologies Symposium*, 2008.

[126] D. McCullagh. Security guide to customs-proofing your laptop. http://www.news.com/8301-13578_3-9892897-38.html, 2008.

[127] D. McCullagh. Dropbox confirms security glitch – no password required. http://news.cnet.com/8301-31921_3-20072755-281/dropbox-confirms-\security-glitch-no-password-required/, 2011.

[128] B. Mediratta. Gallery: Your photos on Your Website. http://gallery.menalto.com/, 2007.

[129] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[130] Microsoft BitLocker. Windows 7 BitLocker Executive Overview. http://technet.microsoft.com/en-us/library/dd548341%28WS.10%29.aspx, 2009.

[131] Microsoft Corporation. Microsoft Passport. http://www.passport.com/, 2007.

[132] F. Monrose, M. Reiter, L. Qi, and S. Wetzel. Cryptographic key generation from voice. In *Proceedings of the USENIX Security Symposium*, 2001.

[133] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(2), 1998.

[134] L. Mummert, M. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[135] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[136] A. Muthitacharoen, S. Gilbert, and R. Morris. Etna: A fault-tolerant algorithm for atomic mutable DHT data. Technical Report MIT-LCS-TR-993, MIT, 2005.

[137] Mysql Database Triggers. http://dev.mysql.com/doc/refman/5.0/en/triggers.html.

[138] S. K. Nair, M. T. Dashti, B. Crispo, and A. S. Tanenbaum. A hybrid PKI-IBC based Ephemerizer system. In *Proceedings of the International Information Security Conference*, 2007.

[139] E. Nakashima. Clarity sought on electronic searches. http://www.washingtonpost.com/wp-dyn/content/article/2008/02/06/AR2008020604763.html, 2008.

[140] New York Times. F.B.I. Gained Unauthorized Access to E-Mail. http://www.nytimes.com/2008/02/17/washington/17fisa.html?_r=1&hp=&adxnnl=1&oref=slogin&adxnnlx=1203255399-44ri626iqXg7QNmwzoeRkA, 2008.

[141] News 24. Think before you SMS. http://www.news24.com/News24/Technology/News/0,,2-13-1443_1541201,00.html, 2004.

[142] A. Nusca. How to: Keep your laptop from being stolen. http://www.zdnet.com/, 2009.

[143] Office of Public Sector Information. Regulation of Investigatory Powers Act (RIPA), Part III – Investigation of Electronic Data Protected by Encryption etc. http://www.opsi.gov.uk/acts/acts2000/ukpga_20000023_en_8, 2000.

[144] P. Ohm. The Fourth Amendment right to delete. *The Harvard Law Review*, 2005.

[145] PBC. http://crypto.stanford.edu/pbc/.

[146] PC Magazine. Messages can be forever. `http://www.pcmag.com/article2/0,1759,1634544,00.asp`, 2004.

[147] D. Peek and J. Flinn. Ensemblue: Integrating distributed storage and consumer electronics. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[148] R. Perlman. The Ephemerizer: Making data disappear. *Journal of Information System Security*, 1, 2005.

[149] R. Perlman. File system design with assured delete. In *Proceedings of the IEEE International Security in Storage Workshop (SISW)*, 2005.

[150] K. Peterson, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[151] L. Peterson, A. Bavier, M. Fiuczynski, and S. Muir. Experiences implementing PlanetLab. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[152] F. A. P. Petitcolas, R. J. Anderson, and M. G. Kuhn. Information hiding: A survey. *Proceedings of the IEEE*, 1999.

[153] PGP Corporation. PGP whole disk encryption. `http://www.pgp.com/products/wholediskencryption/`, 2008.

[154] M. Piatek, T. Isdal, A. Krishnamurthy, and T. Anderson. Do incentives build robustness in BitTorrent? In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[155] Picnik, Inc. `http://www.picnik.com/`, 2007.

[156] M. Pirretti, P. Traynor, P. McDaniel, and B. Waters. Secure attribute-based systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2006.

[157] B. Poettering. "ssss: Shamir's Secret Sharing Scheme". `http://point-at\-infinity.org/ssss/`, 2006.

[158] Project Voldemort. `http://project-voldemort.com/`.

[159] N. Provos. Encrypting virtual memory. In *Proceedings of the USENIX Security Symposium*, 2000.

[160] K. P. N. Puttaswamy, H. Zheng, and B. Y. Zhao. Securing structured overlays against identity attacks. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2008.

[161] M. O. Rabin. Provably unbreakable hyper-encryption in the limited access model. In *Proceedings of the IEEE Information Theory Workshop on Theory and Practice in Information-Theoretic Security*, 2005.

[162] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[163] E. Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley Professional, 2000.

[164] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM)*, 2005.

[165] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1998.

[166] T. Ristenpart, G. Maganis, A. Krishnamurthy, and T. Kohno. Privacy-preserving location tracking of lost or stolen devices: Cryptographic techniques and replacing trusted third parties with DHTs. In *Proceedings of the USENIX Security Symposium*, 2008.

[167] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2009.

[168] W. C. F. Roberto Ierusalimschy, Luiz Henrique de Figueiredo. Lua – an extensible extension language. *Software: Practice and Experience*, 1999.

[169] J. Robertson. Security chip that does encryption in PCs hacked. `http://www.usatoday.com/tech/news/computersecurity/2010-02-08-security-chip-pc-hacked_N.htm`, 2010.

[170] H. Rowaihy, W. Enck, P. McDaniel, and T. L. Porta. Limiting Sybil attacks in structured peer-to-peer networks. In *Proceedings of the IEEE Annual International Conference on Computer Communications (INFOCOM)*, 2007.

[171] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems*, 2001.

[172] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[173] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Proceedings of the International COST264 Workshop on Networked Group Communication*, 2001.

[174] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Proceedings of the Summer USENIX Conference*, 1985.

[175] N. Santos, K. Gummadi, and R. Rodrigues. Towards trusted cloud computing. In *Proceedings of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[176] M. Savage. NHS 'loses' thousands of medical records. `http://www.independent.co.uk/news/uk/politics/nhs-loses-thousands-of-medical-records-1690398.html`, 2009.

[177] J. Schiffman, T. Moyer, H. Vijayakumar, T. Jaeger, and P. McDaniel. Seeding clouds with trust anchors. In *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*, 2010.

[178] B. Schneier and J. Kelsey. Cryptographic support for secure logs and untrusted machines. In *Proceedings of the USENIX Security Symposium*, 1998.

[179] SecondBrain. SecondBrain: All your Internet Content. `http://www.secondbrain.com/`, 2007.

[180] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1996.

[181] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11), 1979.

[182] A. Shamir. Identity-based cryptosystems and signature schemes. In *Proceedings of the Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 1985.

[183] J. Shapiro, J. Smith, and D. Farber. EROS: A fast capability system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1999.

[184] R. Singel. Encrypted e-mail company Hushmail spills to feds. http://blog.wired.com/27bstroke6/2007/11/encrypted-e-mai.html, 2007.

[185] A. Singh, M. Castro, P. Druschel, and A. Rowstron. Defending against Eclipse attacks on overlay networks. In *Proceedings of the ACM SIGOPS European Workshop*, 2004.

[186] A. Singh, T. W. Ngan, P. Druschel, and D. S. Wallach. Eclipse attacks on overlay networks: Threats and defenses. In *Proceedings of the IEEE Annual International Conference on Computer Communications (INFOCOM)*, 2006.

[187] R. Sion and B. Carbunar. On the computational practicality of private information retrieval. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2007.

[188] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[189] Slashdot. Facebook scrambles to contain ToS fallout. http://tech.slashdot.org/article.pl?sid=09/02/17/2213251&tid=267, 2009.

[190] SmugMug, Inc. SmugMug – the ultimate in photo sharing. http://www.smugmug.com/, 2005.

[191] D. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2000.

[192] Sonian, Inc. Sonian email archiving service. http://www.sonian.com/, 2010.

[193] C. Sorrel. Brits send 4,500 USB sticks to the cleaners. http://www.wired.com/, 2010.

[194] M. Steiner and E. W. Biersack. Crawling Azureus. Technical Report RR-08-223, Institut Eurecom, 2008.

[195] M. Steiner, E. W. Biersack, and T. En-Najjary. Actively monitoring peers in KAD. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2007.

[196] M. Steiner, T. En-Najjary, and E. W. Biersack. A global view of KAD. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2007.

[197] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM)*, 2002.

[198] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM)*, 2001.

[199] A. Studer and A. Perrig. Mobile user location-specific encryption (MULE): Using your office as your password. In *Proceedings of the ACM Conference on Wireless Network Security (WiSec)*, 2010.

[200] D. Stutzbach and R. Rejaie. Understanding Churn in Peer-to-Peer Networks. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2006.

[201] D. Stutzbach, R. Rejaie, and Y. Guo. Large-scale monitoring of DHT traffic. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2009.

[202] A. Tanenbaum, S. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 1986.

[203] Y. Tang, P. P. C. Lee, J. C. S. Lui, and R. Perlman. FADE: Secure overlay cloud storage for file assured deletion. In *Proceedings of the International ICST Conference on Security and Privacy in Communication Networks (SECURECOMM)*, 2010.

[204] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *ACM SIGCOMM Computer Communications Review*, 37(5), 1996.

[205] A. Tereshkin. Evil maid goes after PGP whole disk encryption. In *Proceedings of the International Conference on Security of Information and Networks*, 2010.

[206] The Google+ Project. About circles. `http://www.google.com/support/+/bin/static.py?hl=en&page=guide.cs&guide=1257347&rd=1`, 2011.

[207] G. Urdaneta, G. Pierre, and M. V. Steen. A Survey of DHT Security Techniques. *ACM Computing Survey*, 43(2), 2010.

[208] uTorrent. `http://www.utorrent.com`.

[209] A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa. WebOS: Operating system services for wide area applications. In *Proceedings of the International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 1998.

[210] A. Vahdat, P. Eastham, and T. Anderson. WebFS: A global cache coherent filesystem. Technical report, UC Berkeley, 1996.

[211] M. van Dijk and A. Juels. On the impossibility of cryptography alone for privacy-preserving cloud computing. In *Proceedings of the USENIX Workshop on Hot Topics in Security (HotSec)*, 2010.

[212] H. van Heerde, M. Fokkinga, and N. Anciaux. A framework to balance privacy and data usability using data degradation. In *Proceedings of the Conference on Computational Science and Engineering*, 2009.

[213] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1992.

[214] Vuze: the most powerful bittorrent app on earth. `http://www.vuze.com/`.

[215] W3C. SOAP. `http://www.w3.org/TR/soap/`, 2004.

[216] H. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for Web browsers in MashupOS. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[217] H. Wang, Y. Zhu, and Y. Hu. An efficient and secure peer-to-peer overlay network. In *Proceedings of the IEEE Conference on Local Computer Networks*, 2005.

[218] P. Wang, I. Osipkov, N. Hopper, and Y. Kim. Myrmic: Secure and robust DHT routing. Technical report, University of Minnesota, 2007.

[219] WebProNews. Email being used more in divorce cases. `http://www.webpronews.com/topnews/2008/02/11/email-being-used-more-in-divorce-cases`, 2008.

[220] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2008.

[221] D. Wetherall. Active network vision and reality: Lessons from a capsule-based system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1999.

[222] E. J. Whitehead, Jr. and Y. Y. Goland. WebDAV: A network protocol for remote collaborative authoring on the Web. In *Proceedings of the European Conference on Computer Supported Cooperative Work*, 1999.

[223] L. Whitney. Amazon EC2 cloud service hit by botnet, outage. `http://news.cnet.com/8301-1009_3-10413951-83.html`, 2009.

[224] A. Whitten and J. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Proceedings of the USENIX Security Symposium*, 1999.

[225] D. Winer. XML-RPC Specification. `http://www.xmlrpc.com/spec`, 1999.

[226] S. Wolchok, O. S. Hofmann, E. W. Felten, J. A. Halderman, C. J. Rossbach, B. Waters, and E. Witchel. Defeating Vanish with low-cost Sybil attacks against large DHTs. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2010.

[227] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6), 1974.

[228] H. Xie, R. Yang, A. Krishnamurthy, Y. Liu, and A. Silberschatz. P4P: Provider portal for P2P applications. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM)*, 2008.

[229] Yahoo!, Inc. pipes. `http://pipes.yahoo.com/pipes/docs`.

[230] Yahoo, Inc. Browser-Based Authentication (BBauth). `http://developer.yahoo.com/auth/`, 2007.

[231] H. Yu, M. Kaminsky, P. B. Gibbons, and A. D. Flaxman. SybilGuard: Defending against Sybil attacks via social networks. *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM)*, 2006.

[232] K. Zetter. Tor researcher who exposed embassy e-mail passwords gets raided by Swedish FBI and CIA. `http://blog.wired.com/27bstroke6/2007/11/swedish-researc.html`, 2007.

[233] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, 2001.

[234] P. Zimmermann and J. Callas. *The Evolution of PGP's Web of Trust*. O'Reilly, 2009.