Securing and Managing Trust in Modern Computing Applications

by

Andy Sayler

B.S.E.E., Tufts University, 2011

M.S.C.S., University of Colorado, 2013

A dissertation proposal submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2015

This dissertation proposal entitled:
Securing and Managing Trust
in Modern Computing Applications
written by Andy Sayler
has been approved for the Department of Computer Science

_____

Prof. Dirk Grunwald

_____

Prof. John Black

_____

Prof. Sangtae Ha

_____

Prof. Eric Keller

_____

Prof. Blake Reid

Date _____

The final copy of this dissertation proposal has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Today, users routinely push vast troves of private data into a myriad of cloud services, store personal information on mobile devices, and communicate about sensitive topics over public networks. These uses, while convenient, demonstrate a growing gap between users' desired use cases and their ability to control and protect their data. Today, users are often forced to choose between leveraging modern technologies and services, potentially exposing their data to a variety of questionably trustworthy third parties, and maintaining privacy and security of their data.

I feel these two choices represent a false dichotomy that can be resolved by disentangling the security and access control mechanisms used to protect user data from the underlying third parties that store and act on this data. To accomplish this disentanglement, I propose isolating core user secrets (e.g. passwords, cryptographic keys, etc) and storing these core secrets via a dedicated, standardized, and service-oriented secret storage system. Such secrets can be leveraged to bootstrap further privacy and security enhancing techniques (e.g. via the client-side encryption of third-party stored user data). This "Secret Storage as a Service" (SSaaS) model allows users to reduce the trust they must place in any single third party while still providing mechanisms to support the range of cloud-based, multi-user, multi-device use-cases popular today.

In this dissertation proposal, I present an overview of the challenges posed to the security and privacy of user data today. I introduce a framework for evaluating the degree to which we must trust various third parties across a range of popular usage patterns, as well as the mechanisms by which this trust can be violated. I explore the problems with the traditional third party trust model and discuss ways in which the SSaaS model helps alleviate these issues. I discuss the qualities a desirable SSaaS ecosystem would have, propose several applications where the SSaaS model can significantly increase user security and privacy, and present a first-generation SSaaS prototype. Finally, I propose the additional research I will undertake to complete this dissertation, including a survey of modern services and technologies using the proposed trust-analysis framework, the implementation of a second-generation SSaaS prototype, the implementation of a variety of useful SSaaS-backed encryption applications, and an analysis of the security, performance, and capabilities of said systems.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

"A paranoid man is a man who knows a little about what's going on."

– William S. Burroughs, *Friend Magazine, 1970*

1.1 Overview

Over the last decade, computing has undergone a monumental shift from locally stored data on a single personal computer to cloud-based data storage on a multitude of third party servers. This shift has generated many benefits: sharing data with other users is trivial, multi-modal communication between users is easy, and compute devices are largely ephemeral, easily replaced or transitioned between without any significant overhead or loss of user data. This transition, however, has a significant side effect: user data is now largely stored in a manner where it is easily accessible to third parties beyond the user's immediate control. The shift from locally stored and controlled data to third-party stored and controlled data has a number of consequences, from increased risk of data compromise by hackers targeting centralized third-party data stores, to reduced legal protections from government introspection, to its use in "big-data" systems capable of ascertaining more private information than most users likely intend to share.

The popularity of the cloud model leads one to believe that most users are willing to trade the privacy and control afforded by traditional local compute and storage for the convenience and features cloud-based services provide. And yet, a 2014 Pew Research study found that over 90% of American adults agree that they have lost control over the data they store in the cloud, 80%

are concerned about how cloud companies are using their data, and 70% are concerned about the manner is which the government might access their data in the cloud [186]. Furthermore, the myriad of recently publicized data leaks at large companies (e.g. [12]) as well as ongoing government intrusions into third-party user data stores (e.g. [97]) has propelled the debate over user privacy in the age of the cloud to new levels.

The traditional viewpoint holds that users must choose between either the conveniences the cloud provides or the privacy and security of locally stored and processed data. I do not feel that this is true. Instead, I believe that there are mechanisms that can allow users to retain a high degree of control over how their data is stored, accessed, and used while still allowing them to leverage a variety of modern third-party services. The solution lies in developing systems that allow users to place limits on the degree to which they must trust any single third party while still allowing them to leverage the desirable features such parties provide.

To achieve such a solution, I propose a new data storage model called "Secret Storage as a Service" (SSaaS). In an SSaaS ecosystem, a user designates one or more trusted secret storage providers (SSPs), either self hosted or third party, with storing and regulating access to their private secrets (personal information, encryption keys, etc) on their behalf. Existing technologies and services can than interface with these SSPs via a standard interface to access user secrets as allowed by a user-defined set of access control rules. In effect, the SSPs are tasked with regulating access to user data by more traditional feature-oriented third-party services (e.g. Google or Facebook).

The SSaaS model provides a number of benefits over the existing practice of utilizing feature-desirable third-party services, and in doing so, granting them unfettered access to user data. Such benefits include:

No Single Trusted Third Party

In an SSaaS ecosystem, the secret storage provider is separate from the provider of the end-user cloud service (e.g. Dropbox, Gmail, etc). Furthermore, a user may shard their

secrets across multiple SSPs, or even host their own SSP. This ensures that a user is not giving any single entity control over or unfettered access to their data.

Separation of Duties

In an SSaaS ecosystem, a user selects a secret storage provider on the basis of their trust in that provider while selecting a feature service provider on the basis of the end-user services they provide. This allows a user to optimize each selection individually instead of having to chose a single provider on the basis of both trust and feature set, likely having to sacrifice one in favor of the other.

Support for Existing Use Cases

The SSaaS ecosystem is capable of supporting many modern use cases such as sharing data with other users or syncing it across a number of personal computing devices. Thus, SSaaS allows users to gain privacy and security benefits without having to forgo common and popular use cases.

1.2 Motivating Examples

As a motivating example, consider the Dropbox cloud file locker service [42]. Dropbox provides a service through which users may upload arbitrary files in order to sync them between multiple devices and to share them with other users. In order to support this functionally, Dropbox stores a copy of each user's uploaded files on the Dropbox servers. This ensures that Dropbox can provide copies of the files to new user devices or to other users when asked to sync or share a file on the user's behalf.

But how private are a user's files once uploaded to Dropbox? While Dropbox does encrypt files while they are stored on the Dropbox servers as well as while they are in transit between the Dropbox servers and a client machine [43], Dropbox also holds a copy of the associated encryption keys, enabling them to decrypt a user's files whenever they desire. This also means that an adversary may be able to decrypt user files if they are able to compromise both Dropbox's data storage servers as well as their key storage servers. The US government could also decrypt user files

stored via Dropbox should a court Dropbox to provide both the files and the associated encryption keys. Furthermore, a rogue actor within Dropbox could leverage their access to all of Dropbox's infrastructure to access and decrypt user files. Clearly Dropbox's practice of storing both a user's encrypted files as well as a copy of the associated encryption keys provides only marginally more security and privacy of the user data then not using encryption at all. And the added trust it does provide is wholly dependent on placing a high degree of trust in Dropbox to faithfully protect and manage the relevant encryption keys.

An alternative approach would be for Dropbox to put the user in charge of encrypting/decrypting files and storing all necessary encryption keys, ensuring the Dropbox itself never has direct access to unencrypted user files. While this form of client-side encryption could potentially increase the privacy and security of user data in the event that Dropbox's data stores are compromised, searched, monitored, or simply misused, is also has some significant downsides:

(1) It breaks Dropbox's sharing use case. While user's can still share encrypted versions of their files, they would then have to exchange the associated encryption key out of band in order to effectively decrypt, read, or update any shared file. This essentially nullifies Dropbox's appeal as a simple method for sharing files with other users.

(2) It complicates Dropbox's syncing use case. Whereas before a Dropbox user can bootstrap a new Dropbox client device simply by signing into their Dropbox account, users must now both sign into their Dropbox account and provide a copy of their encryption key/keys in order for the Dropbox client to successfully perform the required client-side encryption operations. This adds an additional step to the Dropbox setup process, potentially driving away novice and lay-users.

(3) If the user ever losses their encryption keys, they will effectively lose access to all of their Dropbox-stored files. Similarly, if a user mishandles their keys in a manner that allows others to access them, they have effectively negated the additional privacy or security client-side encryption provides. A user would have to be diligent about ensuring they

maintain access to their keys via backups, etc, while also ensuring their keys do not fall into the wrong hands. Again, such manual management requirements pose a non-trivial burden for many users.

Thus, we're left in a situation where the user must chose between the convenience of using Dropbox as it exists today while also sacrificing a significant degree of privacy over the files they upload to Dropbox, or the burden of employing traditional client-side encryption models where the trust they must place in Dropbox is reduced, but where many core Dropbox features (e.g. simple file sharing) are no longer feasible to use. Neither of these are ideal solutions. We would like a solution that allows the user to leverage the existing convenience and benefits of using Dropbox while also reducing the trust they must place in the Dropbox corporation.

These challenges are not unique to Dropbox. There are many modern technologies and services that force the user to chose between convenience and feature set vs privacy and control. For example:

Mobile Computing Devices

Phones, tablets, and laptops have become ubiquitous modes of modern computing, storing large fractions of our personal data and carrying out computations on our behave. But these devices, while convenient, are also prone to loss, theft, and remote exploitation, exposing the data they store and computations they undertake to a range of external actors.

Cloud Computing Infrastructure

Infrastructure-as-a-Service (IaaS) systems such as Amazon's EC2 [6] or Google's Compute Engine [85] are popular mechanisms for hosting modern compute services. Unfortunately these services require the user to fully trust the backing infrastructure provider and make it difficult to deploy security-enhancing systems like full disk encryption due to the user's lack of physical server access.

User Account Registration

We're constantly being asked to register for services available online. This means providing

the same identity-confirming personal data to third party after third party with little ability to police how this data is stored or used after it is provided.

All of these examples share a common deficiency: they force the user into a position of choosing between desirable feature sets or desirable security and privacy qualities. It is this deficiency that I seek to quantify and resolve via the work presented in this proposal.

1.3 Goals

At the core, the goals of my proposed work are threefold:

- Quantify and analyze the trust and threat models inherent in modern mobile, cloud, and datacenter computing solutions.
- Provide primitives that allow users to minimize, manage, and monitor the degree to which they must trust and expose themselves to third parties within said solutions.
- Use such primitives to create security and privacy enhancing systems well adopted for a range of modern, common, and desirable use cases.

This document works toward these these goals and proposes the necessary future work to complete them. The remainder of this proposal is structured as follows:

Chapter 2 - Background: Presents the necessary background knowledge related to this work.

Chapter 3 - Challenges: Discuses the security and privacy challenges of modern use cases.

Chapter 4 - Related Work: Describes existing work related to enhancing privacy and security.

Chapter 5 - Trust Model: Presents a model for analyzing trust and trust violations.

Chapter 6 - SSaaS: Presents the Secret Storage as a Service model and its merits.

Chapter 7 - Custos: Presents the Custos SSaaS prototype and its design and implementation.

Chapter 8 - Applications: Presents a number of potential trust-limiting SSaaS applications.

Chapter 9 - Proposed Work: Proposes necessary work for the completion of this dissertation.

Chapter 2

Background

This work builds on a number of established topics related to computing, privacy, and security. I touch on the fundamentals of each in this chapter. Chapter 4 touches on more directly related work.

2.1 Cryptography

Many of the topics discussed in this proposal leverage cryptographic primitives as the basis of various security and privacy guarantees. This is largely because cryptography represent a security primitive that does not rely on trusting specific people, platforms, or systems in order to securely function. Instead, it requires that we place our trust in only one thing: the underlying math. This has led to the proliferation of cryptography as the primitive on which many security and privacy enhancing features are built.

2.1.1 Symmetric Cryptography

Modern cryptographic systems come in two flavors: symmetric cryptographic and asymmetric cryptography. Symmetric cryptography algorithms function on the principle that a single "key" is used for both halves of given cryptographic operation (e.g. encryption and decryption). Asymmetric algorithms (discussed in the next section) overcome this limitation.

The core symmetric cryptography operation is symmetric encryption. Symmetric encryption algorithms use a single key to both encrypt and decrypt a message. This message, once encrypted,

can not be deciphered without access to the associated secret key. This key must be securely stored, or if shared, securely exchanged between parties. Anyone with the key can decrypt the corresponding ciphertext the key was used to create; anyone without can not. The security of a symmetric encryption cipher tends to be proportional to the length of the encryption key: the longer the key, the more secure the data encrypted with it is. Common modern symmetric encryption algorithms include block-ciphers such as AES [162] and TwoFish [221] as well as stream ciphers such as (the now deprecated) RC4 [167][1] .

In addition to encryption and decryption, symmetric cryptography algorithms can also be used to provide secure message integrity and authenticity verification. Such keyed hash algorithms leverage a secret symmetric authentication key to generate a message authentication code (MAC) across a given piece of data. This MAC can then be shipped to another user along with the data over which it was generated. Any other holder of the authentication key who receives said data can recompute the MAC value independently, comparing their computed value to the one that was sent and verifying the authenticity and integrity of the associated data in the process. In this manner, users who share a symmetric authentication key can ensure that the data they send to each other is not tampered with in transit and that it comes from another holder of the required key. MAC algorithms can be built from existing hash functions (e.g. HMAC-SHA-256 [118]) using mechanisms such as HMAC [125] or from existing symmetric block ciphers (e.g. AES-CMAC [230]) using mechanisms such as CMAC [20, 47]. In most situations, both encryption and authentication are used in tandem to form a secure challenge capable of communicating data that is indecipherable to outsiders and over which any outside data tampering can be detected.

While symmetric cryptography algorithms are useful in situations where a single actor will be both encrypting and decrypting a piece of data (and thus can hold the required key personally), they pose a major challenge it situations where multiple parties wish to communicate or exchange data securely. In such situations, the parties must find a way to securely communicate the required

[1] Many block ciphers can actually be used as stream ciphers by leveraging operation modes such as the CTR and OFB.

symmetric encryption and authentication keys manually out-of-band. In the absence of additional cryptographic methods, the only real way to securely communicate such keys while avoiding both eavesdroppers and interlopers is to meet in person and generate or exchange keys manually. This task is tedious and impractical for all but the simplest of situations, especially given the modern digital communication landscape where multiple actors may be continents apart. The challenges of securely bootstrapping symmetric cryptography systems led researchers to seek a better method for secure data exchange in the absence of an inherently secure communication channel.

2.1.2 Asymmetric Cryptography

The major breakthrough in solving this challenge came in 1976 when Diffie and Hellman proposed a system for asymmetric cryptography (i.e. public-key cryptography) [40][2] : a cryptography system in which one key is used for encryption while a second related key is used for decryption. When properly designed, it is computationally infeasible to derive one of the keys in an asymmetric cryptography system from the other, allowing a user to publish one of their keys for public consumption while keeping the corresponding key private. A member of the public can then use a user's public key to encrypt a message that only the holder of the corresponding private key will be able to decrypt. If all members of the public maintain such public/private key pairs, it becomes possible for any user to send any other user a message that only the recipient can read without requiring an in-person meeting or similar out-of-band secure communication channel.

Asymmetric cryptography relies on the existence of "trapdoor" functions in order to operate. These functions can be quickly solved in one direction, but are computationally difficult to reverse without a special piece of information (e.g. the 'key'). Factoring large numbers is a classic example of a trapdoor function (and the method on which many modern public key encryption systems are based). Factoring large numbers is computationally difficult in cases where some piece of secret information (e.g. one of the factors) is not known. While, Diffie and Hellman proposed a potential

[2] Asymmetric cryptographic was actually independently developed by the GCHQ (the British signal-intelligence counterpart to the US National Security Agency) in the early 1970s prior to the publication of Diffie and Hellman's paper, but this work was classified and remained a secret until the late 1990s.

implementation of a public key cryptography system, the first practical public key crypto system came a few years latter with the invention of the RSA [197] algorithm. In addition to public/private key systems, Diffie and Hellman also proposed a system for joint key generation where two parties can negotiate a secrete across an insecure connection. Like asymmetric cryptography, such a system can be used to bootstrap secure communications across an insecure channel by allowing two parties to derive a mutual secret that can then be used to facilitate further secure communication using a symmetric encryption and authentication algorithm.

Asymmetric encryption can be used to build the two additional core asymmetric cryptography primitives: cryptographic verification and cryptographic authentication. Cryptographic verification (also called a cryptographic "signature") is essentially the reverse of asymmetric encryption: instead of a member of the public using another party's public key to encrypt a message that only the target party can read, they instead use their own private key to encrypt a message that the public can then decrypt using the signers public key. Since only the owner of a given key-pair should have access to the private key necessary to generate such a message, the owner can "prove" to the public that a given message comes from them. Similar to symmetric MAC systems, asymmetric signatures can also be used to verify that signed data has not been altered in transit since any alteration would result in a verification failure when a member of the public decrypts the message signature.

Just as asymmetric encryption gives rise to cryptographically secure signature algorithms, cryptographically secure signature algorithms give rise to cryptographically secure authentication systems. If a user generates a signed message saying "I am John" and sends it to an authentication server, the server can verify that the message signature is valid using John's public key, authenticating John in the process. The server need only have a list of public keys for each approved user. It can then leverage the assertion that only the indented user has access to the private key corresponding to each approved public key, and is thus the only one capable of generating a signed message on that user's behalf, as the basis of user authentication.

As mentioned above, both symmetric and asymmetric cryptography are often used together. Asymmetric cryptography is useful for bootstrapping a secure communication channel by allowing

the secure exchange of symmetric encryption and authentication keys over a previously insecure channel. These session keys can then be used to continue all further communication using symmetric encryption and MAC algorithms. Symmetric algorithms tend to be more performant then asymmetric algorithms, making such split-type crypto systems desirable.

2.1.3 Secret Sharing

Beyond the rise of public key cryptography, one of the other major cryptographic breakthroughs of the last fifty years was the invention of cryptographically secure secret sharing schemes. In particular, Adi Shamir (the 'S' form "RSA") proposed a practical and robust secret sharing scheme in 1979 [222]. In this work, Shamir lays out the basics of what has come to be known as Shamir Secret Sharing: a method for splitting a piece of information up into two or more pieces such that holders of any subset of the pieces cannot infer any information about the pieces they do not hold or the original block as a whole. Shamir Secret Sharing allows a user to divide a piece of D data into N pieces of which K or more pieces can be used to recompute the original value of D. A user with fewer than K pieces, however, has no more information about the value of D than a user with no pieces. This system provides a highly useful method for distributing information amongst multiple parties or systems in situations where no single party or system can be fully trusted. Such systems can also be used to provide redundancy by selecting N to be greater than K.

Shamir Secret Sharing, unlike all known asymmetric encryption techniques, does not rely on computational complexity as the basis of its security. Instead, it is fundamentally secure based on information theory principles. Thus, unlike computationally secure systems such as RSA, Shamir Secret Sharing can not be broken regardless of the amount of computational power one posses. Shamir Secret Sharing functions on the basis of defining a polynomial of degree (K-1) over a finite field with the D data encoded as the first order-zero term. N points are then selected from this polynomial and distributed to the participants. Since K points (but no fewer) will uniquely identify the original polynomial, and thus allow the derivation of D, K users must combine their pieces in order to re-compute D.

Shamir Secret Sharing (and related systems) are useful in a wide range of situations where one needs to distribute trust across multiple entities. In particular, secret sharing techniques are leveraged in some cryptographically-based access control systems like that described in [93].

2.2 Usability

Strong cryptography provides the basis for many of the secure systems we build today. Unfortunately, strong cryptography has a rather checkered history when it comes to the usability of secure cryptographic systems. Since most cryptographic systems merely reduce the security of a system to the security of the cryptographic keys protecting a system, how one manages such keys is of the utmost importance. Manual key management, the de-facto key management standard for many cryptographic systems, tends to be extremely challenging for the average user to execute properly: often leading to security failures that have little to do with the quality of the cryptography itself.

PGP, one of the longest-running cryptographic security projects, is known to have major usability issues, making it largely incomprehensible to all but the most highly-trained users [264]. These challenge are largely related to the average user's inability to properly manage the various cryptographic keys required for the proper use of PGP [94]. This has led multiple parties to call for the retirement of PGP [95] and/or to suggest alternatives [49, 171, 89]. It remains to be seen if any of the purposed alternatives will be able to provide the level of security offered by PGP while avoiding the usability pitfalls leveraging PGP traditionally entails.

Similar challenges have been observed with other end-user cryptographic systems, ranging from secure storage devices to various communication mediums [243]. In all cases, properly obtaining, storing, and controlling access to cryptographic keys and related cryptographic secrets tends to be a task for which the typical user is ill-suited. Thus, while cryptographic systems like S/MIME have seen limited success in the enterprise where a central authority and staff can handle all key management duties, they have largely failed for individual computer users [193]. How best to

build systems that are both cryptographically secure and easy to use remains an open and pressing question.

2.3 Storage

Data storage has long been one of the core use cases of the digital age. And the amount of data we generate, process, and store is greater today then ever before. Tied tightly with data storage mechanism are the access control mechanisms required to protect the digital data we store. Digital data storage and access control techniques have morphed and changed over the last 50 years, and many of these changes have bearing on the work presented in this proposal.

Early storage and file system technologies often simply neglected data security, lacking robust encryption, verification, and access control primitives. The rise of multi-user operating systems like Unix mandated the creation of basic file-system access control schemes. Thus we gained the traditional Unix file access control and permissioning scheme as part of the virtual file system (VFS) abstraction inherent in all legacy and modern Unix-like operating systems. This system, however, has a number of limitations: it supports only a single, basic access control model (owner, group, and other; R/W/E permissions), it requires a trusted system for enforcement (e.g. the OS kernel), and it is strongly coupled to a specific local system. Systems like NFS attempt to extend Unix file security semantics beyond the local machine allowing remote sharing of files, but even these systems are limited to singular administrative domains and trusted systems.

The Windows NT file system access control model (implemented via the NTFS file system) extends the flexibility of the traditional Unix model by adding support for more expressive access control lists (ACLs). These both allow the control of additional permissions (e.g. delete, create directory, etc) as well as more expressive user to permission mappings beyond the basic owner/-group/other Unix model. Furthermore, the Windows NT model has the ability to delegate user authentication to a local Domain Controller (DC) capable of centrally managing all users from a single location. This expands the ability to control file access beyond the users associated with the local system to the users associated with an entire administrative domain. Yet this system

still has many of the same limitations as the Unix model: the requirement for a trusted system for enforcement and the tight coupling to the local administrative domain.

The rise of the Internet as a reliable and high speed system for connecting multiple machines across the world as well as the move toward cloud computing models where computational resources are outsourced to dedicated providers has increased the demand for secure storage systems capable of spanning multiple systems and domains. In order to overcome the limitations posed by traditional file system security models and accommodate modern multi-user, multi-system use cases, researchers have proposed a number of newer file storage systems. These systems try to address one or more of the limitations mentioned above. Some of them employ cryptographic security models to overcome the need for a trusted enforcement system. Others are designed to extend access control semantics beyond the local machine to large networks or even the global internet. Still others explore the use of novel access control models more expressive then Unix permissions or Windows NT ACLs. Many system combine more than one of these approaches to build a fully featured next generation secure storage system.

[121] presents a survey of the security models of various data storage systems, sorting such systems into basic networked file systems, single-user cryptographic file systems, and multi-user cryptographic file systems. As previously mentioned, basic networked file systems rely on trusted systems and administrators for the enforcement of security rules. Examples of such systems include the Sun Network File System (NFS) [208], the Andrew File Systems (AFS) [105], and the Common Internet File System (CIFS/SMB) [156]. All of these systems are designed for use within local administrative domains and do not scale well to global, loosely-coupled distributed systems. To deal with the scalability issues, researchers have built system like SFS [149] or OceanStore [130] which aim to reduce the administrative burden of large scale distributed file systems.

All of these systems, however, rely on some degree of system, administrative or third-party trust. In order to accommodate situations where users do not wish to place trust on the underlying system or remote servers, there exist a handful of cryptographically-secure file systems. The best of these systems offer end-to-end cryptography, meaning that data is encrypted and decrypted

on the client side and the server never has access to the unencrypted data. Systems like the Cryptographic File Systems (CFS) [21] or eCryptFS [98] provide basic single-user end-to-end file encryption. While end-to-end encryption is a powerful security model for enabling secure storage atop untrusted systems, it does pose challenges with respect to multi-user, multi-device use cases since it generally requires that all clients have access to private cryptographic credentials in order to effectively read or write files. In order to support both end-to-end encryption and multi-user scenarios, researchers have proposed multi-user cryptographic storage systems like SiRiUS [76], cepheus [64], or Plutus [116].

Miltchev, et al. [158] presents a framework for analyzing the suitability of various distributed file systems for modern multi-user, multi-domain use cases by analyzing five underlying file system qualities: authentication, authorization, granularity, delegation, and revocation. They suggests that any secure large scale file system must successfully address the functionality of all five of these factors across multiple administrative domains in order to be an effective multi-user, multi-domain file system. Miltchev, et al. reach the following conclusions regrading successful secure multi-user, multi-domain file systems: the use of public-key cryptography for user authentication is an effective way to support autonomous delegation, capability-based access control systems tend to lack support for auditing and accountability, ACL-based access control systems pose scalability challenges when used across administrative domains, and revocation and user autonomy are often at odds.

Beyond traditional local and networked file systems, many users have turned to cloud-backed storage technologies today. System like Dropbox [42] or Google Drive [88] offer mechanism for storing arbitrary files on third-party cloud servers. Such files can either be accessed via a web browser or synced to a local machine via various client-side utilities. Other cloud services exchange the traditional file storage model all together in favor of various object storage abstractions: Amazon's S3 [7] and Ceph [235] are examples of such systems. These systems can either be used for raw key:object data storage or as a block-oriented backing store for higher level file storage abstractions such as Dropbox or Google Drive. System live Dropbox, Google Drive, or Amazon S3 all rely on a centralized, trusted third party storage provider for "secure" operation. To overcome this

requirement distributed systems such as Tahoe-LAFS [265] propose an alternate model where trust in any single system is reduced and storage is spread across a variety of third-party providers.

2.4 Access Control

Over the years, we have developed a range of access control techniques. All of these techniques share a common goal: controlling access to a specific system, resource, or piece of data. Must access control models have two key components: authentication and authorization. Authentication is used to establish the identity of an actor. Authorization then leverages this identification as the basis of granting or denying specific permissions to the actor.

Computer-based access control systems have been with us since the earliest multi-user (e.g. time sharing) operating systems became popular in the 1970s and 1980s [204]. Early access control systems were primarily focused around the Unix model of access control: users, groups, and read/write/execute file-level permissions. Authentication in these early systems was generally limited to username:password combinations, the mechanisms of which were hard coded into the `login` program. Later, more flexible pluggable authentication systems such as PAM were created [207, 141, 229]. In such Unix-lake access control systems each user is a member of one or more groups and each file has an owner and group. The three file permissions (read, write, and execute) are granted on the basis of a user's relationship to a given file: either the user is the file owner, the user is a member of the file's group, or the user is neither of these things. This model is fairly flexible, and continues to be used today as the core access control model in many Unix-like operating systems (e.g. BSD, Linux, OSX, etc).

Access Control List (ACL) based schemes gained prominence in 1990s and were popularized by the Windows NT family of operating systems. ACLs extend the permission model beyond the basic Unix file permissions to include a wider range of file (e.g. read, write, delete, create, etc) and system-level (e.g. shutdown, connect to network, etc) permissions. ACLs are associated with specific system objects (e.g. files, folders, OS subsystems, etc) and map a user or group to a list of permission that user or group possess. They generalize the Unix access control model to

accommodate a wider range of permissions and mappings between permissions and actors. ACL-based systems have been integrated into many modern Unix-like operating systems as an optional extension beyond the tradition Unix permissioning scheme.

Exiting access control schemes are often grouped into one of two classes: Mandatory Access Control (MAC) systems or Discretionary Access Control (DAC) Systems. While the lines between these two approaches are occasionally blurred, the basic difference between the two lies in which actors within a system have the ability to grant/extend permissions to other actors. In MAC system, all permissions are set by the system administrator and users have no ability to change these permissions themselves or transfer permissions to other users. DAC systems, in contrast, give users the ability to set their own permissions on objects they own or create, and to transfer these permissions to other users. A MAC-based system can be thought of as similar to a DAC system where the system administrator owns all files and never transfer this ownership to any other user. Traditional Unix access control systems as well as ACL access control systems can generally be used in either MAC or DAC based systems. MAC systems are generally preferred in high security environments where the centralized management models they offer lead to tighter control over data. DAC systems are more common in general purpose systems where the extra flexibility they offer reduces the administrative burden. Most Unix-like systems are DAC systems by design, but extensions (e.g. SELinux) can be used to add MAC properties to these systems.

Many of the early access control systems pose a host of manageability challenges. How do you coordinate the permissions of thousands of users across millions of objects? How do you revoke permissions for a defunct user? Or add a new user? Role-Based Access Control Models [209] arose to cope with many of these challenges. Role-Based Access Control (RBAC) inserts an additional layer of indirection between users and permissions. In an RBAC system, users are assigned to one or more roles. Each role is then assigned one or more permissions. This model simplifies management by separating permission assignment from specific users. RBAC permissions are assigned on the basis of specific positions or duties within an organization and mapped to specific roles. Users are then assigned to these roles on the basis of whether or not they hold a specific positions or are required

to perform a specific duty. Thus, adding or removing users does not require any modification to permission mappings, only role mappings. Likewise, adding or removing permissions does not require modifying user mappings, only role mappings.

2.5 The Cloud

The previous 10 years have seen a major shift in the manner in which users and developers obtain various compute resources. Gone are the days where one must purchase there own hardware or operate their own computing systems. Instead, numerous companies are more then happy to sell you any compute service you require for a pre-established time-metered rate. This "Cloud" computing model significantly lowers the barrier to entry to those needing to leverage compute resources, increasing the availability of such services and driving a vast shift in the way we use the Internet, store our data, obtain our entertainment, interact with our friends, and more.

2.5.1 Benefits

Today cloud-services providers like Amazon, Google, Microsoft, Rackspace, and IBM globally sell over $150 billion in cloud services annually [58]. The rapid rise of the cloud computing model is supported by a number of desirable qualities the cloud can provide more effectively then traditional self-hosted computing systems. Namely:

OPEX vs CAPEX

Using cloud-based compute services allows companies to shift what are traditionally one-time up-front capitol expenditures (CAPEX, e.g. large arrays of servers) to regular operational expenditures (OPEX, e.g. a monthly fee). This fact gives rise to a number of potential benefits. Where as spinning up traditional compute infrastructure requires a large initial investment, cloud compute infrastructure can be purchased for as low as a few dollars each month. This drastically lowers the barrier of entry to those requiring such services by eliminating any large up-front costs. Furthermore, moving to the cloud makes

compute infrastructure a regular, predictable expense, easily accounted for when planning budgets. Finally, operational expenditures are often more easily justified at many organizations without requiring major internal review processes, allowing those that purchase compute services via the cloud to retain more direct control over how, when, and what they purchase. All of these factors interact to make the OPEX cloud model a more desirable purchasing model then the traditional CAPEX model.

Flexibility

The "pay-for-what-you-need" cloud purchasing model is also far more flexible then the traditional in-house computing model. Where as the traditional model requires buyers to accurately predict their future compute requirements before making the initial purchase, the cloud allows users to scale infrastructure as required and without any real need for accurate forward demand prediction. This makes it far simpler to start a small project and grow it into a larger project without requiring any large up-front cost or guesswork. Likewise, if a project fails to gain traction, it can be efficiently spun down and no one is left holding a bunch of expensive, but no-longer-useful, hardware.

Efficiency

The cloud models offers efficiencies of scale not available to traditional in-house compute users. At the macro-level, it is rare for end-user systems to require constant load throughout the day. Instead, services tend to see peek usage at certain times related to the diurnal cycles of their users. Large, international cloud service providers can leverage this fact in ways individual hardware operators often fail to do. In particular, such providers can ensure their underlying hardware is uniformly loaded 24/7/365 by spreading the workloads of a variety of diverse tenants across globally-located infrastructure. This allows large cloud services providers to operate their systems at a steady capacity, avoiding the need to oversize systems to account for short-lived peak loads. At the micro level, cloud providers are generally able to co-locate a large number of compute systems in a single data center, allowing them to optimize cooling, power, network, and other resources in manners not available to smaller

in-house server farms. On the power and cooling front, it is not uncommon to see cloud data centers that are over 90% efficient - e.g. a PUE[3] of ≈ 1.1 [86]. On the networking front, such co-location allows for higher speed, lower energy, data transfers between machines. The net result of all these efficiency gains is that cloud providers can generally offer compute resources for less cost then in-house data center deployments.

2.5.2 Service Classes

Modern cloud systems come in a range of classes. These classes generally divide cloud-services up based on the level of abstraction they provide. The common cloud services in use today include:

IaaS: "Infrastructure as a Service" systems describe the lowest-level of cloud services. In an IaaS environment, the user is provided with remote access to a raw computer - generally a virtual machine with a pre-installed operating system - atop which they may build and implement their own services. This class of cloud services represents the more-or-less direct replacement for the traditional in-house compute model where a user would start with a raw physical machine and build up from there. Amazon EC2 [6] and Google Compute Engine [85] are both example of IaaS services.

PaaS: One step up the stack we have "Platform as a Service" offerings. PaaS systems provide the end user with an environment capable of running their code, but abstract away a lot of the lower level details of setting up and managing a full OS and virtual machine. This allows users to trade flexibility for simplicity and ease of use. Google App Engine [82] and Heroku [102] are examples of PaaS offerings.

SaaS: At the top of the cloud service stack, we have "Software as a Service". This class of service is most generally what consumer end-users are referring to when they talk about the cloud. SaaS offerings represent fully-fledged services that provide some form of functionally directly to an end user. Examples of SaaS systems include Dropbox [42], Gmail [90], and Facebook [53].

[3] Power Usage Effectiveness: The ratio of total consumed power to useful IT power.

It is not uncommon for one layer of cloud services to be built atop a lower layer. E.g. an SaaS system might be built atop a PaaS system, itself built atop an IaaS system. Furthermore, the "...aaS" inherent in the names of each of these layers reflects another cloud trend: the popularity of service oriented architectures. Such architectures abstract a set of useful actions into a service that can be consumed by users. Such systems encourage the standardization and commoditization of a wide range of useful computing tasks. This allows developers of new services to lean heavily on existing services: adding only the specific new functionality they need without having to build the supporting infrastructure from scratch. The end result is yet another mechanisms for accelerating the rate of advancement when building systems and services atop the cloud.

2.5.3 Enabling Technologies

It's also important to note the technologies underlying the shift toward cloud-backed computing infrastructure. In particular, several core advances have enabled the modern cloud as we know it. These include:

Commoditization of Hardware

The cloud, by and large, is built using cheap, off-the-shelf commodity hardware. High-end, specialty hardware is rare to find in most cloud data centers. Instead, Google, Amazon, and others leverage more or less the same computing components used in most consumer hardware, but in much larger numbers. Cloud providers have discovered that it is more cost effective to utilize cheap consumer parts and simply design systems to cope with the higher rates of failure such parts exhibit then it is to buy ultra-high end parts with lower failures rates but disproportionately higher costs [13]. This shift has made hardware an easily replaceable and interchangeable commodity in modern data center design, lowering the cost and barriers to entry involved in constructing and maintaining such data centers.

Virtualization

Virtualization, the ability to simulate one or more "virtual computers" running atop a single

physical computer, is not a new concept [79]. But the previous 10 to 20 years have seen the use of virtualization become a commonplace occurrence, well supported by commodity hardware and software alike. Virtualization is what has made it simple and cost effective for cloud-providers to offer their services, slicing discreet physical systems between many paying users. Virtualization also allows providers to separate users from any singular piece of hardware – providers can now migrate users between physical systems in order to meet up-time and load balancing goals without the user even being aware of such a process.

Free and Open Source Software (and Hardware)

The rise of Linux and related Free and Open Source Software (FOSS) systems has closely tracked the rise of the cloud. This is no coincidence. FOSS systems allow users to quickly and cheaply deploy a range of applications without having to worry about purchasing specialty high-cost software, vendor lock-in, or any number of other barriers to deployment mobility. While the cloud provide cheap, commodity hardware resources, FOSS provides cheap, commodity software to make such resources do useful things. Furthermore, many cloud providers have combined the ubiquity of commodity hardware with the ethos of FOSS to create open-hardware ecosystems – making it a lot simpler for service providers to build and deploy cloud-optimized computing hardware [249].

The success of the "cloud" is not due to any singular new idea or major breakthrough in computing. Instead, it represents the confluence of a number of discreet technologies, business cases, and user demands that have coincided at a mutually beneficial moment in time. In doing so, these events have fundamentally shifted the way developers and end-users alike consume and interact with the available computing systems of the 21st century.

Chapter 3

Challenges to Privacy and Security

As mentioned in Chapter 1, the last 10+ years have heralded the rapid expansion of a number of digital data related use cases. In order to accommodate these use-cases, most modern services leverage some form of third-party compute or data storage systems. These third-party systems, however, raise questions about the privacy and security of user data. In particular, to what degree can we trust various third-parties with user data? And as a corollary: are there mechanisms that allow us to control or reduce this degree of trust?

3.1 Modern Use Cases

Before we can answer these questions, it's important to note that any proposed solution that fails to support modern use cases is unlikely to succeed in a market where users are voluntarily turning to third-party services for the features they can provide. Thus, we must start by understanding the predominant modern use-cases. We can divide these cases into two categories: consumer focused use cases and developer focused use-cases. Consumer use cases are those that matter to the average lay computer user. Likewise, developer use cases are those that matter to back-end developers and service providers. Both set of use cases represent requirements that modern security and privacy enhancing solutions must be able to accommodate if they are to be widely adopted and used.

3.1.1 Consumer Use Cases

Today's end-users expect modern software to support a range of common behaviors. Chief amongst these are the ability to support the use of multiple devices per user, the ability to support collaboration and sharing with other users, and the ability to provide turn-key data processing capabilities or other services which act on user data.

3.1.1.1 Multi-Device

It's not uncommon for a single user to utilize multiple computing devices. For example, a user might have a personal laptop, a work desktop, a smart phone, and a tablet. As such, users expect to be able access their data from any of their devices. Similarly, many modern users treat compute devices as disposable: when a user loses a phone or has a laptop stolen, they still expect to be able to continue to access their data on a replacement device. The multi-device and ephemeral-device nature of the average consumer has lead to the rise of a number of solutions that aim to separate a user's data from any single device and to ensure that users may access their data regardless of device. Such solutions can generally be placed into two groups: services that sync user data between devices (i.e. sync services) and services that store all user data in a centralized and globally accessible location and provide a manner for users to access this date from each device (i.e. locker services).

Sync services operate on the premise that user data will be stored locally on each device, but will also be automatically kept in sync across multiple devices. They ensure that the users has the same view of their data regardless of device, even when each device stores independent, localized copies of this data. Such services generally accomplish this by providing either a centralized or a decentralized service that tracks user data on each device and updates data across all devices when it is added, removed, or modified on any single device. Sync services are often a desirable solution to the multi-device data access problem for several reasons:

Bandwidth Efficient: Sync services only require Internet access to update data between devices. The act of reading data already present on a device only requires access to the local copy, avoiding the need to consume bandwidth communicating with an external server. This is a desirable quality in situations where bandwidth is at a premium and would either be cost prohibitive or performance restricting to consume every time data needed to be read. Sync services tend to be bandwidth efficient even when modifying data since they can cache a series of updates locally, only syncing the final differential state to other devices.

Offline Support: As an extension to the previous point, sync services are capable of operating in situations where no Internet or network connection is available. Since all data access is available locally, user may continue to access and modify data even when they can not connect to the sync service. The sync service will simply wait for the network connection to return and then update any local changes on other devices. While this can lead to issues when users make conflicting modifications on multiple devices while offline, in general it represents a more graceful failure model then a system that requires an Internet connection for any form of data access. This also acts as a hedge against a sync service provider shutting down: even if a sync service goes under, the user will still retain local copes of all their data that they could use to bootstrap a new sync service.

No Central Storage: Since sync services are only concerned with syncing changes between devices, it's not inherently necessary for such services to store a copy of user data in a central location. This allows such systems to be built using distributed device-to-device designs when desired. In practice, many sync systems do store a copy of all user data in order to facilitate the bootstrapping of new devices, but this is not an inherent requirement of the sync service architecture.

One of the main challenges to sync services is their local storage requirement. While such a requirement allows for some of the benefits listed above, it also limits the total available storage afforded to each user to the size of their smallest device. Most sync services offer partial-sync

options to help mitigate such issues, but these tend to be burdensome to configure and are often relegated to the purview of advanced users. In situations where a user wishes to store more data then can be fit on any single device, locker services may offer a more desirable solution.

Locker service operate by storing all user data on a central server and then providing mechanisms for users to access and modify this data there. Such services are reminiscent of more traditional networked file systems such as NFS [208] or SMB [156]. Data locker services store all user data in a logically-centralized and globally-accessible location where each individual user device may access it for the propose of reading, modifying, or deleting data. Since such systems only have one logical copy of each file, the user is presented with a single view of their data regardless of which device they chose to access it from. Such services have several desirable qualities:

No Local Storage: Locker services store all data in online locations, generally atop data-center-based servers. Thus, unlike sync services, they require no local storage. This is useful in situations where local storage is limited (e.g. on phones), but where the user still wishes to have access to a large amount of data (e.g. a video collection). Likewise, locker services avoid wasting unnecessary local space by creating multiple copies of each file on each device. Additionally, the lack of local copies may be desirable in situations where local devices are prone to theft or may otherwise not pose a reliable and secure platform for the storage of all user data.

Single Source of Truth: Locker services only store a single (logical) copy of the data at any time. This ensures that situations where the user makes conflicting modifications to a piece of data are far less likely then in sync-based solutions. This property can increase the usefulness of such systems in multi-user scenarios where more then one person might be accessing and modifying data simultaneously.

Centralized Control: Due to the manner in which most locker services store their data, such services often provide more centralized control point then a potentially distributed sync

service. Such control may be desirable in corporate environments where a single administrator wishes to track user file access, modifications, etc.

One of the main downsides to locker services is their requirement for always-online access. Thus, in situations where network access is impossible or where high-bandwidth usage is not practical, locker services can prevent users from access their data. Thus, locker services may operate best on local area networks and in other situations where network bandwidth is reliable and plentifully. In situations where network access is not guaranteed, sync services may offer a more desirable solution.

Today, sync services tend to be the more popular solution for most end-users. Such users generally want access to their data across multiple devices and in multiple locations (home, work, public, etc) – requiring the data to be available in locations where a reliable network is not always available. Thus, sync service dominate multi-device data access solution landscape. Examples of popular centralized sync services today include Dropbox [42] ($\approx$ 300 Million Users [227]), Google Drive [88] ($\approx$ 240 Million Users [227]), and Microsoft OneDrive [155] ($\approx$ 250 Million Users [227]). An examples of a decentralized sync service is BitTorrent Sync [19] ($\approx$ 2 Million Users [227]).

Locker services are also available and tend to be most popular in situations where network access is reliable and where centralized control is desirable: e.g. business environments. In such situations traditional in-house networked file system solution may provide the locker service. Alternatively, there are cloud services that provide users with online locker-like data access: e.g. systems like OwnCloud [181] implement the WebDAV protocol [77] for remote file access over the Internet. Similarly, distributed systems like Least Authority's Simple Secure Storage Service [137] (S4, built atop Tahoe-LAFS [265]) offer Internet-wide multi-device access to a distributed data store. Furthermore, some sync services are also capable of operating more like locker services. Systems like Microsoft OneDrive allow users to specify which files are copied locally (and thus available for offline access) and which are stored only on the server and streamed to the user as required [157]. There have also been a number of popular special-purpose locker services designed

to promote multi-device access to specific classes of data. In the media space, systems like Google Music [91] allow users to upload music files which they can then access and stream to multiple devices. Similarly, distributed systems like Popcorn Time [258] allow users to (often illegally) tap into each others video libraries to stream content to their own devices.

As shown above, users' ownership of multiple computing devices has lead to a strong desire for users to be able to access their data from any device. In response to this desire, a number of third-party backed services have sprung up to provide users with multi-device file access. Any technology aimed at enhancing the security or privacy of end-user data is going to need to account for and support the multi-device nature of modern users.

3.1.1.2 Multi-User

In addition to using multiple compute devices, many users today desire the ability to share and collaborate with other users. In response to these desires, many cloud services offer various mechanisms for multi-user sharing and collaboration. Similar to the multi-device use case, the solutions in this space can be roughly grouped into two categories: distributed services that allows users to share copies of data with other users and centralized services that allow multiple users access to a central copy of the data.

In many cases the same solutions discussed previously that enable the multi-device use case also provide multi-user sharing capabilities. This is true of sync service like Dropbox or Drive that not only allow users to sync files amongst their devices, but also allow them to share files with other users. In many ways, this is just an extension of the sync service model with the extension that now users may also include devices other then their own in the sync set. Similarly, locker-style multi-device solutions often include support for multi-user use cases. For example, traditional networked file systems like NFS provide both multi-device access and multi-user support. Likewise, systems like Least Authority's S4 provide primitives for sharing files with multiple users.

Unlike the sync use case, however, adding support for multi-user sharing requires providing access control primitives in addition to basic file syncing primitives. These primitives allow users

to control the manner in which other users may access and use the shared data. Such controls are necessary to allows users to place limits on the degree to which they trust other users. Many services provide fairly traditional file-like access control schemes were each user may be granted read and/or write permissions to a specific piece of data. Data owners can use these permission to craft access control polices for the group of user with which they wish to share data.

It's also common to see support for various forms of multi-user sharing in a range of hosted services, from social networking apps to web-based document editors. Social network platforms like Facebook [53] have extensive support for sharing photos, videos, status messages, and other user-generated continent. Such systems also provide the data originator with the ability to place limits on how data is shared and who it is shared with (although the effectiveness of such access control settings is often questionable [113]). Similarly, systems like Google Docs [87] or Microsoft Office Online [154] offer users the ability to interactively compose and collaborate on document creation and editing. Such systems are inherently multi-user, generally giving the user the ability to chose who else can view and edit each document.

Multi-user use cases are a key component of many computing systems today. As in the multi-device cases, support for multi-user scenarios is an important component of any privacy and security enhancing technology. Technologies that lock the user out of such use cases are unlikely to be effective solutions for most users today.

3.1.1.3 Hosted Services and Processing

In addition to the multi-device and multi-user scenarios discussed above, many user also expect support for various hosted services and data processing solution. In many ways, this expectation follows from users' multi-device and multi-user expectations: whereas the traditional computing model involves users running locally installed applications for the purpose of processing or interacting with data, such a model fails to properly account for the multi-device and multi-user requirements of many modern applications. Thus, data processing services that would have traditionally been installed locally, are now run as hosted services atop third party infrastructure. Using

such services requires users to be able to share data with third parties for the purpose of leveraging such services. Unlike pure multi-device syncing or multi-user sharing services, data processing service provide some benefit to the user above and beyond the mere storage, transfer, or sharing of data.

Examples of popular hosted services that interact with user-generated data include the social networking and document editing solutions mentioned previously. In both cases, these services take user data and leverage it to provide an additional benefit to the user: e.g. the ability to interact and communicate with one's friends or the ability to create and expand written documents with a colleague. Other examples of hosted services include various "Big Data" systems that leverage vast swaths of user data to provide insights into user behavior or patterns in user actions. For example, "Internet of Things" (IoT) devices that enable a user to track things like their day-to-day power consumption [165] or record their exercise habits [57] are becoming increasingly popular. The data from such devices in generally passed back to third party processing platforms where useful insights are drawn from it and returned to the user. Often such systems leverage their access to data form a multitude of users to return more useful information then the data from any single user could provide. It seems likely that such services will continue to increase in popularity as the cost of deploying IoT devices drops and the collective benefits of access to large data sets grows.

Modern privacy and security enhancing technologies must account for the fact that many users may wish to leverage hosted third party data processing services. Such technologies should provide users with the ability to share data with data processing services in a controlled manner and to transparently audit the manner in which such services are using the shared data. Failure to support such scenarios will negate the benefits for any privacy and security enhancing technology across many current and future use cases.

3.1.2 Developer Use Cases

Beyond end-user use cases, developers are also heavy users of modern third party cloud services. As such, there are a number of backend use cases that would also benefit from privacy and

security enhancements with respect to third party trust. Giving developers the tools to better protect cloud-backed systems allows them to build more secure end-user services. Furthermore, developers are often some of the heaviest users of cloud services, so ensuring that they can adequately protect their data and services hosted atop third party infrastructure significantly expands the total number of computing systems protected. I discuss several common back-end use cases here.

3.1.2.1 IaaS and PaaS Infrastructure

Many production-level systems deployed today run atop third-party cloud IaaS (Infrastructure as a Service) and PaaS (Platform as a Service) systems. This fact leads to two main consequences that must be considered when designing security or privacy enhancing technologies: lack of full-stack control and the need to scale dynamically.

Traditional security and privacy enhancing technologies often rely on full control of the entire deployment stack, from the raw hardware all the way through the user-facing software, in order to guarantee any level of security. In a world where most developers rely on IaaS and PaaS systems for production deployment, such full stack control is generally not possible. The less trust a developer must place in their IaaS or PaaS providers, the more direct control they retain over the security and privacy of their applications. Modern security and privacy enhancing technology systems should be capable of operating securely even when the underlying hardware or platform lies outside of the developers full control.

Additionally, cloud-based deployments are often scaled up and down dynamically as load requires. An application that begins running atop a single virtual machine may need to scale up to 10s or 100s of virtual machines as the load increases. Modern cloud platforms are designed to support such scaling. Thus, any privacy or security enhancing systems designed to protect such systems must also be capable of rapid and dynamic scalability. Failure to support such dynamics will make it difficult for developers to adapt a given security and privacy enhancing technology in an IaaS/PaaS based world.

3.1.2.2 Remote, Headless, and Automated

It is not uncommon for developers to be working atop remote infrastructure when utilizing PaaS and IaaS systems. As such, security and privacy enhancing technologies can should not make assumptions about a user's ability to physically access a machine. Such physical access is sometimes required by security enhancing technologies for the purpose of bootstrapping various encryption systems using SmartCards, USB drives, etc. Unfortunately the remote, cloud-based nature of many modern systems do not allow for such physical-access dependent mechanisms.

Similarly, most IaaS-backed servers or PaaS-backed services are expected to autonomously operate headlessly (i.e. without a human operator present) for long stretches of time. Thus, it is not appropriate to expect developers to be able to provide interactive keyboard input in support of a security or privacy enhancing technology. For example, most existing full-disk encryption systems require a user to enter a pre-boot password each time the system reboots. Such systems can not operate in a modern cloud-backed environment and thus are not useful use in such scenarios. Furthermore, most modern cloud infrastructure is bootstrapped automatically by developers using systems like Puppet [131] or Chef [179]. Thus, modern security and privacy enhancing technologies must also offer support for automated configuration.

Assumptions about a developer's ability to physically access a machine, to interactively provide input to a machine, or to manually deploy each machine no longer hold under the current practice of using ephemeral cloud-backed infrastructure. As such. it's important that any developer-targeted privacy or security enhancing technology avoids making such assumptions, ensuring that it can operate effectively even atop IaaS and PaaS infrastructure.

3.2 Threats to Security and Privacy

The current cloud-computing and third-party solution trends raise a number of security and privacy related questions. To what degree must we trust each service provider to protect our data?

How good our service providers at protecting data? What other threats do modern usage models expose? I explore examples of some of these questions below.

Almost all of the use cases discussed in § 3.1 involve ceding some degree of trust to one or more third parties. This often comes in the form of storing data or executing computations on their third servers. But to what degree is this trust well placed? How likely is such trust to lead to an unintended disclosure or manipulation of private data or a related security failure? Chapter 5 discusses these concepts in more detail. Here I'll present examples of some of the security failures that can occur related to third party trust.

3.2.1 Misuse of Data

One of the main forms of trust we place in third parties is not to intentionally misuse the data we store with them. E.g. Are third parties using our data in unexpected and undesirable ways? Unfortunately, there are a number of examples of such breaches occurring:

Facebook Emotional Contagion Study: In 2014, it came to light that Facebook had engaged in research that involved manipulating what users saw in their news feeds in order to study the effects of one user's emotions on other users [75]. The study was performed on ≈ 700 users without their explicit knowledge or consent. Facebook misused the trust placed in it by its users by leveraging and manipulating their data in unforeseen ways.

Uber User Travel History: Ride-share app Uber recently made headlines when it used the travel-history of a number of its more prominent users to display a live user-location map at a launch party [225]. Similarly, the company also used stored user travel history to compose a blog post detailing its ability to detect a given user's proclivity for "one night stands" [182]. In both cases, Uber leveraged data it had about users in manners users did not approve of or intend.

Target Pregnancy Prediction: In 2012, it became public that Target had developed a statistical system for predicting if its shoppers were pregnant based on the kind of items they bought.

Target would then leverage this data to send customers coupons optimized for pregnant individuals. In one case, this practice even lead to the outing of a pregnant teenage daughter to her previously unaware father [103]. Clearly such outcomes are not within the realm of what most shopper expect when purchasing items at Target.

While not all of these examples are directly related to a user's use of cloud services (or fixable through the use of the SSaaS privacy and security enhancing mechanisms proposed in this document), they do show a range of examples of how third parties can violate the trust placed in them by their users.

3.2.2 Data Breaches

Beyond direct third party misuse of user data, there is also the risk of unintentional leaks of data stored with third parties. This may occur due to a direct attack on third party or through a mistake or oversight on the part of the third party. Thus, even if we trust that a third party won't intentionally misuse our data, we must still question whether or not they are capable of providing adequate protection for our data. Today we are generally fully reliant on third parties to protect the data they store and to faithfully enforce any access control or sharing restrictions we specify.

Unfortunately, there are many examples of third party data breaches resulting in the unintended release of user data. 2014 alone saw the release of over 550 million user identities and associated data online, representing about 1 in 5 internet users globally [245]. While not intentional, such breaches still call into question the degree to which we should trust third parties with our data. Examples include:

Apple iCloud Celebrity Photo Leak: In 2014, a number of celebrity users of Apple's iCloud data storage service [109] were subject to a public release of personal photos they had stored with the service, often in various stats on undress. This leak was the result of a targeted attack on the corresponding users' passwords and iCloud accounts [12]. These attacks appear to have been propagated over several months prior to the public release. While this

leak was not a result of an overt flaw in Apple's iCloud system, the weak default security requirements for iCloud accounts made it relatively simple for attackers to compromise such accounts and steal data.

Anthem and Premera Blue Cross Breaches: In early 2015 two major US health insurance companies were subject to attacks that breached their user records, allowing the release of personal, financial, and medical information on millions of users [128, 129]. While the details of the breaches are not yet public, such attacks demonstrate the risk of trusting a third party with the storage of large quantities of sensitive data.

Heartbleed, Shellshock, Etc: In addition to targeted attacks, third parties are also susceptible to software failures. Prominent examples of such failures include Heartbleed [33], a flaw in OpenSSL [175] that allowed attackers to steal private data from many secure servers, and Shellshock [244], a GNU bash [192] flaw that allowed user to execute arbitrary code on many web servers. Both flaws were widespread and effected large swaths of web-connected sites and services, potentially exposing many users to attacks and data breaches.

Thus, even if we trust the underlying third party provider to properly store and utilize our data, in many cases the data may still be at risk for exposure through attack or oversight.

3.2.3 Government Intrusion

Recent events have revealed yet another threat vector that must be considered when leveraging third-party cloud services: the targeting of such services by various governments for the propose of wide-spread surveillance. In particular, the NSA leaks revealed by Mr. Edward Snowden demonstrate the US government's widespread data surveillance programs targeting popular cloud data providers [97]. While such actions, at least in the US, raise numerous questions of constitutional legality under the 4th Amendment [257], it does not at present change the fact that such searches are known to be occurring. Thus, we are forced to not only consider our trust of the various third

party providers in the cloud, but also our trust of the governments of the jurisdictions in which such providers operate.

Numerous examples of privacy-subverting attacks by government actors have come to light over the previous 5 years. It's worth considering several of these examples as we evaluate how to increase the security and privacy guarantees available atop third-party services. Notable instances of government data introspection include:

PRISM and MUSCULAR: The NSA PRISM program was/is a FISC-approved [256] system for compelling service providers to provide user data to the government [97]. It is believed to be one of the largest mechanisms for the government extraction of user data from various cloud-based services (e.g. Google, Yahoo, Microsoft, etc). Similarly, MUSCULAR was/is a joint NSA and British GCHQ effort to intercept and monitor traffic within Google's and Yahoo's data center to data center networks [71]. Prior to MUSCULAR's disclosure, this intra data center traffic was not generally encrypted, and thus was an ideal point for another entity to intercept and monitor user data. Both cases demonstrate a concerted government effort to access and monitor user data atop popular cloud services.

Lavabit: Lavabit was a private email service with 400,000+ users premised on the idea that poplar free email services such as Gmail lacked adequate security and privacy guarantees (in part due to the lesser legal protections such communications receive under the US Third-party doctrine [241, 242]). In August 2013 Lavabit shuttered its service in service in response to a US government subpoena requiring Lavabit to turn over all of its encrypted user traffic as well as the associated SSL encryption keys necessary to decrypt it [138, 139]. After a legal fight, Lavabit founder Ladar Levison was forced to disclose the encryption keys protecting his service. The Lavabit example shows the government's willingness to compel service operators to aid in the monitoring and collection of user data.

The Great Firewall: Moving beyond US-based government digital surveillance, China has long been known to employ one of the most sophisticated web monitoring and content control

systems in existence [195]. The so called "Great Firewall" effectively monitors all Internet traffic traveling in and out of China, blocking a range of encrypted services that might be capable of subverting such monitoring. Such systems show the willingness of some governments to outlaw certain types of technology in order to ensure government surveillance efforts are not subverted or hindered.

These examples demonstrate the willingness of many governments to ensure they can access and monitor user data in the cloud. When designing security and privacy enhancing systems, we must account for the fact that services providers may find themselves in positions where they are compelled to turn over data (or even collude in its collection) or where they face large scale surveillance of all internal and external network traffic.

3.2.4 Physical Security

The modern usage practices discussed in §3.1 introduce security issues beyond just those related to the trust and exploitation of third-party services. One of the key areas where such issues manifest is in the physical security of modern computing devices. Traditionally the security of a computing device or any data stored there was rooted on the premise that the device itself could be kept physically secure: e.g. an attacker would not posses unrestricted physical access to a device. Modern usage patterns break this assumption in several ways.

First, the multi-device nature of most users increases the number of computing devices potentially storing copies of sensitive user data. This results in an increased physical attack surface. Furthermore, many modern compute devices are inherently mobile, easily carried by the user and moved about. The combination of these facts gratefully increase the likelihood of a computing device storing user data being lost or stolen. As such, it's critical that we design systems that are resilient to data compromise even when they fall into the hands of unauthorized actors. The likelihood of user data compromises occurring due to device loss or theft is far higher today then ever before, and privacy enhancing solutions must account for this fact.

As mentioned previously, physical control over IaaS-based services is also not generally possible. We are thus forced to run many of our modern services atop hardware under another parties control. This lack of physical hardware access has repercussions for modern threat models. To what extent can the hardware provider interfere with or bypass the security of software that runs on their systems? What abilities do they have to introspect data stored on their servers? How much can we trust computations performed on such systems? Thus, modern security and privacy enhancing systems must be designed with the knowledge that the underlying hardware itself may be untrustworthy. This is a significant departure from the more traditional physically-secure-hardware threat model.

Both these cases demonstrate that any security and privacy enhancing systems designed to protect data in either the cloud or atop modern user devices must contend with the fact that the physical security of the devices on which they operate is not guaranteed. We must design such systems with the trustworthiness (or lack there of) of such systems in mind.

3.3 Need for New Solutions

The confluence of modern use cases and modern security concerns place restrictions on the manner in which we must design and build successful privacy and security enhancing systems. These security concerns form the basis for the threat model against which our systems must be able to defend. The requirement that we support modern use cases place limitations on the manners in which we may defend against these threats. These limitations disqualify many existing solutions and underline the need for new approaches.

Traditionally the solution to many of the threats discussed in § 3.2 involve the use of encryption. Encryption is a desirable security primitive since it allows us to protect user data in a manner that is mathematically secure: e.g. encryption does not rely on any system or authority for the enforcement of its security guarantees, it is instead the intrinsic properties of the underlying math itself that give rise to its security claims. Thus encryption and related cryptographic primitives

can serve as the basis for systems that are designed to operate securely atop untrusted hardware or services.

A number of existing encryption systems have been designed and deployed with an aim towards protecting user data in the scenarios discussed previously. For example, full-disk encryption systems such as LUKS [63] protect user data at rest and can help guard against data leaks if a storage device is lost, stolen, or otherwise acquired by an adversary. Unfortunately, such systems fail to account for many modern use cases. In particular, full disk encryption systems generally require the user to interactively supply a pass phrase at boot time in order to bootstrap the decryption of the data on the system. Such requirements are not generally possible to fulfill when using cloud-based infrastructure such as IaaS-backed virtual machines. Simply doing away with such pass-phrases isn't a viable solution either since the security of the entire system rests upon the intended user, and only that user, being able to provide such information. I.e. a full disk encryption system that stores a copy of the user pass-phrase locally to avoid the need for the user to type it in is no more secure then a system without encryption at all. You're either giving an adversary a unlocked box or you're giving an adversary a locked box as well as the key required to unlock it; neither scenario results in a security benefit for the owner of the box.

There also exist various run-time encryption systems that avoid many of the issues associated with using full disk encryption systems in the cloud. In particular, file-level encryption systems like eCryptfs [98] could conceivably be used to locally encrypt user data before uploading it to a cloud service such as Dropbox or Apple's iCloud. Such an action would serve to greatly reduce the degree to which the user must trust either Dropbox or Apple. Unfortunately such actions do not mesh well with many of the desired multi-device and multi-user use cases mentioned previously. For example, encrypting data on one's laptop, storing it on Dropbox, and then trying to access it from another device such as a phone or tablet will fail. This is due to the fact that the keys used to encrypt the data, and that are thus required to decrypt and access the data, only exist on the original laptop, making it impossible to access the data from another device – the very propose behind putting it on Dropbox in the first place. The user could manually move their keys

between devices to overcome this issue, but doing so likely presents an unachievable challenge for most lay users. Furthermore, if the user is capable of easily transferring keys between devices via machines other then Dropbox, why can't they just do that with files as well and avoid Dropbox all together? Likewise, if a user wishes to share data with another iCloud or Dropbox user, they are now also required to exchange the necessary decryption key information out-of-band in addition to enabling the normal sharing mechanisms. Such exchanges are challenging for most users to perform in a practical and secure manner, and requiring them greatly increases the overhead to multi-user sharing, making it unlikely that most users would employ such tactics in the first place.

Thus, I assert that most traditional encryption systems are not good fits for the poplar third-party backed use cases today. Fortunately, in these situations it is not the encryption itself that is flawed. This is fortunate: modern state-of-the-art encryption is quite a powerful tool and it would be a major loss to be unable to leverage it in pursuit of privacy and security enhancing systems. Encryption is a viable tool for helping us reduce the trust we must place in various third parties.

Instead, the issue breaking common use cases today is the lack of secure, usable, and flexible key management systems. Such systems would work in conjunction with traditional encryption solution to allow users to deploy encryption systems while also ensuring that they have access to the associated keys when and where they need them. Likewise, such systems would be tasked with controlling access to such keys to ensure only authorized devices and users may leverage them to decrypt user data. In fact, encryption key storage represents just a subset of the larger secret storage problem: How can a a user securely store various secrets (encryption keys, passwords, personal data, etc) in a manner that allows them to access them when and where desire while also ensuring that no unauthorized access to these secrets is ever allowed?[1] This is the basis of the questions I propose to explore in my thesis.

[1] This question is merely a variant of the Always/Never paradox often encountered when attempting to build secure systems. The problem was probably most famously encountered during the Cold War in the design of nuclear weapons: How can you ensure a nuclear weapon always detonates when its use is intentional, but never detonates when its use is not? [216]

Chapter 4

Related Work

There are a number of related efforts that also seek to fulfill one or more of the goals of this project as outlined in Section 1.3. I discuss some of the more pertinent of these efforts in this Chapter.

4.1 Trust, Threat, and Security Models

Security researchers have developed a number of trust, threat, and security models for a range of computing systems. Some of these models are concerned merely with the technical security of the system. Others expand to account for the many proclivities of human behavior that have bearings on the security and privacy of computing systems. In all cases, such models seek to answer the questions:

- "On what assumptions does the security of a given system rest?"
- "In what manner can those assumptions be violated?"
- "What is the effect of violating such assumptions?"

Security models are a critical part of designing any security system. The security of a given system is only as good as its weakest link. Security models provide an analyses if the various links within and surrounding any given security or privacy enhancing system – links that must be considered when determining the overall guarantees provided by a given system as well as the exposure the system to various threats.

[59] provides an analyses if how trust and related human controls must be considered in the design of any information security system. It builds on work outside the traditional computer science space related to theories of trust and privacy in government, law, and society [30]. These works lay the foundation for trust analysis in computing systems and tie the concept of trust in computing systems to wider theories of trust and privacy as fundamental concepts.

Beyond trust analyses, there are a number of more general computer security models and taxonomies that tend to focus on more traditional technical and operational risks and mitigations. [1] discuses security and threat models for Internet-connected services. [253] focuses on common failures in application-level security. [56] dives into a more abstract analysis of safety and security in computing systems and provides a framework for analyzing software security risks, potential harms, and security requirements. Finally, [32] takes a comprehensive look at operational cyber-security risks across technical, human, process, and external risks.

While all of these efforts provided a basis for trust, threat, and security analysis of computing systems, they do not dive into the specific intricacies of the third-party trust requirements inherent in cloud computing systems. This is a deficiency on which I focus in Chapter 5.

4.2 Minimizing Third Party Trust

Uneasiness with having to trust third parties has led to a number of efforts to reduce, limit, and control such trust. Many of these efforts aim to leverage cryptographic primitives as an alternative to a trusted third party (TPP). As mentioned in Section 2.1, cryptographically-based systems generally require little to no trust in external systems or parties, only in the underlying math. Other efforts use information theory primitives such as the secret sharing schemes (also discussed in Section 2.1) to try to limit how much damage any single TPP can do. Such efforts generally require multiple third parties to collude to accomplish most attacks. In many cases, these efforts also explore the trade-offs between the convenience and capabilities that trusted third party systems can provide and the security risks of requiring one of more trusted third parties. Various third-party trust limiting efforts are discussed below.

4.2.1 Cryptographic Access Control

The primary limitation of all of the access control models mentioned in Section 2.4 is their reliance on a trusted arbiter for enforcement: generally this trusted arbiter is the operating system or third party system in charge of enforcing the access control system. This means that the security of these access control systems is only as good as the security of the system or third party enforcing them. Thus, if the underlying OS or third party is compromised, the access control system falls apart. Likewise, anyone in control of the underlying OS or enforcement system (e.g. an administrator) automatically gains full control over the access control system and the ability to bypass it[1] . This is an acceptable limitations in many situations, especially those based on a centrally managed system with existing physical security and administrative safeguards in place. But in distributed systems or the cloud where physical and administrative control is not guaranteed, a more robust system that lacks this "trusted arbiter" requirement is desirable.

To overcome the need for a trusted enforcement mechanism in access control systems, researchers have turned to cryptographically-based access control systems. [93] and [16] propose several cryptographically-based access control systems. These systems are based on the concept of Attribute-Based Encryption (ABE). ABE schemes allow a user to encrypt a document in a manner such that the access control rules associated with the document are part of the encryption process itself. Thus, in order to decrypt/access a document, a user must satisfy one or more cryptographically guaranteed access control attributes. [93] allows user to encrypt documents that can only be decrypted by users possessing specific attribute polices encoded in their keys. [16] extends this concept to allow documents to be encrypted with a full access control policy embedded in the encryption itself, allowing only users who's private keys meet a generalized set of requirements to access the documents. Both these systems allow the construction of access control systems that do not require any trusted arbiter to regulate access to objects. Instead, the access control policy is enforced by the underlying cryptography itself.

[1] E.g. as in the case of Edward Snowden's collection of large troves of secret NSA data from a facility where he acted as a systems administrator.

Such concepts have not yet been widely deployed in day-to-day use, possibly because of the complexity and computational overhead of building and operating such systems. These systems also still push off the generation, storage, and management of private keys to end users and administrators, raising many of the same key-management challenges discussed in Section 3.3.

4.2.2 Homomorphic Encryption

The rise of the cloud as the home to many modern data processing systems has led to questions about the degree to which third-party cloud providers should be trusted with access to the data processed on their infrastructure. Homomorphic encryption systems are designed to help mitigate this trust. Such systems are designed to perform data processing operations on encrypted data directly, avoiding the need to decrypt it and expose the unencrypted data to a third party.

The previous few years have heralded the arrival of numerous partially-homomorphic encryption systems capable of performing certain classes of data processing and manipulation operations directly on encrypted data. System like CryptDB [187] allow users to search and query encrypted data directly, allowing the storage of such databases on untrusted infrastructure. Other systems like CS2 [117] provided similar protections and capabilities in more generic data storage contexts. Such systems are referred to as partially homomorphic encryption schemes since they only allow a subclass of all possible operations to be performed on encrypted data. Such systems have been fielded and shown to be practical today.

Beyond partially homomorphic encryption schemes, fully homomorphic encryption schemes allow for unrestricted classes of operations to be performed on encrypted data [72]. A number of such systems have been proposed – some of which go so far as to propose the possibility for encrypting computer programs themselves that will be executed by a fully homomorphic processor [25, 24]. Such system are certainly appealing, but thus far, building one with suitably low overhead so as to be practical today has proven elusive.

While homomorphic solutions will likely prove to be part of the solution to the problem of third party trust, they don't inherently solve all trust-related problems. In the practical since,

the available homomorphic systems today are limited to only preforming certain operations. Furthermore, while such systems allow the processing of data atop untrusted infrastructure, they do not provide a solution for user cases where you wish to share plain-text data with other users or otherwise leverage an application that inherently requires data to be decrypted. As in other cases, they also do not provide a general solution for the management of the associated cryptographic keys protecting such operations.

4.2.3 Secure Storage

Beyond data processing, the ability to securely store data atop third party infrastructure while also minimizing the degree of third party trust this entails has been and remains a desirable goal. As such, a number of projects have undertaken efforts aimed at achieving such protections. Secure data storage is one of the most targeted applications by security and privacy enhancing systems. And for good reason – the ability to securely store data is a critical primitive for maintaining the security and privacy of our computing systems.

A number of traditional client-server encrypted file systems exist with the design goal of avoiding server-side trust [121]. File systems like CryptoCache [112], RFS [41], and Plutus [116] are all designed to allow users to store files on servers without trusting the server itself. Other systems like Keypad [69] and CleanOS [248] are aimed at securing data atop user devices and protecting that data when a device is lost or stolen. All of these systems employ various forms of cryptography to obtain their goals.

Beyond traditional client-server secure storage systems, distributed storage systems like Depot [147], OceanStore [130], and Tahoe [265] are all designed to minimize trust in the underlying storage infrastructure. Such systems allow users to distribute and store files across many third-party nodes while also ensuring that the failure, either accidental or intentional, of a subset of nodes does not result in the loss, corruption, or exposure of user data. Depot focuses on data integrity and availability in the presence of untrusted nodes. Tahoe focus on data integrity and privacy in

the presence of untrusted untrusted nodes. OceanStore has elements of both as well as a focus on large scale deployments that provide for properties like data locality to enhance performance.

In many of these systems, however, key management primitives are still ignored or pushed down to the user, leading to usability issues and use-case mismatches. My proposed work could be used to extend such designs to better account for the key management challenges limiting the use of such systems today.

4.3 Enhancing End-User Security

The Edward Snowden leaks, as well as the numerous highly publicized privacy failures of a range of companies from Facebook to Target, have fueled renewed calls for improved security and privacy enhancing technologies aimed at end-users. In response to these calls, a number of organizations have begun offering new classes of security and privacy enhancing tools. Many of these tools share the goals proposed in this work – namely the creation of easy-to-use security and privacy enhancing tools well adapted for modern usage demands.

4.3.1 Communication Tools

Many of the recent security and privacy enhancing tools have focused on securing user-to-user communication. Both the contents of and meta-data associated with such communication has been the focus of many of the recent NSA leaks [220]. In reaction, we've seen the creation of new tools as well as advocacy for the expanded use of existing secure communion tools.

Email is one of the primary communication mediums today. The traditional tools of for securing email, OpenPGP [174] and its various implementation (e.g. GnuPG [122] and Symantec PGP [246]), are not new. But they have seen renewed levels of interest after Edward Snowden revealed his use of such tools and advocated for them as effective NSA counter measures. Unfortunately these tools remain no more usable today then they were 20 years ago, leading to the multitude of usability issues discussed in Section 2.2. In response, tools like Mailpile [49] and Whiteout [262] have been created with the aim of making PGP-based email security more friendly

for the average user. Similarly, Google and Yahoo have both engaged in efforts aimed and making PGP-like encryption and authentication systems available to their webmail users [89, 269]. Other systems such as STEED aim to adapt traditional PGP principles to make the primitives simpler for the average end-user to deal with [123].

But even these recent pushes to re-skin PGP and make it more user friendly can't overcome many of the fundamental issues with the system [95]. Nor are they well suited for securing another major form of modern communion: real-time chat. To overcome these deficiencies, systems like Open Whisper's TextSecure [171] and ChatSecure [14] have been created. Both systems rely on the Off-the-Record secure messaging protocol to achieve encrypted, authenticated, and forward-secure[2] real-time communication between two or more parties [180, 23, 78]. Such systems aim to make secure real-time communication as simple as possible, and have shown promise in terms of popularity and usability. But these systems are not necessarily well suited to replace PGP since they rely on the real-time nature of chat communication to attain forward secrecy. Similarly usable and forward-secure solutions for non-real-time systems like email systems remain elusive.

While secure communication systems are seeing a lot of development and showing promise, all of the solutions discussed above are communication-specific solutions. As such, they are complimentary to the work presented in this proposal related to general methods for reducing third party trust and managing secrets and cryptographic keys.

4.3.2 Password Managers

One of the most common forms of third-party secret storage available today is that of password storage by end-user password managers. The well-documented failure of user-chosen passwords as a reliable security mechanisms [81, 80] has led experts to recommend that users rely

[2] Forward secrecy is a property of secure communication that ensures that the short-term sessions keys used to protect individual messages can not be derived from any long-term authentication or related keys. In short, it guarantees that prior messages can not be decrypted should an adversary manage to compromise a long-term user key at an point in the future. Such systems are commonly implemented by decoupling session keys from user authentication keys, using systems such as Diffie-Hellman key derivation [40] to generate the former while only relying on the latter for the prevention of Man-in-the-Middle attacks during the initial setup process.

instead on a single strong password that unlocks a password management service capable of storing unique and random passwords for each website or service they use [219, 127, 26].

Systems such as LastPass [133] and OnePassword [2] provide users with a hosted platform on which to manage and store their passwords. These systems utilize a single-third party for data storage in order to accommodate the multi-device sync and multi-user sharing use cases presented in Section 3.1. But in doing so, they require the user to place a fair amount of trust in the third-party provider. Such systems do tend to perform client-side encryption and generally are designed to avoid storing the user's master password in a recoverable form, mitigating some of the potential for third party abuse, but at the end of the day, they still require the user to trust a single third party with their credentials, and by proxy, access to their online services and accounts.

Password management systems share some of the same goals I propose in this document – namely, the storage of end-user secrets in a secure and easy-to-use manner. Thus, password storage can be viewed as a special case of the more generic Secret Storage as a Service (SSaaS) model. In this work, I'll present more versatile and generalized solutions to the secret storage problem. Such solution could be used to implement the password manger concept while also decoupling such implementation from needing to trust any single third party.

4.3.3 Storage Tools

Beyond secure communication and password storage lies the more general problem of arbitrary secure user storage. As mentioned in Section 3.1, many user expect such storage to provide multi-user and multi-device support in addition to raw secure storage, disqualifying many of the traditional encrypted file system solutions from the running. Popular storage services like Dropbox [42] or Google Drive [88] are common today, but these solutions provide the associated third parties with unfettered access to user data, making them unsuitable for users who wish their data to remain private.

To overcome issues with systems like Dropbox while still affording users access to modern file-store amenities, systems such as SpiderOak [232], Tresorit [252], and Wuala [132] have been

built with the aim of providing users with a Dropbox-like alternative that does not require trusting the storage provider. Such services shift the encryption of data into the client, helping to ensure the server only ever holds an encrypted copy of said data. None the less, the confidentially and privacy claims made by such services have been shown to be dubious [266], largely due to the fact the a user must still place a large degree of trust in the backing third party in order to facilitate file sharing with other users.

As in previous cases, however, these solutions are purpose built for secure storage, and fail to provide a more general solution for minimizing third party trust. Furthermore, as mentioned above, these services still rely on a moderate degree of trust in a single third party in order to facilitate sharing use cases. The work proposed in this document could be leveraged to re-implement such systems while further suppressing the final vestiges of single-party third-party trust.

4.4 Key Management Systems

As mentioned in Section 3.3, one of the main hindrances to bootstrapping secure systems today is the lack of versatile key-management systems capable of supporting modern use cases while also minimizing third-party trust. Such key management systems are subsets of the larger secret storage problem discussed in this proposal. I am not, however, the first to recognize key management as one of the critical components to building flexible and secure privacy and security enhancing systems. Several related efforts are discussed in this section.

4.4.1 Key Management in Storage Systems

As discussed previously, many existing secure storage systems fail to meet the needs of today's users due to the fact that their restrictive and tightly coupled key management systems fail to accommodate a range of modern use cases. A number of existing secure file systems have acknowledged the issues created by tightly coupling key management with a specific storage solution (or of ignoring key management all together and forcing the user to deal with securing their keys

themselves). These systems represent a step toward more flexible key management, and by proxy, more usable secure file storage.

Systems like Plutus [116] are designed to isolate key management as a dedicated system component. While this is the first step toward solving many of the key management problems, Plutus simply shifts key management into the client, providing some degree of user-facing key management flexibility, but failing to provide a standardized and general key management system. SFS [149] takes this separation a step further by proposing various standardized mechanisms by which a user might manage keys and an externally-facing interface for doing so. But even SFS fails to define a general system for key management, instead focusing on mechanisms that allow the user to select their own key management system.

A system such as SFS, however, could be interfaced with a general secret storage systems such as those proposed in this document. Such applications of standardized secret storage systems are discussed further in Chapter 8.

4.4.2 Key Escrow

Key escrow systems represent on common form of key management system. key escrow systems exist with the aim of fitting client-side key management into hierarchical organizational frameworks where upper level members may need access to subordinate's keys. Such systems have also been proposed in regulatory environments where law enforcement or regulatory personal must be able to access certain forms of otherwise secure data. Escrow systems also prove beneficial in situations where a key holder would like to hedge against the loss of their copy of the key without having to fully trust any other single party with a backup copy. Such systems often leverage secret-sharing schemes such as Shamir [222] to accomplish their goals.

Early key escrow systems merely provided a trusted third party with a secondary copy of a given data encryption key (and/or generated data encryption keys in a manner such that two separate keys could be used to decrypt any given piece of data) [38]; e.g. the infamous NSA-backed Clipper chip. These systems, however, fail the single-trusted-third-party test, requiring

a high degree of user trust in one external entity. To overcome such trust concentrations, Blaze proposes a distributed key escrow system that makes key escrow requests transparent and leverages a distributed consensuses to avoid any single rogue actor using such a system to steal a key or gain unauthorized access to end-user data [22].

While distributed key management systems such as those proposed by Blaze succeed in avoiding trust in any single third-party, they fail to provide for a more general key management system that moves beyond traditional escrow-based use cases (e.g. key sharing for multi-device sync or key sharing for document collaboration). Furthermore, the existing scholarship around key escrow systems fail to explore the benefits and possibilities of the general purpose secret storage systems described in this proposal. The secrets storage systems described in this proposal can, however, fulfill many of the traditional roles of a distributed key-escrow system.

4.4.3 Cloud Key Management

Perhaps the most closely related work to that proposed here is the various cloud-backed key-management systems introduced over the previous five years. These systems are a reaction to many of the developer-facing use cases discussed in Section 3.1. They aim to make key management and secure key storage primitives available to developers leveraging third-party cloud platforms.

Rackspace's CloudKeep[191, 189] (now OpenStack Barbican [176]) aims to create a standardized key management system for use across multiple applications, avoiding the need to re-implement such systems in each application. Similar to the SSaaS system proposed in this document, CloudKeep aims to ease developer burden while increasing the security of end-user applications by focusing security code in a centralized, carefully curated system. Similar commercial systems also exist with the aim of providing developers with turnkey key-management systems [68, 188, 199]. Such systems, however, lack the generic secret-storage flexibility of the systems discussed here. Most also still require placing trust in a single third party – although this party may now differ from the party proving the underlying cloud compute or data storage servers. They also lack the cross-platform

standardization necessary to incentive some of market-driven forms of trust-preserving behavior discussed in Chapter 6.

Finally, various IaaS platforms have begun offering cloud-based Hardware Security Module (HSM) solutions. HSMs traditionally refer to dedicated hardware co-processors capable of storing user cryptographic keys in a manner that prevents them from ever being extracted. Such processors are then tasked with performing all necessary cryptographic operations requiring such keys on the user's behalf. Standardized interfaces such as PKCS11 [51] exist to allow programs to communicate and utilize such hardware. The benefits of such chips lie in their ability to securely store cryptographic keys and to perform cryptographic operations without exposing such keys to the shared computer memory space, potentially leaking them in the process (e.g. as in Heartbleed [33]). Amazon has recently begun offering a cloud-backed HSM service that aims to make the benefits of dedicated HSM hardware available in the virtualized world of the cloud [4]. Such systems, while helpful for developers who wish to build HSM-dependent systems that operate on local or cloud hardware, still require placing a high degree to trust in a single third party to faithfully operate such virtual HSMs. The SSaaS work discussed in this proposal could potentially be used to build similar "soft" HSM systems without requiring single third-party trust [145].

Chapter 5

An Issue of Trust

Trust and privacy are closely related qualities in computing [59]. Can we maintain our digital privacy without having to trust anyone? We can imagine scenarios that maintain privacy without trust, but they generally involve only storing data on self-designed, built, and programmed devices that never leave our possession. Such arrangements are, at best, impractical for the vast majority of users, and at worst, simply not possible to achieve today. The range of manufactures, suppliers, and service providers inherent in the modern computing landscape require that we make decisions regarding whom to trust at every step of any digital interaction in which we partake.

The cloud computing model, by its very nature, further amplifies the number of parties we must trust. But what does it mean to trust a third party in the cloud? Before discussing the details of an SSaaS ecosystem, it's helpful to further define "trust". In this chapter, I define a model for analyzing trust and apply this model to compare traditional cloud-backed offerings to SSaaS-backed offerings.

5.1 Analyses Framework

Generally, when we use the cloud, we must trust third-party service providers with our data. The manner in which this third party trust relates to the privacy of user data has two main factors: how much trust do we place in third parties (e.g. how much of our personal data do we grant them access to), and in what manner can they violate this trust (e.g. how can they abuse the access they are granted). I will thus evaluate cloud trust models across two main axes: the **degree** of trust we

must place in third parties, and the manner in which this trust might be **violated**. Our ideal trust model for a given use case will minimize the degree of third party trust while also minimizing the likelihood that such trust will be violated.

In terms of degrees of trust, we can entrust third parties with the following data-related capabilities:

Storage (S):

Can a third party faithfully store private user data and make it available to the user upon request? Misuse of this capability may result in a loss of user data, but won't generally result in the exposure of user data.

Access (R):

Can a third party read and interpret the private user data they store? Misuse of this capability may result in the exposure of user data.

Manipulation (W):

Can a third party modify the private user data to which they have access? Misuse of this capability may result in the ability to manipulate a user via the manipulation of their private data (e.g. changing appointments on a user's calendar, etc).

Meta-analysis (M):

Can a third party gather user metadata related to any stored private user data? Misuse of this capability may result in the ability to infer private user data (e.g. who a user is friends with based on data sharing patterns).

I'll define a trust violation as occurring when a third party exercises any of the above capabilities without explicit user knowledge and permission. Put another way, a trust violation occurs whenever a third party leverages a capability with which they are entrusted in a manner in which the user does not expect the capability to be leveraged.

I define several types of trust violations based on the manner in which the violation occurs and the motivations behind it:

Implicit (P):

This class of trust violation occurs when a third party violates a user's trust in a manner approved by the third party. An example might be sharing user data with a business partner (e.g. an advertiser). Often these forms of violations aren't really "violations" in the sense that a user may have clicked though a Terms of Service agreement that granted implicit permission for such use, but if the third party is engaging in behavior that the user would not generally expect, an implicit trust violation has occurred.

Compelled (C):

This class of trust violation occurs when a third party is compelled by another actor to violate a user's trust. The most common example would be a third party being forced to turn over user data or records in response to a request from the government with jurisdiction over the party.

Unintentional (U):

This form of violation occurs when a third party unintentionally discloses or manipulates user data. An example would be a coding error that allows either the loss of or unfettered access to user data.

Insider (I):

This class of violation occurs when a privileged adversary within the third party violates a user's trust without the permission or knowledge of the third party. An example would be a cloud service provider employee accessing or disclosing private user data without authorization.

Outsider (O):

This class of violation occurs when an external adversary gains unauthorized access to private user data stored by third party. An example would be an adversary exploiting a bug in a third party's authorization infrastructure to gain unauthorized access to user data.

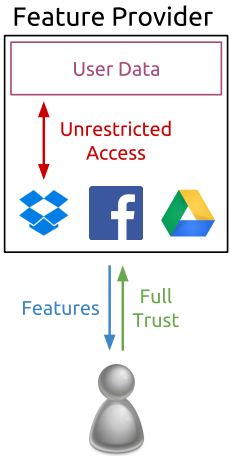


Figure 5.1: Traditional Trust Model

5.2 Traditional Model

Existing cloud services are not generally well optimized for maximizing user privacy. Such services tend to apply an all-or-nothing trust model where a user must seed a high degree of trust to each service in order to reap the benefits each service provides. Figure 5.1 shows the basic trust-for-features relationship between a user, their private data, and a traditional cloud service provider.

We can analyze the Dropbox example from § 1.2 using my proposed framework. To begin, what capabilities is a normal Dropbox user entrusting to Dropbox? Clearly, users must trust Dropbox to faithfully store their data since that is Dropbox's core purpose, so we have granted Dropbox the **S** capability. Furthermore users must also grant Dropbox the ability to read and access their data (i.e. the **R** capability) in order to support Dropbox's sharing and syncing features. While Dropbox doesn't generally utilize it, users are also effectively granting the manipulation (**W**) capability as well since the user has no mechanisms for ensuring that Dropbox can't manipulate their data. Finally, Dropbox has full access to user metadata related to their usage of the service, granting them the **M** capability. Thus, Dropbox users are fully trusting Dropbox with all possible trust capabilities.

But how likely is it that Dropbox might misuse any of these capabilities, thus violating the user's trust and privacy? In terms of implicit violations (**I**), Dropbox charges users for storage, and thus shouldn't generally rely on reading or sharing user data for advertising purposes. Furthermore, such a business model relies on Dropbox remaining in its paying users' good graces, disincentivizing potentially questionable behavior. None the less, there is evidence of Dropbox opening user files for unknown reasons [261], which might indicate a possible **I** violation. The user has no way to prevent an **I** violation when using Dropbox, so we can do no more than give Dropbox the benefit of the doubt in this area.

In terms of compelled (**C**) violations, Dropbox is a US-based company, and is thus susceptible to a variety of government-sponsored data requested, from subpoenas issued under the Third Party Doctrine [251], to probable cause search warrants [257], to National Security Letters [55], to FISC [256] orders. Dropbox publishes a transparency report [44] indicating how frequently they are compelled to violate user privacy. Recent versions of this report indicate that Dropbox receives a few hundred requests for various user data every 6 months.

Unintentional (**U**), Insider (**I**), or Outsider (**O**) violations are all possibilities when using Dropbox. On the **U** front, Dropbox had an incident in 2011 that allowed anyone to log into the service using any password for a 5 hour period [46]. Thus far, Dropbox appears to have avoided any **I**-type violations, but it has been the target of various **O**-type attempted violations, primarily built around advisories who obtain common user passwords [45]. Needless to say, while Dropbox works to avoid these kinds of violations, they are certainly still possible, have occurred in the past, and may well occur in the future.

As we can see, a traditional cloud service like Dropbox both requires an essentially full degree of user trust (i.e. **S**, **R**, **W**, and **M** capabilities) while also being susceptible to a full range of trust violations (e.g. **P**, **C**, **U**, **I**, and **O** type violations). Dropbox is not unique. Other modern cloud services from social media sites, to file lockers, to communication systems all suffer from the same high-trust, high potential for violations paradigm.

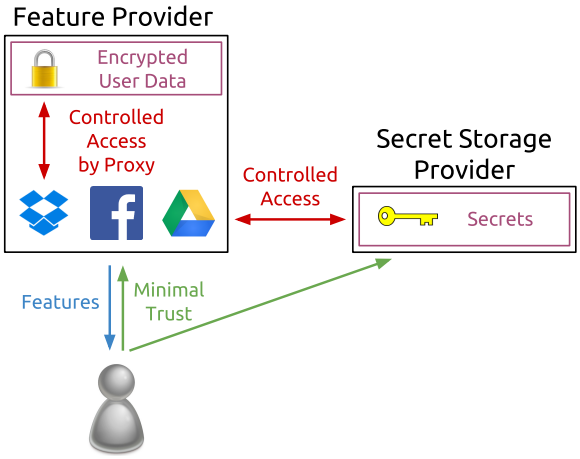


Figure 5.2: SSaaS Trust Model

5.3 SSaaS Model

In order to enhance user privacy in the cloud, we must move beyond the traditional full-trust, easily violated model presented in § 5.2. Of the two components of my trust framework, degree of trust and potential for violation, degree is the easier to control quality: what capabilities to trust a cloud provider with is largely within the user's control, where as the method in which trust might be violated is largely outside of a user's control. I will thus focus my solution on mitigating degree of trust first while disincentivizing methods of violation second.

The ideal trust model begins with the principle of least privilege [205]: we should only afford a cloud provider the minimal degree of trust necessary to provide the intended features. Furthermore, in order to monitor cloud providers for potential trust violations, we would also like to maintain some degree of auditing over any capability we entrust to a provider. Minimal, audited trust is the basis of my ideal trust model.

Applying this ideal trust model to the previous Dropbox example, I conclude that Dropbox should really only be afforded the storage (**S**) capability: after all, there is no real need for Dropbox to ever be able to access (**R**) or manipulate (**W**) user data in order to blindly sync files across device or share them with other users. While Dropbox does not need the metadata (**M**) capability to simply support the desired user case, it is far more difficult for the user to limit this capability

then the **R** or **W** capabilities. I thus will also concede this capability to Dropbox as well. We have thus reduced Dropbox's capabilities from four (full trust) to two (minimal trust).

But how can we enforce this limited trust profile when using Dropbox? Figure 5.2 shows my proposed SSaaS solution to this problem. I introduce a new actor: the Secret Storage Provider (SSP) into the mix. I then restrict the user to only storing encrypted and authenticated data with Dropbox. The user stores the associated encryption and verification keys with the SSP. Assuming we utilize strong encryption and integrity systems like AES [162] + CMAC [47], Dropbox can not feasibly decrypt and access the user's data nor can they manipulate user data without detection. Storing the keys with an SSP, as opposed to simply forcing the user to maintain them manually, has a number of benefits: namely, it affords users continued support for simple sharing and syncing use cases. As long as the user's Dropbox client can communicate with the SSP from any location where they can also communicate with Dropbox, the user can continue to utilize Dropbox as they traditionally would, but with significantly less trust in Dropbox itself.

But have we simply replaced one third party with another? Why is the SSP any more trustworthy than Dropbox itself? There are several reason why we might expect an SSP to be less prone to trust violations then a feature provider like Dropbox. I'll discuss these in §6. But in terms of degrees of trust, we really aren't affording the SSP any greater degree of trust then Dropbox. Both are entrusted with the **S** and **M** capabilities, but neither has the **R** or **W** capability: Dropbox has limited capabilities because it only holds encrypted and authenticated user data, and the SSP has limited capabilities because it holds no direct user data at all, only the cryptographic keys used to protect such data. As long as the SSP faithfully guards the secret keys they store, the trust model holds. There are a few new risks in the SSaaS model, however. If both the feature provider (e.g. Dropbox) and the SSP are located in the same regulatory jurisdiction, it may still be possible for a single entity to compel (**C**-type violation) them both or provide data that would allow an adversary to elevate their capabilities to include **R** and **W**. Likewise, if Dropbox and the SSP collude to violate a user's trust, a similar capability elevation will occur. This latter point introduces a new violation type into my framework:

Colluding (L):

This class of violation occurs when multiple trusted parties collude to gain capabilities over user data beyond what the user intended each individually to have. An example would be an SSP sharing the user's encryption keys with the feature provider storing the corresponding encrypted data.

This framework provided the basis for analyzing third party trust as well as the basic argument in favor of SSaaS as trust-reducing system. I discuss SSP trust and violation mitigation further in Chapter 6.

Chapter 6

Secret Storage as a Service

As discussed in Chapter 3, the reliance on third parties inherent to many popular use cases poses a number of privacy and security related challenges. Fortunately, cryptographic techniques including encryption and authentication provide the necessary primitives for building a systems that increases the security and privacy of users by reducing their exposure to third party-related abuses of trust. Such systems also provide additional security outside of the traditional third-party risk model by ensuring that systems such as mobile devices that are prone to loss or theft remain secure even when outside the position of their owners. Unfortunately, cryptography is not merely "magic fairy dust" that we can sprinkle on any security or privacy problem to make it disappear [228]. Effectively using cryptographic technique to secure our data involves ensuring that cryptography-employing security and privacy enhancing solution are designed in a secure and usable manner.

The crux to designing secure and usable cryptographic data security solutions lies in providing secure and flexible secret storage systems that can be leveraged to manage and store the cryptographic keys associated with any such system. The failure of traditional cryptographic systems to account for key management has led many such systems to be unusable, insecure, and/or ill-adapted to modern use cases. I propose the creation of a standardized Secret Storage as a Service system designed to provide users with the necessary tools for managing secrets such as cryptographic keys in a manner that allows for a range of use cases and that avoids placing high

degrees of trust in any single third party. I present the design and justification of such a system in this chapter.

6.1 Architecture

Secret Storage as a Service (SSaaS) is a cloud architecture where users utilize dedicated Secret Storage Providers (SSPs) in addition to the traditional Feature Providers (FPs) like Amazon, Dropbox, Gmail, or Facebook. An SSP is tasked with the storage of and access control to a variety of user secrets from cryptographic keys to personal data. In the normative case, users will limit themselves to storing cryptographically-protect data on third-party FP servers while storing the associated cryptographic keys protecting such data with a network of SSPs. This allows SSPs to be selected on the basis of their trustworthiness while traditional feature providers can be selected on the basis of their feature sets. The SSaaS model differs from the traditional cloud model by allowing users to distribute trust across multiple third parties (or no third parties at all), ensuring that any single entity need not be fully trusted, while still enabling many popular use cases.

6.1.1 Stored Secrets

What kind of secrets do we store with an SSP? I believe that users should be able to store arbitrary data with any SSP, allowing open ended secret storage based applications. That said, the SSP model works best when secrets stored are not inherently sensitive or revealing when taken alone. This property helps to mitigate the amount we must trust each SSP. Thus, storing secrets like cryptographic keys that alone revel no private user data are generally preferable to storing privacy revealing secrets like plain-text passwords, social security numbers, etc. I do, however, leave the decision of what to store with each SSP up to each user and application. I'll explore various types of secret storage in Chapter 8.

Another consideration related to what secrets to store with an SSP is size. I anticipate SSP-based storage selling at a premium price vs more traditional cloud storage options like Amazon S3 [7]. This is due to the difference in priorities between Secret Storage and generic cloud storage.

An SSP is primarily concerned with safeguarding user secrets and faithfully implementing a user's access control specifications for each secret. These priorities may very well incur additional costs not present in more traditional cloud storage environments: e.g. the need to locate data centers in specific legal jurisdictions, a greater emphasis of resistant to compelled violations via legal representation, etc. Thus, it may be desirable for the user to minimize the amount of data stored with an SSP as a cost optimization: again making use cases such as storing cryptographic keys with an SSP while storing the encrypted data with a more traditional provider desirable.

For these reasons, I feel that storing cryptographic keys with an SSP is a common enough use case that some SSPs may specifically optimize for it. Such "Key Storage as a Service" (KSaaS) SSPs represent a subset of the general SSaaS model.

6.1.2 Secret Storage Providers

In the SSaaS model, SSPs will offer a standard set of features. These include a standardized interface, access control primitives, and auditing capabilities. These features provide the basis of building privacy-preserving SSaaS-backed applications.

6.1.2.1 Secret Storage

At its core, an SSP provider is offering a key:value data storage model. Each secret is tagged with a key: a unique identifier, potentially a UUID [135] or similar unique ID standard. The value associated with each key is then the user secret itself, be it a cryptographic key, personal user data, or arbitrary secret value. Users are able to query each SSP for the value associated with a given ID, or to add new secrets to each SSP.

Optionally, SSPs may provide versioning of each id:secret pair. This may be desirable for use cases where the user wishes to share data with other users while maintaining the ability to revoke shared access to future versions of a data set. Such "lazy revocation" [116] capabilities can be built atop versioning schemes that maintain access control information on a per-version basis.

I foresee each SSP exposing its key-value secret store via a RESTful HTTPS-based API. The ubiquity of RESTful interfaces in modern applications ensures that such an interface will allow simple communication between a client and the SSP across a wide variety of platforms. This interface will expose create, read, modify, delete semantics similar to most existing key:value stores. In fact, I assume that most SSP implementations will use an off-the-shelf key-value store as the backend for storing user secrets. I intend for SSPs to utilize a standard API in order to allow user to interact with multiple SSPs and transfer their secrets between SSPs.

6.1.2.2 Access Control

The SSP data model associates an access control specification with each id:secret pair (or in a versioned system, with each id:version:secret set). This specification governs the manner in which a given secret can be accessed. Such specification will be provided and controlled by individual users for each secret they store with the SSP. The SSP is in charge of faithfully enforcing the access control specification.

An access control specification dictates who can create, access, modify, or delete each secret. It contains information regarding both authentication (how a user proves they are who they claim to be) as well as authorization (what permissions each authenticated user is granted). I foresee SSPs offering a standard access control framework in order to promote interoperability between multiple SSPs.

It's important that the SSP access control model remain flexible. Since an SSP may be asked to store a variety of secrets in support of a range of use cases, the access control model must be expressive enough to avoid artificially limiting the user to specific use cases or secrets. For example, one use case might require a single user to satisfy multiple challenges in order to gain access to a highly sensitive secret while another might allow autonomous access from system possessing an approved token during specific times of day to access secrets used by autonomous processes (e.g. a data-center server booting an encrypted hard disk or using a secure backup system).

In addition to the key:value storage API operations discussed previously, an SSP will also expose API endpoints for manipulating the access control parameters associated with each secret as part if the standard REST API. This interface will allow users to update access control information to allow data sharing with other users, revoke prior shared access, etc. This interface will, in turn, require it's own access control specification to ensure that only approved modifications can be made to any secret's access control rules.

6.1.2.3 Auditing

In addition to access control, each SSP should provide auditing information related to the manner in which each id:secret pair is accessed or modified. This information is useful to the user in order to provide additional transparency into the manner in which secrets are utilized. This auditing can be as simple as basic logging of all secret access or as complex as a system that automatically analyzes access patterns to try to detect anomalies that might indicate potential trust violations.

Auditing information is useful to users for several reasons. In the event that user data or secrets are ever unintentionally leaked or compromised, audit information can provide a valuable indication of the scope of the damage. Furthermore, auditing plays an important role in allowing users to understand the semantics of access revocations: since it's unfeasible to revoke access to data another user has already read (and potentially copied out-of-band), audit information provides a user with the scope of potentially revocable outstanding authorization allowances. E.g. if User A shares a secret with User B by granting them read access to it via their SSP, but then decides they'd rather revoke that access, User A can check the audit logs to determine if User B has yet accessed the shared secret and thus whether or not guaranteed revocation is even possible.

As in the prior cases, an SSPs auditing capabilities will need to be exposed via a REST interface to allow client applications to leverage audit data. Likewise, audit API functions will need their own set of access control specifications in order to control who has access to audit

information or the ability to delete that information. As before, standardizing this interface is desirable from an SSP interoperability standpoint.

It may also be desirable for SSPs to employ some form of publicly-verifiable audit trail, similar to the concepts discussed in [22] or [134]. Alternatively, such systems might today be constructed using block-chain-based primitives such as those available in the BitCoin crypto-currency network [161]. Such systems might provide more robust variants on the "warrant cannery" concept that has recently become popular amongst a range of third-party services providers as a counter measure against secret warrants, subpoenas, and court orders [178]. The semi-centralized nature of SSPs make them a desirable point at which to audit and detect unauthorized access requests for user data.

6.1.3 Clients

While SSPs form one half of the SSaaS architecture, the other half is formed by clients connecting to and leveraging data from SSPs. Chapter 8 discusses potential SSaaS use cases and applications in detail. I outline some of the basics of SSaaS client deign and operation here.

An SSaaS client is any system designed to connect to and utilize a Secret Storage Provider service. Clients communicate with one or more SSP via the SSaaS API. Clients can store and retrieve secrets with each SSP, managing secret access control settings, and retrieve secret access audit info. Examples of SSaaS client applications include encrypted file systems, secure communication systems, dedicated crypto-processing systems, etc. Any system that stands to benefit from offloading secret storage and management to a dedicated system, either for the purpose of gaining benefits from the logically centralized nature of an SSP (e.g. for the purpose of accessing secrets from multiple devices or for sharing them with multiple users) or simply to avoid implementing a full secret management and access control stack locally, is a good candidate for integration into an SSaaS architecture.

Is the simplest case, each SSaaS client communicates with a single upstream SSP via a standard SSaaS protocol. The standardized protocol allows the end-user to specify which SSP they

wish to use, and to change SSPs as desired. The downside to such arrangements lies in the fact that storing each secret with only a single SSP raises both trust and availability concerns (the details of which are discussed below). To overcome such concerns, I believe it will often be desirable for each SSaaS client to interact with a network of several upstream SSPs. In this situation, each secret can be sharded between multiple SSPs - increasing reliability while also reducing trust.

Such multi-SSP arrangement, however, raise client side management issues. How are access control rules shared between SSPs? How does the client keep multiple SSPs in sync? And so on... Answering such questions is some of the work I propose to undertake in Chapter 9. The complexity of such solutions will likely call for the creation of a standardized SSaaS client library that can be utilized by multiple applications. Such a library avoids the need for each SSaaS client to reimplement SSaaS communication and management primitives directly. Furthermore, it may also be desirable to create a common SSaaS management application. Many of the SSaaS-related management features, e.g. setting up access control requirements, checking audit logs, etc, are common across all SSaaS client applications. It may thus be desirable to offload such functions to a general SSaaS client management applications in cases where such functions need not be tightly coupled with a given SSaaS-backed applications. Such offloading allows the primary SSaaS-backed applications to focus purely on the secret storage and retrieval side of the SSaaS API while dedicated management applications handle the access control and auditing side of the SSaaS API.

6.2 Economics

Part of the the argument in favor of the SSaaS model is economics. Today, users primarily select cloud services on the basis of their features. When they pay for these services, they're primarily paying to support the core features such service provide. Privacy and security, while concerns, are often secondary goals. Furthermore, on many free cloud services, the ability to harvest user data is the basis of the service provider's business model. These situations create a number of perverse incentives in terms of a traditional feature provider's goals with respect to user security and privacy [8]. In the first case, the feature provider simply does not prioritize user

security since that's not the primary basis on which users are choosing to pay for a service. In the second case, a feature provider might actively work to subvert user security and privacy in order to further leverage user data to generate income.

The SSaaS model aims to rectify these issues by introducing SSP actors whose primary goal is the protection of user secrets and from whom users purchase secret storage services on the basis of security and privacy guarantees first and foremost. Thus, SSaaS's ability to separate secret storage duties from feature provider duties allows users to purchase each service on the basis of its associated merits, avoiding the issues associated with putting features in direct competition with security and privacy: a competition that security and privacy have historically lost. Given such separation, independent markets can form around feature provision and secret protection, all optimized for the respective priorities of each field.

Beyond the removal of perverse incentives brought about by the separation of SSPs for FPs, it will also be desirable to encourage a competitive market amongst multiple SSP providers. In order to achieve such a market, I advocate for the standardization of a single inter-compatible SSaaS protocol. Such a standard protocol gives users a high degree of mobility between competing SSPs providers, avoiding vendor lock-in. This mobility, in turn, increases the competitive pressures between providers. In short, the aim of an SSaaS ecosystem is to make security and privacy tradable commodities, and to leverage market powers to price and improve both. A competitive market for secret storage has a number of security and privacy enhancing benefits:

Reputation: If users can easily switch between SSPs, it forces SSPs to compete on the basis of their security and privacy preserving reputations. SSPs who can do a superior job avoiding the trust violations discussed in Chapter 5 can attract more users and/or command a higher price for their services. Since a SSP's reputation is tied solely to their ability to faithfully protect user secrets, they will not be able to "iron over" any privacy-related reputation failings with superior end-user feature sets – as traditional cloud providers do today[1] .

[1] As an example, consider Facebook's numerous trust violations [75, 144, 254] and the fact that such violations have had no noticeable impact on the number of people using Facebook [60]. An SSP would enjoy no such network

Multiple Providers: A healthy ecosystem of competing SSPs will allow users to select from multiple independent providers over which they may shard a single secret. As I'll discuss below, this multi-SSP practice provides a number of benefits over relying on a single SSP, from additional trust reduction to redundancy.

Cost: As in other competitive markets, having a number of competing providers will allow the user to select a provider that offers the best combination of cost and service.

The SSaaS model also provides business benefits related to insurance. Having a dedicated entity in charge of protecting user secrets (and by proxy, any other data protected by those secrets) simplifies the process of evaluating risk and liability related to the protection of sensitive data. Similar to the model used by Certificate Authorities, SSPs could provide insurance polices to their users to indemnify them against any loss relating from a trust violation on the part of the SSP. Likewise, SSPs would underwrite such user-facing insurance polices with their own insurance polices provided by independent third-party insurers. These insurers would need to perform independent audits of SSP infrastructure and polices in order to evaluate trust violation risk, further enticing SSPs to fundamentally deign themselves for the avoidance of such trust violations. Such insurance benefits are not as readily available in the mixed trust + feature cloud ecosystem of today since it's far harder to evaluate the privacy violation risk of a company whose primary objectives are more complex then secret storage alone. The increasing regulation of user privacy rights and the penalties associated with violating user privacy further incentive a system where privacy and security are severable properties that can be independently regulated, evaluated, and indemnified – unconnected to the user-facing feature set of a given third-party service.

Likewise, SSaaS provides compliance benefits to users storing highly regulated data. Instead of having to individually verify that each end-user cloud service meets the requirements of a specific data storage regulations, a user could instead simply make sure that their SSP meets the necessary regulations. Once verified, a single SSP could be reused with multiple FPs without having to

benefit from additional services beyond secret storage were they to violate user's trust; instead, users would simply switch to a new SSP.

undergo further compliance verification. SSPs might even proactively obtain specific compliance certifications to make it easy for their users to comply with specific regulations, regardless of which feature-providing cloud service a user wishes to leverage. Such practices would likely prove highly beneficial in tightly-regulated fields such as health care (e.g. requiring HIPPA [260] compliance), education (e.g. requiring FERPA [259] compliance), and online payment processing (e.g. requiring PCI DSS [184] compliance).

6.3 Security and Trust

As mentioned in Chapter 5, use cases that involve splitting user data into cryptographically-protected core data and the associated cryptographic keys (i.e. secrets) and storing each with separate providers inherently decreases the level of trust that any single provider must be afforded. If, however, we wish to store unencrypted user data directly with an SSP, potentially because a given use case can't easily be split into encrypted data and encryption keys, we must place a higher degree of trust in an SSP. In the split encrypted data + cryptographic secrets use case, we must only trust the SSP with the storage (**S**) and metadata (**M**) capabilities, the same capabilities with which we must trust the FP. But in the case where an SSP directly stores raw user data, we must expand their capability profile to also include the access (**R**) and manipulation (**W**) capabilities. Are SSPs any more worthy (i.e. less likely to violate) this increased level of trust than a traditional FP?

Even in such a heightened degree of trust scenario, I believe that SSPs are less likely to violate user trust than traditional FPs. As outlined in § 6.2, SSPs have economic incentives well aligned with upholding a user's trust where as traditional FPs often have economic incentives in direct conflict with upholding user trust. This fact alone decreases their likelihood of trust violation by disincentivizing implicit violations (**P**-type) and putting a higher premium on avoiding unintentional (**U**-type), insider (**I**-type), and outsider (**O**-type) violations. I thus feel that users are better off trusting SSPs with raw sensitive data than FPs.

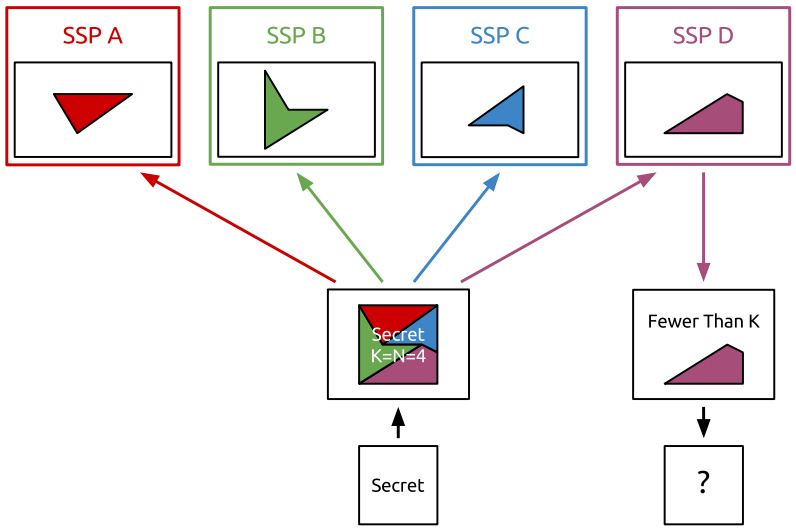


Figure 6.1: Sharding Secrets Between Multiple Providers

None the less, it's best if users can avoid placing high degrees of trust in any third party, FP or SSP. Fortunately, there is a viable alternative for avoiding additional trust even in the raw-secret storage SSP use case (and for further reducing trust in the split data + keys use case): sharding a single user secret across multiple SSPs. As shown in Figure 6.1 secure secret sharing systems such as Shamir's (k, n) threshold scheme [222] ensure that a secret split into n shares can not be reassembled using fewer than k shares in an information theoretically secure manner[2] . Such schemes could easily be applied in an SSaaS ecosystem to avoid affording (**R**) and (**W**) capabilities to an SSP, even when storing raw user data. In addition, secret sharing schemes have benefits related to redundancy and data availability. Since (k, n) threshold schemes include redundant shares in any case where $n > k$, users can split their secrets into n shares stored with n independent SSPs. Doing so insures that the user can recover their secret as long as at least k SSPs remain operational and in possession of their designated share. As such I foresee sharding secrets, be they cryptographic keys or raw data, across multiple SSPs being a standard best-practice in SSaaS ecosystems for both the trust reduction and increased reliability benefits such a practice provides.

[2] Information-theoretic security is an even higher level of security guarantee then traditional cryptographic security. Where as cryptographic security relies on assumptions about what kinds of math problems are or are not hard for computers to solve, information-theoretic security requires no such assumptions – instead deriving security from fundamental and universal properties of information theory.

Even with secret sharing users are still at risk of a successful trust violation should multiple SSPs collude to betray a user's trust (**L**-type violation) or should multiple SSPs be compelled to turn over shares of user secretes to a single entity (**C**-type violation). As such, we are interested in selecting a set of SSPs across which to shard our secrets that minimize **L** and **C** type violation risk. Thus, in addition to the reputation-based selection criteria outlined in § 6.2, I propose several additional SSP selection criteria:

Geopolitical Diversity: An ideal set of SSPs would be located across a range of non-cooperating geopolitical domains. This will greatly decrease the likelihood of a compelled (**C**-type) violation by ensuring that no single entity has the ability to compel multiple SSPs to surrender user secret shares.

Ownership Diversity: Similarly, a user should only select SSPs owned and operated by independent entities. This decreases the likelihood of an (**L**-type violation) by ensuring no single entity can instruct multiple SSPs to collude to reconstruct a user secret.

If the user selects a diverse set of SSPs, all with strong privacy-preserving reputations, I believe that they have a reasonable likelihood of avoiding a successfully series of trust violations resulting in the compromise of user privacy or security.

Chapter 7

Custos: An SSaaS Prototype

Custos[1] is a prototype SSP server, SSaaS protocol, and SSaaS client ecosystem used to demonstrate and explore SSaaS concepts. Custos is optimized for cryptographic key storage, making it primarily a Key Storage as a Service (KSaaS) implementation. I provide an overview of Custos and some of its design insights here. Additional information is available at [213] and [215].

7.1 Architecture

Figure 7.1 shows the core Custos components. The bulk of Custos's functionality is handled on the SSP server side. The Custos SSP server implements the following components:

API: Handles all Custos requests, including requests for key:value objects, requests for audit data, and requests to modify access control parameters. The API is designed to promote a variety of Custos-compliant server implementations.

Access Control: Compares the set of provided authentication attributes (calling into the authentication system to verify them) to the set of required authentication attributes to determine if a Custos request should be allowed or denied.

Authentication: Verifies the validity of any authentication attributes associated with a given Custos request via a pluggable authentication module interface capable of supporting a variety of authentication primitives.

Data: Handles data requests (e.g. get, set, create, and delete of key:value objects).

[1] Custos is Latin for "guard".

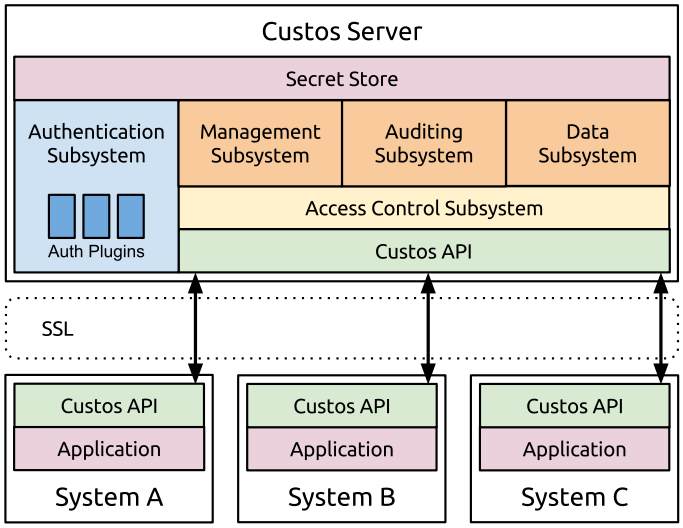


Figure 7.1: Custos's Architecture

Auditing: Handles audit requests and logs all Custos requests.

Management: Handles management requests (e.g. the manipulating access control parameters).

Key-Value Secret Store: Stores persistent data such as end-user secrets (e.g. encryption keys) as well as internal state (e.g. access control requirements).

7.1.1 Access Control

Custos employs a unique access control scheme called Access Control Chains (ACCs). In this scheme each Custos permission (e.g. reading a specific secret) is associated with one or more Access Control Chains. Each Access Control chain consists of an ordered list of Authentication Attributes (AA). Each Authentication Attribute represents a single authentication primitive (e.g. supplying the correct password, or verifying one's identity via a cryptographic authentication certificate). In order to gain a specific permissions in Custos, a user must be able to satisfy all AAs in at least one of the permission's associated ACCs. This scheme enables highly flexible access control semantics, allowing Custos secrets to be stored for a variety of applications.

In order to discuss the Custos access control system, I must first explain the Custos **organizational units** (OUs): the core Custos data structures. The Custos architecture specifies three

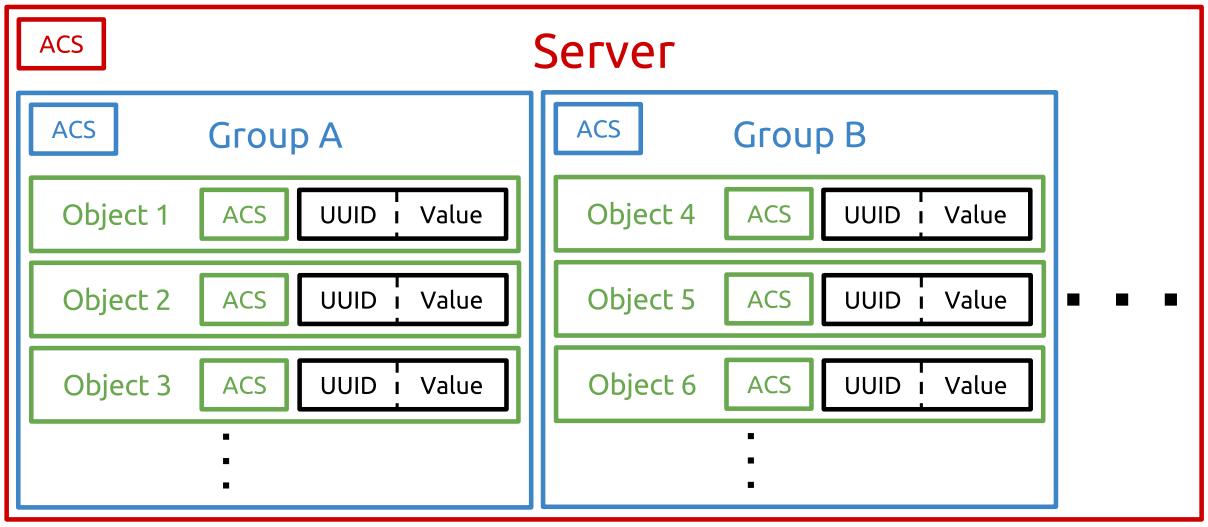


Figure 7.2: Custos's Organizational Units

organizational units (Figure 7.2): a server, a group, and a key:value object. The server unit is used to specify server-wide configuration. A server has one or more groups associated with it. A group is used to slice a server between a variety of administrative domains (e.g. separate customers). A group, in turn, has an arbitrary number of key:value objects associated with it. Each OU is responsible for the creation of OU instances beneath it: i.e. servers create groups and groups create objects.

The Custos access control abstraction revolves around designating an **Access Control Specification** (ACS) for each OU in the Custos architecture. An ACS consists of three components (Figure 7.3). Each ACS contains a full list of the applicable **permissions** for the given OU. Associated with each permission is one or more **access control chains** (ACCs). Each ACC consists of an ordered list of **authentication attributes**.

7.1.1.1 Permissions

Each Custos ACS contains a list of permissions: rights to perform specific Custos actions. Custos defines permissions for each OU: i.e. per-server permissions, per-group permissions, and per-object permissions (Table 7.1). Unlike many access control systems, Custos has no notion of object

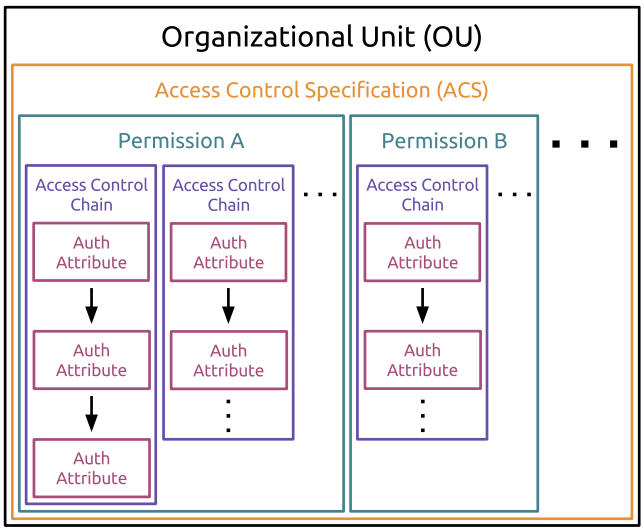


Figure 7.3: Access Control Specification Components

Permission	OU	Rights
srv_grp_create	Server	create groups on a Custos server
srv_grp_list	Server	list groups on a Custos server
srv_grp_override	Server	escalate to any group-level permission, overriding the per-group ACS
srv_audit	Server	read all server-level audit information
srv_clean	Server	delete all server-level audit information
srv_acs_get	Server	view the server-level ACS controlling the permissions in this list
srv_acs_set	Server	update the server-level ACS controlling the permissions in this list
grp_obj_create	Group	create a key:value objects within the given group
grp_obj_list	Group	list key:value objects within the given group
grp_obj_override	Group	escalate to any object-level permission, overriding the per-object ACS
grp_delete	Group	delete the given group on a Custos server
grp_audit	Group	read all group-level audit information
grp_clean	Group	delete all group-level audit information
grp_acs_get	Group	view the group-level ACS controlling the permissions in this list
grp_acs_set	Group	update the group-level ACS controlling the permissions in this list
obj_delete	Object	delete the given key:value object within the given group
obj_read	Object	read the given key:value object within the given group
obj_update	Object	create a new version of the given key:value object within the given group
obj_audit	Object	read all object-level audit information
obj_clean	Object	delete all object-level audit information
obj_acs_get	Object	view the object-level ACS controlling the permissions in this list
obj_acs_set	Object	update the object-level ACS controlling the permissions in this list

Table 7.1: Custos Permissions

ownership. Instead, it relies on explicitly granting each right an owner would traditionally hold via explicit permissioning. Custos permissions are initially set when the associated OU is created.

After creation, each ACS can be updated by anyone granted the necessary `acs_set` permission for the specific OU instance. Custos group and server ACSs also include an "override" permission. This permission can be used to override the permissions of a lower-level OU's ACS. For example, anyone gaining the `srv_grp_override` permission can use it to gain any of the rights normally granted via a group-level permission. Likewise, anyone gaining the `grp_obj_override` permission can use it to gain any of the rights normally granted via an object-level permission. These overrides exist for administrative tasks: allowing server admins to manipulate group data, and allowing group admins to manipulate object data.

7.1.1.2 Access Control Chains

Each ACS permission has one or more associated access control chains (ACCs). An access control chain is an ordered list of authentication attributes (discussed in § 7.1.1.3). In order for a request to be granted a specific permission, it must be able to provide authentication attributes satisfying at least one of the ACCs associated with that permission. If a user wishes to disable access to a permission, they can do so by associating the null ACC with that permission. If the user wants to provide unrestricted access to a permission, they may do so by associating an empty ACC with the permission. For example, consider a key:value object whose `obj_read` permission has the following ACC set:

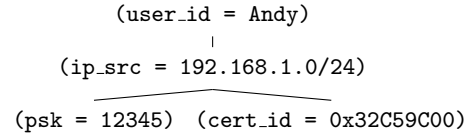
```
[ (user_id = Andy),
  (ip_src = 192.168.1.0/24),
  (psk = 12345) ]
[ (user_id = Andy),
  (ip_src = 192.168.1.0/24),
  (cert_id = 0x32C59C00) ]
[ (user_id = John),
  (psk = Swordfish) ]
```

In order for our read request for the associated key:value object to succeed, we would have to make sure that our request contained all the authentication attributes in at least one of the lists

above. In the case of the first ACC, that would mean attaching the 'user_id' attribute with a value of 'Andy', as well as attaching the 'psk' attribute with a value of '12345'. The 'ip_src' attribute is an implicit attribute (see § 7.1.1.3) and will be automatically appended to our request when received by the Custos server. In order to satisfy it, we would have to send the request from the 192.168.1.0 subnet. In the case of the second ACC, we still need the 'Andy' username and must satisfy the IP restriction, but this time we must prove that we have access to the private key associated with the specified authentication certificate instead of providing a password. In the third ACC, we have granted access to an additional user, John, with his own password. As long as we can satisfy at least one ACC in a set of ACCs for a given permission, we are granted the right to perform actions associated with the permission.

This system is highly flexible. Take, for example, the lack of explicit username support anywhere in the Custos specification. As was done above, usernames simply become another authentication attribute. Often a username will be the first attribute in a ACC to allow for all following attributes to be specified relative to a given username (as shown in the example above). But there's nothing special about usernames. We could just have easily started each ACC with an IP attribute, requiring a separate password based upon the location a user is making their request from. The combination of simple ordered attribute lists and a wide range of flexible attributes makes for powerful access control semantics.

Another point worth noting is that sets of ACCs can be converted into ACC trees, often simplifying the understanding or verification of their semantic intent. ACC lists are converted into ACC trees by combining common attributes across multiple ACC lists into single nodes in an ACC tree. For example, the first two ACCs in the previous set of ACCs could also be represented as:



Finally, where desired [2] , the Custos API can continue to prompt the user for the next N missing attribute types in a chain. When in use, this feature allows a Custos server to engage in a back-and-forth message exchange with a client to prompt the client through all required attribute types in an ACC. For example, in the case where N is equal to 1 and the previously mentioned ACCs are in effect, the following set of transactions would occur:

(1) The user sends a read request with no attributes

(2) The server respond that a username is required

(3) The user resubmits the request with an attached username attribute equal to 'Andy'

(4) The server responds that a password or a certificate is required (the IP attribute is implicit and is thus not prompted for)

(5) The user resubmits the response with a password equal to '12345'

(6) As long as the user's request originates from the specified IP range, the server will grant the request.

7.1.1.3 Authentication Attributes

Type	Class	Description
`ip_src`	implicit	Request source IP
`time_utc`	implicit	Request arrival time
`user_id`	explicit	Arbitrary user ID
`psk`	explicit	Arbitrary pre-shared key

Table 7.2: Example Authentication Attributes

Each Access Control Chain contains one or more Authentication Attributes (AAs). An authentication attribute is a generic container for authentication data. AAs contain the following information:

Class The top level classification property of an AA. It is used to designate the nature of a given AA. Currently, Custos specifies two possible values for class: "implicit" and "explicit".

[2] Custos's attribute prompting feature is a form of information leakage, so its use, and the associated trade-offs, are optional.

Implicit attributes are those that are automatically associated with a request (like an IP address). Explicit attributes are those that the user provides directly to Custos (like a username).

Type Within a given class, the AA type specifies which authentication plugin should handle a specific attribute.

Value The value contains the arbitrary data associated with a given attribute.

The Custos specification supports a flexible set of authentication types. Each AA is processed by a specific AA plugin module allowing for extensible authentication primitive support similar to systems like PAM [207]. Examples of potential Custos AA types are shown in Table 7.2.

7.1.2 Protocol

Custos employs a JSON-based RESTful HTTPS protocol for client-server communication. The protocol exposes endpoints for secret management (e.g. create, list, read, delete), access control (e.g. adding, removing, and modifying access control rules), and auditing (e.g. reading or clearing secret access history). Custos allows arbitrary cryptographic key data to be associated with each ID for secret storage purposes. Each ID:secret pair in Custos is associated with a specific group. These groups can be used to allow multiple users to administrator a set of secrets and to control who can create new secrets within a group.

The Custos API is secured via SSL/HTTPS. Custos servers are authenticated over SSL via the standard public key infrastructure (PKI) mechanisms (e.g. certificate authorities, etc)[3] . API requests are made to specific server HTTPS endpoints. The standard HTTP verbs (`GET`, `PUT`, `POST`, and `DELETE`) are used to multiplex related operations atop a specific endpoint. Each combination of endpoint and verb defines a specific API method. Each method requires a specific permission to complete. All API message formats are composed in JSON. Binary data is encoded as Base64 ASCII text. Authentication attributes are passed via query string as URL-encoded JSON. Custos uses

[3] In situations were the traditional PKI mechanisms are deemed undesirable, it may be possible to authenticate Custos servers via self-certifying mechanisms [50, 149].

UUIDs [135] as keys, each associated with an arbitrary object for values. The full API specification, including detailed message formats, example messages, and a list of API methods and endpoints see [213].

In the Custos protocol, the server is completely agnostic to the secret cryptographic keys it stores. Thus, a user may shard their secret keys across multiple providers/servers if they wish. Because of this fact, however, users must still manually set the appropriate ACCs for a given secret across all Custos servers holding a share of that secret. Likewise, a user must manually query each server holding a share of a secret for audit data in order to aggregate the full audit history for a given secret. Developing methods for more easily coordinating ACC and audit information for a set of secret shares across multiple SSPs is one of the proposed efforts in Chapter 9.

7.2 Implementation

I've created prototype implementations of both a Custos server and a sample Custos client. These implementations exist to test and demonstrate the basic SSaaS concepts proposed here. Expanding these prototypes is part of the proposed work in Chapter 9.

7.2.1 SSP Server

The Custos SSP server [210] is implemented in Python. It is designed to use off-the-shelf key-value stores for backing secret storage. The bulk of the Custos server code base is involved with implementing the verification process for Access Control Chains and the associated Access Attribute modules. The server leverages the Flask [198] web framework to implement the Custos API, and provides HTTPS support via the Apache [9] web server. The server is capable of handling a few hundreds key requests per second, depending on the complexity of the associated ACCs. Increasing the performance of the SSP server is the topic of future work.

7.2.2 EncFS

To test the Custos SSP server and demonstrate an SSaaS-aware application, I created `EncFS`, a sample SSaaS client [211]. `EncFS` is a simple FUSE-based [247] layered file system that provides transparent client side-encryption atop an underlying file store. `EncFS` offloads its key storage duties to the the Custos SSP Server using the Custos SSaaS protocol. It accomplishes this via `libcustos`, a C-library implementation of the Custos protocol [212].

`EncFS`'s utilization of the SSaaS model allows it to support many cloud-based use cases not readily supported by traditional encrypted file-systems, all while minimizing the trust placed in cloud-based storage providers and Custos SSPs. For example, when mounted atop a Dropbox-backed directory, `EncFS` allows a user to ensure that Dropbox has no access to the plain-text contents of a user's files. None the less, users can still use Dropbox to sync a file across multiple devices or share it with other users. The user must simply update the access control attributes for a given file's encryption key stored via their Custos SSP to allow access from other user-owned devices or by other collaborating users. `EncFS` accomplishes this without any need to modify Dropbox's storage interface or otherwise change the manner is which existing cloud file stores operate.

Adding the SSP encryption key request operation to the file read process does incur an additional 10 to 100 milliseconds of latency on file reads, primarily dependent on the network latency between the client and the SSP [215]. None the less, I don't believe that most users will notice this additional latency in normal day-to-day use (e.g. editing a text file, playing a media file, viewing a photo, etc). I also don't feel this latency is any worse then most existing distributed or networked file system protocols would incur, and in situations where an SSaaS client like `EncFS` is used in conjunction with a distributed or networked file system, secret-access operations can be performed in parallel with encrypted data-access operations, ensuring that the SSP latency is not incurred in addition to existing networked file system latency.

Chapter 8

Applications

As mentioned in previous chapters, the SSaaS model can be utilized across a range of applications. In each case, the model allows users to reduce the trust they must place in any single third-party while also supporting popular use cases. I present several potential SSaaS applications in this chapter. Some of these applications are also discussed in [215].

8.1 Storage

One of the primary applications of the SSaaS model is to secure storage systems, both local and hosted. These applications are all variations on the previously discussed encrypted data + SSaaS-backed keys model. In such applications, applications handle the encryption and verification of data locally, sending only encrypted data to third parities or storing such data on high risk devices. In order to ensure that such client-side encryption does not break traditional features associated with cloud and mobile data storage, clients store the associated encryption keys with one or more Secret Storage Providers.

8.1.1 Cloud File Sync/Storage

Building on the original motivating example in this proposal, i.e. using Dropbox without trusting Dropbox, we can construct an SSaaS-backed cloud sync and storage client. Figure 8.1 illustrates this application. As in the general storage case, this application involves applying client-side encryption/decryption (e.g. using AES [162]) and authentication/verification (e.g. using

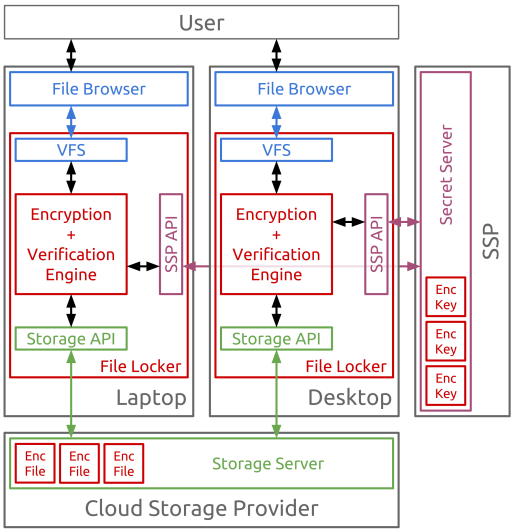


Figure 8.1: SSaaS-Backed Cloud File Locker System

CMAC [47]) on every read and write to a third-party backed data store. The third-party data store holds only encrypted and authenticated data, ensuring that the third party need not be granted Access (**R**-type) and Manipulation (**W**-type) capabilities.

In order to ensure that users can still share data with other users and sync it across devices, all required encryption and authentication keys are stored with one or more SSPs. When a user wishes to sync data to a new device, they grant said device access to the necessary keys via their SSP's management interface. The new device can then download the encrypted files from the minimally trusted storage provider and decrypt/verify them using the keys provided by the SSP. Device authentication can be provided via certificates, shared-secrets (e.g. passwords), or contextual information. When a user wishes to share data with another user, they grant the new user access to encrypted data via the storage provider's normal sharing mechanisms. They then also grant the new user access to the necessary keys via the SSP's management mechanisms. The user can now download the data from the storage provider and decrypt and verify it with the keys from the SSP. As in the multi-device sync case, authentication may be performed via a variety of mechanisms, allowing the data owner to select the authentication primitives best suited for a given situation.

This type of application overcomes the traditional deficiencies of secure cloud-based data storage. It minimizes trust in the third party storage provider by only granting them access to encrypted and authenticated data. It also maintains support for the multi-device and multi-user use cases traditionally associated with cloud-backed data storage. The SSaaS model allows users to enhance their privacy and security by reducing exposure to third parties without incurring additional usability burdens or denying access to desirable features.

8.1.2 Server Data Encryption

Beyond consumer-oriented SSaaS-backed encryption systems, there's strong case for using SSaaS-backed encryption systems for datacenter-based servers. Leveraging virtual (as well as physical) servers hosted in cloud data centers is an extremely popular deployment method. Unfortunately, as mentioned in Chapter 3, the administrator's lack of physical access to such servers makes it difficult to utilize privacy-enhancing technologies like Full Disk Encryption (FDE) or file-system level encryption. In the FDE case, users are generally required to provide some form of decryption pass-phrase or physical dongle at boot time in order to securely bootstrap the system. Similarly, even in the file-system level encryption case, encryption systems generally require some form of interactive mechanism to provide the necessary security pass-phrase bootstrapping the system. Since administrators generally lack easy physical access to datacenter-based servers as well as interactive presence on headless servers using traditional encryption systems with most cloud servers remains difficult, if not impossible.

These deficiencies can largely be resolved by relying on SSaaS-backed encryption systems - either full disk or file-system oriented. Using SSaaS, a user would configure each server to store their file encryption and verification keys with an SSP (or SSPs), configuring each server to request the keys from the SSP at boot or on access to an encrypted file. Non-interactive authentication could be provided using contextual security techniques [106] (e.g. do you expect a server to be rebooted at a certain time of day each day?) to allow for such encryption systems to operate largely autonomously. In more sensitive situations, the SSP's access control system could even

keep the user in the loop as in the traditional case by asking the user to provide a pass-phrase or decryption approval via text message or similar out-of-band real time communication method each time key access is required.

Such systems would allow users to store sensitive data on servers while minimizing the degree of trust they must place in third-party data center hosts. Such servers would store all data in an encrypted form, ensuring that a physical search of the server or other interference from the data center host would not be easy. Encrypted data could be decrypted on-demand, making it very difficult for the data center host to access it without employing high degrees of subterfuge. Even in cases where the data center host does manage to leverage their control of the underlying physical systems to trick an SSP into providing decryption keys, the SSP would still be able to log and audit the event – making it extremely difficult for a data center host to access secure user data in an undetectable manner.

Possible implementations of server-based encryption efforts are discussed further in Chapter 9. Such implementations would provide a mechanism to significantly decrease the amount of trust developers (and by proxy, their users) must place in the providers of hosted server infrastructure without significantly raising the cost or overhead of leveraging such infrastructure.

8.1.3 Mobile Device Encryption

Beyond of the world of hosted storage systems, the SSaaS model provides benefit for the protection of local storage as well. Mobile devices such as phones tables and laptops store huge quantities of personal information. Yet these devices are more prone to theft and loss then traditional computing systems such as desktops and servers. The encryption and protection of such devices is thus a high priority when working to increase the privacy and security of many users.

Traditional device encryption systems have become prevalent in Android and iOS-base mobile devices [3, 54]. Such systems are useful for ensuring that device-stored data can not be accessed when devices are shutdown or (when properly designed) locked, but they do little to protect data when the phone is (are has recently been) in use. Furthermore, the encryption keys for such

systems are stored locally (often via an HSM) and are generally only protected with a short pin or pass-phrase.

Such encryption systems could be converted to an SSaaS model where the keys are stored with one or more SSPs, providing a number of benefits over local key storage. First, since keys are stored off-device in an SSaaS-backed encryption system, it's easy to revoke access to the keys remotely in the event that a device is lost or stolen, ensuring that an attacker can not guess a user's PIN and access their data. Second, such a systems could be used to protect individual pieces of data separately vs the current practice of unlocking all data when the device starts up. In doing so, it would be possible for a user to set per-app access control rules with their SSP, effectively reducing the trust the user must place in the third-party apps they install on their devices. Users could leverage the SSaaS model to audit all app data access and limit apps to accessing only the data they are expected to need. Finally, an SSaaS system could provide more flexible access control semantics's on mobile devices then the standard Everything/Nothing access model common today. For example, as SSaaS system could be setup to provide keys to one set of data to User A while providing keys to a separate set of data for User B – effectively protecting per-user data on multi-user devices. Similarly, an SSaaS system could be configured to provide access overrides in specific situations – e.g. allowing an authorized secondary user to access the device and the data it stores in the event that the primary user becomes incapacitated (e.g. similar to the functionality provided by Google's Inactive Account Manager [200], but with cryptographic guarantees).

8.1.4 Personal Data Repository

Moving past encrypted storage applications, we could also leverage the SSaaS model to create a dedicated repository of personal user info (e.g SSN, address, etc). Entering personal user data on various websites is a common user requirement for placing orders, creating accounts, etc. This practice is undesirable for several reason. First, it is highly burdensome: users are continually forced to reenter the same info again and again and must ensure they keep it up to date across multiple sites. Second, it distributes a lot of sensitive user data across a large number of actors in a

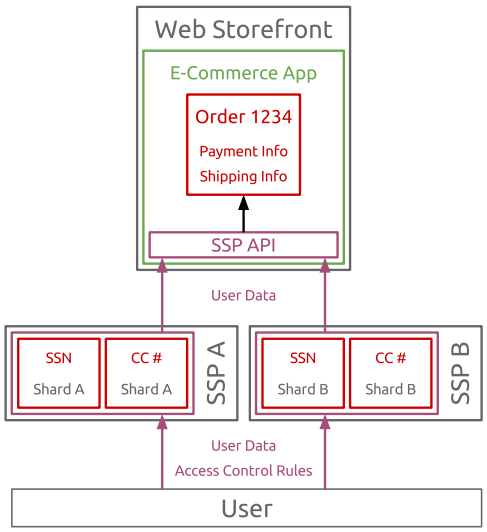


Figure 8.2: SSaaS-Backed Personal Data Repo

manner that is very difficult for a user to track or control. Instead, I propose that a user shard their data across several SSPs and then point websites that require it directly at the SSPs themselves. The user can then specify Access Control rules with each SSP allowing specific websites access to the minimum data they require.

Not only does this approach provide a central repository where the user can enter personal data and keep it up to date (e.g. when moving), but it also allows the user to limit each site's access to only certain data and provides an audit trail for the user to track exactly which sites have accessed which data. Figure 8.2 illustrates such a use case. In this case, as opposed to the previous applications, we'll be storing sensitive user data directly with our SSPs. As such it will be important to shard the data across multiple SSPs in order to minimize the trust we must place in each individual SSP.

Such applications could go a long ways toward minimizing the amount of sensitive personal data we have stored with large number of third parties in favor of concentrating it on only a handful of SSPs. Doing so both reduces the trust we must place with any single SSP and ensures the data is being stored with parties better incentivized to protect it (see the discussion in Section 6.3).

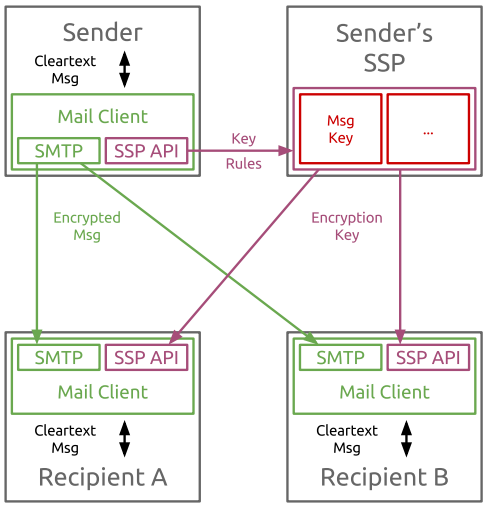


Figure 8.3: SSaaS-Backed Secure Email System

8.2 Communication

The applications of SSaaS are not limited to data storage. Another potential area of application involves secure communication between two or more parties. Such secure messaging systems are in growing demand, especially after the Snowden revelations related to US Government monitoring of email and associated digital communication systems [71, 96, 97]. While solutions like GnuPG [122] exist to allow users to secure the contents of their mail, their complexity tends to limit their use to a small subset of advanced users [94, 264]. The secure communication paradigm could be simplified through the use of SSPs and the SSaaS model.

Figure 8.3 illustrates a potential SSaaS secure communication use case. In this application, a user's mail client would encrypt the contents and subject of a user's email prior to sending it with a locally-generated single-use symmetric encryption key. The client would then upload this key to the user's SSP where the user would specify that their intended recipients should have read access to the key. The user could then send the encrypted mail. When received, the recipient's mail client would request the required key from the sender's SSP, providing appropriate credentials to prove their identity in the process, decrypt, and read the email. This system would be capable of supporting multi-recipient systems like newsgroups and would also allow the user to grant read

access to additional users after the fact (e.g. should the recipient wish to forward the email to an additional trusted party), both capabilities that traditional public-key based email encryption systems struggle to support. Similar applications could be designed for real-time communication such as chat.

As in previous cases, third party trust can be minimized by sharding keys across multiple SSPs. Furthermore, such systems could guarantee some degree of forward secrecy by having the SSP automatically expire and delete keys after a certain period of time – something that traditional email encryption systems struggle to support. Such designs have the potential to raise the default level of security inherent in mass digital communication without significantly inconveniencing users.

8.3 Authentication

SSaaS also has applications in the authentication realm. Cryptographic authentication systems (i.e. certificate-based authentication) are considered to be some of the most secure forms of digital authentication – far better then more commonly deployed shared-secret (i.e. password) authentication systems. Where as shared secret authentication merely relies on both the user and the verifier (e.g. server) knowing a common secret such as a pass-phrase, certificate-based authentication systems rely on a user to be able to cryptographically prove they posses the private key corresponding to a specific public key. Such systems have a range of applications, from authentication on websites to authentication on servers and workstations. In the latter case, SSH [270] is one of the more common remote-access protocols for Unix-like computing systems such as Linux and OSX. SSH uses keys on both the server side, where they provide server verification and encryption for connecting clients, as well as on the client side, where they can (optionally) be used to authenticate users instead of passwords. SSH is thus a useful example to demonstrate the potential benefits of the SSaaS model to authentication systems. SSaaS can provide usability and security benefits to SSH on both the client and server.

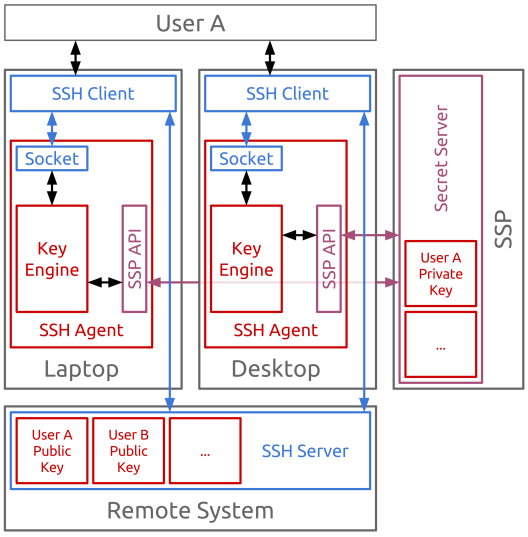


Figure 8.4: SSaaS-Backed SSH Agent

8.3.1 SSH Agent Key Management

The management of private cryptographic keys has long been a challenge for users. To help mitigate this challenge the systems community has developed the concept of an "agent" program. Agent programs sit between the user and an authentication system, providing the required cryptographic keys on the user's behalf to the authentication system when required [34]. Agents are commonly used with popular computing utilities like SSH [270] and GnuPG [122]. Unfortunately, existing agent solutions are designed for legacy usage models: single-device, non-portable desktop environments. They do not provide a mechanism for managing private keys across multiple devices, securing keys if the associated device is lost or stolen, or managing keys for a large group of users across an organization.

The locality and management challenges associated with traditional cryptographic agent programs can be overcome by using an SSaaS-backed agent. Figure 8.4 shows the basic design for an SSaaS-backed SSH-agent. Instead of storing private keys locally, such an agent would defer private key storage to one or more dedicated SSPs. When the agent requires the user's private keys, it requests them from the SSP. Thus, the user and any associated agent programs are able to securely access the necessary private keys from multiple devices (e.g. laptop, desktop, tablet).

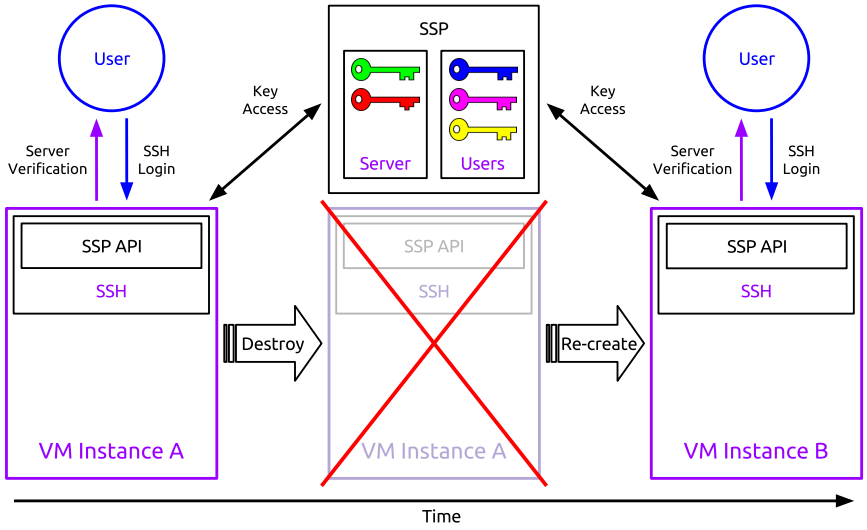


Figure 8.5: SSaaS-Backed SSH Server Key Management

Furthermore, if the user ever loses one of their devices, they can greatly reduce the risk of exposing any of their private keys by revoking the lost device's access to the off-site SSaaS data (e.g. similar to [69] and [248]). Such a system would also allow large organizations to manage SSH or other cryptographic keys for all their users from a centralized SSaaS management applications.

Divorcing private SSH key storage from local devices opens up a range of use case possibilities, increases security by keeping keys off of frequently lost or stolen portable devices, and relieves the user of the usability overhead required to manually manage their private keys.

8.3.2 SSH Server Key Management

Similarly, the SSaaS model can provided benefits when applied to the server side of SSH. An SSH server must manage both a server key pair, used to prove the server's identity to a user and to bootstrap the negotiation of a session key, as well as a list of user public keys for any users granted server access. Unfortunately, managing both sets of keys manually poses a number of challenges. Integrating server-side SSH key management with an SSaaS system can help alleviate these challenges.

In the server key case, the ephemeral nature of modern IaaS-based systems causes issues with locally manged keys. For instance, what the user perceives of as a single server may actually be multiple load-balanced servers, or may be a single virtual server that is destroyed and recreated frequently in response to upgrades and changes in demand. In either case, it's important that the user be able to properly authenticate that the server they are accessing is in fact the correct server and not malicious man-in-the-middle attacker. Unlike SSL, however, SSH does not rely on a central certificate authority structure for the verification of keys. Instead it uses a "trust on first use" model that fingerprints a server key the first time the user connects and then assumes that this key will remain associated with a given server address indefinitely. If an ephemeral cloud server generates a new server SSH key each time it's recreated, or if multiple logically equivalent, load-balanced servers all have different SSH keys, a user will be unable to verify that the server is equivalent to the one they originally accessed. This issue could be overcome by storing the SSH verification keys for a server with one or more SSPs as shown in Figure 8.5. Then, when a server is recreated, or when a single logical server is balanced between multiple instances, it still has access to the same SSH keys originally in use, ensuring that previous users can properly authenticate the server as valid.

The server's list of authorized user keys poses similar challenges. These lists contain the public keys of all users authorized to access a given server. Maintaining such lists across large numbers of servers, and ensuring they stay up to date even as individual server instances are added, removed, and upgraded is challenging. Failing to properly manage such lists can either lead to legitimate users being locked out of systems, or worse, allowing illegitimate users (e.g. a previous employee) to access systems they should not. Indeed, such misconfiguration errors are one of the leading causes of security failures today [18, 119]. Centralizing the management of such authorized keys lists with an SSP allows administrators to manage and audit server access across a wide collection of infrastructure from a single place. Such a solution is the SSaaS equivalent of cloud user management solutions such as JumpCloud [115]. But unlike existing services, an

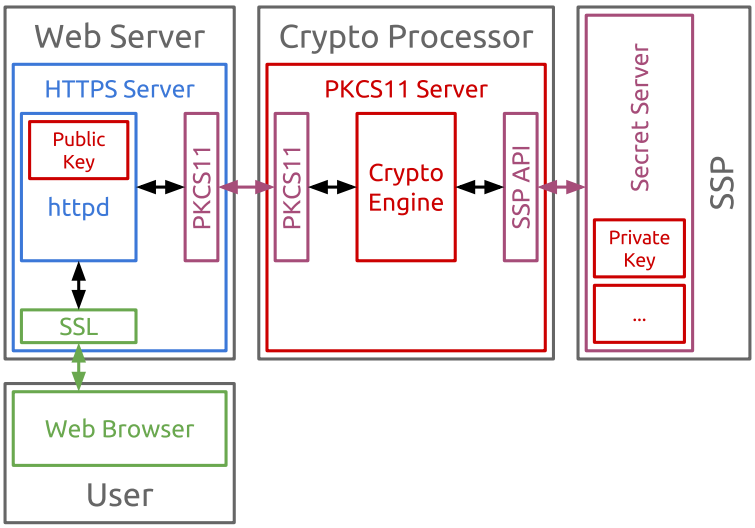


Figure 8.6: SSaaS-Backed Dedicated Crypto Processor

SSaaS-backed SSH user management system avoids trusting any single third party by leveraging the key-sharding possibilities available in an SSaaS ecosystem.

Both of these setups illustrate the benefits an SSaaS system can provide to common authentication and user management problems in cloud servers. In many ways, SSaaS represents a more secure variant of traditional configuration management solutions such as Chef [179] or Puppet [131]. But unlike such solutions, the SSaaS model is deigned with the security and privacy of the configuration data it stores as a first principle, not a secondary concern.

8.4 Dedicated Crypto Processor

The recent Heartbleed bug [33] exposed one of the main risks of embedding cryptographic secret storage within the applications requiring access to these secrets: when applications break, they also risk exposing access to the private keys stored in the same memory segments. The Heartbleed fallout has forced everyone to reevaluate whether or not giving public-facing services direct access to cryptographic keys is a good idea. There has long existed an alternative: using a hardware security module (HSM) to perform dedicated crypto processing and key storage on behalf of other services. Such systems ensure that cryptographic keys are never exposed outside of

the secure hardware module. Programs communicate with the HSM via a standard protocol like PKCS#11 [51]. In such systems, an application sends the HSM clear-text data to encrypt and gets back the encrypted ciphertext in return. Unfortunately, most existing HSM solutions don't scale to the performance levels required by high-volume services. This has led some to suggest moving to a software-based "HSM" model [145]. Such softHSM systems would still ensure cryptographic keys remain stored in separate isolated memory spaces while also out-performing traditional HSM systems.

A software-based dedicated crypto processing system is an ideal use case an SSaaS-backed system. Figure 8.6 shows the potential design of such a system. Here, a Secret Storage Provider supplies the back-end key storage for a dedicated crypto processing server which performs cryptographic operations (e.g. SSL encryption and authentication) on behalf of a web-server. In this setup, the web-server never accesses any private cryptographic keys directly, mitigating one of the major risks Heartbleed exposed. Furthermore, the logically centralized nature of an SSaaS SSP allows dedicated crypto processing servers to scale horizontally (e.g. multiple load-balancing instances) as demand requires: storing all keys via one or more SSPs allows new crypto processing instances to immediately access these keys without the need to utilize ad-hoc key-syncing or configuration management interfaces. Furthermore, The SSP's auditing functionality ensures that you always know which keys have been accessed by which systems, placing hard bounds on what an attacker may or may not have had access too: a luxury that Heartbleed-prone servers did not have.

Such a design combines the benefits of a dedicated crypto-processor with the third-party trust limiting nature of the SSaaS model. This combination allows for high speed cryptographic processing without placing a high degree of trust in any single system or party.

Chapter 9

Proposed Work

Thus far, I've laid out an overview of the SSaaS model as a means for limiting third party trust and increasing the security and privacy of today's computing users across a range of modern use cases. In this chapter, I propose the remaining work that will be completed for inclusion in my dissertation. This work all builds on or expands the work discussed thus far.

9.1 Existing Services Surveys

The first body of work I propose involves taking the trust framework presented in Chapter 5 and applying it to a variety of existing services. Such a survey will be useful in further demonstrating the security and privacy challenges posed by existing systems. I propose analyzing the trust models of several classes of existing cloud services, each of which is discussed briefly in the following subsections.

To conduct these surveys, I will analyze the publicly available information related to the services in question, including service provider documentation, independent analyses, and examples of known security and trust failures. In some cases it may also be necessary to experiment with the services directly for the purpose of further exploring how they should be categorized within the proposed trust analyses framework. These analysis will provide the basis for improving the design of SSaaS-backed solutions to exhibit more desirable trust profiles then existing systems without unduly restricting desirable use cases.

9.1.1 Consumer Web Services

The first set of proposed trust surveys will include an in-depth analysis of several prototypical consumer-facing cloud services. Potential candidates for analysis include:

- Cloud storage services such as Dropbox [42], SpiderOak [232], and BitTorrent Sync [19].
- Social media, communication, and transportation services such as Facebook [53], Gmail [90], and Uber [255].
- Password managers such as LastPass [133] and OnePassword [2].
- Financial services such as banks, finance aggregators like Mint [111], and payment services such as Google Wallet [92] and Apple Pay [11].

9.1.2 Developer Web Services

A second set of proposed trust surveys will focus on common developer-facing cloud services. Potential candidates for analysis include:

- IaaS providers such as Amazon EC2 [6], RackSpace [190], and Google Compute Engine [85].
- Configuration management solutions such as Chef [179] and Puppet [131].
- Recently-released Key management services such as OpenStack Barbican [176], Amazon Cloud HSM [4], and Gezzang zTrustee [68]. This may also include end-user facing key-reputation services such as KeyBase.io [120].

9.2 Implementation

Beyond applying the proposed trust analysis framework to a range of existing services, I propose several implementation advancements of my existing SSaaS prototype. These advancements will build on the Custos work discussed in Chapter 7, either improving or replacing existing components. I discuss the main facets of the proposed implementation projects below.

9.2.1 Second-Generation SSP Server and v2 API

I propose creating an updated SSP server implementation to replace the Custos first-generation SSP server presented in Chapter 7. The existing Custos server is performance limited due to its lack of high performance backing data store. I propose a rewrite of the system capable of leveraging production-level backing stores such as Redis [250]. Furthermore, the existing system would benefit from improvements with respect to its authentication and authorization subsystems. These systems will be improved to include simpler interfaces, making it easier to add plugins to support new access control systems. Finally, I plan to make the auditing system simpler to interface with third party auditing systems such as LogRythm [143] or Splunk [233]. Collectively, this work will constitute the creation of the Gen2 SSP prototype.

In addition to updating the SSP server prototype, this will be a good opportunity to make any necessary changes to the Custos SSaaS API. Potential API changes will be made with an aim toward ensuring support for the range of use cases discussed in Chapter 8 and explored in the proposed trust surveys. Furthermore, API updates will be undertaken to insure compliance with current REST standards [107] and best practices [202]. Updates will also be made with an eye to learning from similar recently released key-management APIs such as those of OpenStack Barbican [176]. These changes will result in the creation of the Version 2 Custos SSaaS API.

9.2.2 Multi-Party Sharding

One of the major unexplored parts of the discussed SSaaS model is the use of multi-party sharding to avoid having to trust a single SSP. In order to avoid centralizing trust in a single location, such sharding must be performed and managed from the client-side. This raises a number of questions and challenges:

- How do we properly abstract the secret sharding process so that SSaaS clients may leverage it without undue overhead?

- How do we help users to chose a proper set of SSPs to minimize their exposure to the varies multi-SSP trust failures (e.g. collusion violations and compelled violations) outlined in Chapter 5?[1]
- How do we best distribute access control rules across multiple SSPs in a manner that minimizes SSP collusion as well as management overhead?

I propose exploring potential answers to these questions by integrating multi-party sharding support into an updated version of the Custos client-side interface libraries. Making multi-SSP sharding simple to use and straightforward to manage will be a key component of using it to minimize the degree of trust users must place in individual SSPs. Having a working implementation that is easy to integrate with existing SSaaS applications will go a long ways toward achieving such simplicity and ease of use.

9.2.3 Client Applications

In addition to improving the server-side implementation and adding more robust client support for multi-party sharding, I propose to build out several SSaaS client applications similar to those discussed in Chapter 8. These applications will allow further demonstration of the potential benefits of the SSaaS model and will provide usable systems against which various SSaaS concepts may be evaluated. In particular, I propose focusing on several SSaaS data encryption clients including:

Ceph Encryption Support: Ceph [235] is a popular distributed file system commonly used by IaaS systems such as OpenStack [177]. As such, adding SSaaS-backed encryption support to Ceph would allow developers to utilize Full Disk Encryption mechanisms on cloud-hosted VMs, a use case that is currently not possible to achieve in most situations. I propose adding SSaaS-backed encryption support to the librbd [237, 238] and/or rbd-fuse [239] components of Ceph, allowing VMs and other programs that leverage such components to

[1] The answer to this question would appear to have parallels with existing failure-resistant storage systems such as RAID [183] and Ceph's CRUSH maps [236].

perform encrypted reads and writes to the underlying Ceph storage cluster. In doing so, the user will be able to reduce their trust in the Ceph provider while still leveraging the benefits of using Ceph-backed VMs in the cloud.

File-system Encryption Support: In addition to Ceph-based FDE, it would also be useful to have a workable file-system level SSaaS-backed encryption solution. Thus, I propose improving the existing FUSE-based EncFS system discussed in Chapter 7 and/or adding SSaaS support to an existing encrypted file system such as eCryptFS [98]. Such a system will allow users to leverage SSaaS-backed file encryption atop existing file-system-like interfaces such as NFS [208], Dropbox [42], or Google Drive [88].

Cloud Encryption Support: I also propose exploring the benefits that would come from integrating SSaaS encryption support directly with one or more cloud storage clients such as Dropbox [42]. Such a cloud-client-integrated solution has potential benefits over a general-purpose file-system encryption systems due to its the ability to unify the sharing support of both the SSaaS ecosystem as well as the backing storage provider in a single interface.

9.3 Analysis

The final component of my proposed work will be to analyze the trust, security, performance, and capabilities of the proposed implementations discussed above. The analysis will focus on the improvements and trade-offs offered by SSaaS-backed applications and the SSaaS model relative to existing and alternate solutions. This analysis will serve to test on the assertions made in this proposal and evaluate the degree to which they hold up in real-world systems.

To analyze trust and security, I'll apply the trust framework discussed in Chapter 5 to the proposed client application implementations. Doing so will also allow comparisons with existing solutions analyzed as proposed in Section 9.1. The trust and security analysis will also focus on the proposed secret sharding implementation with the goal of evaluating the suitability of such methods for practical third-party SSP risk mitigation.

While ideal performance is not a stated goal of the proposed work, I aim to undertake basic performance measurements such as those published in [215]. The goal of such an analysis will be to established the performance trade-offs and overhead using SSaaS methods entails relative to non-SSaaS solutions. Understanding this overhead will help users and designers to make informed decisions about where and how to employ SSaaS methods in a manner that maximizes the security and privacy benefits will minimizing potential performance losses.

Finally, the capability analyses will take stock of the capabilities and features of the proposed implementations relative to existing solution. The aim of such an analysis will be to confirm or refute the assertions that SSaaS systems can increase privacy and security without unduly hindering access to popular features. The analysis will also detail the management and access control capabilities of the proposed SSaaS implementations.

These analyses, combined with the proposed surveys and implementation advances, constitute the body of work proposed for the completion of my doctoral degree. The results of the proposed work will be used to expand the ideas outlines in this document, leading to the creation of my full desertion. I hope to complete all such proposed work over the course of the next year for submission by the end of the Spring 2016 semester.

Bibliography

[1] Ali Abbas, Abdulmotaleb El Saddik, and Ali Miri. A State of the Art Security Taxonomy of Internet Security: Threats and Countermeasures. Computer Science and Engineering, 1(1):27–36, 2005.

[2] AgileBits. 1password. `agilebits.com/onepassword`.

[3] Ron Amadeo. Android L will have device encryption on by default. Ars Technica, Sep 2014.

[4] Amazon. Cloud hsm. `aws.amazon.com/cloudhsm`.

[5] Amazon. Dynamodb. `aws.amazon.com/dynamodb`.

[6] Amazon. Elastic cloud compute. `aws.amazon.com/ec2`.

[7] Amazon. Simple storage service. `aws.amazon.com/s3`.

[8] Ross Anderson. Why information security is hard - an economic perspective. In Seventeenth Annual Computer Security Applications Conference, pages 358–365. IEEE Comput. Soc, 2001.

[9] The Apache Software Foundation. Apache http server project. `httpd.apache.org`.

[10] The Apache Software Foundation. Cassandra. `cassandra.apache.org`.

[11] Apple. Apple pay. `www.apple.com/apple-pay/`.

[12] Apple Inc. Update to celebrity photo investigation. `www.marketwatch.com/story/apple-media-advisory-2014-09-02`, September 2014.

[13] Jeff Atwwod. Building a Computer the Google Way. Cosing Horror: Programming and Human Factors, Mar 2007.

[14] Chris Ballinger. Chatsecure: Encrypted messenger for ios and android. `chatsecure.org`.

[15] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando Andre, and Paulo Sousa. DepSky: Dependable and secure storage in a cloud-of-clouds. In Proceedings of the sixth conference on Computer systems, pages 31–46, 2011.

[16] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-Policy Attribute-Based Encryption. In IEEE Symposium on Security and Privacy, 2007, pages 321–334. IEEE, May 2007.

[17] Bharat Bhargava, Noopur Singh, and Asher D Sinclair. Privacy in Cloud Computing Through Identity Management. SPIE Defense, Security and Sensing 2011 Conference, 298(0704):7, 2011.

[18] Matt Bishop. Unix security: threats and solutions. In SHARE 86.0, number 916, pages 1–38, 1996.

[19] BitTorrent. Sync. `www.getsync.com`.

[20] John Black and Phillip Rogaway. CBC MACs for arbitrary-length messages: The three-key constructions. Journal of Cryptology, 18(2):111–131, 2005.

[21] Matt Blaze. A cryptographic file system for UNIX. the 1st ACM conference, pages 9–16, 1993.

[22] Matt Blaze. Oblivious Key Escrow. Information Hiding, 1174, 1996.

[23] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-Record Communication, or, Why Not To Use PGP. In Workshop on Privacy in the Electronic Society, 2004.

[24] Michael Brenner, Jan Wiebelitz, Gabriele Von Voigt, and Matthew Smith. Secret program execution in the cloud applying homomorphic encryption. IEEE International Conference on Digital Ecosystems and Technologies, 5(June):114–119, 2011.

[25] Peter T. Breuer and Jonathan P. Bowen. A fully homomorphic crypto-processor design: Correctness of a secret computer. In Proceedings of the 5th International Conference on Engineering Secure Software and Systems, ESSoS'13, pages 123–138, Berlin, Heidelberg, 2013. Springer-Verlag.

[26] Jon Brodkin. The secret to online safety: Lies, random characters, and a password manager. Ars Technica, 2013.

[27] Milan Broz. dm-crypt. `code.google.com/p/cryptsetup/wiki/DMCrypt`.

[28] Jose M. Alcaraz Calero, Nigel Edwards, Johannes Kirschnick, Lawrence Wilcock, and Mike Wray. Toward a Multi-Tenancy Authorization System for Cloud Services. IEEE Security & Privacy Magazine, 8(6):48–55, November 2010.

[29] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. RFC 1880: OpenPGP Message Format. Technical report, 2007.

[30] L. Jean Camp. Designing for trust. In Trust, Reputation, and Security. Rino Falcone, Springer-Verlang, Berlin, 2003.

[31] Martìn Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. ACM SIGCOMM Computer Communication Review, 37(4), 2007.

[32] James L. Cebula and Lisa R. Young. A taxonomy of operational cyber security risks. Technical Report December, 2010.

[33] Codenomicon. The heartbleed bug. `heartbleed.com`.

[34] Russ Cox, Eric Grosse, Rob Pike, Dave Presotto, and Sean Quinlan. Security in Plan 9. In USENIX Security, pages 3–16, 2002.

[35] Douglas Crockford. Introducing json. `www.json.org`.

[36] Joan Daemen and Vincent Rijmen. AES Proposal: Rijndael. Technical report, 1999.

[37] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. ACM SIGOPS Operating Systems Review, 41(6):205, October 2007.

[38] Dorothy E. Denning and Dennis K. Branstad. A taxonomy for key escrow encryption systems. Communications of the ACM, 39(3):34–40, March 1996.

[39] T. Dierks and E. Rescorla. RFC 5246: The Transport Layer Security (TLS) Protocol - Version 1.2. Technical report, 2008.

[40] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. IEEE Transactions on Information Theory, 22(6):29–40, 1976.

[41] Yuan Dong, Jinzhan Peng, Dawei Wang, Haiyang Zhu, Sun C. Chan, and Michael P. Mesnier. RFS: a network file system for mobile devices and the cloud. ACM SIGOPS Operating Systems Review, 45(1):101–111, 2011.

[42] Dropbox Inc. Dropbox. `www.dropbox.com`.

[43] Dropbox Inc. Dropbox security. `www.dropbox.com/security`.

[44] Dropbox Inc. Dropbox transparency report. `www.dropbox.com/transparency`.

[45] Dropbox Inc. Dropbox wan't hacked. `blog.dropbox.com/2014/10/dropbox-wasnt-hacked/`.

[46] Dropbox Inc. Yesterday's authentication bug. `blog.dropbox.com/2011/06/yesterdays-authentication-bug/`.

[47] Morris Dworkin. Recommendation for Block Cipher Modes of Operation: The CMAC Mode of Authencation. Technical Report 800-38B, National Institute of Standards & Technology, 2005.

[48] P. J. Eby. Pep 3333: Python web server gateway interface. `www.python.org/dev/peps/pep-3333`.

[49] Bjarni Einarsson, Smair McCarth, and Brennan Novak. Mailpile: Let's take email back. `www.mailpile.is`.

[50] C Ellison. Establishing identity without certification authorities. USENIX Security Symposium, 1996.

[51] EMC2. Pkcs #11: Cryptogrphic token interface standard. `www.emc.com`.

[52] The Enigmail Project. Enigmail. `www.enigmail.net`.

[53] Facebook, Inc. Facebook. www.facebook.com.

[54] Cyrus Farivar. Apple expands data encryption under iOS 8, making handover to cops moot. Ars Technica, Sep 2014.

[55] Federal Bureau of Investigation. FOIPA: National Security Letters. vault.fbi.gov/National%20Security%20Letters%20(NSL), 2007.

[56] Donald G. Firesmith. A taxonomy of security-related requirements. International Workshop on High Assurance Systems, 2005.

[57] Fitbit. About Fitbit. http://www.fitbit.com/about.

[58] Gary Flood. Gartner Tells Outsourcers: Embrace Cloud Or Die. http://www.informationweek.com/cloud/infrastructure-as-a-service/gartner-tells-outsourcers-embrace-cloud-or-die/d/d-id/1110991, 2013.

[59] Stephen Flowerday and Rossouw Von Solms. Trust: An Element of Information Security. In Simone Fischer-Hübner, Kai Rannenberg, Louise Yngström, and Stefan Lindskog, editors, Security and Privacy in Dynamic Environments, volume 201 of IFIP International Federation for Information Processing, pages 87–98. Kluwer Academic Publishers, Boston, 2006.

[60] Ben Foster. How many users on Facebook. www.benphoster.com/facebook-user-growth-chart-2004-2010/, 2014.

[61] Python Software Foundation. shelve - python object persistence. docs.python.org/2/library/shelve.html.

[62] A. Freier, P. Karlton, and P. Kocher. RFC 6106: The Secure Socket Layer (SSL) Protocol - Version 3.0. Technical report, 2011.

[63] Clemens Fruhwirth. Luks. code.google.com/p/cryptsetup.

[64] Kevin E. Fu. Group Sharing and Random Access in Cryptographic Storage File Sytems. Master's thesis, Massachusetts Institute of Technology, 1998.

[65] S. M. Furnell, A. Jusoh, and D. Katsabas. The challenges of understanding and using security: A survey of end-users. Computers & Security, 25(1):27–35, February 2006.

[66] Steven Furnell. Usability versus complexity striking the balance in end-user security. Network Security, (12):13–17, December 2010.

[67] Steven Furnell, Adila Jusoh, Dimitris Katsabas, and Paul Dowland. Considering the Usability of End-User Security Software. In Simone Fischer-Hübner, Kai Rannenberg, Louise Yngström, and Stefan Lindskog, editors, Security and Privacy in Dynamic Environments, volume 201 of IFIP International Federation for Information Processing, pages 307–316. Kluwer Academic Publishers, Boston, 2006.

[68] Gazzang. ztrustee. www.gazzang.com/products/ztrustee.

[69] Roxana Geambasu, John P. John, Steven D. Gribble, Tadayoshi Kohno, and Henry M. Levy. Keypad: an auditing file system for theft-prone devices. In Proceedings of EuroSys '11, pages 1–16, New York, New York, USA, 2011. ACM Press.

[70] Roxana Geambasu, Tadayoshi Kohno, Amit A. Levy, and Henry M. Levy. Vanish: Increasing Data Privacy with Self-Destructing Data. In Proceedings of the 18th Conference on USENIX Security Symposium, pages 299–316, 2009.

[71] Barton Gellman and Ashkan Soltain. Nsa infiltrates links to yahoo, google data centers worldwide, snowden documents say. The Washington Post, Oct 2013.

[72] Craig Gentry. A fully homomorphic encryption scheme. PhD thesis, Stanford University, 2009.

[73] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. ACM SIGOPS Operating Systems Review, 37(5):29–43, December 2003.

[74] NightLabs Consulting GmbH. Cumulus4j. `www.cumulus4j.org`.

[75] Vindu Goel. Facebook Tinkers With Users' Emotions in News Feed Experiment, Stirring Outcry. `www.nytimes.com/2014/06/30/technology/facebook-tinkers-with-users-emotions-in-news-feed-experiment-stirring-outcry.html`, June 2014. New York Times.

[76] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. SiRiUS: Securing Remote Untrusted Storage. NDSS, (0121481), 2003.

[77] Y. Goland, W. Whitehead, A. Faizi, S. Carter, and D. Jensen. RFC 1528: HTTP Extensions for Distributed Authoring – WEBDAV. Technical report, Feb 1999.

[78] Ian Goldberg, Matthew D. Van Gundy, Hao Chen, and Berkant Ustaoglu. Multi-party Off-the-Record Messaging. In Conference on Computer and Communications Security, 2009.

[79] R.P Goldberg. A Survey of Virtual Machine Research. IEEE Computer, 7(4):34–45, 1974.

[80] Dan Goodin. Why passwords have never been weaker, and crackers have never been stronger. Ars Technica, 2012.

[81] Dan Goodin. How the bible and youtube are fueling the next frontier of password cracking. Ars Technica, 2013.

[82] Google. App engine. `cloud.google.com/appengine/docs`.

[83] Google. Authenticator. `code.google.com/p/google-authenticator`.

[84] Google. Chrome. `www.google.com/chrome`.

[85] Google. Compute engine. `cloud.google.com/compute/`.

[86] Google. Data centers - efficiency. `www.google.com/about/datacenters/efficiency/internal/`.

[87] Google. Docs. `docs.google.com`.

[88] Google. Drive. `drive.google.com`.

[89] Google. End-to-end. `github.com/google/end-to-end`.

[90] Google. Gmail. mail.google.com.

[91] Google. Google play music. music.google.com.

[92] Google. Google wallet. wallet.google.com.

[93] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In Proceedings of the 13th ACM conference on Computer and communications security - CCS '06, page 89, New York, New York, USA, 2006. ACM Press.

[94] Matthew Green. The daunting challenge of secure e-mail. The New Yorker, 2013.

[95] Matthew Green. What's the matter with pgp? A Few Thoughts on Crytophraphic Engineering, 2014.

[96] Glenn Greenwald. The crux of the nsa story in one phrase: 'collect it all'. The Guardian, July 2013.

[97] Glenn Greenwald and Ewen MacAskill. Nsa prism program taps in to user data of apple, google, and others. The Guardian, June 2013.

[98] Michael Halcrow. ecryptfs. ecryptfs.org.

[99] Michael Austin Halcrow. eCryptfs : An Enterprise-class Cryptographic Filesystem for Linux. In Ottawa Linux Symposium, pages 201–218, Ottawa, 2005. International Business Machines, Inc.

[100] Eric Haszlakiewicz. json-c. github.com/json-c/json-c/wiki.

[101] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and Alex J. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In Proceedings of the 21st USENIX Security Symposium, 2012.

[102] Heroku. Heroku platform. www.heroku.com.

[103] Kashmir Hill. How Target Figured Out A Teen Girl Was Pregnant Before Her Father Did. www.forbes.com/sites/kashmirhill/2012/02/16/how-target-figured-out-a-teen-girl-was-pregnant-before-her-father-did/, Feb 2012. Forbes.

[104] JH Howard. An overview of the andrew file system. 1988.

[105] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. ACM Transactions on Computer Systems, 6(1):51–81, February 1988.

[106] R. J. Hulsebosch, A. H. Salden, M. S. Bargh, P. W. G. Ebben, and J. Reitsma. Context sensitive access control. In Proceedings of the tenth ACM symposium on Access control models and technologies, page 111, New York, New York, USA, 2005. ACM Press.

[107] IBM. Restful web services: The basics. www.ibm.com/developerworks/webservices/library/ws-restful.

[108] Tarik Ibrahim, Steven M. Furnell, Marira Papadaki, and Nathan L. Clark. Assessing the Usability of End-User Security Software. Trust, Privacy and Security in Digital Business, 6264:177–189, 2010.

[109] Apple Inc. icloud. www.apple.com/icloud.

[110] MongoDB Inc. mongodb. www.mongodb.org.

[111] Intuit. Mint: Money manager, bill pay, credit score, budgeting and investing. www.mint.com.

[112] Christian D. Jensen. CryptoCache: a secure sharable file cache for roaming users. In Proceedings of the 9th ACM SIGOPS European Workshop, page 73, New York, New York, USA, 2000. ACM Press.

[113] Maritza L Johnson. Toward Usable Access Control for End-users: A Case Study of Facebook Privacy Settin PhD thesis, Columbia University, 2012.

[114] Michael K. Johnson. A tour of the linux vfs. www.tldp.org/LDP/khg/HyperNews/get/fs/vfstour.html, 1996.

[115] JumpCloud. Jumpcloud: Identity management reimagined. jumpcloud.com.

[116] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In Proceedings of the 2nd USENIX Conference on File and Storage Technologies, pages 29–42, 2003.

[117] Seny Kamara, C Papamanthou, and Tom Roeder. Cs2: A searchable cryptographic cloud storage system. Technical report, Microsoft Research, 2011.

[118] S. Kelly and S. Frankel. RFC 4868: Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec. Technical report, May 2007.

[119] Zeus Kerravala. Configuration management delivers business resiliency. The Yankee Group, 2002.

[120] Keybase. Keybase: Get a public key, safely, starting just with someone's social media username(s). keybase.io.

[121] Vishal Kher and Yongdae Kim. Securing distributed storage: challenges, techniques, and systems. In Proceedings of the 2005 ACM workshop on Storage security and survivability, page 9, New York, New York, USA, 2005. ACM Press.

[122] Werner Koch. Gnupg. www.gnupg.org.

[123] Werner Koch and Marcus Brinkmann. STEED - Usable End-to-End Encryption. Technical report, 2011.

[124] John T. Kohl, B. Clifford Neuman, and Theodore Y. Ts'o. The evolution of the Kerberos authentication service. In European Conference Proceedings, 1991.

[125] H. Krawczyk, M. Bellare, and R. Canetti. RFC 2104: HMAC - Keyed-Hashing for Message Authentication. Technical report, Feb 1997.

[126] Hugo Krawczyk. Secret Sharing Made Short. In Douglas R. Stinson, editor, Advances in Cryptology-CRYPTO'93, volume 773 of Lecture Notes in Computer Science, pages 136–146. Springer Berlin Heidelberg, Berlin, Heidelberg, July 1994.

[127] Brian Krebs. Safeguarding your passwords. Krebs on Security, 2008.

[128] Brian Krebs. Data breach at health insurer anthem could impact millions. Krebs on Security, Feb 2015.

[129] Brian Krebs. Premera blue cross breach exposes financial, medical records. Krebs on Security, Mar 2015.

[130] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. ACM SIGPLAN Notices, 35(11):190–201, 2000.

[131] Puppet Labs. Puppet. `puppetlabs.com`.

[132] LaCie AG. Wuala: Secure cloud storage. `www.wuala.com`.

[133] LastPass. Lastpass. `lastpass.com`.

[134] B. Laurie, A. Langley, and E. Kasper. RFC 6962: Certificate Transparency. Technical report, 2013.

[135] P. Leach, M. Mealling, and R. Salz. RFC 4122: A universally Unique IDentifier (UUID) URN Namespace. Technical report, 2005.

[136] Marcos A. P. Leandro, Tiago J. Nascimento, Daniel R. dos Santos, Carla M. Westphall, and Carlos B. Westphall. Multi-tenancy authorization system with federated identity for cloud-based environments using shibboleth. In The Eleventh International Conference on Networks, pages 88–93, 2012.

[137] Least Authority. Simple secure storage service (s4). `leastauthority.com`.

[138] Ladar Levison. Lavabit. `lavabit.com`.

[139] Ladar Levison. Secrets, lies and snowden's email: why i was forced to shut down lavabit. The Guardian, May 2014.

[140] Yan Li, Nakul Sanjay Dhotre, Yasuhiro Ohara, Thomas M. Kroeger, Ethan L. Miller, and Darrell D. E. Long. Horus: Fine-Grained Encryption-Based Security for Large-Scale Storage. In Proceedings of the 11th Conference on File and Storage Systems, 2013.

[141] The Linux-PAM Team. Linux pam. `www.linux-pam.org`.

[142] Witold Litwin, Sushil Jajodia, and Thomas Schwarz. Privacy of data outsourced to a cloud for selected readers through client-side encryption. In Proceedings of the 10th annual ACM workshop on Privacy in the electronic society, page 171, New York, New York, USA, 2011. ACM Press.

[143] LogRhythm Inc. Logrhythm: Siem 2.0, log and event management, log analysis. www.logrhythm.com.

[144] Natasha Lomas. Facebook Data Privacy Class Action Joined By 11,000 And Counting. techcrunch.com/2014/08/04/europe-vs-facebook-class-action, Aug 2014. TechCrunch.

[145] Perry Lorier. Heartbleed and private key availability. plus.google.com/106751305389299207737/posts/WM4i4Rqxs5n, 2014.

[146] Philip MacKenzie and Michael K. Reiter. Delegation of cryptographic servers for capture-resilient devices. Distributed Computing, 16(4):307–327, December 2003.

[147] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud Storage with Minimal Trust. ACM Transactions on Computer Systems, 29(4):1–38, December 2011.

[148] A. Matsui, J. Nakajima, and S. Moriai. RFC 3713: A Description of the Camellia Encryption Algorithm. Technical report, 2004.

[149] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. ACM SIGOPS Operating Systems Review, 33(5):124–139, December 1999.

[150] Michelle L. Mazurek, Saranga Komanduri, Timothy Vidas, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Patrick Gage Kelley, Richard Shay, and Blase Ur. Measuring Password Guessability for an Entire University. Technical report, 2013.

[151] Carnegie Mellon. Captcha. www.captcha.net.

[152] Carnegie Mellon. Sasl. asg.web.cmu.edu/sasl.

[153] A. Menezes, P. van Oorschot, and S. Vanstone. Overview of Cryptography. In Handbook of Applied Cryptography, pages 1–48. 1996.

[154] Microsoft. Office Online. office.live.com.

[155] Microsoft. OneDrive. onedrive.live.com.

[156] Microsoft. Server Message Block (SMB) Protocol Versions 2 and 3. Technical Report MS-SMB2-46.0, 2014.

[157] Microsoft. Online-only files and files avaible offline. windows.microsoft.com/en-us/windows-8/onedrive-online-available-offline, 2015.

[158] Stefan Miltchev, Jonathan M. Smith, Vassilis Prevelakis, Angelos Keromytis, and Sotiris Ioannidis. Decentralized access control in distributed file systems. ACM Computing Surveys, 40(3):1–30, August 2008.

[159] MIT Media Lab. OpenPDS Software. Technical report, Human Dynamics, MIT Media Lab, 2012.

[160] Mozilla. Persona. developer.mozilla.org/en-US/Persona.

[161] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, 2008.

[162] National Institute of Standards & Technology. Announcing the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication, 2001.

[163] Ted Nelson. Computer Lib / Dream machines. Self published, 1974.

[164] B. Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. IEEE Communications Magazine, 32(9):33–38, 1994.

[165] Neurio. Neurio Home Intelligence. `www.neur.io`.

[166] National Institute of Standards & Technology (NIST). FIPS 140-2: Security Requirements for Cryptographic Modules. Federal Information Processing Standards Publication, 2001.

[167] nobody@jpunix.com. Thank you bob anderson: Rc4 source code. `http://cypherpunks.venona.com/archive/1994/09/msg00304.html`.

[168] Donald A Norman. The design of everyday things. Basic books, 2002.

[169] OASIS. Saml. `www.oasis-open.org/committees/tc_home.php?wg_abbrev=security`.

[170] The OAuth Team. Oauth. `oauth.net`.

[171] Open Whisper Systems. Security, simplified: Open source securty for mobile devices. `whispersystems.org`.

[172] The OpenBSD Team. Openssh. `www.openssh.com`.

[173] The OpenID Foundation. Openid. `openid.net`.

[174] The OpenPGP Alliance. Openpgp. `www.openpgp.org`.

[175] OpenSSL. Openssl. `www.openssl.org`.

[176] OpenStack Project Team. Barbican. `https://wiki.openstack.org/wiki/Barbican`.

[177] OpenStack Project Team. Openstack. `https://wiki.openstack.org/wiki/Barbican`.

[178] Kurt Opsahl. Warrant Canary Frequently Asked Questions. Technical report, April 2014.

[179] Opscode. Chef. `www.opscode.com/chef`.

[180] OTR Development Team. Off-the-record messaging protocol version 3. Technical report.

[181] OwnCloud. Owncloud server. `owncloud.org`.

[182] Jose Pagliery. Uber removes racy blog posts on prostitution, one-night stands. `money.cnn.com/2014/11/25/technology/uber-prostitutes/`, Nov 2014. CNN Money.

[183] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In Proceedings of the 1988 ACM SIGMOD Conference on Management of Data. ACM, 1988.

[184] PCI Securty Standard Council. Pci ssc data security standards overview. `www.pcisecuritystandards.org/security_standards/index.php`.

[185] Alen Peacock, Xian Ke, and Matthew Wilkerson. Typing patterns: a key to user identification. IEEE Security & Privacy Magazine, 2(5):40–47, September 2004.

[186] Pew Research Center. Public perceptions of privacy and security in a post-snowden era. www.pewinternet.org/2014/11/12/public-privacy-perceptions, November 2014.

[187] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP '11, page 85, New York, New York, USA, 2011. ACM Press.

[188] Porticor. Porticor cloud security. www.porticor.com/technology/.

[189] Rackspace. Cloud keep. github.com/cloudkeep.

[190] RackSpace. Rackspace cloud servers. www.rackspace.com/cloud/servers.

[191] Jarret Raim and Matt Tesauro. Cloud keep: Openstack key management as a service. www.openstack.org/summit/portland-2013/session-videos/presentation/cloud-keep-openstack-key-management-as-a-service.

[192] Chet Ramey. Gnu bash. http://www.gnu.org/software/bash/.

[193] B. Ramsdell and S. Turner. RFC 5751: Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification. Technical report, 2010.

[194] Kenneth Reitz. Requests: Http for humans. www.python-requests.org.

[195] Reporter WIthout Borders. The internet in china. http://surveillance.rsf.org/en/china/, 2012.

[196] Jason K. Resch and James S. Plank. AONT-RS: Blending Security and Performance in Dispersed Storage Systems. In 9th USENIX Conference on File and Storage Technologies, 2011.

[197] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2):120–126, 1978.

[198] Armin Ronacher. Flask: web development one drop at a time. flask.pocoo.org.

[199] Alon Rosen. Analysis of The Porticor Homomorphic Key Management Protocol. Technical report, Porticor, 2012.

[200] Rebecca J. Rosen. Google Death: A Tool to Take Care of Your Gmail When You're Gone. The Atlantic, Apr 2013.

[201] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. ACM SIGOPS Operating Systems Review, 35(5):188, December 2001.

[202] Vinay Sahni. Best practices for designing a pragmatic restful api. www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api.

[203] Saltstack. Salt. `www.saltstack.com`.

[204] Jerome H. Saltzer. Protection and the control of information sharing in multics. Communications of the ACM, 17(7):388–402, July 1974.

[205] JH Saltzer and MD Schroeder. The protection of information in computer systems. Proceedings of the IEEE, 63(9):1278–1308, 1975.

[206] Bharath K. Samanthula, Gerry Howser, Yousef Elmehdwi, and Sanjay Madria. An efficient and secure data sharing framework using homomorphic encryption in the cloud. In Proceedings of the 1st International Workshop on Cloud Intelligence - Cloud-I '12, pages 1–8, New York, New York, USA, 2012. ACM Press.

[207] Vipin Samar. Unified login with pluggable authentication modules (PAM). In Proceedings of the 3rd ACM Conference on Computer and Communications Security, pages 1–10, New York, New York, USA, 1996. ACM Press.

[208] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In Proceedings of the Summer 1985 USENIX Conference, pages 119–130, Portland, OR, 1985.

[209] Ravi S. Sandhu, Edward J. Coynek, Hal L. Feinstein, and Charles E. Youman. Role-Based Access Control Models. IEEE Computer, 29(2):38–47, 1996.

[210] Andy Sayler. Custos server repo. `github.com/asayler/custos-server`.

[211] Andy Sayler. Encfs repo. `github.com/asayler/custos-client-encfs`.

[212] Andy Sayler. libcustos repo. `github.com/asayler/custos-client-libcustos`.

[213] Andy Sayler. Custos: A Flexibly Secure Key-Value Storage Platform. Master's thesis, University of Colorado Boulder, December 2013.

[214] Andy Sayler, Junho Ahn, and Richard Han. Os programming assignment: An encrypted filesystem. `github.com/asayler/CU-CS3753-PA5`.

[215] Andy Sayler and Dirk Grunwald. Custos: Increasing security with secret storage as a service. In 2014 Conference on Timely Results in Operating Systems (TRIOS 14), Broomfield, CO, October 2014. USENIX Association.

[216] Eric Schlosser. Command and Control: Nuclear Weapons, the Damascus Accident, and the Issusion of Safety. The Penguin Press, New York, 2013.

[217] Bruce Schneier. Applied Cryptography. John Wiley & Sons, 1996.

[218] Bruce Schneier. Secrets and Lies. John Wiley & Sons, 2000.

[219] Bruce Schneier. Password advice. Schneier on Security, 2009.

[220] Bruce Schneier. Nsa doesnt need to spy on your calls to learn your secrets. Wired, March 2015.

[221] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, and Chris Hall. Twofish: A 128-bit block cipher. Technical report, 1998.

[222] Adi Shamir. How to share a secret. Communications of the ACM, 22(11):612–613, November 1979.

[223] The Shibboleth Consortium. Shibboleth. shibboleth.net.

[224] Jyh-Ren Shieh, Ching-Yung Lin, and Ja-Ling Wu. Recommendation in the end-to-end encrypted domain. In Proceedings of the 20th ACM international conference on Information and knowledge management - CIKM '11, page 915, New York, New York, USA, 2011. ACM Press.

[225] Peter Sims. Can We Trust Uber? medium.com/@petersimsie/can-we-trust-uber-c0e793deda36, Sep 2014. Medium.

[226] Abe Singer, Warren Anderson, and Rik Farrow. Rethinking Password Policies (Uncut). ;login, 38(4), 2013.

[227] Craig Smith. How Many People Use 800+ of teh Top Social Networks, Apps, and Digital Services. expandedramblings.com/index.php/resource-how-many-people-use-the-top-social-media, March 2005.

[228] S. W. Smith. Fairy dust, secrets, and the real world. Security & Privacy, IEEE, 1(1):89–93, January 2003.

[229] Dag-Erling Smorgrav. Openpam. www.openpam.org.

[230] J.H. Song, R. Poovendran, J. Lee, and T. Iwata. RFC 4493: The AES-CMAC Algorithm. Technical report, June 2006.

[231] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In Proceedings of the 8th USENIX Security Symposium, 1999.

[232] SpiderOak. Spideroak: Store. sync. share. privately. spideroak.com.

[233] Splunk Inc. Splunk: Operational intelligence, log management, application management, enterprise security and compliance. www.splunk.com.

[234] Daniel Stenberg. curl. curl.haxx.se.

[235] Inktank Storage. Ceph: The future of storage. ceph.com.

[236] Inktank Storage. Crush maps. ceph.com/docs/master/rados/operations/crush-map/.

[237] Inktank Storage. Librbd (python). docs.ceph.com/docs/master/rbd/librbdpy/.

[238] Inktank Storage. Librbd source code. github.com/ceph/ceph/tree/master/src/librbd.

[239] Inktank Storage. Rbd-fuse - ecpose rbd images as files. docs.ceph.com/docs/master/man/8/rbd-fuse/.

[240] Ahren Studer and Adrian Perrig. Mobile user location-specific encryption (MULE): Using Your Office as Your Password. In Proceedings of the third ACM conference on Wireless network security - WiSec '10, page 151, New York, New York, USA, 2010. ACM Press.

[241] Supreme Court of United States. United states v. miller. scholar.google.com/scholar_case?case=15052729295643479698, April 1976.

[242] Supreme Court of United States. Smith v. maryland. scholar.google.com/scholar_case?case=3033726127475530815, June 1979.

[243] Michael Sweikata, Gary Watson, Charles Frank, Chris Christensen, and Yi Hu. The usability of end user cryptographic products. In 2009 Information Security Curriculum Development Conference, page 55, New York, New York, USA, 2009. ACM Press.

[244] Symantec Corperation. Shellshock: All you need to know about the bash bug vulnerability. http://www.symantec.com/connect/blogs/shellshock-all-you-need-know-about-bash-bug-vulnerability.

[245] Symantec Corporation. Internet Security Threat Report 2014. 19(April):97, 2014.

[246] Symantic. Pgp. www.symantec.com/encryption.

[247] Miklos Szeredi. Fuse: Filesystems in userspace. fuse.sourceforge.net.

[248] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. CleanOS: Limiting Mobile Data Exposure with Idle Eviction. In Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation, pages 77–91, 2012.

[249] The Open Compute Project Team. The open compute project. http://www.opencompute.org/.

[250] The Redis Team. Redis: Remote dictionary server. redis.io/.

[251] Richard M Thompson, II. The Fourth Amendment Third-Party Doctrine. Technical Report R43586, Congressional Research Service, 2014.

[252] Tresorit. Tresorit: The ultimate was to stay safe in the cloud. tresorit.com.

[253] K. Tsipenyuk, B. Chess, and G. McGraw. Seven pernicious kingdoms: A taxonomy of software security errors. Security & Privacy, IEEE, 3(6):81–84, November 2005.

[254] Hayley Tsukayama. Facebook draws fire from privacy advocates over ad chnages. www.washingtonpost.com/blogs/the-switch/wp/2014/06/12/privacy-experts-say-facebook-changes-open-up-unprecedented-data-collection/, June 2014. Washington Post.

[255] Uber. Uber: Your ride, on demand. www.uber.com.

[256] United States. Foreign Intelligence Surveillance Court. www.fisc.uscourts.gov/.

[257] United States. US Constituion: Amendment 4. www.gpo.gov/fdsys/pkg/CDOC-110hdoc50/pdf/CDOC-110hdoc50.pdf, Sep 1789.

[258] Unknown. Popcorn time: Watch movies and tv shows instantly. popcorntime.io.

[259] U.S. Department of Education. Family educational rights and privacy act. www2.ed.gov/policy/gen/guid/fpco/ferpa/index.html.

[260] U.S. Department of Health and Human Services. Understanding health information privacy. www.hhs.gov/ocr/privacy/hipaa/understanding/index.html.

[261] vintsurf. Dropbox... opening my docs? www.wncinfosec.com/dropbox-opening-my-docs/, Sep 2013.

[262] Whiteout. Whiteout mail: Email encryption for the rest of us. whiteout.io/.

[263] Alma Whitten and J. D. Tygar. Usability of security: A case study. Technical Report 102590, 1998.

[264] Alma Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In Proceedings of the 8th USENIX Security Symposium, pages 679–702, 1999.

[265] Zooko Wilcox-O'Hearn and Brian Warner. Tahoe: the least-authority filesystem. In Proceedings of the 4th ACM international workshop on Storage security and survivability, pages 21–26, New York, New York, USA, 2008. ACM Press.

[266] Duane C. Wilson and Giuseppe Ateniese. To Share or Not to Share in Client-Side Encrypted Clouds. arXiv preprint arXiv:1404.2697, 2014.

[267] Ted Wobber, Aydan Yumerefendi, Martín Abadi, Andrew Birrell, and Daniel R. Simon. Authorizing applications in singularity. ACM SIGOPS Operating Systems Review, 41(3):355, June 2007.

[268] 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):337–345, June 1974.

[269] Yahoo. End-to-end. https://github.com/yahoo/end-to-end.

[270] Tatu Ylonen. SSHsecure login connections over the Internet. Proceedings of the 6th USENIX Security Symposium, 1996.

[271] Yubico. Yubikey standard. www.yubico.com/products/yubikey-hardware/yubikey.